

What You See Is What You Get? It Is Not the Case! Detecting Misleading Icons for Mobile Applications

Linlin Li
lill3@mail.sustech.edu.cn
Southern Univ. of Sci. and Tech.
Shenzhen, China

Ying Wang*
wangying@swc.neu.edu.cn
Northeastern University
Shenyang, China

Ruifeng Wang
twilight_wang@163.com
Northeastern University
Shenyang, China

Cuiyun Gao
gaocuiyun@hit.edu.cn
Harbin Institute of Technology
Shenzhen, China

Xian Zhan
chichoxian@gmail.com
Southern Univ. of Sci. and Tech.
Shenzhen, China

Sinan Wang
wangsn@mail.sustech.edu.cn
Southern Univ. of Sci. and Tech.
Shenzhen, China

Yepang Liu[†]
liuyp1@sustech.edu.cn
Southern Univ. of Sci. and Tech.
Shenzhen, China

ABSTRACT

With the prevalence of smartphones, people nowadays can access a wide variety of services through diverse apps. A good Graphical User Interface (GUI) can make an app more appealing and competitive in app markets. Icon widgets, as an essential part of an app's GUI, leverage icons to visually convey their functionalities to facilitate user interactions. Whereas, designing intuitive icon widgets can be a non-trivial job. Developers should follow a series of guidelines and make appropriate choices from a plethora of possibilities. Inappropriately designed or misused icons may cause user confusion, lead to wrong operations, and even result in security risks (e.g., revenue loss and privacy leakage). To investigate the problem, we manually checked 9,075 icons of 1,111 top-ranked commercial apps from Google Play and found 640 misleading icons in 312 (28%) of these apps. This shows that misleading icons are prevalent among real-world apps, even the top ones.

Manually identifying misleading icons to improve app quality is time-consuming and laborious. In this work, we propose the first framework, ICONSEER, to automatically detect misleading icons for mobile apps. Our basic idea is to find the discrepancies between *the commonly perceived intentions of an icon* and *the actual functionality of the corresponding icon widget*. ICONSEER takes an Android app as input and reports potential misleading icons. It is powered by

a comprehensive icon-intention mapping constructed by analyzing 268,353 icons collected from 15,571 popular Android apps in Google Play. The mapping includes 179 icon classes and 852 intention classes. Given an icon widget under analysis, ICONSEER first employs a pre-trained open-set deep learning model to infer the possible icon class and the potential intentions. ICONSEER then extracts developer-specified text properties of the icon widget, which indicate the widget's actual functionality. Finally, ICONSEER determines whether an icon is misleading by comparing the semantic similarity between the inferred intentions and the extracted text properties of the widget. We have evaluated ICONSEER on the 1,111 Android apps with manually established ground truth. ICONSEER successfully identified 1,172 inconsistencies (with an accuracy of 0.86), among which we further found 482 real misleading icons.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

Android Apps, Icon Design, Discrepancy Detection, Deep Learning

ACM Reference Format:

Linlin Li, Ruifeng Wang, Xian Zhan, Ying Wang, Cuiyun Gao, Sinan Wang, and Yepang Liu. 2023. What You See Is What You Get? It Is Not the Case! Detecting Misleading Icons for Mobile Applications. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23)*, July 17–21, 2023, Seattle, WA, USA. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3597926.3598076>

1 INTRODUCTION

“One picture is worth a thousand words.”

- Fred R. Barnard, in *Printers' Ink*, 1927.

Nowadays, smartphones are playing significant roles in our daily lives. People can access various services through a wide range of apps. Graphical User Interfaces (GUIs) serve as a bridge between

*Ying Wang is also affiliated with Hong Kong University of Science and Technology.

[†]Yepang Liu is the corresponding author. He is affiliated with both the Department of Computer Science and Engineering and the Research Institute of Trustworthy Autonomous Systems at Southern University of Science and Technology.


Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0221-1/23/07...\$15.00

<https://doi.org/10.1145/3597926.3598076>












apps and users. A well-designed GUI can make an app more appealing and usable, which helps increase app downloads and usage [23, 27, 37, 40, 41, 47, 50]. As an important part of GUIs, UI widgets (e.g., buttons) are ubiquitous and provide a convenient way for users to interact with the apps. Many UI widgets leverage icons (small images, e.g., ) to visually convey their functionalities. These widgets are called icon widgets in the literature [53].




Designing intuitive icons can be non-trivial, requiring designers and developers to follow certain guidelines [1]. A well-designed icon can accurately convey the corresponding widget's functionality to users, while a poorly-designed icon may confuse users and even mislead them to perform wrong operations [38]. Figure 1 shows three examples of real misleading icons. Figure 1(a) shows two screenshots from v2.5 and v3.5 of a drawing app "Paint and Drawing Fun" [19] for kids. The left screenshot contains an icon (in the red box) for removing drawings, but the icon looks like a "photo", which makes users misinterpret its real functionality. We found users' complaints about that on Google Play: "Can you replace the top left icon with a dustbin icon? My baby can't figure out what the icon is for." In the new version (v3.5), developers replaced the "photo" icon with a "dustbin" icon. Figure 1(b) shows two screenshots from version 1.8.1 and 2.0.0 of *BabyCam* [7]. The left one (v1.8.1) contains an icon that is a "share" button but many users think it is the logo of this app. To avoid confusion, in the new version (v2.0.0), the share functionality was removed by developers, and the icon became non-clickable. The two screenshots in Figure 1(c) are from version 6.6 and 7.0.5 of *Speak and Translate Languages* [20]. The left one (v6.6) contains an icon-widget whose real functionality is to exchange the two languages but the icon contains two arrows pointing to the same direction, which usually indicates "randomization". In v7.0.5, developers replaced it with an appropriate one, a "two-way arrow" icon. All these three misleading icons are from popular apps with over one million downloads.

