Measuring User Perception for Detecting Unexpected Access to Sensitive Resource in Mobile Apps

Trung Tin Nguyen
CISPA Helmholtz Center for Information Security
tin.nguyen@cispa.saarland

Duc Cuong Nguyen
CISPA Helmholtz Center for Information Security
duc.nguyen@cispa.saarland

Michael Schilling
CISPA Helmholtz Center for Information Security
michael.schilling@cispa.saarland

Gang Wang
University of Illinois at Urbana-Champaign
gangw@illinois.edu

Michael Backes
CISPA Helmholtz Center for Information Security
backes@cispa.saarland

ABSTRACT

Understanding users' perception of app behaviors is an important step to detect data access that violates user expectations. While existing works have used various proxies to infer user expectations (*e.g.*, by analyzing app descriptions), how real-world users perceive an app's data access when they interact with graphical user interfaces (UI) has not been fully explored.

In this paper, we aimed to fill this gap by directly measuring how end-users perceive app behaviors based on graphical UI elements via extensive user studies. The results are used to build an automated tool - GUIBAT (Graphical User Interface Behavioral Analysis Tool) - that detects sensitive resource accesses that violate user expectations. We conducted three user studies in total (N=904). The first two user studies were used to build a semantic mapping between user expectations of sensitive resource accesses and the common graphical UI elements (N=459). The third user study (N=445) was used to validate the performance of GUIBAT in predicting user expectations. By comparing user expectations and the actual app behavior (inferred by static program analysis) for 47,909 Android apps, we found that 75.38% of the apps have at least one unexpected sensitive resource access in which third-party libraries attributed to 46.13%. Our analysis lays a concrete foundation for modeling user expectations based on UI elements. We show the urgent need for more transparent UI designs to better inform users of data access, and call for new tools to support app developers in this endeavor.

CCS CONCEPTS

• **Security and privacy** → *Usability in security and privacy*; • **Human-centered computing** → *User studies*; *Graphical user interfaces.*

KEYWORDS

Android Security; User Interface; User Privacy; Permission; Unexpected Sensitive Resource Access; Usable Security

ACM Reference Format:
Trung Tin Nguyen, Duc Cuong Nguyen, Michael Schilling, Gang Wang, and Michael Backes. 2021. 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 (ASIA CCS '21), June 7–11, 2021, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3433210.3437511

1 INTRODUCTION

In recent years, efforts such as the General Data Protection Regulation (GDPR) have been made to protect user privacy online [21]. A key principle of the regulation is *transparency*, *i.e.*, users should be well-informed regarding how their data is collected, used, and shared [47]. For mobile applications (apps), however, achieving real transparency of data access is still challenging due to the diverse app types and the complex contexts of data usage.

Mobile platforms such as Android use a permission-based system to regulate the access to sensitive data. However, in practice, users are still in a disadvantaged position to protect their privacy. Due to a lack of transparency, it remains unclear for the users when (for example, when using which function of an app) a certain data access takes place. Apps can often combine different granted permissions to access permission-protected resources[1] (referred to as *"sensitive resources"*) in ways that would surprise users [5, 10].

Researchers have worked to detect sensitive resource accesses that violate user expectations, by matching *user expectations* and *apps' actual behaviors*. While app behavior can be inferred by analyzing the app code, user expectation is much more difficult to measure. Most existing works infer user expectation using certain proxies such as app descriptions (*i.e.*, users would expect apps to behave the way described on the description page) [22, 56, 58]; Unfortunately, app description is too coarse-grained and not all users would read app descriptions in practice. More recently, researchers have started to look into another source of information that directly shapes user expectations: user interface (UI) elements (*e.g,* texts

[1]In the Android platform, permission-protected resources are organized into groups regarding the device's capabilities or features. In this work, we focused on the dangerous permission groups (*i.e.*, Contact, Phone, Calendar, Camera, Location, Storage, Microphone, SMS), since these permission groups deal with user-sensitive information.

and images) [2, 28, 68, 69]. However, while text descriptions are self-explanatory with natural language meanings, visual images (referred to as *"icon"*) often convey information in non-verbal ways, and the end-users interpret the meaning of an icon using their pre-existing experience and knowledge [29, 67]. Existing works either only look at *textual* UI elements (*e.g,* the text associated with the button) to examine user expectations [2, 28], or try to predict the intention of apps' icon UI elements but without understanding how real-world users perceive data access when they interact with such graphical UI(s) [68, 69] — which we found less effective in detecting data access that violates user expectations.

In this paper, we want to fill this gap by directly analyzing user expectations and making them measurable. Particularly, we aim to take into account the influence of all UI elements (*icons* and *texts*) on users expectations and to build *a tool* (GUIBAT) that can automatically detect violations of such expectations in mobile apps. This leads us to the following research questions: *(RQ1) Does the output of GUIBAT reflect users' expectations of apps' sensitive resource access?* - and if it does - *(RQ2) How widespread is sensitive resource access that violates users' expectations in the wild?*

First, we conducted two user surveys to understand how end-users perceive graphical UI(s), and what users' expectations are (Section 3). The goal was to build a mapping between user expectations of sensitive resource accesses and apps' icon UI elements. Then, we built a new tool called GUIBAT that learns from users' perception to identify unexpected sensitive resource accesses (Section 4). More specifically, using the study results, we trained a classifier to infer user expectation from icons on the app's UI. For text elements, we followed similar Natural Language Processing approaches used in prior works to examine user expectation [28, 49, 65] since text elements are self-explanatory with natural language meanings. Next, we leveraged static program analysis to find the app's actual accessed sensitive resources. Finally, a sensitive resource access is considered *unexpected* if it is deviating from user expectations.

Our evaluations showed that GUIBAT significantly outperforms prior works in terms of identifying user expectation of sensitive resource accesses when they interact with the app's UI, and revealed the deficient of text-based only approaches. More importantly, we validated GUIBAT's effectiveness with the third user study, our results showed that GUIBAT can accurately reflect users' expectations of sensitive resource accesses in apps. Thereby, we applied GUIBAT on 100,000 Android apps to look at the landscape of unexpected access to sensitive resources in the wild. GUIBAT identified 47,909 apps with UI elements accessing sensitive resources, and 75.38% of the apps have at least one unexpected sensitive resource access. Among these apps, 38.20% have unexpected sensitive resource accesses exclusively attributed by third-party libraries. We believe that our results will shed new light on the transparency of sensitive resource accesses in mobile apps. GUIBAT would (1) help to inform end-users about unexpected access to sensitive resources and (2) help app stores to better control the compliance of apps to the transparency policies. In summary, we make the following contributions:

- We for the first time performed two user studies to build a semantic mapping between user expectations of sensitive resource accesses and common apps' icon UI elements. Our results open a new perspective for identifying the relation between users' perception of icons and the associated sensitive resource accesses. This can lead to better design of apps' graphical UI and enhancement of transparency for accessing sensitive resources. More importantly, we showed that prior works that predict the intention of apps' icon UI elements without understanding how real-world users perceive data access (*e.g.*, relying on benign apps' icon-to-permission association), in large parts do not reflect user expectation.
- We built GUIBAT[2] — a new tool that accounts users' perception for detecting *unexpected sensitive resource accesses* in Android apps, based upon the knowledge gained from two user studies and static control-flow analysis. We further performed the third user study to validate that GUIBAT can accurately reflect user perceptions of sensitive resource accesses, and its efficiency that enables the large-scale study.

Our paper is organized as follows. We give an overview of related work in Section 2, describe how we designed and conducted the series of studies in Section 3, and how we built GUIBAT in Section 4. We present our results in Section 5, and discuss our findings and suggest actionable items in Section 6. Section 7 concludes our paper.

2 RELATED WORK

Understanding and Supporting Users. Many studies showed that users have low attention and comprehension rate to install-time permission dialogs in Android apps [17, 32]. Further, with both install-time and runtime permission mechanisms, users were often surprised at many permissions that they have granted to their apps and possible associated risks [5, 10, 31]. A number of tools were developed to help users better manage their privacy. For example, Roesner et al. proposed an access control gadget where permission granting is built along with user actions [59]. Li et al. introduced PERUIM that shows how permissions are used by different UI elements by combining static and dynamic analysis [37].

Detecting Unexpected Sensitive Resource Access. A related line of work aims to detect sensitive resource accesses that deviate from the app descriptions [22, 56, 58]. The idea is to use app descriptions as a proxy for user-expected behavior. However, Yu et al. showed that relying on the app description alone is not sufficient and could lead to large errors [73]. Recently, researchers have started to use app UIs as a proxy to study user expectations. They try to detect the mismatches between expected sensitive resource access in the text of UI elements and its actual accessed sensitive resources [2, 28].

Unfortunately, how users perceive data access when they interact with icons which is one of the most important UI elements for user interaction [20, 51], have been neglected in most research so far. A key reason is that it is difficult to map graphical icons to user expectations without actually performing user studies. Recently, Xiao et al. built Icontent to identify expected sensitive resource accesses of apps' icons by manually labeling images on Google Image. DeepIntent used benign apps' icon-to-permission association as the "norm", and aimed to detect malicious apps that deviate from the norm to detect malware [68]. However, both Icontent and DeepIntent did not measure end-users' expectations of icons.

[2]GUIBAT is available at https://github.com/ttincs/guibat.

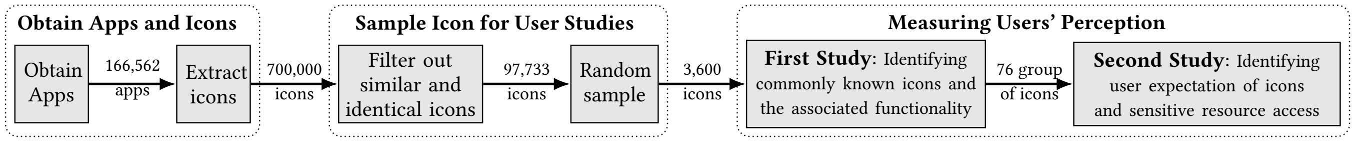

Obtain Apps and Icons
Obtain Apps
166,562 apps
Extract icons
700,000 icons
Sample Icon for User Studies
Filter out similar and identical icons
97,733 icons
Random sample
3,600 icons
Measuring Users' Perception
First Study: Identifying commonly known icons and the associated functionality
76 group of icons
Second Study: Identifying user expectation of icons and sensitive resource access

Figure 1: Overview of the methodologies to measure user's perception.

Different from the prior works, our first goal was to build a semantic mapping between user expectations of sensitive resource accesses and apps' icon UI elements via extensive user studies. Second, we aimed to take into account the influence of all UI elements (icons and texts) on users expectations and to detect unexpected sensitive resource accesses in mobile apps. Our results show that even benign apps can have misleading UIs that lead to violation of users' expectation. More importantly, it shows that relying on benign apps' icon-to-permission association as the "norm" to detect behavior discrepancy is not enough.

3 MEASURING USERS' PERCEPTION

We aimed to build a comprehensive mapping between app icons and their associated sensitive resource accesses in mobile apps. Figure 1 gives an overview of our approach. We first developed a crawler to obtain a large number of Android apps, and extracted the icon UI elements from their apps' UI (Section 3.1). Next, we conducted a series of studies[3] to measure how real-world users perceive an app's data access when they interact with such apps' icons. More specifically, we divided our studies into two parts: one is to identify *commonly-known* icons (*e.g.,* icons are frequently used) to users (Section 3.2), since it is practically impossible to study all available icons of apps with users because icons are extremely diverse in appearance, and not all icons are supposed to convey meaning to users [13]; the other is to measure user expectation of sensitive resource accesses of these icons (Section 3.3). Once we had established the expectations of end-users between the app's icons and sensitive resource accesses, we leveraged this knowledge to build `GUIBAT` which we describe in Section 4.

