IconIntent: Automatic Identification of Sensitive UI Widgets based on Icon Classification for Android Apps

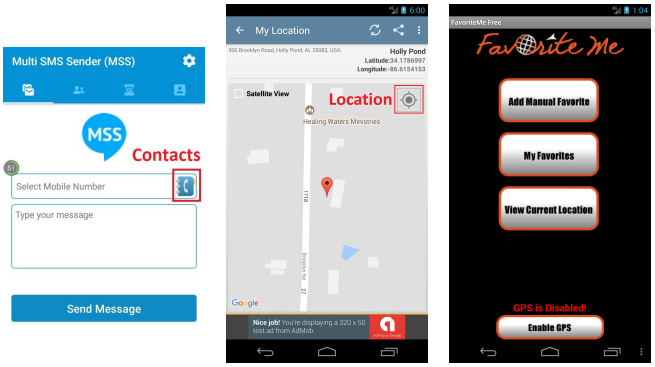
Xusheng Xiao[1] Xiaoyin Wang[2] Zhihao Cao[1] Hanlin Wang[1] Peng Gao[3]
[1]Case Western Reserve University, [2]The University of Texas at San Antonio, [3]Princeton University
[1]xusheng.xiao@case.edu, [2]xiaoyin.wang@utsa.edu, [3]pgao@princeton.edu

***Abstract*—Many mobile applications (i.e., apps) include UI widgets to use or collect users' sensitive data. Thus, to identify suspicious sensitive data usage such as UI-permission mismatch, it is crucial to understand the *intentions of UI widgets*. However, many UI widgets leverage icons of specific shapes (*object icons*) and icons embedded with text (*text icons*) to express their intentions, posing challenges for existing detection techniques that analyze only textual data to identify sensitive UI widgets. In this work, we propose a novel app analysis framework, IconIntent, that synergistically combines program analysis and icon classification to identify sensitive UI widgets in Android apps. IconIntent automatically associates UI widgets and icons via static analysis on app's UI layout files and code, and then adapts computer vision techniques to classify the associated icons into eight categories of sensitive data. Our evaluations of IconIntent on 150 apps from Google Play show that IconIntent can detect 248 sensitive UI widgets in 97 apps, achieving a precision of 82.4%. When combined with Supor, the state-of-the-art sensitive UI widget identification technique based on text analysis, Supor +IconIntent can detect 487 sensitive UI widgets (101.2% improvement over Supor only), and reduces suspicious permissions to be inspected by 50.7% (129.4% improvement over Supor only).**

I. Introduction

Mobile apps are playing an increasingly important part in our daily life [1], [2]. Despite the capabilities to meet users' needs, the increasingly access to users' sensitive data, such as location and finance information [3]–[5], raises privacy concerns. Prior works on smartphone privacy protection focus on analyzing mobile apps' code to detect information leaks of the sensitive data managed by the framework APIs, such as device identifiers (e.g., IMEI), location, and contact [6]–[8]. But this line of works are limited because they cannot address *sensitive user inputs*, where apps express *their intentions to use or collect users' sensitive data*. Many apps today include UI widgets such as buttons and text boxes, which expect users' consensus to use their sensitive data (e.g., pressing a button), or users' input of sensitive data (e.g., filling financial information in a text box).

It is crucial to understand the *intentions of UI widgets* by analyzing apps' UIs, for the app stores to inspect suspicious permissions (i.e., UI-permission mismatches [9]), for lawyers or managers to write more precise privacy policies [10], and for developers to better inform users about sensitive data usages. For example, given an app that requests a permission (e.g., microphone), an inspection of the app's UIs can determine that the permission is suspicious if this permission cannot be justified by the text and / or icons on any UI widget. Recent works have made progress in detecting disclosure of sensitive user inputs [9], [11], [12] by analyzing textual data in the UIs. However, UI widgets' intentions can also be expressed via images, especially icons of specific shapes (*object icons*). For example, the icons in Figure 1 indicate that the app will access users' contacts (Figure 1a) and GPS data (Figure 1b).



(a) (b) (c)

Figure 1: UIs containing icons that indicate the uses of sensitive data in mobile apps

Understanding the intentions of icons is a challenging problem. First, there are numerous types of icons in mobile apps. Icons representing the same intention can have different styles and can be shown in different scales and angles. Due to small screens of smartphones, icons are often not co-located with texts that explain their intents. As exemplified by Figure 1b, Google Map uses the icon shown in red square to center the map to the user's current location, without any text around the button. Second, some icons are embedded with text, referred to as *text icons*. For example, the third button from the top shown in Figure 1c indicates that the app will access users' GPS data. The diversified colors and opacities in fonts and backgrounds (e.g., ghost button [13]) make it difficult to directly apply Optical Character Recognition (OCR) [14], which works best for icons having black texts and white backgrounds.

To address this problem, we propose a novel framework, IconIntent, that *synergistically combines program analy-*

sis and icon classification to associate icons with UI widgets and classify the intentions of icons (both object icons and text icons) into eight pre-defined sensitive user input categories (including *Camera*, *Contacts*, *Email*, *Location*, *Phone*, *Photo*, *SMS*, and *Microphone*). The classified icons can be directly used to detect the mismatch of UI intentions and permissions. We target Android since they are the most popular mobile platform with the most users, but the general research is applicable to other mobile platforms such as iOS. Our proposed framework is based on three *key insights*.

First, while UIs contain unstructured information, the association between icons and UI widgets can be inferred from the structured information in UI layout files and app's code. This inspires us to develop static analysis techniques on UI layout files and app's code to infer such associations. Second, mobile apps are expected to have an intuitive UI where most usage scenarios of an app should be evident to average users, so icons indicating the same type of sensitive user input should have *similar looks*. This inspires us to develop object icon classification techniques to detect similar icons based on the sensitive icons collected from interactive widgets. Third, in order for users to easily recognize the objects or text in icons, the colors / opacity between the foreground and the backgrounds must be contrasted. This inspires us to develop icon mutation techniques to amplify and normalize this contrast, making icons easier to be recognized by the icon classification techniques.

ICONINTENT consists of three modules: icon-widget association module, icon mutation module, and icon classification module. The icon-widget association module provides a UI layout analysis technique to identify the associations between icons and UI widgets defined in the UI layout files. This module further provides a dataflow analysis technique that analyzes the program code to identify such associations. The icon mutation analysis module extracts icons from an app, and produces mutated icons for each of the extracted icon. The icon classification module adapts SIFT [15], a state-of-the-art image feature engineering technique, with our novel *key-location increasing* and *relative one-to-one matching* techniques to enhance its effectiveness in classifying icons. Additionally, this module adapts OCR techniques to extract text from the icons, and then classifies the icons using the edit-distance based similarity between the extracted text and the keywords in each category.