We found that misleading icons are prevalent in real-world apps. According to our manual investigation of 9,075 icons in 1,111 top-ranked Android apps from 32 different app categories on Google Play, 28% (312/1,111) of the apps contain at least one misleading icon. Such misleading icons can lead to poor user experience, which may further affect app rating and downloads [25, 55]. Unfortunately, in practice, spotting them is not an easy task. It may require developers to carefully check each icon widget in their app or find potential issues from users' feedback via review analysis. Such a process can be time-consuming and laborious, especially when the app contains a large number of icon widgets or user reviews. It is therefore highly desirable to have an automated tool to assist developers or app companies in detecting misleading icons. However, little effort has been devoted into designing such tools in both industry and academia. To fill the gap, we propose the first framework, ICONSEER, to automatically identify misleading icons in their mobile apps.

To detect misleading icons, a possible solution is to analyze whether there is any discrepancy between *the actual functionality of an icon widget* and *the commonly perceived intentions of its corresponding icon*. To infer the functionality of an icon widget, we may leverage two data sources. First, it is possible to analyze widget functionality by performing program analysis on the corresponding event handler (i.e., the callback). However, it is known to be difficult to accurately inferring high-level semantics from low-level code

features such as framework API calls. Besides, many techniques, e.g., code obfuscation and hardening, can hide or encrypt the real intention of code, which can dramatically increase the analysis difficulty. Second, textual properties such as the annotations around the icon widgets or the content descriptions [9] (cf. Section 2) can also help infer widget functionality. After a full consideration of various factors (e.g., efficiency, effectiveness, the availability of code, etc.) and a pilot study (cf. Section 3), we decided to leverage the textual properties to infer the functionalities of icon widgets. On the other hand, to obtain the accurate and complete commonly perceived intentions for an icon, we need to tackle two specific challenges.

- **The flexibility of icon design.** When it comes to icon design, developers or designers can have an unlimited number of choices, ranging from simple symbols with obvious meaning (e.g., , ) to complex images with delicate details. The designed icons may differ in shapes, sizes, or colors. For example, all of these icons , , , ,  can represent "play music". Both  and  can represent the delete operation.
- **Multiple intentions of icons.** The same icon can represent different meanings. For instance, the icon  can represent a filter, a setting button or a volume adjuster. The icon  can represent numerous intentions, such as refresh, repeat, playback, update, reload, sync, spin, rotate, loading, and so on.

To address the first challenge, we manually analyzed our collected 1,111 top Android apps and made an important observation. We found that although developers have abundant choices when designing icons, for app usability, they tend to use similar icons (e.g., , , ) for widgets providing similar functionalities, as common icons require little cognitive effort and thus users can easily understand their meanings. Inspired by this observation, we automatically extracted 268,353 icons from 15,571 popular Android apps of different categories. We sampled 10% of these icons and categorized them into 179 classes via an open coding process. Each icon class contains similar icons with different styles. To alleviate the interference caused by the flexibility of design, we used the labelled 26,835 icons and trained an open-set deep learning model [24] to automatically capture deep linkages among icons. This model was then employed to classify unlabelled icons into the 179 icon classes or an "unknown" class. In this way, we ensured the diversity of icons in each class. To address the second challenge, for each of the 179 icon classes, we extracted the icons' corresponding text properties by static analysis of the app layout files and code. By applying statistical analysis, we obtained frequent text properties of each icon class to represent the commonly-perceived intentions of the icons. After careful manual validation, we finally mapped the 179 classes of icons to 852 classes of intentions, which is referred to as icon-intentions mapping hereinafter.

With the comprehensive icon-intentions mapping, our framework, ICONSEER, performs the following steps to help developers identify misleading icons in Android apps. Given an icon widget under analysis, ICONSEER first classifies the icon into possible classes and retrieves the corresponding intentions from the icon-intentions mapping. It then extracts the developer-specified text properties of the icon widget as a representation of the widget's actual functionality. Finally, it compares the semantic similarity of the retrieved