3.1 App and Icon Dataset

App Dataset. We built a crawler to crawl Android apps from the Google Play store and successfully obtained about 600,000 apps. Among them, we only selected apps that request dangerous permissions because they access users' sensitive data. Further, we used the permission mappings of `PScout` [1] and `Axplorer` [4] to identify accessed sensitive resources of a given Android app. As the Android's permission documentation is incomplete, the combination of `PScout` and `Axplorer` provides a complete mapping from Android version 2.2 up to 7.1. We only selected apps that declare at least one dangerous permission and filtered out apps that are not supported by `PScout` and `Axplorer` based on the *min* and *max* SDK versions (min/max platform version to which the app is compatible). Besides, apps that belong to game categories were excluded since they have little to no UI elements or their UI is mainly made of drawings on canvases, which are out of the scope for this work. After filtering, our app dataset contains 166,562 apps.

Icon Dataset. From the 166,562 apps, we then leveraged existing works [2, 19] to perform static analysis on both the UI layout files and the app's code to extract apps' icons (see Section 4.1 on implementation details). This resulted in about 700,000 icons. To get an overview of these icons, we then used perceptual hashing [74] for filtering out identical and very similar icons at pixel levels — which resulted in 97,733 dissimilar icons (see Appendix A.1.1 for details).

3.2 First Study: Identifying Commonly-Known Icons and The Associated Functionality

We conducted this study online via Amazon Mechanical Turk (MTurk)[4] to determine which icons that are commonly-known and familiar to users based on population stereotype [13] (*e.g.,* an icon known by most of the participants is considered commonly-known). We randomly selected 3,600 icons from our set of dissimilar icons, and asked participants about their subjective feelings, if they have any concrete expectation(s) of a given icon or a similar looking icon (*i.e., "Do you have a concrete expectation what could happen if you press this icon or a similar symbol?"*). The participant's responses are binary: *"yes"* and *"no"*. To identify the careless respondents, we used the instructed response items which is the most popular form of attention check [44, 63], for example, items are embedded with an obvious correct answer (*e.g., "please select yes"*). The survey was designed to ask each participant 90 icons and 10 more attention check icons. This resulted in 40 batches. For each batch, we tested with three different participants to examine whether an icon is commonly-known. If a participant failed attention checks, the corresponding batch would be re-conducted with another participant. An icon was considered commonly-known to users if all participants know it (*i.e.,* 100% recognizable).

The survey lasted for 2 days and we collected valid responses from 120 Turkers. Their median age is 31.5 years (65% male and 35% female, see Table 8 in Appendix A.2 for full demographics). We identified 972 commonly-known icons and found that the majority of surveyed icons (73%) are not linked to any concrete expectations of users and therefore have no intrinsic meaning to them.

Identifying Associated Functionality. We then conducted a follow-up survey to find the perceived function an icon represents. This helps us not only group icons based on their functionality but also quickly filter out icons that do not represent any functionality (given that concrete expectations can only be formed if users are

[3]We compensated participants based on an average hourly wage of $14 (above US minimum wage, https://www.dol.gov/general/topic/wages/minimumwage). More importantly, all user studies in this work were approved by the ethical review board of our university. Web access to the server was secured with an SSL certificate issued by the university's computing center, and all further access was restricted to the department's intranet and only made available to maintainers and collaborating researchers. Participants could leave the studies at any time.

[4]https://www.mturk.com/

Table 1: Icon groups and user expectation of sensitive resource accesses . An icon group has multiple similar looking icons. The agreement rate is in parenthesis.

Icon group	Icon group	Icon group
Calendar (1.0)	Calendar (0.89)	Storage (0.88)
Contacts (1.0)	Contacts (0.88)	Storage (0.75)
Contacts (0.67) SMS (0.67)	Contacts (0.67) SMS (0.67)	Storage (0.75)
Camera (1.0) Storage (0.75)	Camera (1.0) Storage (0.67)	Storage (0.75) Microphone (0.67)
Location (1.0)	Location (1.0)	Storage (0.67)
Microphone (1.0)	Phone (1.0)	Storage (0.67)
SMS (0.75)	Storage (1.0) Camera (0.86)	

somehow familiar with an object [43]). This would significantly reduce the number of icons we need to study later. To do that, we used the icon intuitiveness test to learn about users' pre-existing knowledge of known and familiar icons [50]. An icon was shown to a group of participants without contexts (*e.g.,* without textual description). We asked participants to describe their understanding of a given icon, and their expectations of what would happen if they interact with it (*i.e., Q1: What does this icon symbolize?, Q2: What you would expect to happen if you interact with this icon or an icon that has a similar symbol?*). In this survey, we showed each participant 10 icons, and for each icon, we also asked three different participants to describe their interpretations (free-text responses).

The survey involved 294 participants and lasted 8 days. Participants' median age is 34 years (58.16% male, 41.50% female, see Table 8 in Appendix A.2 for full demographics). We obtained 2,916 feedbacks for 972 commonly-known icons. After extracting users' responses, we constructed a mapping between commonly-known icons and the function they represent by performing description-based icons clustering (see Appendix A.1.2 for details): (1) we first clustered similar icons using textual descriptions of their visual representation, since apps' icons are extremely diverse in appearance (*e.g.,* icons representing the same camera object may look very different); (2) we then further clustered these clusters of icons using descriptions of their functions. In the first step, we identified 76 groups of icons from 972 commonly-known icons. Finally, we identified 44 groups of functions from 76 groups of icons. Our results showed that although icons are extremely diverse in appearance, users will have the same expectations about the associated functionality if they represent the same object. This suggests that by focusing on the most commonly-known icons, our technique could cover most of the icons on which users have perception.

3.3 Second Study: Identifying User Expectation of Icons and Sensitive Resource Access

We had at this point a set of commonly-known icons and their associated functionalities. Our studies' final step was to build a semantic mapping between user expectations of sensitive resource accesses and the apps' icons. From 76 clusters of commonly-known icons, we randomly selected 3 icons per cluster to conduct this study. Our survey first provided the participants with an instruction page that gave a detailed explanation of each sensitive resource to minimize technical terms. We further asked participants an attention and comprehension check question to see if the participants understand these sensitive resources (see Appendix A.2.1 for details).

Afterward, we provided multiple choice answers, where the participants could choose the sensitive resources that an icon could associate with (*i.e., "Which of the following sensitive resources that you would expect this app to access to perform the function it represents?"*). Each of the sensitive resources was accompanied by a corresponding explanation taken from the Android system to leverage user prior experience. Participants could choose "NONE" if no sensitive resource is expected. Also, we asked participants to explain their answers (*i.e., "Why would you think the above selected sensitive resources are needed?"*). This question helps us to see if participants provide meaningful responses. To minimize the bias introduced by a single participant for each icon, we asked three different participants. The survey involved 45 participants and lasted for 3 days. Participants' median age is 33 years (68.89% male and 31.11% female, see Table 8 in Appendix A.2 for full demographics). Finally, we got 684 feedback on the commonly-known icons and their related sensitive resources.

To build a comprehensive mapping between app icons and their associated sensitive resource accesses, we based on the majority of votes for the associated sensitive resources among the participants. Specifically, for each cluster of icons, we calculated the agreement rate on an associated sensitive resource (R_i) by the number of votes for R_i divided by the total number of votes. We considered a cluster of icons relating to R_i if its agreement rate was at least 0.66 which meets the standard of the icon recognition ISO 3864 [30] (*i.e.,* 66.7% for signs). Details of the icons and their agreement rates are listed in Table 1. Specifically, we identified 20 groups of icons that associate with sensitive resources from users' perspective. In this study, we found that one specific icon could be associated with more than one sensitive resource from users' perspective. For example, the icons that represent a camera (see Table 1) can be associated with CAMERA and STORAGE sensitive resources (*e.g.,* explained by a participant *"It's an icon of a camera so it'd have to access your camera to work, and I think it'd have access to your storage to store the photos you take with the app."*). This suggests that approaches only consider one-to-one mapping between icons and sensitive resources might not correctly reflect users' expectations (*e.g.,* `IconIntent` [69]).

4 DETECTING UNEXPECTED SENSITIVE RESOURCE ACCESS

So far, we have conducted two studies to establish a semantic mapping between common apps' icons and user expectation of apps' sensitive resource accesses. Recall that our goal is to have a fully automatic and scalable solution to detect *unexpected sensitive resource accesses* based on user perception. We now leveraged the knowledge gained from the two user studies to build `GUIBAT` that estimates user expectation to detect *unexpected sensitive resource accesses* in Android apps. Figure 2 presents the architecture of `GUIBAT`. For a given app, *GUI Extractor* first extracts all the app's UI elements (*e.g.,* buttons, images), and their associated callbacks which handle users

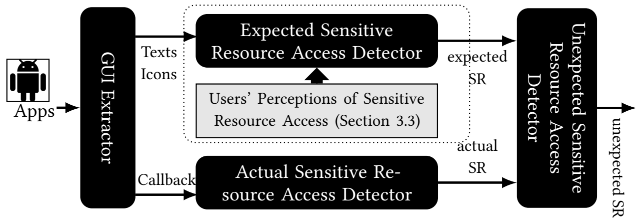
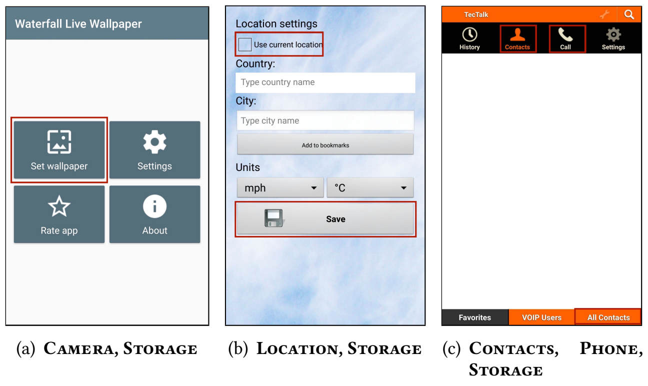

Apps
GUI Extractor
Texts
Icons
Callback
Expected Sensitive Resource Access Detector
Users' Perceptions of Sensitive Resource Access (Section 3.3)
Actual Sensitive Resource Access Detector
expected SR
actual SR
Unexpected Sensitive Resource Access Detector
unexpected SR

Figure 2: Overview of the GUIBAT's architecture.



(a) Camera, Storage (b) Location, Storage (c) Contacts, Phone, Storage

Figure 3: Example of expected sensitive resource accesses.

interaction with the app's UI (Section 4.1). Then, using our study results in Section 3.3, the *Expected Sensitive Resource Access Detector* utilizes a classifier built on top of Natural Language Processing and Image Recognition techniques to identify the app's expected sensitive resource accesses based on the texts and icons of the extracted UI elements (Section 4.2), which we refer as *expected SR*. On the other hand, for the corresponding extracted UI element, the *Actual Sensitive Resource Access Detector* leverages Static Control-flow Analysis techniques to detect the app's actual sensitive resource accesses from sequences of callbacks by looking for permission-protected API calls (Section 4.3), which we refer as *actual SR*. An *actual SR* will be considered unexpected (referred to as *unexpected SR*) by the *Unexpected Sensitive Resource Access Detector* (Section 4.4) if it is not contained in the *expected SR*.