We evaluate the effectiveness of ICONINTENT using a dataset of 150 Android apps that collect sensitive data. We manually labeled 5,791 icons from the apps as ground truth. The results show that ICONINTENT detects 248 sensitive UI widgets (achieving 82.4% precision) from 97 apps, indicating that both sensitive icons and sensitive UI widgets are common. We also evaluate the effectiveness of ICONINTENT in complementing SUPOR [9], the state-of-the-art sensitive UI widget detection technique based on text analysis. The results show that SUPOR +ICONINTENT identifies 487 sensitive UI widgets, which achieves 101.2% improvement over 242 sensitive UI widgets identified by SUPOR. Also, we evaluate the effectiveness in reducing the inspection effort of suspicious permissions: if an identified intention of a UI widget matches a requested permission, then the permission is considered not suspicious. The results show that SUPOR +ICONINTENT reduces suspicious permissions to be inspected by 50.7%, compared with 22.1% identified by SUPOR, achieving 129.4% improvement. We further evaluate the effectiveness of icon classification techniques on the 5,791 icons. The results show that ICONINTENT effectively identifies object icons with the average F-score of 87.7%, compared with 48.6% of off-the-shelf SIFT. ICONINTENT identifies text icons with the average F-score of 89.8%, compared with 36.6% of off-the-shelf OCR.

This paper makes the following major contributions:

- We are the first to investigate the intents of icons in mobile apps' UIs, and study their uses in UI widgets.
- We propose a novel framework, ICONINTENT, that synergistically combines *program analysis* and *icon classification* to associate icons with the corresponding UI widgets and classify the intents of the icons into eight pre-defined sensitive categories.
- We conduct evaluations on 150 market apps. The results show that ICONINTENT effectively detects sensitive UI widgets (82.4% in precision) and reduces 50.7% of the suspicious permissions detected by SUPOR.

II. BACKGROUND AND MOTIVATING EXAMPLES

A. Android UI Rendering

An Android app usually consists of multiple activities, where each activity provides the window to draw the UI [16]. A UI is defined by a layout, which consists of UI widgets (e.g., buttons and image views) and layout models (e.g., linear layout) that describe how to arrange UI widgets. The UI framework provides a declarative language based on XML for developers to define UI layouts.

Example UI with a Sensitive Icon. Figure 2 shows a simplified UI layout file from *Animated Weather* and its rendered UI. This UI layout file contains three UI widgets: an image view widget (`ImageView`), a text box that accepts user inputs (`EditText`), and a button (`Button`). They are aligned horizontally based on the `LinearLayout` at Line 1. Figure 3 shows the code snippet of the corresponding activity for the layout file. Line 3 indicates the activity class `SearchForm` uses the layout file identified by the resource id `R.layout.search`. Line 4 first finds the `ImageView` widget using the API `findViewById` with the resource id `R.id.img`, which refers to the `ImageView` widget with the attribute `android:id="@+id/img"`. Line 4 then binds the event handler `onClick` to the click event of the widget via `setOnClickListener`. The handler `onClick` simply calls `startAsincSearc` (Line 5), which in turn calls `ManagerOfLocation.findPosition` (Line 9) that retrieves users' current location.

```
<LinearLayout android:orientation="horizontal">
  <ImageView android:id="@+id/img" android:src="@drawable/
     loc" .../>
  <EditText android:id="@+id/TxtCity" ... />
  <Button android:text="@string/search" .../>
</LinearLayout>
```

(a) UI layout file (search.xml)

(b) Rendered UI

Figure 2: Simplified layout file for a search UI

```
public class SearchForm extends Activity {
  public void onCreate(Bundle savedInstanceState) {
    setContentView(R.layout.search); // bound to layout
     file search.xml in Fig. 2
    ((ImageView) findViewById(R.id.img)).
     setOnClickListener(new OnClickListener() {
      public void onClick(View v) {startAsincSearch();} })
     ;
      ... } // bound to OnClick handler
  private void startAsincSearch() {
    ...
    ManagerOfLocation.findPosition(); // use GPS data
  . .. } }
```

Figure 3: Simplified UI Handler for Animated Weather

In the rendered UI (Figure 2b), the *ImageView* widget shows the icon `loc.png` specified by the resource id `drawable/loc`, which indicates to use users' current locations. Note that the UI does not have descriptive texts to explain the intention of the icon (i.e., retrieving users' current location). Such UI design indicates that for widely used icons, the UI assumes the users' knowledge in the semantics of the icon. This motivates us to collect a set of commonly used sensitive icons, and propose icon classification techniques that detect sensitive icons based on the collected icons.

Example UI with a Sensitive Text Icon. Figure 1c shows a UI from *Favorite.Me*. This UI has four buttons that use stylish text icons. The third icon from the top is embedded with the text "View Current Location", indicating the use of a user's GPS data. When a user clicks on the icon, the app retrieves users' current location. Existing works [9], [11], [12] that analyze texts in the UIs face challenges in identifying this sensitive UI widget, since no sensitive texts can be extracted from the UI. This motivates us to adapt OCR techniques to extract texts from text icons, and perform text classification to identify sensitive UI widgets.

B. App Icon Varieties

To make apps' UI unique and stylish in the small screen, app icons have different combinations of colors and transparencies in texts, backgrounds, and object shapes. As such, icons in Android apps are usually *small*, *diversified*, and *partially* or *totally transparent*. Figure 4 shows seven sensitive icons that pose different challenges for the icon classification technique and the OCR technique: (1) the SMS icon in Figure 4a and location icon in Figure 4b are too small; (2) the SMS icon in Figure 4c and the contact icon in Figure 4d have low contrast between the colors of the texts/objects and the background; (3) the Email icon in Figure 4e shows the text in bright color and the background in dark color, while OCR performs better with deep color texts in bright color backgrounds; (4) the Photo icon in Figure 4f is a ghost button, which uses transparencies to hide the background color. (5) the Camera icon in Figure 4g is an icon with low color contrast and uses transparency and shadow to show contrast.

Our preliminary study on 300 text icons extracted from apps in Google Play shows that directly applying existing OCR techniques can infer semantic information from less than 10% of the studied icons [17]. This further motivates us to perform image mutations on the icons such as converting the transparency differences to color differences, and apply the icon classification technique on the mutated icons.

III. Approach

A. Overview

Figure 5 shows the overview of IconIntent. IconIntent consists of three modules: icon-widget association, icon mutation, and icon classification. IconIntent accepts an app APK file as input and outputs the identified sensitive UI widgets with the associated icons, where each icon is annotated with the corresponding categories of sensitive data. The icon-widget association module performs static analysis on the UI layout files and the code to identify the associations between UI widgets and the icons. The icon mutation module extracts the icons from the resources, and performs image mutations on the extracted icons to generate a set of mutated icons. The icon classification module accepts the mutated icons as input, and classifies icons into eight categories of sensitive data.

B. Icon-Widget Association

IconIntent performs static analysis on both the UI layout files and the code to identify the associations between icons and UI widgets. We next formally define Android's UI layouts and our static analysis.

UI layouts and UI widgets. We first formally define UI layouts and their IDs.

Definition 1 (UI Layout): A UI layout is a tree $L(W, E)$, where each node $w \in W$ denotes a UI element and each edge $e(a, b) \in E$ denotes a parent-child relationship from a to b. L is uniquely identified by the layout ID $L.id$.