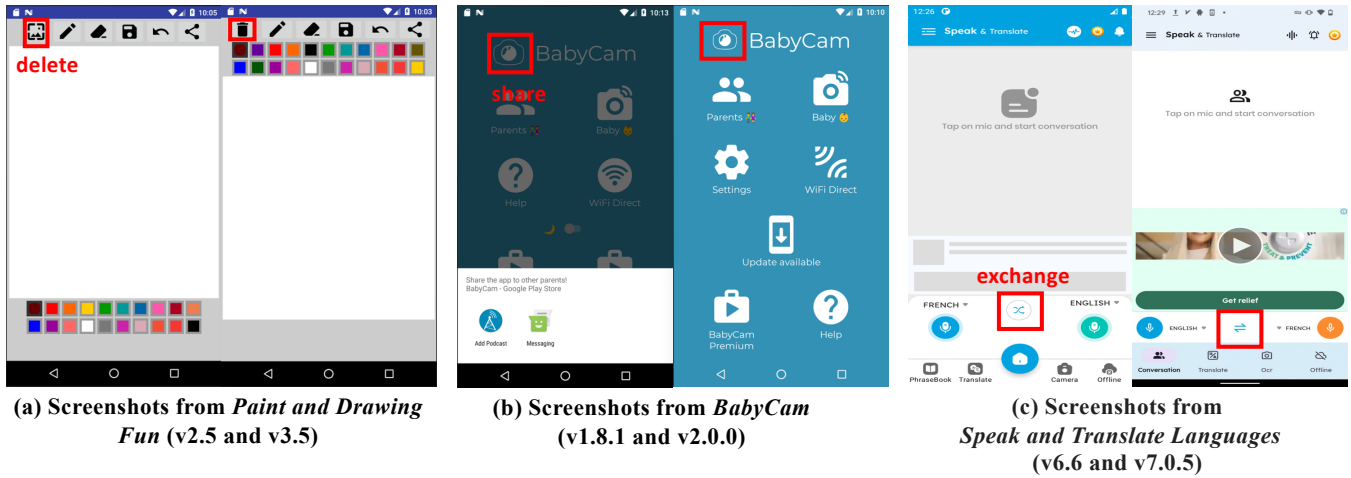


Figure 1: Examples of misleading icons

intentions and the extracted text properties. If there exists any discrepancy, it will report a warning to developers, who can further validate whether the icon is really misleading or not.

To evaluate our work, we first compared our constructed icon-intention mapping with those constructed by two prior studies [29, 39]. Compared with the two existing mappings, our mapping involves a much larger number of icons (268,353 vs. ~100k or 73,449), more icon classes (179 vs. 40 or 80), and more intention classes (852 vs. 184 or 320). Second, we conducted experiments to evaluate the performance of the trained open-set deep learning model. Results show that the model can achieve an F1-score of 0.8519, outperforming different baselines [31, 32, 34, 46]. Finally, to evaluate ICONSEER’s capability of detecting misleading icons, we conducted experiments on 1,111 real-world apps. The results show that ICONSEER is able to effectively detect inconsistencies between icon intentions and widget functionalities and find 482 real misleading icons. To summarize, this paper makes the following major contributions.

- To the best of our knowledge, we are the first to investigate the misleading icons problem in real-world apps. Our pilot study reveals the prevalence of this problem.
- We propose a novel framework, ICONSEER, to automatically detect misleading icons in Android apps. Experiments on popular Android apps show that ICONSEER can effectively help developers find real icon design issues to further improve app quality.
- We release our constructed icon-intentions mapping and all experiment data [15]. In future, we will continue to maintain and improve the mapping. We believe that our mapping can support many follow-up studies. For example, it can be used to detect and repair accessibility issues (e.g., misused or meaningless content descriptions) and confusing textual annotations of icons. It can also be used to detect malicious behaviors of UI widgets.

2 BACKGROUND

This section introduces text properties related to icon widgets.

- **Content description of icon widgets.** Users with vision impairment often have difficulties in interacting with apps. To improve app accessibility [4], Google recommends developers to provide a

content label [9], a short textual description of functionality, for each interactive UI widget in their apps, which will be read out by smartphones’ built-in screen reader to help vision-impaired users to use the apps. [4]. Developers provide content label of an icon widget by setting its content_description attribute. Therefore, the value of content_description can be leveraged to reveal the functionality of icon widgets.

- **Identifier of icon widgets.** When defining an icon widget, developers often assign it a unique ID by setting the widget’s id attribute. Then the widget can be retrieved from the view tree (a.k.a. GUI hierarchy) with View.findViewById() or Activity.findViewById() when needed. For app maintenance consideration, developers often supply meaningful ids (e.g., “searchButton”) for icon widgets. Hence, it is feasible to obtain the functionality of icon widgets by analysing the value of their id attributes.

- **Surrounding text of icon widgets.** To ease user interaction, developers often provide textual annotations around icon widgets to explain its functionalities. For example, in Figure 1(b), “Parents” indicates the functionality of the icon widget 👨‍👩‍👧. We can obtain icon-surrounding-text by extracting the value of the text component satisfying the following conditions[42]: (1) being a sibling of the icon-widget in the GUI hierarchy; (2) being on a horizontal or vertical line with the icon (the abscissa or ordinate difference between the center point of the icon and that of the text component is less than 20 pixels); (3) being close to the icon (the abscissa or ordinate difference between the icon boundary and the text component boundary is less than 200 pixels); (4) not being the child components of Toolbar components (the text in a Toolbar, e.g., “Speak & Translate” in Figure 1(c), is typically the title of an activity, which does not indicate widget functionalities).

3 PILOT STUDY

In this section, we perform a pilot study to investigate: (1) the significance of the misleading icons problem and (2) the feasibility of detecting misleading icons via finding semantic inconsistencies between the commonly perceived intention of an icon and the text properties of its corresponding icon widget.

Table 1: Statistics of Discrepancies and Misleading Icons






	# Quadruples	# Discrepancies	# Misleading Icons
content_description	1,920	254 (13.2%)	41 (16.1%)
id	5,505	1,138 (20.7%)	224 (19.7%)
icon-surrounding-text	4,104	1,156 (28.2%)	508 (43.9%)
all three text properties	9,075	1,576 (17.4%)	640 (40.6%)

• **Data collection.** Our study requires a data set of icons and the text properties associated with the corresponding widgets. For this purpose, we first crawled the latest version of the top-100 commercial apps for 32 categories from Google Play. We successfully downloaded 3,033 apps. We then adopted DroidBot [36], a light-weight action (e.g., click, scroll) generator, to automatically explore each of the collected apps for half an hour. 24,731 pairs of screenshots and GUI hierarchies were collected after the exploring process. We cropped the icons from the screenshots following the heuristics proposed by Liu et al. [39]: (1) an icon should be visible to user, (2) its occupied area should be less than 5% of the total screen area, and (3) its aspect ratio should be greater than 0.75. We extracted the content_description, id, and icon-surrounding-text of the icon widgets from the GUI hierarchies. In this way, we obtained 15,668 unique quadruples, in the form of $\langle icon, content_description, id, icon_surrounding_text \rangle$, each containing at least one non-empty text property. Here, we regard two quadruples as the same if the three text properties are the same and the Structural Similarity Index (SSIM) of the two icons is larger than 0.9 [51]. It is worth mentioning that these 15,668 quadruples are from 1,111 apps instead of all the 3,033 apps. The main reasons that many apps have no icons extracted are as follows: (1) we performed dynamic analysis using the Android emulator but some apps cannot run in emulators due to their internal protection mechanism [44]; (2) DroidBot failed to start some apps; (3) some apps have a login page requiring the username and password, which could not be skipped by DroidBot. We will further discuss this in Section 6.