In this section, we describe each component in details. We then present how we designed and conducted the third user study to validate GUIBAT's results in Section 5.1.

4.1 GUI Extractor

This component aimed to identify the app's UI elements and their associated callbacks and to extract the associated graphical content (*i.e.,* texts and icons) that comprise an app's UI. Prior works have proposed different approaches to analyze the app's GUI (e.g., GATOR [60, 72], Backstage [2]). Among them, GATOR is a widely-used static analysis toolkit for Android that analyzes an app's UI by providing an over-approximation algorithm to infer the relationship between app's callbacks and app's UI elements [68, 69]. Further, GATOR identifies both static layout files and dynamically generated UI components. Moreover, the authors showed that it achieved good precision, takes less computational memory. Therefore, to discover the apps' UI elements and their callbacks, we extended GATOR with some improvements (*i.e.,* better coverage and higher precision, see Appendix A.1.3).

Second, to extract texts and icons, we followed similar approaches used in prior works [2, 68]. Specifically, *GUI Extractor* first statically parses the app's UI layout file[5] (*i.e.,* layout files are in XML to define app's UI), and then uses the XML text-related attributes (*e.g., android:text, android:title*) to extract UI texts, and image-related attributes (*e.g., android:drawable, android:icon*) to extract UI icons. Further, by analyzing app's byte-code, *GUI Extractor* extracts UI texts and icons that are dynamically created at runtime by identifiying the used of text-related APIs (*e.g., setText(), setTooltipText()*), or image-related APIs (*e.g., setImageResource(), setIcon()*). *GUI Extractor* further extracts additional information that are shown to users after they press on the app's UI elements. It first finds all the instantiations of *Toast, Snackbar, Dialog* and *AlertDialog* objects which allow an app to send feedback messages to end-users. Then it extracts the text message based on the used API (*e.g., Toast.makeText*). For example, clicking on a button shows the *"Start recording"* text.

[5]https://developer.android.com/guide/topics/ui/declaring-layout

4.2 Expected Sensitive Resource Access Detector

After having all the extracted UI elements of a given app, we want to automatically detect if they represent any sensitive resource accesses from users' perspective. Our idea is based on the intuition that an app's UI (*i.e.,* presented as texts and icons) depicts the user's expectation of app's behavior [28]. More importantly, we can not treat a UI element as a stand-alone element on a screen as an app activity's UI may be comprised of different UI elements, and their *expected SR* can complement each other. Therefore, from each extracted UI element, GUIBAT's *Expected Sensitive Resource Access Detector* first uses *GUI Extractor* to extract the contextual information (the texts and icons of the surrounding UI elements) where the UI element is represented. Then, it performs icons classification, in combination with the acquired user perception (Section 3) to automatically identify the *expected SR* of icons. Additionally, we also leverage prior works to identify *expected SR* in texts [28, 49, 65].

Examples of *expected SR* (according to our user studies) are depicted in Figure 3. In Figure 3(*a*) the *expected SR* is accessing Camera, Storage represented by a gallery icon (see Table 1). In Figure 3(*b*) the *expected SR*s are Location represented by the *Use current location* text (see Table 2), and Storage represented by the *Save* text with a floppy disk icon. Finally, in Figure 3(*c*) the *expected SR*s are Contacts represented by *Contacts* text with the human icon, and Phone represented by *Call* text with the phone icon.

4.2.1 *Icon Classifier.*

This component aimed to identify user expected sensitive resource accesses based on apps' icons. To this end, we used Convolutional Neural Networks (CNNs), which is a deep learning model that achieves state-of-the-art results in image recognition challenges, and widely used in image classification tasks [24]. We abstained from using perceptual hashing [74] because it can only identify identical or very similar looking images, while

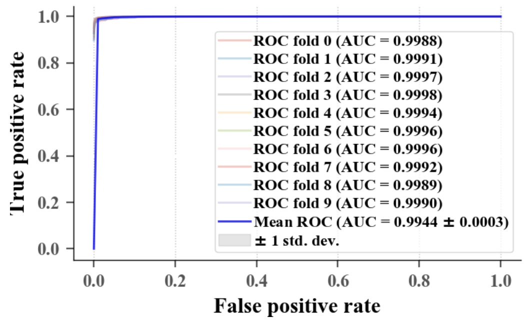


Figure 4: ROC curves of the 10-Fold cross-valication.

our goal is to maximize the robustness of GUIBAT in identifying 20 groups of icons in Table 1, a more challenging classification task.

Our *Icon Classifier* is a multi-class CNN classifier. Given an input (*i.e.*, an icon), the Classifier produces the probability of this icon associating with the 20 groups of icons in Table 1. To determine whether a given icon is associated with sensitive resources (from user perspectives), we first use the *Icon Classifier* to predict the probabilities for 20 groups of icons. If the icon's prediction result has the highest confidence probability at group$_i$, and also its probability is higher than 0.99, we then consider that the icon is associated with the corresponding sensitive resource[6]. Our user studies have already collected a labeled dataset reflecting user-perceived association between icons and sensitive resource accesses. As such, we expect the Classifier to capture user expectations of icons.

Constructing Training Dataset. For each icon group in Table 1, we manually selected 500 similar icons based on the similarity of their visual feature from our icon dataset. The manually labeling method helps us enrich the generalization of the *Icon Classifier*. An icon is considered similar to one of the sensitive related icons in Table 1 if it represents the same object (*e.g.,* camera). Each labeled icon was reviewed by two volunteers independently. If there was a disagreement between the two volunteers, we would ask another volunteer to join the discussion. If an agreement could not be reached, we simply excluded that icon. Further, our dataset as any real dataset contains noisy data such as icons with different sizes, or icons with different formats (*e.g.,* RGB, RGBA) which potentially lead to low-quality models. Therefore, the following preprocessing methods were applied to obtain a quality training dataset [34, 68]: (1) resizing icons to 128x128 pixels (which is most common icon size); (2) converting the RGBA to RGB without affecting the image's content. Finally, our training dataset has 10,000 labeled icons pertaining to sensitive resources across 20 groups of icons.

Training. We employed a self-training method to leverage unlabeled data at scale, which has been widely used in image processing [61, 62, 70]. Specifically, we first implemented the SimpleNet architecture, a light-weighed deep CNN architecture that achieves high precision [24]. Then we trained the *SimpleNet* model on the labeled dataset. We used it as a "teacher" to generate pseudo labels on 700,000 unlabeled icons. Subsequently, we trained a larger *SimpleNet* on the combination of labeled and pseudo labeled icons to produce more "student" data. We iterated this process by putting back the "student" into the "teacher" data. During this process, we kept increasing the size of the "student" data to improve performance. This iterative process was stopped when the size of "student" data became stable (*i.e.*, our final training dataset has 46,578 labeled icons across 20 groups of icons).

[6]We exclude the icons that could not be mapped to any group in Table 1 (probability < 0.99). If an icon has the same probability for 2 groups (or more), we consider it to be associated with the corresponding sensitive resources behind these groups.

Table 2: Sensitive Related Keywords.

Sensitive Resources	Keywords
Calendar	*calendar, calender, event, reminder, meeting, schedule, agenda*
Contacts	*contact, account, call*
Camera	*take picture, camera, capture, scan*
Location	*location, map, gps, track*
Microphone	*microphone, recording, record, audio, voice, mic*
Phone	*call, telephone*
SMS	*sms, mms, send, incoming, voicemail*
Storage	*storage, sd card, file, save*

When putting back the "student" data to the training dataset, our training dataset became imbalanced among some classes which could significantly affect the *classifier*'s performance. Therefore, we first leveraged the Cost Sensitive Learning method in which the weights of each class (calculated based on sample frequencies) are integrated into the cost function [25]. Further, to reduce overfitting and to build a quality image classifier, we employed data augmentation technique that helps diversitize our training dataset which is randomly zooming into images, and the zoom ranges from 0 to 10% of the original images [14].

Evaluation. To validate our "teacher" model, the k-Fold cross validation was selected, together with k = 10 as this was shown to be the best method for cross validation by prior work [33]. Further, to evaluate the model, we used the area under the curve (AUC) of the receiver operating characteristic (ROC) [23]. The AUC is the most commonly used for evaluation of imbalanced data classification [12, 57], where the area near up to 1.0 represents a perfect model, and the area of 0.5 represents a random guessing model. Figure 4 shows the AUC values for 10-Fold cross validation. Our *classifier* has an AUC's mean value of 0.9944.

4.2.2 *Identifying Expected Sensitive Resource Accesses in Texts.*

This component aimed to identify the user-expected sensitive resource access based on texts of UI elements, since in different UI contexts, similar icons may reflect different intentions [68]. From the extracted texts of UI elements[7], we followed prior works [49, 65] to rely on keywords to map texts to *expected SR*. Most of the visible texts on UI elements are short (combination of *verbs*, or *nouns*) but they are self-explanatory. Therefore, we first collected security and privacy relevant keywords from existing works [49, 65], then we manually examined the Android documentation[8] regarding the permission-protected resources to expand the keyword list. The final list is shown in Table 2. Given an input text *t* (*i.e.,* a list of words), GUIBAT first uses a set of Natural Language Processing techniques

[7]The embedded texts in icons we are also extracted using Optical Character Recognition (OCR) techniques (https://github.com/tesseract-ocr/tesseract).
[8]https://developer.android.com/reference/android/Manifest.permission

to preprocess t: normalizing and lemmatizing, removing generic stop words [26, 40, 65]. Then by searching the sensitive related keywords (see Table 2) in t, GUIBAT can identify the corresponding *expected SR*.

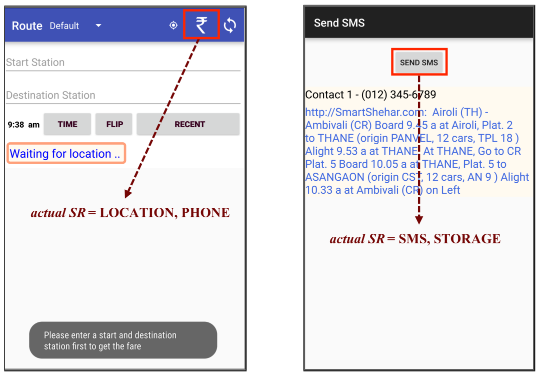
4.3 Actual Sensitive Resource Access Detector

To detect actual sensitive resource accesses of apps' UI elements, we first leveraged GATOR to build the control-flow graph (CFG) that includes all the reachable API calls from a UI element's callbacks. We further extended this component to support multi-threading (*e.g.*, *AsyncTask.execute*) which was not covered by default. GATOR was then used to expand the built CFG by analyzing the inter-component communication (ICC) (*e.g.*, permission-protected APIs may be triggered via services or broadcast receivers), since many of the vulnerabilities and malicious behavior in Android apps addressed in the literature are related to the ICC mechanism [35, 52, 53]. After identifying all reachable API calls, we used PScout and Axplorer to find out which API is permission-protected APIs (*i.e.*, sensitive resources) [1, 4]. Accesses of content providers where apps can access sensitive resources were also detected by querying the content provider with URIs. For instance, by providing the URI content *"content://com.android.contacts"* to the *ContentResolver.Query* method, the app can access user CONTACTS.