Figure 2a shows a UI layout loaded from `search.xml`, and its layout ID can be referenced in the code via `R.layout.search`. In this layout, there are four UI elements: a `LinearLayout`, an `ImageView`, an `EditText`, and a `Button`. The `LinearLayout` is the parent of the other three UI elements. Based on these definitions, we next define UI widgets.

Definition 2 (UI Widget): In a UI layout $L(W, E)$, a UI widget $w \in W$ is a type of UI element that can interact with the user (e.g., a button). w is uniquely identified by a pair $\langle L.id, w.id \rangle$, where $w.id$ represents the element ID of w.

(a) (b) (c) (d) (e) (f) (g)

Figure 4: Icon varieties in mobile apps

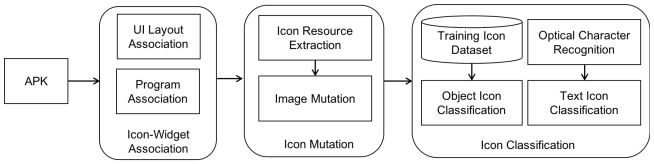


Figure 5: Overview of ICONINTENT

In Figure 2a, all the UI elements except the `LinearLayout` are UI widgets. In particular, the ID of the `ImageView` widget is $\langle R.layout.search, R.id.img \rangle$. Based on the definitions of UI layouts, we next formally define the binding from the UI widgets in the layout to the variables in the code.

Variable Binding. The UI layout files are loaded into activities at runtime via the layout-loading API calls, mainly `setContentView` and `Inflate`. The layout ID is used as the parameter to determine which layout file to load into an activity. We next define the variable-layout binding.

Definition 3 (Variable-Layout Binding): A variable v_L is said to be bound to a UI layout $L(W, E)$, represented as $v_L \rightsquigarrow L$ if (1) a layout binding API is invoked with v_L as the receiver object and $L.id$ as the parameter, or (2) v_L is an alias to another variable v'_L that is bound to L.

Once a layout is bound to an activity, the UI widgets in the layout can be bound to variables via invoking the widget-binding APIs, mainly `findViewByID`, with the UI widget ID.

Definition 4 (Variable-Widget Binding): Given that $v_L \rightsquigarrow L(W, E)$, a variable v is said to be bound to a UI Widget $w \in W$ if (1) a widget binding API is invoked with l as the receiver object , v as the return value, and $w.id$ as the parameter, or (2) v is an alias to another variable v' that is bound to w.

In Figure 3, Line 3 loads the layout file with the ID `R.layout.search` to the activity. Line 4 binds the `ImageView` widget by invoking the API `findViewById()` with `R.id.img` as the parameter to a temporary variable (omitted in Figure 3).

Icon Association. Following the definitions of UI layouts and widgets, we define icons as follows.

Definition 5 (Icon): An icon c is a type of resource. It is uniquely identified by $c.id$, which is the resource ID.

Icons can be associated with UI widgets via specifications in layout files directly. In the layout files, icons are often referred to using resource names in the `android:src` attributes of UI widgets. These resource names (e.g., `@drawable/loc` in Figure 2a) may be directly mapped to file names in the resource folder (`res/drawable/loc.png`). Besides `android:src`, icons can be associated using other attributes. Based on our preliminary study on icons used in the top 10,000

```
<selector>
  <item android:state_checked="true" android:drawable="
    @drawable/btn_radio_to_on_mtrl_015" />
  <item android:drawable="@drawable/
    btn_radio_to_on_mtrl_000" />
</selector>
```

Figure 6: Example Resource XML File for Icons

```
void onCreate(Bundle savedInstance) {
  View g = this.findViewById(R.id.button_esc); // FindView
  ImageView h = (ImageView) g; // cast to ImageView
  h.setImageResource(R.drawable.icon2); // change icon
  ... }
```

Figure 7: Example OnCreate Event Handler

apps downloaded from Google Play, most of the icons of interest are used in interactive UI widgets, with the top frequent widgets being `ImageView`, `Button`, `TextView`, and `ImageButton`; while the icons used in container and layout widgets, such as `ListView` and `LinearLayout`, are typically for beautifying backgrounds. In addition, icons specified in the attribute `android:background` of UI widgets are mainly used for beautifying backgrounds and not permission related. Thus, our work focuses on analyzing the icons specified in the `android:src` attributes.

Besides resource names for icons, the `android:src` attributes can specify *drawable objects*, which are frequently observed in check boxes or radio buttons. Drawable objects manage several different images, organizing the images in layers or showing different images based on the state of the UI widgets that use the drawable objects. Figure 6 shows the definition of a drawable object. This example XML file specifies two icons via the attributes `android:drawable` in the `item` elements, where the first icon will be shown if the UI widget's state is "checked" and the second icon will be shown otherwise. Based on the `android:src` attribute, we define the icon-widget association via UI layout files as:

Definition 6 (Icon-Widget Association (UI Layout)): Given a layout $L(W, E)$, an icon c is associated with a UI widget $w \in W$ if (1) $c.id$ is specified in the attribute $w.src$ where $w.src$ represents the `android:src` attribute of w, or (2) a drawable object d is specified in the attribute $w.src$ and $c.id \in D_d$, where D_d represents the set of resources IDs contained in the drawable object d.

Besides specified using XML, UI widgets may use different icons when certain events occur (e.g., switching activities). Based on our preliminary study, on average each app uses the image loading API `setImageResource` 7.4 times[1]. We next define the icon-widget association via API calls.

[1]The other image-loading APIs `setImageBitmap` and `setImageBitmap` are mainly used to load images through network or external storages but not resources included in the app's APK file.

$$\frac{may\_alias(x,y)}{\begin{array}{c}\Gamma \vdash newwid(y,x,w.id) : [y_t \mapsto \Gamma(y) \bigcup \{wid\}] \\ \Gamma \vdash newwid(y_i,x,w.id) : \Gamma_i(y_i \in dom(\Gamma))\end{array}}{\Gamma \vdash x = findViewByID(w.id) : \bigcup_i \Gamma_i}$$

Figure 8: Transfer functions for `findViewByID`

$$\frac{may\_alias(x,y)}{\begin{array}{c}\Sigma \vdash newrid(y,x,c.id) : [y_t \mapsto \Sigma(y) \bigcup \{rid\}] \\ \Sigma \vdash newrid(y_i,x,c.id) : \Sigma(y_i \in dom(\Sigma))\end{array}}{\Sigma \vdash x.setImageResource(c.id) : \bigcup_i \Sigma}$$

Figure 9: Transfer functions for `setImageResource`

Definition 7 (Icon-Widget Association (API Calls)): Given that $v_L \rightsquigarrow L(W,E)$, $w \in W$ and $v \rightsquigarrow w$, an icon c is associated to a UI widget w if an image loading API is invoked with v as the receiver object and $c.id$ as the parameter.

As shown in Figure 7, Lines 2 and 3 associate the ImageView widget to variables `g` and `h`, and Line 4 indicates that `h` will use the icon identified by `R.drawable.icon2`.