• **Data cleaning.** We performed the following steps to further remove noises from our data set. First, we removed 1,652 quadruples containing icons with pure color. Such icons carry little semantic information¹ and may be accidentally collected due to the glitches of image cropping (i.e., the real icons may not be pure-colored ones). Second, we removed 1,058 quadruples containing icons with text using *PaddleOCR* [18]. Such icons are less likely to cause serious confusions since users may rely on the embedded text to infer icon intention. Third, we removed 3,883 quadruples containing three meaningless text properties such as “null”, “icon”, “default”, and “no image found”². Finally, 9,075 quadruples remained after our data cleaning. Among these 9,075 quadruples, there are 1,920 ones containing non-empty content_description, 5,505 ones containing non-empty id and 4,104 ones containing non-empty icon-surrounding-text, as shown in Table 1 (column #2).

• **Misleading icons identification.** To identify misleading icons, we first manually checked each of the 9,075 quadruples to find semantic inconsistencies between the icon and the three text properties. If there are inconsistencies, we then installed and ran the concerned apps on a real device, i.e., Google Pixel 4 with Android 11,

Table 2: Examples of misleading icons

App Name	Version	Downloads	Icon	Widget	Functionality
HP Print Service Plugin [13]	22.1.0.38	500M+		Advanced settings	
ABC [3]	10.26.0.101	10M+		Open my list	
HD Wallpapers [12]	1.7.1	10M+		Close menu	
Cool DJ Club Theme [10]	7.5.0_0527	5M+		More keyboard themes	
10times [2]	3.8.10	100k+		Explore new events	

to understand the functionalities of the corresponding icon-widgets to decide whether the icons are really misleading or not. We performed such dynamic validation because the inconsistencies may also be caused by inaccurate or wrong text properties. This manual checking process was independently performed by two authors of this paper. When there were conflicting decisions, they would invite another author to discuss and then vote to get the final results.

• **Results.** Table 1 presents the results of our pilot study. The second column shows the number of checked quadruples that contain each type of text properties. The third column shows the number of discrepancies (and the corresponding proportion) by examining the icons and the corresponding text properties. The last column shows the number of real misleading icons and the proportion in the number of discrepancies. For example, by examining the 1,920 content_descriptions and the corresponding icons, we found 254 inconsistencies, from which we finally identified 41 real misleading icons. In total, among the 9,075 quadruples, we found 640 misleading icons, which are from 312 different apps. Table 2 lists several examples of such identified misleading icons.

• **Reporting issues to developers.** We randomly selected 111 misleading icons from 640 founded ones and reported them to the app developers via emails. So far, we have received 31 responses. 20 (64.5%) of them confirm our reported issues and the remaining 11 (35.5%) say that the issues will be further examined by experts for confirmation. More details can be found on our website [15].

• **Users’ perceptions of misleading icons.** To understand how users think about the detected misleading icons, we randomly selected 100 icons from the 640 misleading ones and collected users’ opinions on them by issuing questionnaires. As the example in Figure 2(a) shows, for each of the 100 icons, we prepared a question, which contains the following content: (1) the misleading icon, (2) an alternative icon, (3) the associated screenshot of the app with a red bounding box marking the location of the misleading icon, and (4) the actual functionality of the corresponding widget. We shared our questionnaire with 13 volunteers, who have used smartphone apps for at least five years, and asked them to rate the qualities of the two icons from 1 to 5, where 1 means the icon conveys incorrect information to the user, 2 means the user can’t figure out the icon’s intention, 3 means it took the user a while to correctly figure out the icon’s intention, 4 means the user can correctly figure out the icon’s intention at the first glance but thinks that there are better options, and 5 means the icon is good enough. We also collected the confidence levels of the volunteers’ responses for each question by letting them select high, low or medium. For each question, we made sure that we collected responses from at least 10 volunteers. In the end, we received 1300 answers in total. After filtering

¹We do not deal with misleading icons that are pure-colored in this work. So far, there is no evidence that such misleading icons are common in practice.

²The full list of meaningless text can be found on our website [15].

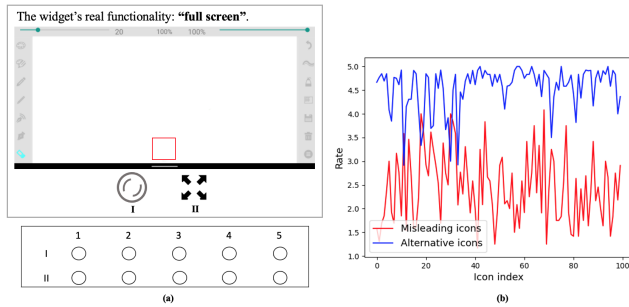


Figure 2: An example question and the study results

answers with low confidences, 1214 answers remained. We then calculated the average rating for each misleading icon and its paired alternative icon. As shown in Figure 2(b), misleading icons received low ratings, with an average of 2.46. Some of them could convey wrong information to users, which may cause serious confusions and wrong user interactions. In comparison, the ratings of alternative icons are much higher, with an average of 4.55. These results indicate that whether icons can precisely convey app functionality information indeed matters to many users.