We further leveraged LibScout [3] to identify *actual SR* of third-party libraries (*i.e.*, providing the information of 501 commonly used libraries). Additionally, to cover cases of unknown third-party code that LibScout could not identify, we extended its implementation to identify libraries using app package name as a heuristic [2, 36, 49]. It is not practical to build a CFG, which includes all of the reachable code statements due to execution time. Therefore, we needed to find a threshold to limit the depth of the CFG analysis counting from a callback. In our experiments, we randomly selected and analyzed 10,000 apps from the app dataset. With a maximum depth of 350 calls from the corresponding callback we could successfully identifies 96.52% of the accessed sensitive resources of app's UI elements (see Figure 8 in Appendix A.1.4 for details).

Evaluation. To evaluate the precision of GUIBAT on mapping *actual SR* to app's UI elements, we first randomly selected 200 Android apps from 32 categories (*i.e.*, predefined categories on Google Play), and used dynamic analysis to extract runtime *actual SR* of each app's UI element. Specifically, we used the dynamic instrumentation toolkit Frida[9] to instrument the Android system to monitor access of sensitive resources while running the app by logging at each permission-protected API calls and URI queries, and used DroidBot [38] an automatic event generation tool, to simulate user-interaction with the apps (performs best in comparison with similar tools [6]). This way, we could automatically create ground truth for apps' UI elements and their *actual SR*. In this experiment, we limited the automated analysis time to 30 minutes for each app. As the results, we successfully identified 1,284 UI-sensitive resources mappings. Then we used GUIBAT's *Accessed Sensitive Resource Detector* to extract app's UI elements that access sensitive resources from these apps, and compare its results with the ground truth. The average precision and recall of GUIBAT is 90.19% and 96.65%

[9]https://frida.re/



Please enter a start and destination station first to get the fare

Contact 1 - (012) 345-6789
http://SmartShehar.com: Airoli (TH) - Ambivali (CR) Board 9.45 a at Airoli, Plat. 2 to THANE (origin PANVEL, 12 cars, TPL 18) Alight 9.53 a at THANE, At THANE, Go to CR Plat. 5 Board 10.05 a at THANE, Plat. 5 to ASANGAON (origin CST, 12 cars, AN 9) Alight 10.33 a at Ambivali (CR) on Left

(a) ***unexpected SR*:PHONE** (b) ***unexpected SR*:STORAGE**

Figure 5: *Unexpected SR* of a benign app (on Play Store).

respectively. We did not calculate the recall of this component for the whole apps, because it is practically impossible to calculate how many mapping relations are missed by GUIBAT as there is no ground truth available, and due to the limitation of dynamic analysis which can not guarantee all the app code are analyzed.

4.4 Unexpected Sensitive Resource Access Detector

The final step in our work-flow (see Figure 2) is to find unexpected access to sensitive resources in a given app. Specifically, GUIBAT first identifies the user expectation of sensitive resource accesses from the apps' UIs (*expected SR*), then the actually accessed sensitive resources (*actual SR*) using *Expected Sensitive-Resource Access Detector*, and *Actual Sensitive-Resource Access Detector* respectively. An *actual SR* is considered unexpected if it is not contained in the *expected SR*. For example, in Figure 5 (a), when users click on the highlighted button, the app then accesses both LOCATION, and PHONE (*e.g.,* to get the unique user's device ID such as the IMEI) sensitive resources. GUIBAT detects an *unexpected SR* since PHONE sensitive resource is not expected in any related context, *e.g.,* on the surrounding UI elements, or in the feedback message at the bottom of the screen when users click on this button. Similarly, in Figure 5 (b), GUIBAT identifies the *Send SMS* as an *unexpected SR*, since it accesses user's phone STORAGE in addition to SMS while no further information is provided on this screen.

5 RESULTS

In this section, we present our results regarding the research questions stated in Section 1. Our results show that GUIBAT is superior than prior works on detecting *unexpected SR* with users' expectations. After that, we used GUIBAT to analyze a representative sample of 100,000 Android apps for *unexpected SR*.

5.1 RQ1: Does the output of GUIBAT reflect users' expectations of apps' sensitive resource access?

To answer RQ1, we carried out a further user study and tested users' actual expectations while interacting with an app. The goal was to check whether GUIBAT's output reflects these expectations. The main hypothesis was: *If our tool works as intended, sensitive resource access classified by GUIBAT as "**unexpected**" should be less expected by users than the one classified as "**expected**".*

However, previous work showed that other variables might also influence users' perception, attitudes, and expectations in the context of mobile apps. Particularly, the following factors have been primarily identified: 1) the users' individual characteristics (*e.g.*, gender, age, whether the persons have computer science background, or which mobile operating system people use) [20]; 2) the apps' characteristics (*e.g.*, which sensitive resources the app uses) [10, 49]; and 3) the user perception of the app descriptions [22, 56, 58]. To ensure that such effects do not overshadow our results, and, to be able to test whether GUIBAT provides added value compared to these known influential factors of user expectations, we included them in our study design as well.

Study Design and Procedure. The goal of our study was to simulate the users' interaction with an app as close to reality as possible. However, it was also important to cover different types of apps and different sensitive resources to be as representative as possible in our evaluation. After balancing these two goals, we decided on a survey design in which participants were shown an app, and then mentally put themselves (with the help of the app description and screenshots) in the situation that they were using this app.

We first presented participants the description of an app retrieved from Google Play. After reading the description, participants could select from a list of 8 sensitive resources (*e.g.*, or none of them) those they expected the app to access. Participants were then asked to explain briefly why they thought the app needed the selected resources (see *Q1* and *Q2* in Appendix A.2.2). This step prepared participants to put themselves in the situation of using the app, as it is the same information they see in Google Play before downloading and installing an app. This setup also allowed us to measure the effect of the app descriptions on the users' expectations of sensitive resource accesses and to control for this effect. Next, we asked participants to imagine that they were using the app and showed them a concrete screenshot taken from the corresponding app. For all *actual SR* that are detected by GUIBAT, participants should then indicate on a 7-point Likert scale to what extent they expect the app to access these resources (see *Q3* in Appendix A.2.2). We then asked participants to what extent they feel comfortable with the accessing of these resources (on a 7-point Likert scale) (see *Q4* in Appendix A.2.2). Lastly, we collected demographic data of our participants, including questions about their predominantly used mobile operating system and their background in computer science.

Selection of Apps. We randomly selected 1,000 apps from the app dataset that was already used to extract the icons during the development of GUIBAT. Then we used DroidBot [38] to simulate user-interaction and automatically took screenshots of these apps. We limited the time for the dynamic analysis to 10 minutes for each app. Using this setting, DroidBot successfully analyzed 977 apps (97.7% of the total apps). Among 977 apps, we successfully extracted screenshots of *unexpected SR* in 311 apps by using GUIBAT. In a final step, we manually filtered out apps that were not in English or that were protected by FLAG_SECURE whose UI was not shown on screenshots. This resulted in a final set of screenshots of N_{apps} = 243 apps in which GUIBAT identified a total of 810 cases of sensitive resource access (see Table 3).

Table 3: Detected sensitive resource accesses in our sample.

	Number of Sensitive Resource Accesses	
	Expected	Unexpected
Phone	19	104
Location	55	89
Storage	130	76
Contacts	118	16
Camera	22	13
Microphone	33	8
Calendar	18	1
SMS	108	0
Total	**503**	**307**

Survey. The survey lasted 6 days and 486 Turkers had participated (for each of the selected apps, we surveyed with two participants). After collecting all the responses, we removed 41 careless responses based on attention checks. Participants' median age is 34 years (41.35% male, 57.98% female, and 0.67% others, see Table 8 in Appendix A.2 for full demographics). Since we have included gender as a potential influencing factor in our analyses and only 3 persons indicated a non-binary gender, we unfortunately had to exclude these answers, since the model calculation based on such few data points would have been very error-prone. In the end, our final sample included N_{total} = 1,444 individual expectation ratings for sensitive resource access given by N_{users} = 442 participants and based on N_{apps} = 243 apps.

Data Analysis. To analyze the effects of all the variables listed at the beginning of this section, we used a regression approach that predicts participants expectations regarding sensitive resource accesses. Since each participant in our survey indicated his/her expectation for several types of sensitive resource access, these data points are not independent from each other. The same applies to the data from any particular app, which was always presented to several participants. To account for this fact, we designed our analysis as a multi-level approach and included these two grouping aspects of our data as *random effects* in our model. In concrete terms, this means that the intercept of our regression function is not fixed for all persons and all apps, but can vary freely. Since we measured the expectations of our participants regarding sensitive resource accesses with a 7-point Likert scale, we used an ordinal regression based on a cumulative link model with a *logit* linkage function and *flexible* thresholds between the categories. This regression method provides a regression function as well as ordered thresholds that describe the six category boundaries between the values of our 7-point Likert scale. In an iterative process, we specified different regression models and compared them using the Akaike information criterion (AIC), as well as direct comparisons of the goodness of fit using Chi^2-tests to find the model with the best trade-off between complexity and fit to the empirical data [27, 42].

Results of the Evaluation. We started with a base model without any independent variables and without the random effects and added successively the following effects/predictors in the next steps:

(1) the random effects
(2) the users' individual characteristics

Table 4: Final regression model predicting users' expectation regarding sensitive resource accesses.

	Estimate (b)	SE	z value	p
Users' characteristics				
Computer background (Yes)	0.384	0.223	1.722	0.090
Age	-0.007	0.009	-0.881	0.379
Gender (Male)	-0.190	0.180	-1.058	0.290
Used devices (iPhone)	-0.547	0.179	-3.061	**0.002**
Apps' characteristics				
Contacts	1.061	0.455	2.331	**0.002**
SMS	1.565	0.466	3.362	**0.001**
Phone	1.618	0.503	3.388	**0.001**
Microphone	2.078	0.478	4.131	**<0.001**
Location	2.087	0.465	4.492	**<0.001**
Storage	2.621	0.456	5.751	**<0.001**
Camera	2.647	0.519	5.099	**<0.001**
Users' expectations based on app description				
Expected access (Yes)	3.637	0.173	21.046	**<0.001**
GUIBAT				
Detected as *unexpected SR*	-0.569	0.145	-3.930	**<0.001**

Thresholds for category boundaries					
1|2	2|3	3|4	4|5	5|6	6|7
1.980	2.756	3.354	4.004	4.787	5.775

Used devices (Baseline=Android); Sensitive resources (Baseline=Calendar); GUIBAT (Baseline=*expected SR*); SE = Std. Error=; significant p-values are printed bold.

(3) the apps' characteristics
(4) the users' expectations based on the description
(5) **the classification based on GUIBAT**
(6) the interaction of the previous variables

The results of this process showed that every more complex models up to step 5 could explain the empirical data significantly better than the preceding/simpler model (see Table 10 in Appendix A.2 for the goodness of fit of each relevant step in the model building process). The interaction effects (step 6) do not add any further value and therefore Model 5, which explains 63.5% of the empirical variance is the final model for our analysis. Thereby, the significant difference between the model without the predictor based on GUIBAT's output (Model 4) and with this predictor (Model 5) shows that our tool can explain the variance of user expectations regarding sensitive resource accesses beyond the effects on which previous research has focused on. Table 4 presents the final regression model's results, including the estimates and standard errors of the predictors, as well as the 6 thresholds of the Likert scales category boundaries. The proportional odds assumption as a prerequisite for a cumulative linking model was fulfilled for all predictors.

Our results showed that among the users' characteristics only the mobile operating system that people use predominantly had a significant effect on users expectations (b = -.55, p = .002). Particularly, iOS users were significantly less likely to expect that apps use sensitive resources than Android users. We further found differences in the prevalence of expectations regarding access to different types of sensitive resources. Thus, for instance, access to the CALENDAR was significantly less likely expected by users than access to all other sensitive resources, and was therefore picked as the baseline category in the regression model ($b_{\text{all others}}$ = 1.06–2.65, $p_{\text{all others}}$ ≤ .002). This results are in line with prior work that shows that users do not consider all permission to be equally sensitive [10, 49].