Static Analysis on UI Layout Files. We develop a static analysis technique that leverages a XML parser to parse the extracted UI layout files to build the formal UI layouts, and inspects all the UI widgets in each layout to identify the associations between the icons and the UI widgets. Figure 2a shows an example UI layout file, where the layout model `LinearLayout` is used to place three UI widgets. The UI widget `ImageView` at Line 2 is associated with an icon identified by the resource name `@drawable/loc`, which refers to the icon `loc.png` in the `res/drawable` folder. By traversing the UI tree from the root `LinearLayout` to its child node `ImageView`, our analysis can infer the association between the `ImageView` widget with id `@id/img` and the icon with the resource name `@drawable/loc`.

The analysis technique identifies the resource names of icons and the UI widgets. These resource names may be directly mapped to file names in the resource folder, or XML files that represents drawable objects as shown in Figure 6. To handle drawable objects, our analysis further parses the XML resource files and identifies all the resource names from the attribute `android:drawable` in each XML element.

Static Analysis on App Code. To compute the icon-widget associations, ICONINTENT provides a data flow analysis technique that overapproximates the associations between variables and the widget IDs and the associations between variables and the icon IDs. Figures 8 and 9 show the transfer functions of `findViewByID` and `setImageResource` in the form of inference rules. The data flow value for each variable is initialized as $\{\bot\}$ and the join operator is defined as set union. If the variable x may alias the variable y, we simply union the data flow facts from x to y. We use the environment Γ to denote data flow facts as a mapping from each variable to widget IDs. Given the statement `x = findViewByID(w.id)` where x is a variable and $w.id$ is the ID of w, we may infer the fact that x is bound to the UI widget w whose widget ID is $w.id$ (i.e., $x \rightsquigarrow w$ and $\Gamma(x) = \Gamma(x) \bigcup \{w.id\}$). If another variable y is an alias of x, then y is associated with the widget ID $w.id$ as well (i.e., $\Gamma(y) = \Gamma(y) \bigcup \{w.id\}$). The association between widget IDs and variables can also be done via the API `setID`, which follows the similar rules as `findViewByID`'s.

Our analysis also infers the association between image resource IDs and variables that represent UI widgets. This is done via using the similar transfer function as `findViewByID` 's to analyze the API method `setImageResource`. We use the environment Σ to denote data flow facts as a mapping from each variable to its resource IDs. Consider the statement `x.setImageResource(c.id)` where x is a variable bound to a UI widget w (i.e., $x \rightsquigarrow w$) and $c.id$ is the resource ID of the icon c. Whenever we observe such API in the code, we may infer the fact that x is associated with the icon c whose resource ID is $c.id$ (i.e., $\Sigma(x) = \Sigma(x) \bigcup \{c.id\}$) and w is associated with c since $x \rightsquigarrow w$. Similarly, if y may alias x, then y is associated with c (i.e., $\Sigma(y) = \Sigma(y) \bigcup \{c.id\}$).

Based on the analysis result, ICONINTENT can determine which UI widgets are associated with a given icon. Specifically, if $\Sigma(x_t)$ does not contain $\bot$, the UI widgets identified by the widget IDs (i.e., $\Gamma(x_t)$) are considered to be associated with the resource IDs $\Sigma(x_t)$. That is, we will have the icon-widget associations $\{w_t \mapsto i_t | w_t \in \Gamma(x_t), i_t \in \Sigma(x_t)\}$.

Example Analysis. Consider the example shown in Figure 7. For the UI widget variable `g`, we have $\Gamma(g) = \{$`R.id.button_esc`$\}$. Since `g` and `h` are aliases (Line 3), we have $\Gamma(h) = \{$`R.id.button_esc`$\}$. Due to the `setImageResource` at Line 4, we have $\Sigma(g) = \{$`R.drawable.icon2`$\}$, and $\Sigma(h) = \{$`R.drawable.icon2`$\}$. Thus, we have the icon-widget association $\{$`R.id.button_esc` $\mapsto \{$`R.drawable.icon2`$\}\}$.

C. Icon Mutation

This module extracts icons from the input APK file and performs image mutations to produce a set of mutated icons for each of the extracted icons. Motivated by the app icon variety shown in Figure 4, ICONINTENT leverages five commonly-used image mutation techniques [18], [19]. These techniques mutate the colors and transparencies of images in different ways, and can be combined together to produce different mutated icons (thus producing $2^5 = 32$ mutated images for each icon). We next briefly describe the color model used in digital images and the mutation techniques.

Image Mutation. A digital image is represented as a rectangular grid of pixels with fixed rows and columns, where a pixel represents a single color dot. A color in the RGB color model [20] is expressed as an RGB triplet $\langle r, g, b \rangle$, where "r", "g", and "b" are the numeric values that describe how much red, green, and blue are included in the color, respectively. To express the opacity degree of the color, the RGBA color model, $\langle r, g, b, a \rangle$, is used, which provides an extra numeric value ("a") besides the RGB triplet used in the RGB model. Using the RGBA color

Table I: Sensitive user-input categories and keywords

Category	Keywords
Camera	camera, retake
Contacts	contact, group
Email	email, mail
Location	location, locate, gps, map, place, address
Microphone	microphone, micro, karaoke, interview, voice, audio
Phone	phone, call
Photo	photo
SMS	sms, message

model, a digital image with $m * n$ pixels can be represented as a matrix M with m rows and n columns, where each cell (i.e., a pixel) in the matrix is a RGBA tuple. Image mutation techniques apply various transformations to mutate the RGBA tuples in M to produce a mutated image. We next describe the five mutation techniques.

- **Image Scaling:** This technique enlarges or shrinks the image by changing the resolution (pixels per inch) of image. A commonly used technique is to resample pixel values using nearby pixels [21] for the scaled image.
- **Grayscale Conversion:** This technique converts an image to another image in which the value of each pixel just represents only the amount of light [18].
- **Color Inversion:** This technique inverts the colors of each pixel in the image.
- **Contrast Adjustment:** This technique adjusts the contrast of colors in the image.
- **Opacity Conversion:** This technique converts the transparency differences between the objects(/texts) and the background to the color differences.

D. Icon Classification

The icon classification module classifies two types of icons (i.e., *object icons* and *text icons*) into one of the eight sensitive user-input category. We next describe the sensitive user-input categories and the two techniques in detail.

1) Sensitive User-Input Categories: Table I shows eight sensitive user-input categories and their keywords. The keywords are used to search for training icons and identify text icons. We choose these eight sensitive user-input categories because the app functionality related to these categories are popular, represent tangible sensitive resources that users can understand, and have significant security and privacy implications [9], [22], [23]. Furthermore, developers often use these icons in the UI widgets that accept the user inputs.

2) Object Icon Classification: ICONINTENT leverages object recognition to classify object icons based on a training icon set labeled with sensitive user-input categories. Given an icon as input, the technique recognizes whether the training icon (deemed as the reference object) appears in the input icon (deemed as the scene picture), and labels the input icon using the sensitive user-input category that has most recognized icons, or labels it as not sensitive if none are recognized. Algorithm 1 shows this general process.