• **Developers’ actions to misleading icons.** Our email communications with app developers show that misleading icons can indeed draw developers’ attention. However, it is still unclear whether developers would take actions to fix icons that can potentially mislead users. Since the 312 apps with misleading icons were collected in October 2022 and 213 of them have been updated, we further studied whether the 436 misleading icons found in the 213 apps have already been fixed by developers. For the study, we crawled the latest version of the 213 apps. We installed and ran these updated apps on a Google Pixel 4 device with Android 11, to check whether our found misleading icons were fixed. A misleading icon was considered fixed if (1) the shape of the icon was changed, not just the style such as color and background, or (2) the icon was removed, but the relevant functionality remained. With our manual checking, we found that 26 misleading icons were already fixed, in which 19 icons were changed and 7 icons were removed (the functionality remained). More information can be found on our website [15]. These results show that the problem of misleading icons is indeed relevant to real-world app developers.

Conclusion: We can make four observations from our pilot study. First, it is feasible to detect misleading icons by checking semantic inconsistencies between the icon intentions and the text properties of the corresponding widgets. Second, misleading icons are widespread. Third, users generally agree that the detected misleading icons could convey wrong or unclear information to them and could be replaced with better ones. Lastly, 26 of the detected misleading icons were quickly fixed in the apps’ new versions, showing that developers indeed care about such issues.

4 METHODOLOGY

We propose a framework, ICONSEER, to detect misleading icons. As shown in Figure 3, ICONSEER is based on icon-intentions mapping, which maps commonly perceived intentions with each icon class.

It takes an Android app as input and reports potential misleading icons. In the following, we first introduce the construction of the mapping and then present the detection process of ICONSEER.

4.1 Icon-Intentions Mapping Construction

4.1.1 Apps Collection. ICONSEER relies on a high-quality and comprehensive icon-intentions mapping that maps icon classes to their commonly perceived intentions to identify misleading icons in real-world Android apps. In order to construct the mapping, we need a large set of representative and diverse apps. Thus, we crawled the latest version of the top-500 apps for 32 categories from Google Play and 15,571 apps were successfully downloaded.

4.1.2 Icon-Text Pairs Extraction. This step aims to identify icons from the apps and extract the text properties associated with them. Here, we use the term “icon-text pair” to denote an icon and all of its associated text properties. In the pilot study, we collected quadruples through dynamic analysis, where a quadruple includes an icon and the three text properties. However, the dynamic approach suffers from two limitations: first, it takes a long time to run each app (half an hour per app in the pilot study); second, it cannot collect all icons in the app due to the limited coverage. These two limitations prevent us from collecting a comprehensive dataset. So ICONSEER adopts static analysis to extract icon-text pairs from an app’s code and resources files. Notice that, unlike the dynamic approach, with static analysis, we can only extract two text properties, i.e., `content_description` and `id`. This is because the *icon-surrounding-text* is rendered at runtime, while static analysis cannot accurately extract such runtime information in a lightweight manner. In the following part, we illustrate how ICONSEER performs static analysis on two types of app resource: the layout XML files and the program code.

• **Static analysis on layout XML files.** Android developers usually define an app’s GUI components as layout files, which are stored in XML format under the `res/layout/` directory in the app archive [16]. ICONSEER parses the layout files and extracts the values of `android:src`, `android:id` and `android:contentDescription` attributes of the image components [39]. Here, `android:src` specifies image resource (i.e., *drawable objects*), which can mapped to certain files (e.g., `res/drawable/search_btn.png`) in the resource directory. The other two attributes are used to specify the text properties mentioned above. Below is an example layout file `player_widget.xml` from the app *AntennaPod* [6], which defines an image button widget with two associated text properties.

```
1. <ImageButton android:id="@+id/butPlay"
2.   android:src="@drawable/ic_play_arrow_white_24dp"
3.   android:contentDescription="@string/play_label"/>
```

In some situations, developers do not directly bind a fixed image to an icon widget. The reason is that the app needs to adapt its GUI appearances to different styles or themes. In this case, developers usually assign a reference value to the `android:src` attribute [5]. Depending on the configurations, the app can select a suitable image as the widget’s icon. To ease understanding, we provide a code snippet from `mediaplayerinfo_activity.xml` in the app *AntennaPod* below. In this case, ICONSEER will extract the value of `android:src` by retrieving the `av_play` item name in the `styles.xml` file.