We also found that expectations regarding sensitive resource accesses based on app descriptions had an impact on the expected access to sensitive resources during app usage. If participants expected access to a certain sensitive resource based on the description, they were also more likely expecting access to this sensitive resource during their further interaction with the app (b = 3.64, p < .001). This is in line with prior work and particularly supports the basic assumptions of research that analyzes the descriptions and links this content to the users' expectations [22, 56, 58].

Most importantly, however, we found a significant effect of the predictor that represents the output of our tool. If sensitive resource access was classified by GUIBAT as unexpected, participants significantly less likely expected an app to behave in such a way, than if the sensitive resource access was classified as expected (b = -0.57, p < .001). This result supports the hypothesis that GUIBAT works as intended. Thereby, this conclusion is also supported by the results of the model comparison process, which showed that our tool can explain the users' expectations significantly beyond previously considered influencing factors (such as app description).

5.2 RQ2: How widespread is sensitive resource access that violates users' expectations in the wild?

To explore the status quo of *unexpected SR* in the wild (RQ2), we utilized GUIBAT to perform a large-scale analysis on Android apps collected from Google Play. The run-time of GUIBAT's static analysis depends on the complexity of the analyzed apps, hence it is not practically feasible to analyze all apps with-in a reasonable amount of time. Therefore, we randomly selected 100,000 apps[10] (see Section 3.1) and ran GUIBAT on each app for 5 minutes, and then terminated it if no results were outputted. As the result, GUIBAT successfully analyzed 89,371 (89,37% of 100,000) apps. Among 89,371 apps, GUIBAT detected that only 60,485 (67.68% of 89,371) apps trigger permission-protected API calls. This means that not all declared permissions are actually used (*i.e.,* over-privileged apps). Further, GUIBAT identified 656,110 UI elements belonging to 47,909 (79,21% of 60,485) apps that access sensitive resources.

GUIBAT detected 386,285 (58.88% of 656,110) UI elements from 36,115 (75.38% of 47,909) apps that access at least one sensitive resource which is not expected by end users. Figure 6 shows the distribution of *unexpected SR* by sensitive resources as well as the attribution to the *unexpected SR* (*i.e.,* by app or by third-party libraries). We see that STORAGE sensitive resource has the highest number of *unexpected SR* accounting for 44.50% (221,536) of the total *unexpected SR*, followed by LOCATION, and PHONE resources, accounting for 32.01%, and 15.50% respectively. The *unexpected SR* pertaining to these three sensitive resources contribute to 92.01% of the total *unexpected SR*. More importantly, third-party library is the contributing factor to the high number of *unexpected SR* with 43.03% of STORAGE, 62.71% of LOCATION, and 38.53% of PHONE.

To study to which extent third-party libraries contribute to the *unexpected SR* of apps and found 178,206 (46.13% of 386,285) UI elements from 17,674 apps that have *unexpected SR* from third-party libraries. Per apps, 38.20% of 36,115 apps have *unexpected SR* that are exclusively attributed by third-party libraries. We see that LOCATION, PHONE, and STORAGE sensitive resources have the highest number of *unexpected SR* caused by using third-party library APIs

[10]These apps declare at least one dangerous permission in the apps' manifest file.

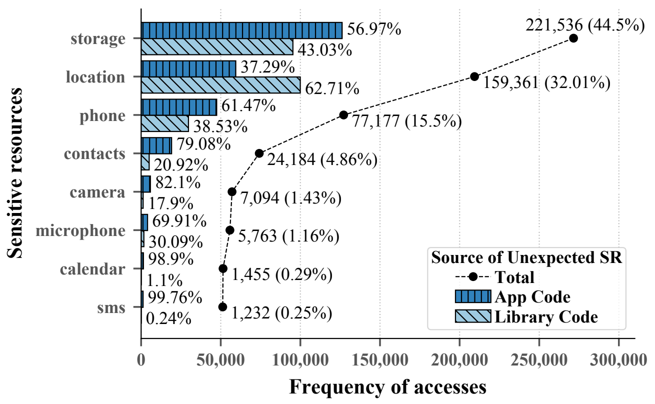


Figure 6: Distribution of *unexpected SR* among sensitive resources in 36,115 apps. Attribution of app code and library code sum up to 100% in each sensitive resources.

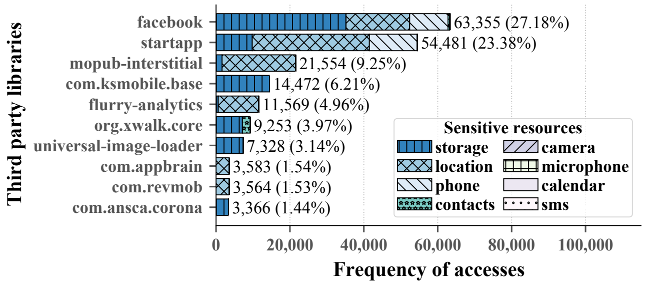


Figure 7: Top 10 libraries with *unexpected SR*.

(see Figure 7). A prior work of Book et al. showed that LOCATION and PHONE sensitive resources have constantly been the most frequently used of advertisement libraries [11]. We found that among the top 10 libraries that contribute to the total *unexpected SR* of the PHONE, LOCATION, STORAGE sensitive resources, the majority of them are advertising and analytic libraries (see Figure 7).

6 DISCUSSION

6.1 GUIBAT Compares to Prior Works

To see how effective GUIBAT is in identifying *unexpected SR*, we compare GUIBAT with the most recent work on detecting intention-behavior discrepancy namely DeepIntent, and with text-based only approaches.

Compare with DeepIntent. We applied GUIBAT on the dataset that DeepIntent took as the ground truth to train their icon-behavior classifier. The dataset contains 7,691 data points that were extracted from a set of benign apps, and each data point is a triple of *<icon,text,actual_SR>*. We found that 22.91% (1,020 out of 4,452) of the data points are *unexpected SR*[11]. All of these *unexpected SR* were then manually verified independently by two of the authors (disagreements were excluded) based on our user study results

[11]We filtered out data that was out of scope for this work (*e.g.*, icons have embedded foreign languages).

Table 5: Distribution of *expected SR* on icons/texts.

	Number of UI elements	
	w/o Contexts	w/ Contexts
Exclusively by Texts	177,022	225,702
Exclusively by Icons	18,740	58,181
Both Texts & Icons	11,594	146,561
Total	207,356	430,444

(whether the *actual SR* are justified by the associated icons/texts), and we found that 958 of them (93.92% of 1,020) indeed are *unexpected SR* (see Table 11 in Appendix A.2 for examples). However, it is unknown why apps make these *unexpected SR* due to not having the app's code or app's name of these data points. Determining the root cause for such *unexpected SR* is outside the scope of this paper. Our results show that relying on benign apps' icon-to-permission association as the "norm" to detect behavior discrepancy is not enough. There are *actual SR* (in this case, 22.91%) of benign apps that were previously considered normal by DeepIntent but are unexpected by users.

Besides, to show the effectiveness of our *Icon Classifier* in identifying icons and their associated sensitive resource accesses, we compared it with the *icon-only* classifier of DeepIntent[12]. We first ran the *icon-only* classifier on our labeled dataset that reflects user perception of sensitive resource accesses (see Section 4.2.1). This resulted in a low accuracy (AUC=0.50, see Table 9 in Appendix A.2 for the details). Further, for a fair comparison, we ran our *Icon Classifier* and the *icon-only* classifier on an "*unseen*" dataset. To create the labeled *unseen* dataset we followed the approach used in IconIntent [69]. Specifically, to obtain icons belonging to each sensitive resource, sensitive resource name (*e.g.*, "camera") together with the keyword "icon" were used to search for icons from Google Image. The top 100 results were then downloaded. This resulted in 800 images in total. Images that were not icons were then filtered out, resulted in 657 labeled icons. On this *unseen* dataset, our *Icon Classifier* significantly outperforms the DeepIntent's *icon-only* classifier in identifying sensitive resource related icons when taking user perception into account (details are included in Table 9, Appendix A.2). In particular, our *Icon Classifier* achieved an AUC value of 0.88, while the *icon-only* had an AUC value of 0.51. Our results indicate that approaches without considering how end users perceive app's data access is less effective in detecting sensitive resource accesses that violate user expectations.

Compare with text-based only approach. In Table 5, when we consider the contexts where UI elements are represented (*e.g.*, including surrounding UI elements), among UI elements that advertise their sensitive resources usage, 47.56% are advertised using icons. This shows that relying solely on textual UI elements would result in a severely high number of false positive (*i.e.*, sensitive resource accesses that are represented on UI but missed by the text-based detectors). Text-based only approaches [2, 28] did not cover such UI elements, hence their analysis would miss a significant portion of UI elements (*e.g.*, about 47.56%) affecting the final results. Further, GUIBAT also identified 28,145 UI elements (which access sensitive

[12]For a fair comparison, we trained these classifiers multiple times (K=10) based on the provided source code (https://github.com/deepintent-ccs/DeepIntent) to achieve the authors' reported performance.

resources) that associate with icons on UI screens that comprise solely of icons (*i.e.*, no textual UI elements were presented) from 6,730 apps (14.06% of apps in our dataset). This means, text-based only approaches would completely miss such screens, hence the analysis results of the corresponding apps (*e.g.,* 14.06%) would be affected.

6.2 Transparency of Sensitive Resources Access

GUIBAT revealed that 36,115 apps (75.38% of 47,909) have at least one *unexpected SR.* This is a non-negligible number given its implications to user's privacy. Further, this shows that even seemingly benign apps (*i.e.*, apps from Google Play) can have misleading UIs that lead to *unexpected SR.* Users already expressed concerns about the permissions they grant to apps, and the most common permissions that users worried about and were uncomfortable after they granted apps access to were STORAGE, PHONE, and LOCATION [10, 49]. This is also the same set of sensitive resources that were most frequently associated with *unexpected SR.* Besides, Micinski et al. showed that user actions such as pressing a button could be interpreted as authorization [45]. Thus, it is important to inform users whether such actions would lead to *unexpected SR.*

Further, in the third study (see Section 5.1), we have observed a significant correlation (z = 19.7, p < 0.001) between the users' expectation and comfort ratings. In other words, users' expectations of apps' *actual SR* are directly connected to their subjective feelings. Users are uncomfortable when an app's *actual SR* is not expected on the app's UI (inline with a prior study [39]). Tools like GUIBAT could support users in making decision to protect their privacy by learning about app's *unexpected SR.*

GUIBAT can also help developers to self-assess the informativeness of their apps' UIs. Future work can look into building new tools to support developers in this endeavor (*e.g.*, by suggesting changes). Besides, developers may also use privacy policies to inform users about their access to sensitive user data. However, privacy policies are often long and complicated that is difficult for users to understand [54]. Further, in practice, the majority (71%) of apps lack privacy policy even though they are obligated to have one [76]. Hence, they would have a limited impact if only a few users would actually read and understand the privacy policy.