Algorithm 1: Object Icon Classification

Input: I as the input object icon, $Category$ as the set of sensitive user-input categories, M as the hashmap that stores the mapping from the training icons to their corresponding categories,
Output: $Cout$ as the predicted category for I

```
1  Cout ← null, CatCount ← Map.Empty();      // CatCount records how may icons are recognized in I for each category
2  foreach c ∈ Category do
3  |  CatCount[c] ← 0;
4  end
5  foreach k ∈ M.keys() do
6  |  if Recog(k, I) then
7  |  |  CatCount[M[k]] ← CatCount[M[k]] + 1;
8  |  end
9  end
10 foreach c ∈ Category do
11 |  if Cout == null or CatCount[c]) > CatCount[Cout]) then
12 |  |  Cout ← c;
13 |  end
14 end
15 return Cout;  // the category with most recognized icons
```

Classifying icons based on objects inside it brings new challenges to object recognition techniques such as **Scale-Invariant-Feature-Transform (SIFT)** [15], [24] in computer vision. SIFT is a state-of-the-art technique for object recognition from pictures. It extracts key locations that are invariant with respect to image translation, scaling, and rotation. They are often image snippets with enough details such as tree textures and fur patterns. When doing the matching, the key locations are used as input to a nearest-neighbor indexing method that identifies candidate object matches. Specifically, we already use image mutation to resolve icon format and quality issues such as transparent background and low contrast. However, such adaptation is not sufficient, and we found that direct adoption of SIFT is not be effective for the following two reasons.

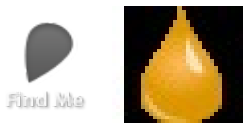


Figure 10: Rotation

- **Too Few Key Locations.** Compared with common objects such as animals and human faces in object recognition, software icons typically consist of basic shapes with smooth edges, such as lines, ovals and circles. Thus, SIFT cannot extract sufficient key Locations from the icons, and too few key locations will lead to inaccuracy in object recognition.
- **Lower Tolerance for Changes.** App icons have lower tolerance of changes such as rotation and distortion. For example, a cat is still a cat no matter how the image is rotated. However, the icon in the left sub-figure of Figure 10 no longer corresponds to "location" if it is rotated upside-down, which becomes a liquid drop icon (the right sub-figure) that is often used in weather apps. On the other hand, certain extent of change tolerance

is still helpful. The left sub-figure of Figure 10 still represents location if rotated slightly anticlockwise.

To address these two challenges, we propose corresponding techniques as follows.

Increasing Key Locations. To increase the number of key locations, we first enlarge the icon images. If an image for matching is smaller than 200×200, we automatically enlarge it to 200×200. Second, we switch to FAST algorithm [25], an alternative technique extracting many low quality key locations instead of a few high quality key locations (as in SIFT). Figure 11 presents the comparison of key locations (as blue circles) extracted by SIFT as shown in Column (a) and FAST algorithms as shown in Column (b). The low quality of key locations will be addressed together with low change tolerance in our *Relative One-to-one Matching* technique described as follows.

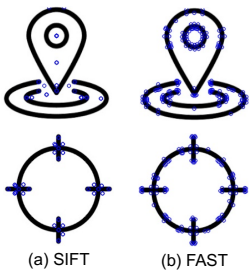


Figure 11: SIFT vs. FAST

Relative One-to-One Matching. Standard object recognition allows multiple key locations in the sample image to be matched to one key location in the scene image, which helps find smaller and blurred instances of the objects in the scene image (e.g., a cat hiding in the grass). But this is not suitable for icon classification, where change tolerance is much lower. Furthermore, the usage of low-quality key locations extracted by FAST algorithm further increases noise in key locations. Thus, we propose a novel matching technique that allows mapping a key location in the object image to only one key location in the same area in the scene image (e.g., a key location at the left-top corner of object image can only be matched to one key location in the left-top area of the scene image). We use a relative distance threshold (percentage of image weights and heights) to determine the size of the area. To achieve one-to-one mapping, we use a greedy algorithm to match the key-location pairs with the highest similarity, until no key locations left.

3) Text Icon Classification: IconIntent analyzes the embedded texts of the icons to determine whether the texts are similar to keywords in the sensitive user input categories (Table I). Based on our preliminary studies on about 300 text icons collected from top Google Play apps, more than 95% of the text icons contain 1 to 3 words [17]. This indicates that most of the text icons contain only short phrases or words. Therefore, keyword matching [9], [11], [12] can be adapted to effectively classify the text icons. However, the extracted texts from the icons are often not accurate, which may include wrong characters (e.g., "lcafiion"), extra characters (e.g., "llocation"), or miss some characters (e.g., "emai"). Thus, it is unlikely for the words in the extracted texts to exactly match a sensitive keyword.

To address these issues, we develop an edit-distance-based algorithm to compute similarities between words in the extracted texts and the keywords, and identify the *most similar keyword* based on the computed similarities; if the similarity is over a pre-defined threshold, then we classify the icon to the corresponding sensitive user input category. Our algorithm adapts edit distance [26] with n-gram substring generation, and relative length computation to compute the similarity between the extracted text from a text icon and the keywords in the sensitive user-input categories. Edit distance is a widely used approach to quantify how dissimilar two strings are by computing the minimum cost of operations required to transform one string to another. However, edit distances faces two problems *Redundant Word* and *Multi Keyword Matching* during the classification.

Redundant Word. The extracted text usually contains redundant words that do not express intentions in using sensitive data. For example, for a text icon that contains the text "enablegps" (extracted without spaces based on OCR), the word "gps" indicates the intention to use users' GPS data, which is sensitive, while the word "enable" is redundant. Redundant words may also come from the inaccurate character recognition of OCR. To address this problem, we introduce n-gram substring generation [27]. Given a keyword k, we generate a sequence of substrings from a word w where the length of each substring is within $[length(k) - 1, length(k) + 1]$. We then compare each n-gram with the keyword "gps" and find an exact match.

Keyword Similarity. Since keywords have different lengths, in many cases the Levenshtein distance [26] does not reflect the similarity as expected. For example, given a word "locatl", we compare it with two keywords "locate" and "call". The edit distance between "locatl" and "locate" is 1. For the keyword "call", we generate a n-gram list which is {"loca", "ocat", "catl"}, and the edit distance between "catl" and "call" is also 1. While they both have the same distance, we know "locatl" is more similar to "locate" rather than "call". To address this issue, we propose to measure the similarity by considering the keyword length k: $Sim_{w,k} = 1 - \frac{Ed}{length(k)}$, where Ed is the original edit distance and $length(k)$ is the length of the keyword k.

IV. Evaluation

We implemented IconIntent upon Gator [28]–[30] for static analysis and upon OpenCV [31] and Asprise OCR [14], [32] for icon classification. We evaluate IconIntent on 150 Android apps, and 5,791 manually labeled icons from the apps. We seek to answer the following research questions:

- **RQ1:** Is IconIntent effective on identifying sensitive UI widgets?
- **RQ2:** How effective is our technique on detecting suspicious permissions without GUI indication?