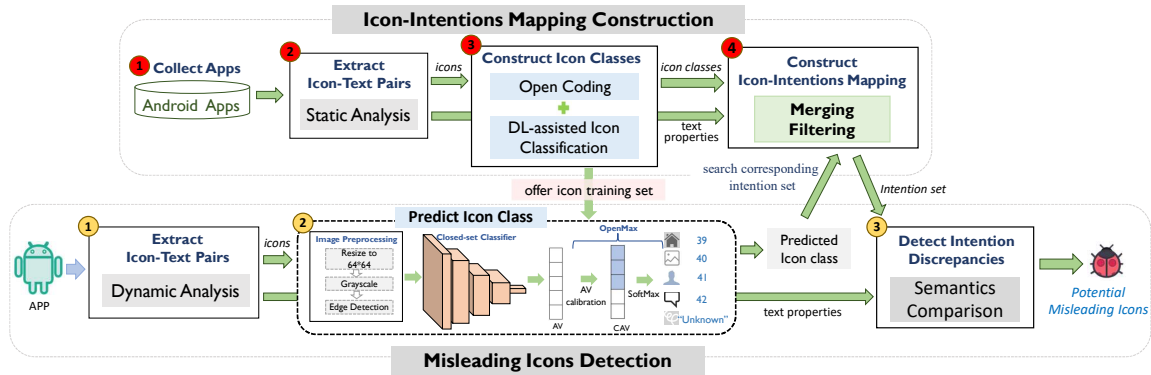


Figure 3: Overview of the ICONSEER framework

```

1. <ImageButton android:id="@+id/butPlay"
2.   android:src="@attr/av_play"
3.   android:contentDescription="@string/pause_label"/>
    
```

```

styles.xml
<resources>
<style name="Theme.Base.AntennaPod.Dark">
  <item name="av_play">@drawable/play_arrow_white</item>
</style>
...
<style name="Theme.Base.AntennaPod.Light">
  <item name="av_play">@drawable/play_arrow_grey600</item>
</style>
</resources>
    
```

For the above two cases, ICONSEER extracts triples in the form of $\langle image, id, content_description \rangle$ from app layout files.

• **Static analysis on program code.** In Android framework, developers can bind image resources (i.e., icons) and corresponding text properties to icon widgets programmatically. There are three APIs widely used: `setImageResource()`, `setImageDrawable()`, and `setImageBitmap()`. For example, the code below binds the icon with the resource ID `R.drawable.imgIcon` to the widget object `img`. To deal with such cases, ICONSEER performs backward slicing on the arguments of these API methods to obtain their definition statements. With the actual values of these arguments (e.g., the `R.drawable.imgIcon`), ICONSEER can then retrieve the icons from the app’s resource directory (e.g., `res/drawable/imgIcon.png`).

```

1. Integer resource = R.drawable.imgIcon; // R$drawable.imgIcon
2. img.setImageResource(resource);
    
```

There are two ways to identify the `id` attribute of an icon widget. Similar to identifying image resources, ICONSEER uses Soot to obtain the argument of the API `findViewById()` or `setId()`. As the first code snippet below shown, the string with the resource ID `R.id.imgIcon` is the `id` attribute of the widget object `img`. In the second code snippet, an `ImageView` object `img` is created and then bound with a unique `id` string resource by calling the `setId()` API.

```

1. Integer resource = R.id.imgIcon; //R$id.imgIcon
2. ImageView img = (ImageView)findViewById(resource);
    
```

```

1. ImageView img = new ImageView();
2. Integer id = R.id.imgIcon; //R$id.imgIcon
3. img.setId(id);
    
```

Developers invoke the API method `setContentDescription()` to set the `content_description` attribute of an icon widget, as

shown in the code below. For such cases, ICONSEER can also obtain the attribute `content_description` via static analysis, which is similar to the way it extracts the image resources or IDs.

```

1. String contentDesc = R.string.imgIcon; //R$string.imgIcon
2. img.setContentDescription(resource);
    
```

It is essential to perform object-sensitive analysis when extracting an icon and its corresponding text properties. For example, in the above code snippets, we need to make sure that the `img` variable should point to the same object, so that the icon and the texts can be paired. Since these API invocations only consist of a few statements, to match the objects, we perform an intra-procedural analysis on the static single assignment (SSA) form of the caller’s method body. In Soot, this intermediate representation corresponds to the Shimble code, in which an identifier can only be assigned once, thus the same identifier always points to the same object. In this way, we can extract $\langle image, id, content_description \rangle$ triples.

During analysis, we only kept the triples (obtained from both the layouts and the code) whose image can satisfy the definition of icons mentioned in Section 3. As we could not obtain the screen size through static analysis, we limited the length and width of the images to less than 300 pixels rather than setting limits on their occupied areas. With these restrictions, we extracted 277,395 distinct triples, in the form of $\langle icon, id, content_description \rangle$. Then, we removed noises from the dataset. First, we removed 9,042 triples containing icons with pure color or with text. The reason and processing are same as that in Section 3. Then, we removed meaningless text properties for each remaining triple. Finally, we obtained 268,353 triples, of which 237,993 ones contain at least one text property (232,960 triples contain non-empty `id` and 31,628 triples contain non-empty `content_description`).

4.1.3 Icon Classes Construction. In the previous step, we obtained a total of 268,353 icons. Intuitively, these icons are not unique. They should belong to some groups (i.e., icon classes) that share similar action semantics. Thus, the goal of this step is to classify the collected icons into a smaller number of groups. Below, we introduce how we determined the number of icon classes and how we trained deep neural networks to automatically classify icon.

• **Open coding for determining icon classes.** Firstly, we should identify the appropriate classes for the 268,353 extracted icons. Two prior studies [29, 39] have constructed icon classes by open

coding [22]. However, they only presented 80 and 40 icon classes, respectively, which cannot cover many icons in our dataset. Given that the existing classifications are insufficient and manually inspecting all collected icons are time-consuming, we performed an open coding process on a subset of icons to determine the appropriate icon classes. To ensure a 99% confidence level with a 1% confidence interval, we randomly sampled 26,835 (10%) of our collected icons to decide the icon classes.

The open coding process is as follows. First, we merged the icon classes of two prior works [29, 39], which resulted in 92 unique classes (called “initial classes”). Then, for each icon, three authors (annotators) independently inspected it and determined which initial class it should belong to. The belonging class would be confirmed if two or more authors could reach a consensus. After that, 16,291 (60.7%) icons were successfully classified into the initial classes. For the remaining unclassified icons, we conducted an iterative process of creating, adjusting, and removing icon classes, until all the annotators had no conflicting opinions. Moreover, if an icon class contained too many icons, we would further categorize its containing icons into smaller classes. For example, the class “emoji” was split into three sub-classes: “smile”, “sad”, and “others”. At the end, 179 icon classes were finally identified.