Runtime permission. The goal of using runtime permission is to improve the permission decision-making and avoid undermining users' expectations [16]. It enables apps to embed their permission requests in contexts, so that the end-users can understand better. Our analysis found 21,843 apps with runtime permission (60.48% of 36,115 apps) that have at least one *unexpected SR.* We found a significant difference (Kruskal-Wallis, $\chi_2 = 197.99, df = 1, p < .001$) between the number of *unexpected SR* of apps with install-time permission (mean = 10.99) and of apps with runtime permission (mean = 10.05). This suggests that apps with runtime permissions have significantly smaller numbers of *unexpected SR.* However, we believe that the runtime permission is not the panacea for resolving the transparency issues of accessing sensitive data. Particularly, in this analysis, the effect size of runtime permissions is negligible ($epsilon - squared = 0.00413$). An app behavior may defy users' expectations, depending on whether the app provides users enough semantics to justify the access — not merely whether the app was authorized to receive data the first time it asked for it [55].

6.3 The Impact of Third-party Libraries

GUIBAT revealed that 38.20% of apps have *unexpected SR* that are exclusively attributed by third-party libraries. This suggests that developers should be extra careful about the *actual SR* by libraries. More specifically, developers should be informed when they use an API that accesses sensitive resources. Library developers should also provide information on what the library does, which sensitive resources will be accessed under which conditions. Future work can further analyze libraries to provide permission-protected API mappings. This could put app developers in a better position in informing their users about the apps' permission access. Central repositories such as Jcenter and Maven also play an important role in making permission accesses of libraries more transparent. For example, these central repositories can enforce third-party libraries to follow a permission-transparency policy, *e.g.*, having an explanation for usages of permission-protected API(s).

6.4 Limitations and Future Work

Similar to any other static analysis approaches, GUIBAT is over-approximation in identifying associated callbacks of UI elements. Besides, while `LibScout` is resilient against common obfuscation techniques (*e.g.*, identifiers renaming, API hiding), it would fail when apps leverage more advanced obfuscation techniques (*e.g.*, package flattening, class repackaging). Hence, this would affect our results in attributing the source of *unexpected SR* in apps. Future work could adopt more advanced de-obfuscation tools [8, 71] to pre-process these obfuscated apps. Note that Wemker et al. showed that only 24.92% apps on Google Play are obfuscated by developers [66]. Specifically related to UI, apps could also obfuscate their UI *e.g.*, by using `UIObfuscator` [75]. This would affect our results in statically extracting UI elements. Future work could look into developing de-obfuscators that address the challenges that `UIObfuscator` introduces, for example, by removing invisible layout from view hierarchies, resolving Java reflection [7], and dynamically instrumenting and analyzing apps to unveil apps' UI. Especially, with the techniques used in GUIBAT, we could build an icon classifier that learnt from the unobfuscated apps (majority of apps on Google Play [66]), and combine it with dynamic analysis (and instrumentation techniques) to analyze the UI of obfuscated apps.

Further, we did not consider UI elements on webviews or other dynamic app content, which would potentially affect the number of the identified UI elements. These limitations might affect the precision of GUIBAT in detecting the *unexpected SR.* Besides, we did not consider SENSORS permission as only 0.09% (319 of 600,000) of apps in our dataset request this permission. In our analysis, we did not consider inter-app interaction (*e.g.*, an app can launch another app). For such cases, users have to choose the provider apps (receiving intent) explicitly. Thus, further analysis of such providers are needed. This would affect the number of false positive in our analysis. We leave these challenges for future work.

To examine the coverage of the icons in our studies, we extended our *Icon Classifier* with the results of the first studies (see Section 3.2) to identify *all* commonly-known icons in our dataset (700,000 icons). With the confidence probability higher than 0.5 (random guessing), the *Extended Icon Classifier* successfully identified 72.35% of the icons across 76 groups of commonly-known icons. Applying the *Extended Icon Classifier* on the Material Design Icons [18], we successfully identified 83.22% of the icons (352 of 423). We focused on the commonly-known icons (72.35%), and ignored icons that users have not seen before since it is not possible to formulate user's mental models of such icons. This limits us from studying the user's perception of *not yet seen* icons. Thereby, our detection result can only serve as a lower-bound of *unexpected SR* in mobile apps. Further, our studies were conducted online which might suffer from opt-in bias and the inherent bias from MTurk's users.

In this paper, we referred to prior works [49, 65] and Android's documentation to create the mapping for sensitive related keywords (see Table 2). This mapping might be incomplete, and hence, could limit GUIBAT in identifying *expected SR* of textual UI elements. One alternative approach would be to build the mapping by mining the app dataset and then picking the most common keyword associated with *actual SR*. However, even with a complete mapping, the text-based detector is deficient for identifying *expected SR* of UI elements that comprise solely of icons (14.06% of apps in our dataset).

Finally, an interesting direction for future work is to identify the root causes of the *unexpected SR* in mobile apps, *e.g.*, by conducting an in-depth study with developers and end users to characterize the identified *unexpected SR*.

7 CONCLUSION

In this paper, we proposed GUIBAT - a new tool that learns from users' perception to detect *unexpected SR* in Android apps. Our evaluations showed that GUIBAT correctly reflects users' expectations, and revealed the deficient of prior work. Having applied on a set of 47,909 apps, GUIBAT identified that 75.38% of apps have at least one *unexpected SR*. Further, GUIBAT also revealed that 46.13% of the *unexpected SR* were attributed by third-party libraries. We showed the urgent need for more transparent UI designs to better inform users of data access, and called for new tools to support app developers in this endeavor.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback and our shepherd Mu Zhang at the University of Utah for his guidance in the shepherding process.

REFERENCES

[1] KWY Au, YF Zhou, Z Huang, and D Lie. 2012. PScout: Analyzing the Android Permission Specification. In *CCS*.
[2] V Avdiienko, K Kuznetsov, I Rommelfanger, A Rau, A Gorla, and A Zeller. 2017. Detecting Behavior Anomalies in Graphical User Interfaces. In *ICSE-C*.
[3] M Backes, S Bugiel, and E Derr. 2016. Reliable Third-Party Library Detection in Android and Its Security Applications. In *CCS*.
[4] M Backes, S Bugiel, E Derr, P McDaniel, D Octeau, and S Weisgerber. 2016. On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis. In *USENIX Security*.
[5] R Balebako, J Jung, W Lu, LF Cranor, and C Nguyen. 2013. "Little Brothers Watching You": Raising Awareness of Data Leaks on Smartphones. In *SOUPS*.
[6] L Bao, TD B Le, and D Lo. 2018. Mining sandboxes: Are we there yet?. In *SANER*.
[7] P Barros, R Just, S Millstein, P Vines, W Dietl, M d'Amorim, and MD Ernst. 2015. Static Analysis of Implicit Control Flow: Resolving Java Reflection and Android Intents (T). In *ASE*.
[8] B Bichsel, V Raychev, P Tsankov, and M Vechev. 2016. Statistical deobfuscation of android applications. In *CCS*.
[9] R Bonett, K Kafle, K Moran, A Nadkarni, and D Poshyvanyk. 2018. Discovering Flaws in Security-Focused Static Analysis Tools for Android using Systematic Mutation. In *USENIX Security*.
[10] B Bonné, ST Peddinti, I Bilogrevic, and N Taft. 2017. Exploring decision making with Android's runtime permission dialogs using in-context surveys. In *SOUPS*.
[11] T Book, A Pridgen, and DS Wallach. 2013. Longitudinal Analysis of Android Ad Library Permissions. In *MOST*.
[12] NV Chawla. 2005. *Data Mining for Imbalanced Datasets: An Overview.*
[13] HI Cheng and PE Patterson. 2007. Iconic hyperlinks on e-commerce websites. *Applied Ergonomics* (2007).
[14] F Chollet. 2016. Building powerful image classification models using very little data. *Retrieved December* 13 (2016).
[15] SR Choudhary, A Gorla, and A Orso. 2015. Automated Test Input Generation for Android: Are We There Yet?. In *ASE*.
[16] AP Felt, S Egelman, M Finifter, D Akhawe, DA Wagner, et al. 2012. How to Ask for Permission. *HotSec* (2012).
[17] AP Felt, E Ha, S Egelman, A Haney, E Chin, and D Wagner. 2012. Android Permissions: User Attention, Comprehension, and Behavior. In *SOUPS*.
[18] Flaticon. 2020/04/30. Material. https://www.flaticon.com/packs/material-design.
[19] GATOR. 2018/08. GATOR. http://web.cse.ohio-state.edu/presto/software/gator/.
[20] C Gatsou, A Politis, and D Zevgolis. 2011. From icons perception to mobile interaction. In *FedCSIS*.
[21] GDPR. 2020/11/02. The EU General Data Protection Regulation. https://gdpr.eu/.
[22] A Gorla, I Tavecchia, F Gross, and A Zeller. 2014. Checking App Behavior Against App Descriptions. In *ICSE*.
[23] JA Hanley and BJ McNeil. 1982. The meaning and use of the area under a receiver operating characteristic (ROC) curve. *Radiology* (1982).
[24] SH HasanPour, M Rouhani, M Fayyaz, and M Sabokrou. 2016. Lets keep it simple, Using simple architectures to outperform deeper and more complex architectures. *CoRR* (2016).
[25] H He and EA Garcia. 2009. Learning from Imbalanced Data. *TKDE* (2009).
[26] A Hotho, S Staab, and G Stumme. 2003. Ontologies improve text document clustering. In *ICDM*.
[27] JJ Hox, M Moerbeek, and R Van de Schoot. 2017. *Multilevel analysis: Techniques and applications*.
[28] J Huang, X Zhang, L Tan, P Wang, and B Liang. 2014. AsDroid: Detecting Stealthy Behaviors in Android Applications by User Interface and Program Behavior Contradiction. In *ICSE*.
[29] S Isherwood. 2009. Graphics and semantics: The relationship between what is seen and what is meant in icon design. In *EPCE*.
[30] International Standards Organization (ISO). 1984. International standard for safety colours and safety signs: ISO 3864.
[31] J Jung, S Han, and D Wetherall. 2012. Short Paper: Enhancing Mobile Application Permissions with Runtime Feedback and Constraints. In *SPSM*.
[32] PG Kelley, S Consolvo, LF Cranor, J Jung, N Sadeh, and D Wetherall. 2012. "A Conundrum of Permissions: Installing Applications on an Android Smartphone. In *FC*.
[33] R Kohavi. 1995. A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection. In *IJCAI*.
[34] A Krizhevsky, I Sutskever, and GE Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *NIPS*.
[35] L Li, TF Bissyand, M Papadakis, S Rasthofer, A Bartel, D Octeau, JK, and L Traon. 2017. Static analysis of android apps: A systematic literature review. *IST* (2017).
[36] L Li, TF Bissyandé, J Klein, and Y Le Traon. 2015. An Investigation into the Use of Common Libraries in Android Apps. In *Technique Report*.
[37] Y Li, Y Guo, and X Chen. 2016. PERUIM: Understanding Mobile Application Privacy with permission-UI Mapping. In *UbiComp*.
[38] Y Li, Z Yang, Y Guo, and X Chen. 2017. DroidBot: A Lightweight UI-guided Test Input Generator for Android. In *ICSE-C*.
[39] J Lin, S Amini, J I Hong, N Sadeh, J Lindqvist, and J Zhang. 2012. Expectation and purpose: understanding users' mental models of mobile app privacy through crowdsourcing. In *Ubicomp*.
[40] RTW Lo, B He, and I Ounis. 2005. Automatically building a stopword list for an information retrieval system. In *DIR*.
[41] K Mao, M Harman, and Y Jia. 2016. Sapienz: Multi-objective Automated Testing for Android Applications. In *ISSTA*.
[42] P McCullagh. 1980. Regression models for ordinal data. *Journal of the Royal Statistical Society: Series B (Methodological)* (1980).
[43] S McDougall and S Isherwood. 2009. What's in a name? The role of graphics, functions, and their interrelationships in icon identification. *Behavior research methods* (2009).
[44] AW Meade and SB Craig. 2012. Identifying careless responses in survey data. *Psychological methods* (2012).
[45] K Micinski, D Votipka, R Stevens, N Kofinas, ML Mazurek, and JS Foster. 2017. User Interactions and Permission Use on Android. In *CHI*.
[46] GA Miller. 1995. WordNet: a lexical database for English. *COMMUN ACM* (1995).
[47] N Momen, M Hatamian, and L Fritsch. 2019. Did App Privacy Improve After the GDPR? *IEEE Secur Priv* (2019).