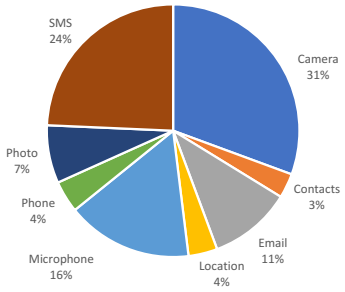


Figure 12: Icons over sensitive categories

- **RQ3:** How effective are our techniques on associating icons with UI widgets?
- **RQ4:** How effective is our technique on identifying sensitive object icons?
- **RQ5:** How effective is our technique on identifying sensitive text icons?

A. Dataset Construction

1) Training Dataset: Our classification technique of object icons requires a training dataset with positive examples in each sensitive user-input category. We do not need negative examples in our training set since our algorithm determines an icon as sensitive if it is similar enough to any positive examples in a sensitive category. To make IconIntent more extensible, our training set must be constructed within reasonable effort. We collect positive examples from two sources. First, we use the name of each sensitive category (e.g., camera) with the keyword "icon" to search for representative icons from Google Image Search, and downloaded the first 100 retrieved icons for each category. Second, we used the keywords from each category to search for icons in other apps (top 10,000 apps excluding the 150 apps in our test dataset), and fetched the first 500 icons for each category. Then, we manually labeled these icons to identify 776 unique sensitive icons used in the apps. Combined with the icons from Google Image Search, our training dataset has 1,576 icons as positive examples.

2) Test Dataset: We build our test dataset from the top apps in the Google Play market. Since apps that have UI widgets to collect sensitive user inputs are not distributed evenly across app categories [9], [11], [12] and we have to manually label all the apps in our test dataset to obtain the ground truth, we choose as our test set the top 150 apps with appearances of sensitive keywords as mentioned in Section III-D1 in their UI layout files. From the 150 apps, we manually labeled 5,791 icons, in which we identified 539 sensitive object icons, and 49 sensitive text icons. Note that during labeling we checked the context information of icons to confirm whether they are related to a sensitive category.

The distribution of sensitive icons in our test dataset on different sensitive user-input categories are presented in Figure 12. From the chart, we can see that among 588 sensitive icons identified from our test dataset, icons from all sensitive user-input categories exist. Specifically, camera, SMS, and microphone are the top three categories on the number of icons being used. The reason is that icons in these categories are popular, consistent and easy-to-understand, so they can be easily recognized by end users.

Table II: Detected Sensitive Icons (SI) and Sensitive UI Widgets (SW)

Category	#Detected SIs			#Apps with SIs	#Detected SWs	#Apps with SWs
	Object	Text	All			
Camera	148	1	149	47	65	35
Contacts	14	1	15	6	10	6
Email	44	5	49	16	25	12
Location	19	11	30	9	12	9
Microphone	75	3	78	26	65	19
Phone	20	1	21	6	38	4
Photo	41	12	53	13	19	13
SMS	125	11	136	23	24	10
All	486	44	530	135	248	97

B. Evaluation Results

1) Identifying Sensitive UI Widgets (RQ1): To answer RQ1, we consider two application scenarios in detecting sensitive UI widgets: (1) using IconIntent alone and (2) using IconIntent to complement Supor.

RQ 1.1: Using IconIntent Alone. In Table II, we present the results of IconIntent on identifying sensitive UI widgets from our test dataset. Columns 2-5 presents the number of detected sensitive object icons, sensitive text icons, all sensitive icons, and the number of apps containing detected sensitive icons. Columns 6-7 presents the number of associated sensitive UI widgets, and the number of apps containing detected sensitive UI widgets. The last row of the table shows the data combining all categories. Note that since one app may have icons / UI widgets from different categories, sum of the numbers in rows 2-9 of Column 5 and 7 is not equal to the number in Row 10.

From the table, we have the following observations. First, IconIntent effectively detects most sensitive icons (530 out of 588) from most apps (135 out of 138 apps that contains sensitive icons) in our test dataset. Second, IconIntent effectively associates the detected icons with 248 UI widgets from 97 apps. Such results indicate that not only sensitive icons are common, but sensitive UI widgets are also common, and IconIntent detects many sensitive UI widgets from most apps in our test dataset. Third, icons from different categories have different characteristics on the association with UI widgets. For example, while 125 SMS icons are detected, only 24 UI widgets from 10 apps are associated with some icons. This shows that SMS icons are often not directly associated with UI widgets. One possible reason is that SMS icons such as text bubbles are often used as static labels in chatting apps. By contrast, 20 Phone icons are associated with 38 UI widgets, which shows that phone icons are generally associated to UI widgets (e.g., buttons).

RQ 1.2: Combining with Supor. Supor is designed to detect only sensitive input fields that accept textual user inputs. Thus, to fairly evaluate effectiveness of our

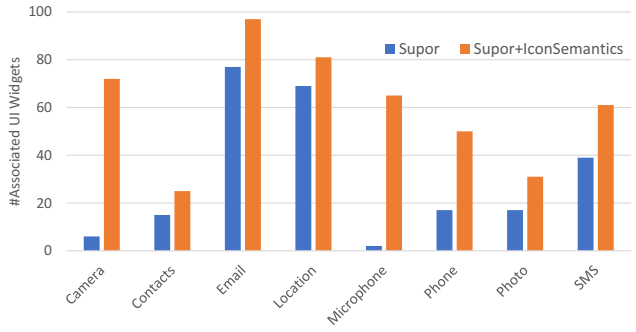


Figure 13: Effectiveness of our technique on identification of sensitive UI widget when combined with SUPOR

technique, we make two improvements to make SUPOR applicable to other UI widgets: (1) we expand the UI widgets types to include buttons, radio buttons, check boxes, and other commonly used UI widgets that accept user inputs; (2) we leverage dex2jar [33] to convert dex bytecode in APK files to Java bytecode, so that SUPOR can support custom widgets. We will use "SUPOR" to refer to this improved version of SUPOR in the rest of the section.

We applied SUPOR to the apps in our test dataset to associate text labels with UI widgets. For each category, we compare the UI widgets associated by SUPOR and by SUPOR +ICONINTENT. To enable fair comparison, we fed into SUPOR the keywords defined in Section III-D1 and collected all the UI widgets associated with sensitive keywords in sensitive text labels. The results are presented in Figure 13.

We can see that ICONINTENT can effectively complement SUPOR. SUPOR identified 242 sensitive UI widgets, while combining SUPOR and ICONINTENT can identify 487 sensitive UI widgets, achieving 101.2% improvement. Involving ICONINTENT can help SUPOR to identify 17.4% (Location) to 3150% (Microphone) more sensitive UI widgets. Specifically, ICONINTENT results in the most improvement in the categories of Camera and Microphone where icons instead of texts are used more often among apps. We also found that UI widgets found by SUPOR are almost disjoint with the UI widgets found by ICONINTENT, as only 3 UI widgets are identified by both SUPOR and ICONINTENT. It indicates that developers rarely add textual descriptions to icons for UI widgets, saving space on mobile screens.