• **Deep learning-assisted icon classification.** After identifying the icon classes, our next step is to train a classifier for classifying all the remaining icons. We used the 26,835 icons with manually labeled classes to train and evaluate icon classifiers (90% for training and 10% for testing). We considered four state-of-the-art convolutional neural network (CNN) models for this task: ResNet50 [31], DeseNet161, DenseNet201 [32] and, EfficientNetB7 [49].

Before training classifiers, we performed image preprocessing. First, we resized the icons in the data set to 64×64 pixels. Second, we converted the icons to Grayscale³ to eliminate the effect of colors. Then, we enhanced the edges of the icons using canny edge detection [8]. The values of the parameters *minVal* and *maxVal* in edge detection algorithm were set to 10 and 40, respectively.

After preprocessing, we fine tuned these models by replacing their classification layers with our self-defined head model and trained the re-structured models on our dataset. The head model contains three fully connected layers: *Linear*(*numFeatures*, 512), *Linear*(512, 256), and *Linear*(256, 179). The two numbers in () indicate the numbers of the input and output dimensions of the layer, and *numFeatures* is same as the number of input dimensions of models’ original classification layer. During training, the weights in the body of the original models were frozen, that is, we only trained the weights of self-defined three linear layers on the dataset. The *epochs* and *Batch Size* were set to 50 and 64. We chose Adam [33] as the optimizer and set the learning rate to 0.001. We used the cross-entropy loss [11] as the loss function and F1-score (weighted) to measure the performance of the models. The results showed that DenseNet201 outperformed others, with an F1-score of 0.8001.

As the dataset to be classified is an open one [48, 52], containing icons from “unknown” classes (i.e., not all icons belong to the 179 icon classes), it is not suitable to directly apply the trained closed-set DenseNet201 to classify all the icons. In the trained model

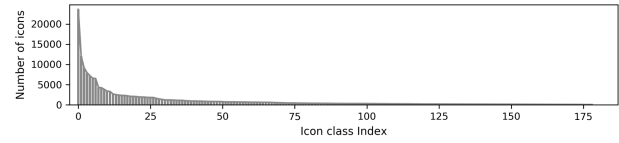


Figure 4: Distribution of the size for the 179 icon classes

DenseNet201, the output of the fully-connected layer (penultimate layer) will be fed to the SoftMax layer, which will produce the probability for each class. The ideal result for an “unknown” image is that all existing classes would have low probabilities. However, a deep network often likes to assign a most-likely existing class for an “unknown” image [48, 52]. OpenMax [24], proposed by Bendale and Boulton, is a classical solution towards this problem. It is designed to adjust the *activation vector* (AV), i.e., the output of the penultimate layer. To apply OpenMax, it is recommended to use grid search to select its hyper-parameters η , α , ϵ , using a set of training images plus a sampling of open set images, such that the F1-score can be optimized over the set [24]. We constructed the dataset by combining the 2,415 icons (10%) sampled from the training dataset and 200 “unknown” icons. We did a grid search on the dataset to select the three parameters. We searched η from 1 to 29, with a step size of 2; α from 1 to 30, with a step size of 1; ϵ from 0 to 1, with a step size of 0.05. Among the 9,000 combinations of the three parameters, we selected the combination that can lead the models to obtain the highest F1-score. The best combination was (5, 10, 0.5).

During icon class prediction, the output of DenseNet201 was fed to the OpenMax layer to classify the icon into one of 179 classes or “unknown”. We used the open-set classifier to classify the 241,518 (=268,353-26,835) icons without labels. There were 60,858 icons being predicted as “unknown”, and the other 180,660 icons were classified into one of the 179 classes. To validate the result, we manually checked the predicted class of the 180,660 icons, removed mis-classified icons from each icon class. Here, 21,515 mis-classified icons were removed, resulting in 159,145 icons with ground truth labels. We combined these correctly classified icons with the 26,835 manually classified icons to obtain a dataset of 185,980 icons. Figure 4 shows that the distribution of the size for the 179 icon classes. The final icon classes in the dataset exhibit an imbalanced distribution, with a maximum size of 23,619 for the class containing icons similar to “✘”, and a minimum size of 56 for the class containing icons similar to “🚲”. The average class size is 912, the median is 336. There are 165 (92%) classes containing more than 100 icons.

4.1.4 *Icon-Intentions Mapping Construction.* To extract textual descriptions of icon intention for a particular class, we gathered all the text properties of icons with the same class label. For each text property in a set, we converted it into lowercase and tokenized it with common text delimiters (e.g., empty space, underscore, and camel case). In the resulting set of words (i.e., document consisting of terms), we calculated term frequency-inverse document frequency (TF-IDF) for each term. We kept only the terms with a TF-IDF score greater than 0.005 for each document. Across all 179 documents, a total of 10,094 terms remained, of which 3,204 were unique. For each icon class, three authors checked the corresponding words and discussed whether they could be commonly perceived intentions for the icons in the class. After several rounds of checks and

³We compared the performances of classifiers when trained on gray images and colored images and found that the former way can achieve a better performance.