[48] K Moran, M Tufano, C Bernal-Cárdenas, M Linares-Vásquez, G Bavota, C Vendome, M Di Penta, and D Poshyvanyk. 2018. MDroid+: A Mutation Testing Framework for Android. In *ICSE*.

[49] DC Nguyen, E Derr, M Backes, and S Bugiel. 2019. Short Text, Large Effect: Measuring the Impact of User Reviews on Android App Security & Privacy. In *SP*.

[50] J Nielsen and D Sano. 1995. SunWeb: User interface design for Sun Microsystem's internal web. *Computer Networks and ISDN Systems* (1995).

[51] D Norman. 2014. *Things that make us smart: Defending human attributes in the age of the machine.*

[52] D Octeau, D Luchaup, M Dering, S Jha, and P McDaniel. 2015. Composite constant propagation: Application to android inter-component communication analysis. In *ICSE*.

[53] D Octeau, P McDaniel, S Jha, A Bartel, E Bodden, J Klein, and Y Le Traon. 2013. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *USENIX Security*.

[54] A Oltramari, D Piraviperumal, F Schaub, S Wilson, S Cherivirala, TB Norton, NC Russell, P Story, J Reidenberg, and N Sadeh. 2017. PrivOnto: A semantic framework for the analysis of privacy policies. *Semantic Web* (2017).

[55] X Pan, Y Cao, X Du, B He, G Fang, R Shao, and Y Chen. 2018. FlowCog: Context-aware Semantics Extraction and Analysis of Information Flow Leaks in Android Apps. In *USENIX Security*.

[56] R Pandita, X Xiao, W Yang, W Enck, and T Xie. 2013. WHYPER: Towards Automating Risk Assessment of Mobile Applications. In *SEC*.

[57] RC. Prati, GEAPA Batista, and MC Monard. 2004. Class Imbalances versus Class Overlapping: An Analysis of a Learning System Behavior. In *MICAI*.

[58] Z Qu, V Rastogi, X Zhang, Y Chen, T Zhu, and Z Chen. 2014. Autocog: Measuring the description-to-permission fidelity in android applications. In *CCS*.

[59] F Roesner, T Kohno, A Moshchuk, B Parno, HJ Wang, and C Cowan. 2012. User-Driven Access Control: Rethinking Permission Granting in Modern Operating Systems. In *SP*.

[60] A Rountev and D Yan. 2014. Static Reference Analysis for GUI Objects in Android Software. In *CGO*.

[61] T Salimans, I Goodfellow, W Zaremba, V Cheung, A Radford, and X Chen. 2016. Improved techniques for training gans. In *NIPS*.

[62] H Scudder. 1965. Probability of error of some adaptive pattern-recognition machines. *IEEE Transactions on Information Theory* (1965).

[63] MK Ward and SB Pond III. 2015. Using virtual presence and survey instructions to minimize careless responding on Internet-based surveys. *Computers in Human Behavior* (2015).

[64] Joe H Ward Jr. 1963. Hierarchical grouping to optimize an objective function. *Journal of the American statistical association* (1963).

[65] T Watanabe, M Akiyama, T Sakai, and T Mori. 2015. Understanding the Inconsistencies between Text Descriptions and the Use of Privacy-sensitive Resources of Mobile Apps. In *SOUPS*.

[66] D Wermke, N Huaman, Y Acar, B Reaves, P Traynor, and S Fahl. 2018. A Large Scale Investigation of Obfuscation Use in Google Play. In *ACSAC*.

[67] S Wiedenbeck. 1999. The use of icons and labels in an end user application program: an empirical study of learning and retention. *Behaviour & Information Technology* (1999).

[68] S Xi, S Yang, X Xiao, Y Yao, Y Xiong, F Xu, H Wang, P Gao, Z Liu, F Xu, et al. 2019. DeepIntent: Deep Icon-Behavior Learning for Detecting Intention-Behavior Discrepancy in Mobile Apps. In *CCS*.

[69] X Xiao, X Wang, Z Cao, H Wang, and P Gao. 2019. IconIntent: automatic identification of sensitive UI widgets based on icon classification for Android apps. In *ICSE*.

[70] Q Xie, E Hovy, MT Luong, and QV Le. 2019. Self-training with Noisy Student improves ImageNet classification. *arXiv preprint arXiv:1911.04252* (2019).

[71] L Xue, H Zhou, X Luo, L Yu, D Wu, Y Zhou, and X Ma. 2020. PackerGrind: An Adaptive Unpacking System for Android Apps. *IEEE Trans. Softw. Eng.* (2020).

[72] S Yang, D Yan, H Wu, Y Wang, and A Rountev. 2015. Static Control-flow Analysis of User-driven Callbacks in Android Applications. In *ICSE*.

[73] L Yu, X Luo, C Qian, and S Wang. 2016. Revisiting the Description-to-Behavior Fidelity in Android Applications. In *SANER*.

[74] C Zauner. 2010. Implementation and benchmarking of perceptual image hash functions. (2010).

[75] H Zhou, T Chen, H Wang, L Yu, X Luo, T Wang, and W Zhang. 2020. UI Obfuscation and Its Effects on Automated UI Analysis for Android Apps. In *ASE*.

[76] S Zimmeck, Z Wang, L Zou, R Iyengar, B Liu, F Schaub, S Wilson, N Sadeh, S Bellovin, and J Reidenberg. 2017. Automated analysis of privacy requirements for mobile apps. In *NDSS*.

A APPENDIX

A.1 Methodology

A.1.1 Identifying Identical or Similar Icons. To identify whether two icons are identical or very similar, *pHash* technique was used [74], and a *hamming distance* value of less than 11 between 2 icons is considered as similar. To select this threshold, we first randomly selected 1,000 dissimilar icons (*i.e.*, 1,000 categories) by comparing their exact *pHash* value. For each category, we applied four transformations techniques (see Table 6) to generate more samples, resulted in 8,400 icons across 1,000 categories. Then, we clustered these icons into the same group with different hamming distance thresholds (ranging from 0 to 50). For example, with the threshold is 5, two icons are considered in the same group if their hamming distance is less than or equal 5. To select the best threshold for identifying very similar icons, we considered at the tradeoff between the quality of the clustering against the number of clusters. A measure that allows us to make this tradeoff is normalized mutual information or NMI (*i.e.*, a number between 0 and 1), and a higher value is better. After running this evaluation 10 trials, we chose the value of 11 for the hamming distance threshold as the NMI's score reached its peak at 0.88.

Table 6: The transformation techniques that are used.

Name	Range	Notes
Zoom	0-0.3	Float range for random zoom
Rotation	0-30	Degree range for random rotations
Width shift	0-0.2	Float range for random horizontal shift
Height shift	0-0.2	Float range for random vertical shift

A.1.2 Description-based Icons Clustering Module. To analyze natural language text from user responses, we applied several text preprocessing methods. These methods are broadly classified into three tasks: (1) text preprocessing, (2) feature extraction, (3) cluster analysis. Here are the details.

We first applied the following widely-used text preprocessing techniques [26, 40, 65]: correcting misspelling by the *autocorrect*[13] (*e.g.*, *"imagec"* to *"image"*); normalizing and lemmatizing all words, *e.g.*, removing punctuations, converting letters to lowercase and reducing the inflectional forms of a word (*e.g.*, *"sent"*, and *"sending"* to *"send"*); and removing generic stop words such as *"are"* and *"the"*; lastly we employed the term-based sampling approach [40] to create our domain-specific stop words list (taking top 50 words that have the least weighted) so that we could additionally remove words that are not generic stop words but are commonly used in user responses from our survey (*e.g.*, *"icon"*, *"symbol"*, *"button"*).

Then, using the preprocessed texts of user responses, we adopted the bag-of-words model to convert each user response into a text feature vector as follows. Let $T = \{t_1, t_2, ..., t_n\}$ be a set of all unique terms in the corpus of user responses. A text feature of vector of the i^{th} user response is denoted as $t_i = \{t_1, t_2, ..., t_k\}$. For example, the raw text is *"I would expect it to take a photo"*, after applying the preprocessing, the generated preliminary text vector is {"*photo*"}. Further, to resolve the problem of synonyms, we employed the hypernym strategy (*hypdepth = 5*), which is suggested by Hotho et al. [26] to enrich the text vectors with concepts from the Wordnet [46]. Specifically, we added to each term of the feature vectors all subconcepts of the 5 levels below it in the Wordnet corpus, after that, we performed word stemming on all terms

[13]https://github.com/phatpiglet/autocorrect/

Table 7: The evaluating results of GUIBAT and Gator 3.5 on a set of 34 apps. V=Views. C=Callbacks. (+)=New. (-)=Removed.

App	GUIBAT/GATOR 3.5		GUIBAT's Refinement upon GATOR		
	V	C	(+) V	(+) C	(-) C
AardDictionary 1.4.1	218/218	105/105	0	0	0
FTP Server 1.0.4	14/0	25/0	14	25	0
Battery Circle 1.4.1	30/30	7/19	0	0	12
LolcatBuilder 0.2.1	9/1	5/0	8	5	0
SpriteMethodTest 2.0a	4/4	0/0	0	0	0
Alarm Clock 1.4.1	46/40	20/32	6	2	14
Manpages 0.2.1	13/13	7/7	0	0	0
Auto Answer 1.4.1	10/0	8/0	10	8	0
RandomMusicPlayer 2.0a	9/9	6/6	0	0	0
AnyCut 1.4.1	15/15	5/5	0	0	0
HNDroid 1	65/62	105/106	3	0	1
DivideAndConquer 1	1/1	0/0	0	0	0
Photostream 2.0a	7/2	0/0	5	0	0
Multi SMS 0.2.1	47/47	30/30	0	0	0
World Clock 2.0a	24/24	12/18	0	0	6
Ringdroid 2.0a	34/34	56/68	0	0	12
Yahtzee 2.0a	30/30	12/12	0	0	0
aagtl 2.8.11	60/60	44/44	0	0	0
WhoHasMyStuff 2.0a	40/40	58/58	0	0	0
Mirrored 0.2.1	30/21	15/15	9	0	0
WeightChart 2.0a	65/32	115/21	33	100	6
ADSdroid 1.4.1	6/6	2/2	0	0	0
myLock 0.2.1	33/16	3/9	17	0	6
LockPatternGenerator 0.2.1	19/10	19/13	9	6	0
MunchLife 0.2.1	20/14	20/19	6	1	0
CountdownTimer 1.0.31	45/35	8/10	10	0	2
NetCounter 0.1.1	28/20	10/6	8	4	0
TippyTipper 2.0a	95/83	28/188	12	0	160
BaterryDog 1.4.1	15/15	32/32	0	0	0
Dialer2 1	55/45	57/57	10	0	0
DalvikExplorer 2.11	134/134	126/126	0	0	0
Blokish 2.2	38/32	13/13	6	0	0
ZooBorns 2.0a	12/12	8/8	0	0	0
Wordpress_394 2.0a	46/40	21/30	6	3	12
Total	1,317/1,145	982/1,059	172 (15.02% of 1,145)	154	231 (21.81% of 1,059)