2) Detecting Suspicious Permissions (RQ2): To answer **RQ2**, we studied the permissions requested by the subject apps, and compared them with identified sensitive icons to detect the suspicious requested permissions that are not indicated in the user interface. Previous approaches such as SUPOR [9] can achieve the same goal, but consider only textual labels on the app's GUI. Thus, we can check how many more icon-permission matches ICONINTENT finds, which reduces the suspicious permissions that require further inspection. The results are shown in Table III. We consider the six permissions that can be directly mapped to our sensitive categories. Note that Email and Photo are not listed as they cannot be easily mapped to permissions.

Table III: Suspicious Permissions Detected with SUPOR and ICONINTENT

Permission	All	SUPOR	SUPOR+ICONINTENT
CAMERA	31	27 (-12.9%)	9 (-71.0%)
CONTACTS	39	31 (-20.5%)	24 (-38.5%)
AUDIO	20	19 (-5.0%)	4 (-80.0%)
LOCATION	68	48 (-29.4%)	36 (-47.1%)
PHONECALL	27	21 (-22.2%)	16 (-40.7%)
SMS	28	20 (-28.6%)	16 (-42.9%)
TOTAL	213	166 (-22.1%)	105 (-50.7%)

Also, AUDIO permission is related to microphone icons, and PHONECALL permission is related to phone icons. Columns 2-4 show all sensitive permissions requested, the suspicious permissions identified by SUPOR, and identified by SUPOR +ICONINTENT, respectively.

From the table, we have two observations. First, ICONINTENT effectively reduces suspicious permission requests from 166 to 105 (37%), so much fewer suspicious permission requests need to be inspected. Second, ICONINTENT achieves different effectiveness on different categories. In particular, ICONINTENT is most effective in CAMERA (reducing 18 of 27) and AUDIO (reducing 15 of 16), indicating that icons are dominantly used in these categories. Note that this is not a fair comparison between SUPOR and ICONINTENT, as we consider only the categories where icons are commonly used. However, the result does show that in these popular categories, considering only text is far from sufficient, and applying ICONINTENT can significantly reduce the suspicious permissions to be inspected.

3) Associating Icons with UI Widgets (RQ3): To answer RQ3, we studied the number of UI widgets and their icons associated by ICONINTENT. From the 5,791 icons in the test dataset, ICONINTENT associated them with 5,408 UI widgets. Specifically, UI analysis helped to associate 4,165 UI widgets, and icon-association analysis helped to associate additional 1,243 UI widgets. Among the associated UI widgets, 248 are associated with sensitive icons and are thus sensitive UI widgets, while 53 are associated with mistakenly classified icons so they are false positives. Note that we will evaluate our icon classification techniques in Sections IV-B4 and IV-B5, and we only include true positives in Figure 13. Among these 248 UI Widgets, 234 can be detected with UI analysis and icon-association analysis can help to identify additional 14 sensitive UI widgets.

4) Object Icon Classification (RQ4): The results of object icon classification on the test dataset are presented in Table IV. Columns 2-4 show the precision, recall, and F-score of our technique for each icon category. We can observe that ICONINTENT can achieve an average F-score of 87.7% (with distance threshold as 0.3). Furthermore, we compare the effectiveness (F-Score) of our techniques described in Section III-D2 with that of default SIFT technique, and turning off each of our techniques: without increasing key locations and without change-aware matching. From the figure, we can see that ICONINTENT's F-score (87.7%) outperforms those of SIFT (48.1%), without mutation (75.8%), without

Table IV: Results for Object-Icon Classification

Setting	P (%)	R(%)	F (%)
SIFT	43.0	54.5	48.1
Without Mutation	91.2	64.9	75.8
Without Increkey	63.7	90.0	74.6
Without ROM	76.0	89.4	82.2
IconIntent	88.2	87.3	87.7

Table V: Results for Text-Icon Identification

Setting	P (%)	R(%)	F (%)
Without Mutation	91.7	22.9	36.6
IconIntent	89.8	89.8	89.8

key location increase (without Increkey, 74.6%), and without Relative One-to-one Mapping (without ROM, 82.2%).

5) Text Icon Classification (RQ5): The results of text icon classification are presented in Table V, which is of the same format as Table IV. We can see that our text icon classification results have few false positives and false negatives, and the image mutation techniques can improve the recall significantly. We set the threshold for text similarity as 0.8 since it achieves the best results in terms of precision, recall, and F-score. For false positives, there are two types of misclassification. First, the OCR recognized texts from some icons which do not contain any text, and the extracted texts matched some sensitive keywords. Second, there is one application breaking apart company English logos, which confused our keyword matching algorithm. For false negatives, there are three types of misclassification. First, we missed some keywords. If we add particular keywords, it will be solved. Second, when IconIntent performed image mutations, IconIntent did not consider the rotated images. We can implement a new image mutation to get a better result. Third, OCR cannot recognize the correct text from some blurred images.

C. Threats to Validity

The main internal threat comes from the mistakes we may make during icon labeling. To reduce the threat, we check the context of the icons when we cannot tell whether an icon is related to a sensitive category. There are two main external threats to validity. First, our experiment evaluates only the apps with many sensitive UI widgets, but this is reasonable because these apps are also the ones IconIntent will be mainly applied to. Second, since the keywords we used as queries are from eight sensitive categories, our evaluation may be limited to apps collecting data in these categories. This threat is unavoidable because the difference between sensitive data and insensitive data must be defined in some way. IconIntent can be easily extended to support other categories of sensitive data, but more evaluations will be required for those categories.

V. Related Work

Computer Vision Techniques for Software Engineering Tasks. REMAUI [34] applies computer vision techniques for reverse engineering UIs of mobile apps. Sikuli [35], [36] uses image recognition to identify and control UI components for automating UI testing. WebDiff [37] and XPERT [38] leverage computer vision techniques to detect visual differences, assisting the task of detecting cross browser rendering issues. Instead of detecting standard UI elements and comparing visual appearances, our approach uses computer vision techniques to find icons similar to our collected icons and extract texts from icons, which are combined with program analysis techniques to understand association between icons and UI widgets.

UI Analysis of Mobile Apps. SUPOR [9], UIPicker [11], and UiRef [12] are among the first works to analyze the descriptive texts in apps' UI for determining whether the corresponding user inputs contain sensitive data. AsDroid [39] checks the compatibility of the descriptive texts and the intentions represented by the sensitive APIs. PERUIM [41] extracts the permission-UI mapping from an app based on both dynamic and static analysis, helping users understand the requested permissions. Liu et al. [42] propose an automatic approach for annotating mobile UI elements with both structural semantics such as buttons or toolbars and functional semantics such as add or search. In these works, the security and privacy implications of icons remained unexplored, and our approach opens up a new direction in analyzing sensitive icons in UIs. Furthermore, our static analysis associates UI widgets with variables in the code, which cannot be inferred by just analyzing the UI.

Textual Analysis of Mobile Apps. WHYPER [22] and AutoCog [43] adapt natural language processing techniques for analyzing apps' descriptions and infer the mapping between sentences in app descriptions and permissions. CHABADA [44] clusters app descriptions' topics and identifies outliers in each cluster with respect to their API usage. BidText [45] detects sensitive data disclosures by performing bi-directional data flow analysis to detect variables that are at the sink points and are correlated with sensitive text labels. IconIntent complements these techniques to better understand apps' intentions.