(*e.g.*, "location" and "locating" to "locat"). For instance, with the preliminary text vector above, the final text feature vector will be {"*photo*", "*photograph*", "*exposur*", "*pictur*", "*pic*"}. To this end, we converted each term, in the text vector to numeric one by using the term-frequency inverse document-frequency (TF-IDF). The TF-IDF value for each element was calculated as:

$$tfidf(t,d) = tf(t,d) * idf(t) = \frac{1+N}{1+df(d,t)}$$

where t refers to the selected term, d refers to the text vector, tf is the absolute frequency of a term, *i.e.*, $tf(t,d)$ is the number of times a term t occurs in a given d, idf is the term's inverse document frequency, N is the number of text lists in the corpus, and $df(d,t)$ returns the number of text lists that contain the target term t. Lastly, the agglomerative hierarchical clustering was used to identify which icons are commonly known by users and the function these icons represent. Specifically, Ward's method was used [64], and the similarity score or distance between two vectors is calculated by cosine similarity:

$$similarity(\vec{v_i}, \vec{v_j}) = cos(\vec{v_i}, \vec{v_j}) = \frac{\vec{v_i}.\vec{v_j}}{||\vec{v_i}||.||\vec{v_j}||}$$

To this end, we first clustered similar icons using textual descriptions of their visual representation. We then further clustered these clusters of icons using descriptions their functions. In the first step, we identified 77 groups of icons from 972 commonly-known icons. However, we filtered out one group that contains icons whose (participant) responses were not comprehensible (*i.e.,* this group was also considered an outlier by hierarchical clustering). Finally, we identified 44 groups of functions from 76 groups of icons.

A.1.3 Improving GATOR. We leveraged GATOR and further overcame the following limitations:

(1) GATOR does not consider some important Android UI components which leads to missing app's UI elements along with their associated callbacks in app's GUI model (*Navigation Drawer*, *Preference*, and *Fragment*). Richard et al. showed that 91% of popular apps contain fragment code in their apps which indicates that *Fragment* is widely used [9]. GUIBAT additionally considers these UI components; (2) GATOR's static reference analysis that maps app's UI elements to their callbacks introduce over-approximation due to the over-approximate of variable mappings (*e.g.*, incorrectly map a UI element to multiple callbacks). To be more precise, GUIBAT, on the other hand, further strictly on object id (of UI elements) to correctly map callback methods to UI elements.

Evaluation. We selected an Android benchmark suite from the work of Choudhary et al. [15] which is widely used by other static and dynamic analysis [41, 48]. It contains 68 open source Android apps. We filtered out those apps that are not supported by GUIBAT such as apps in the games category, which resulted in 43 apps. Then we ran GUIBAT and GATOR on 43 apps to generate app's GUI models. Among these 43 apps, those apps that have less than 200 callbacks were chosen for a comprehensive manual examination, resulted in 34 apps. From 34 apps, we compare the number of extracted UI elements, and the number of extracted callback methods between GATOR and GUIBAT. If there are any differences between the results of GUIBAT and GATOR, we manually verify the differences (see Table 7). In general, GUIBAT is better than GATOR in GUI analysis as the following: (1) in 14 (41.17%) apps, both GUIBAT and GATOR have identified the same number of UI elements and their callbacks; (2) GUIBAT correctly has removed 231 (21.81% of 1,059) of the callback methods that are made-up by the over approximation of GATOR; (3) GUIBAT successfully identified the Preference UI component and their associated callbacks in 20 apps which GATOR missed. In particular, there are 172 (15.02% of 1,145) components that GUIBAT identified while GATOR could not.

A.1.4 Identifying threshold to limit the depth of the CFG analysis. Figure 8 provides details on the relative cumulative distribution of the different values of the depth of our CFG analysis.

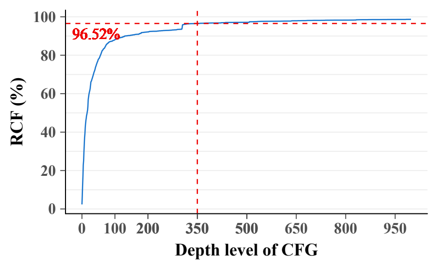


Figure 8: Relative cumulative frequency (RCF) of accessed sensitive resources depth level of CFG in 10,000 apps.

Table 8: Demographics of participants from our studies.

	Study 1.1	Study 1.2	Study 2	Study 3
Number of participants	120	294	45	445
Gender				
Female	42 (35.0%)	122 (41.5%)	14 (31.11%)	258 (57.98%)
Male	78 (65.0%)	171 (58.16%)	31 (68.89%)	184 (41.35%)
Other	0	1 (0.34%)	0	2 (0.45%)
No answer	0	0	0	1 (0.22%)
Age group				
18–30	52 (43.33%)	103 (35.03%)	17 (37.78%)	155 (34.83%)
31–40	43 (35.83%)	117 (39.8%)	23 (51.11%)	173 (38.88%)
41–50	14 (11.67%)	47 (15.99%)	3 (6.67%)	66 (14.83%)
51–60	6 (5.0%)	17 (5.78%)	1 (2.22%)	39 (8.76%)
61–70	5 (4.17%)	8 (2.72%)	1 (2.22%)	11 (2.47%)
>=71	0	2 (0.68%)	0	1 (0.22%)
Computer science background				
Yes	25 (20.83%)	69 (23.47%)	12 (26.67%)	76 (17.08%)
No	95 (79.17%)	225 (76.53%)	33 (73.33%)	369 (82.92%)
Primary phone				
Android	44 (36.67%)	199 (67.69%)	31 (68.89%)	275 (61.80%)
iPhone	75 (62.5%)	93 (31.63%)	14 (31.11%)	170 (38.20%)
Other	1 (0.83%)	2 (0.68%)	0	0
Education level				
Less than high school	0	1 (0.34%)	0	1 (0.22%)
High school graduate	19 (15.83%)	31 (10.54%)	5 (11.11%)	38 (8.54%)
Some college, no degree	32 (26.67%)	65 (22.11%)	13 (28.89%)	87 (19.55%)
Associate's degree	9 (7.5%)	38 (12.93%)	2 (4.44%)	56 (12.58%)
Bachelor degree	51 (42.5%)	128 (43.54%)	17 (37.78%)	189 (42.47%)
Master degree	6 (5.0%)	29 (9.86%)	7 (15.56%)	59 (13.26%)
Ph.D	1 (0.83%)	1 (0.34%)	0	6 (1.35%)
Graduate/prof. degree	2 (1.67%)	1 (0.34%)	1 (2.22%)	8 (1.80%)
Others	0	0	0	1 (0.22%)
Ethnicity				
White/Caucasian	91 (75.83%)	218 (74.15%)	30 (66.67%)	333 (74.83%)
Black/African American	9 (7.5%)	33 (11.22%)	6 (13.33%)	42 (9.44%)
Asian	11 (9.17%)	22 (7.48%)	5 (11.11%)	41 (9.21%)
Hispanic/Latino	4 (3.33%)	12 (4.08%)	1 (2.22%)	17 (3.82%)
Native American/Alaska	1 (0.83%)	2 (0.68%)	1 (2.22%)	3 (0.67%)
Native Hawaiian/Pacific Islander	0	0	0	1 (0.22%)
Middle Eastern	1 (0.83%)	0	0	1 (0.22%)
Other	3 (2.5%)	7 (2.38%)	2 (4.44%)	7 (1.57%)

Table 9: The AUC score of GUIBAT's Icon Classifier and `DeepIntent`'s models.

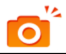
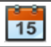
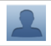
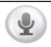
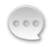
Sensitive Resources	"unseen" dataset from Google Image		GUIBAT's Icon Training Dataset
	GUIBAT	DeepIntent icon-only	DeepIntent icon-only
Camera	0.93	0.51	0.49
Calendar	0.98	-	-
Contact	0.85	0.61	0.52
Location	0.94	0.55	0.55
Microphone	0.92	0.53	0.55
Phone	0.97	0.40	0.42
SMS	0.70	0.47	0.50
Storage	0.75	0.50	0.55
Average	0.88	0.51	0.50

Table 10: Model comparison based on Goodness of fit.

	AIC	logLik	df	*p*
Base model	4,853.8	-2,420.9		
+ Random effects	4,798.8	-2,391.4	2	**<0,001**
+ Users' characteristics	4,785.2	-2,380.6	4	**<0,001**
+ Apps' characteristics	4,624.3	-2,293.1	7	**<0,001**
+ Users' expectations based on app description	3,957.7	-1,958.9	1	**<0,001**
+ GUIBAT	3,944.2	-1,951.1	1	**<0,001**
+ Interaction	3,945.4	-1,950.7	1	0.369

All regression models listed above predict the users' expectations regarding apps' sensitive resource access; AIC = Akaike Information Criterion; df = degree of freedom; logLik = log likelihood; *p* quantifies statistical significance; significant *p*-values are printed bold.

Table 11: Example of *unexpected SR* that are detected by `GUIBAT` from `DeepIntent`'s benign training dataset.

Icons	Texts	*Actual SR*	*Unexpected SR* by GUIBAT
	tencent enter purchas btn take pictur	Camera, Location, SMS	Location, SMS
	time search	Location	Location
	default user icon	Location	Location
	holder mic	Contacts, SMS	Contacts, SMS
	post float letter refresh system friend topic new subject normal	Location	Location
	to to leg menu save	Contacts, Location, SMS, Microphone	Contacts, Location, SMS, Microphone

A.2 Results

A.2.1 User Study 2: Instruction Page. In mobile apps, sensitive resources are the resources that involve your private information, or could potentially affect your stored data or the operation of other apps. For example, your contacts is a sensitive resource. In this study, we focus on the following sensitive resources: **Contacts**: *involve the contacts information such as modify your contacts, find accounts on the device, read your contacts, etc*; **Phone**: *involve the telephone features such as read call log, read phone status, access your phone number, directly call phone numbers, write call log, reroute outgoing calls, etc*; **Calendar**: *involve calendar information such as read calendar events, add or modify calendar events, etc*; **Camera**: *access the camera such as taking pictures and videos, turning the flashlight on, etc*; **Location**: *access device location such as precise location (GPS and network-based), approximate location (network-based), etc*; **Storage**: *involve the storage such as read the contents (e.g., photos, media, or files) of your device, modify or delete the contents of your SD card, etc*; **Microphone**: *access the microphone such as record audio, etc*; **SMS**: *involve SMS information such as read your text messages (SMS or MMS), receive text messages, send and view SMS messages, read cell broadcast messages, etc.*
Q. A mobile app that accesses the camera to take photos and then saves them to SD card in your phone, what are sensitive related functions that the app involves?

A.2.2 User Study 3. Main Questions:
Q1. Which of the following sensitive resources would you expect this app to access? (*a list of 8 sensitive resources and their descriptions*)
Q2. Please explain briefly why in your opinion the app requires the sensitive resources you have selected (Free text).
Q3. To serve the app's functionality on the current screen, on a scale from 1 to 7 how much do you expect this app to access the following data? (*a list of sensitive resources that are detected by GUIBAT*)
Q4. Suppose you are pressing the highlighted function (red rectangle), how much do you feel comfort letting this app to access the following data? (*a list of sensitive resources that are detected by GUIBAT*)