VI. Conclusion and Future Work

In this work[2], we present a novel framework IconIntent that performs program analysis techniques to associate icons and UI widgets and adapts computer vision techniques to classify the associated icons into eight sensitive categories for Android apps. We have conducted evaluations on 150 market apps. The results show that IconIntent effectively identifies sensitive UI widgets (248 UI widgets in 97 apps), and reduces suspicious permissions to inspect. IconIntent can be integrated with various privacy analysis tools, such as GUILeak [10] to help developers trace information types mentioned in privacy policies to icons. In future work, we plan to adopt deep learning techniques to further improve the accuracy in icon recognition.

[2]The work is supported in part by NSF grants CNS-1755772 and CNS-1748109.

REFERENCES

[1] K. W. Miller, J. Voas, and G. F. Hurlburt, "Byod: Security and privacy considerations," *IT Professional*, vol. 14, no. 5, pp. 53–55, 2012.

[2] ZDNet, "Research: 74 percent using or adopting byod," 2015. [Online]. Available: http://www.zdnet.com/article/research-74-percent-using-or-adopting-byod/

[3] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *IEEE Symposium on Security and Privacy (IEEE S & P)*, 2012.

[4] B. Liu, B. Liu, H. Jin, and R. Govindan, "Medusa: A programming framework for crowd-sensing applications," in *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2015.

[5] S. Rosen, Z. Qian, and Z. M. Mao, "Appprofiler: A flexible method of exposing privacy-related behavior in android applications to end users," in *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2013.

[6] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[7] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2014.

[8] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: Statically vetting android apps for component hijacking vulnerabilities," in *the ACM Conference on Computer and Communications Security (CCS)*, 2012.

[9] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, and G. Jiang, "SUPOR: precise and scalable sensitive user input detection for android apps," in *USENIX Security Symposium*, 2015.

[10] X. Wang, X. Qin, M. B. Hosseini, R. Slavin, T. D. Breaux, and J. Niu, "GUILeak: Tracing privacy policy claims on user input data for android applications," in *International Conference on Software Engineering (ICSE)*, 2018.

[11] Y. Nan, M. Yang, Z. Yang, S. Zhou, G. Gu, and X. Wang, "UIPicker: User-input privacy identification in mobile applications," in *USENIX Security Symposium*, 2015.

[12] B. Andow, A. Acharya, D. Li, W. Enck, K. Singh, and T. Xie, "UIRef: Analysis of sensitive user inputs in android applications," in *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2017.

[13] N. Babich, "Ghost buttons in UX design," 2016, https://www.techinasia.com/talk/ghost-buttons-ux-design.

[14] S. Mori, H. Nishida, and H. Yamada, *Optical character recognition*. John Wiley & Sons, Inc., 1999.

[15] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International Journal of Computer Vision (IJCV)*, vol. 60, no. 2, pp. 91–110, 2004.

[16] Google, "Android view system," 2017, https://developer.android.com/guide/topics/ui/declaring-layout.html.

[17] Z. Cao, "Sensitive text icon classification for android apps," Master thesis, Case Western Reserve University, 2017.

[18] C. Kanan and G. W. Cottrell, "Color-to-grayscale: does the method matter in image recognition?" *PloS one*, vol. 7, no. 1, 2012.

[19] R. J. Schalkoff, *Digital image processing and computer vision*. Wiley New York, 1989, vol. 286.

[20] "RGB color model," 2003, https://en.wikipedia.org/wiki/RGB_color_model.

[21] K. Turkowski, "Filters for common resampling tasks," in *Graphics gems*. Academic Press Professional, Inc., 1990, pp. 147–165.

[22] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "WHYPER: Towards automating risk assessment of mobile applications," in *USENIX Security Symposium*, 2013.

[23] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: User attention, comprehension, and behavior," in *Symposium on Usable Privacy and Security (SOUPS)*, 2012.

[24] D. G. Lowe, "Object recognition from local scale-invariant features," in *International Conference on Computer Vision (ICCV)*, 1999.

[25] E. Rosten, R. Porter, and T. Drummond, "Faster and better: A machine learning approach to corner detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, vol. 32, no. 1, pp. 105–119, 2010.

[26] E. S. Ristad and P. N. Yianilos, "Learning string-edit distance," *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, vol. 20, no. 5, pp. 522–532, 1998.

[27] G. Kondrak, "N-gram similarity and distance," in *International Symposium on String Processing and Information Retrieval (SPIRE)*. Springer, 2005, pp. 115–126.

[28] A. Rountev and D. Yan, "Static reference analysis for gui objects in android software," in *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2014.

[29] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev, "Static control-flow analysis of user-driven callbacks in Android applications," in *International Conference on Software Engineering (ICSE)*, 2015.

[30] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev, "Static window transition graphs for android (t)," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015.

[31] O. team, "OpenCV," 2017, http://opencv.org/.

[32] Asprise, "Asprise ocr and barcode recognition," 2017, http://asprise.com/royalty-free-library/java-ocr-api-overview.html.

[33] B. Pan, "dex2jar: Tools to work with android .dex and java .class files," 2017, https://github.com/pxb1988/dex2jar.

[34] T. A. Nguyen and C. Csallner, "Reverse engineering mobile application user interfaces with REMAUI," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015.

[35] T. Yeh, T. Chang, and R. C. Miller, "Sikuli: using GUI screenshots for search and automation," in *the ACM Symposium on User Interface Software and Technology (UIST)*, 2009.

[36] T. Chang, T. Yeh, and R. C. Miller, "GUI testing using computer vision," in *International Conference on Human Factors in Computing Systems (CHI)*, 2010.

[37] S. R. Choudhary, H. Versee, and A. Orso, "WEBDIFF: automated identification of cross-browser issues in web applications," in *IEEE International Conference on Software Maintenance (ICSM)*, 2010.

[38] S. R. Choudhary, M. R. Prasad, and A. Orso, "X-PERT: a web application testing tool for cross-browser inconsistency detection," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2014.

[39] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "AsDroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction," in *International Conference on Software Engineering (ICSE)*, 2014.

[40] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "AppIntent: analyzing sensitive data transmission in android for privacy leakage detection," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2013.

[41] Y. Li, Y. Guo, and X. Chen, "PERUIM: Understanding mobile application privacy with permission-ui mapping," in *ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp)*, 2016.

[42] T. F. Liu, M. Craft, J. Situ, E. Yumer, R. Mech, and R. Kumar, "Learning design semantics for mobile apps," in *ACM Symposium on User Interface Software and Technology (UIST)*, 2018.

[43] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, "Autocog: Measuring the description-to-permission fidelity in android applications," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

[44] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *International Conference on Software Engineering (ICSE)*, 2014.

[45] J. Huang, X. Zhang, and L. Tan, "Detecting sensitive data disclosure via bi-directional text correlation analysis," in *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2016.