Table 3: Examples of icon-intentions mapping

No.	Example Icons	Intentions
8		select, submit, right, checked, check, done, success, choose, tick, confirm,...
19		rating, rate, like, save, recommend, favor, star, ...
24		more, menu, category, dashboard, grid, kits, ...
41		settings, sets, filter, adjuster, adjust, config, equalizer, ...
63		refresh, repeat, reset, retry, playback, update, reload, sync, spin, rotate, ...

discussions, we finally obtained 1,425 intentions for 179 classes, among which there are 852 unique ones. Table 3 shows part of the mapping. The whole mapping can be found on our website [15].

4.2 Misleading Icons Detection

We will introduce the process for detecting misleading icons below.

4.2.1 Icon-Text Pairs Extraction. First, ICONSEER performs dynamic analysis to extract icons and the three text properties of their corresponding widgets from the app under test. It extracts unique quadruples (*icon*, *content_description*, *id*, *icon-surrounding-text*) from the screenshots and GUI hierarchies collected by DroidBot and then performs data cleaning. The process is the same as that in Section 3.

4.2.2 Icon Class Prediction. In the previous section, ICONSEER obtains 179 classes of icons and maps each icon class to a set of possible intentions. In this section, it classifies each icon under analysis by a pretrained classifier to obtain the icon’s commonly perceived intentions to be further compared with the three text properties. Before classification, ICONSEER preprocesses the icon under analysis by resizing, grayscaling, and edge enhancement, which is same as that introduced in Section 4.1.3. To do classification, ICONSEER leverages DenseNet161 after comparing its performances with other state-of-the-art models, including ResNet50, DeseNet201 and EfficientNetB7. The classifier was trained and tested using the 179 classes of icons obtained earlier (see Section 4.1.3). Since the 179 icon classes have imbalanced sizes (Figure 4), we first performed downsampling on classes over 500 icons to reduce the class size to 500. After such processing, the total number of icons was reduced to 57,991. As the trained icon classifier will be evaluated on the dataset with ground truth built in our pilot study, we further excluded 197 icons, which are also in the evaluation dataset, to prevent data leak. After this, 57,794 icons were left. We then trained the classifier on 90% of the icons (52,015) and used the remaining 10% (5,779) as a hold-out test set. The training process and parameters setting are the same as that in Section 4.1.3. The output of trained DenseNet161 was fed to an OpenMax layer, whose three hyper-parameters η , α , ϵ were set to 5, 15, 0.5, identified follow a same process in Section 4.1.3.

4.2.3 Intention Discrepancies Detection. If an icon is classified to the “unknown” class, ICONSEER will not further process it. Otherwise, ICONSEER will proceed to check whether the semantic intentions of the icon are inconsistent with all the three text properties. Algorithm 1 describes the semantic similarity calculation process. It takes two inputs: (1) an icon’s intentions set and (2) the three

Algorithm 1: Calculating the similarity score between an icon’s intentions set and three text properties

Input: Semantic intentions set *intens*; Text properties set *text_props*

Output: Similarity score *sim_score*

```

1 Initiate list text_sims
2 for text_prop in text_props do
3   for inten in intens do
4     if text_prop.Contains(inten) then
5       text_sim ← 1
6     else
7       text_prop_words ← PreProcess(text_prop)
8       inten_words ← PreProcess(inten)
9       text_prop_vec ←
10        Avg(GloveWordEmbedding(text_prop_words))
11        inten_vec ← Avg(GloveWordEmbedding(inten_words))
12        text_sim ← CosineSim(text_prop_vec, inten_vec)
13      text_sims.Add(text_sim)
14 sim_score ← Max(text_sims)
15 return sim_score

```

text properties of the widget. It outputs a similarity score between them. The algorithm begins by initializing a list *text_sims* to store the similarities (Line 1). For each text property *text_prop* and each icon intention *inten*, it checks if *text_prop* contains *inten* (Line 4). If so, *text_sim* will be directly assigned with 1 (Line 5). Otherwise, the similarity will be calculated using text embedding technique. First, *text_prop* and *inten* are preprocessed (Lines 7-8). The preprocessing contains four steps: punctuation removal by *nlTK* [17], word segmentation by *wordninja* [21], lowercasing, lemmatization by *nlTK*. The preprocessing will return two lists of words: *text_prop_words* and *inten_words*. Then, *GloveWordEmbedding()* will be called to obtain the list of vectors corresponding to the words in *text_prop_words* (Line 9) and *inten_words* (Line 10). *GloveWordEmbedding()* leverages word vectors, pretrained by *Glove* [45], to embed a given word. The two lists of vectors obtained by word embedding will be averaged and assigned to *text_prop_vec* and *inten_vec*. Then, cosine similarity between the two vectors will be calculated (Line 11) and added to the list *text_sims* (Line 13). When the nested loop finishes, *text_sims* between each *text_prop* and *inten* have been added to *text_sims*. Finally, the max value of *text_sims* will be returned as *sim_score* (Lines 13-14). If *sim_score* < 0.9, ICONSEER will consider that the icon has semantic inconsistencies with the three text properties, and report it as an suspicious misleading icons.

5 EVALUATION

The evaluation aims to investigate the following research questions:

- **RQ1 (Classifier Performance):** What is the performance of *ICONSEER*’s open-set icon classifier?
- **RQ2 (Mapping Comprehensiveness):** How does *ICONSEER*’s icon-intentions mapping compared with existing mappings?
- **RQ3 (Discrepancy Detection Capability):** Can *ICONSEER* effectively detect discrepancies between icons and the three text properties of their corresponding widgets?
- **RQ4 (Misleading Icon Detection Capability):** Can *ICONSEER* identify real misleading icons through detecting the discrepancies?

Table 4: The performances of different open-set classifiers

Models \ Metrics	Performance Metrics (Closed-Set Classifiers)							OpenMax Parameters			Performance Metrics (Open-Set Classifiers)			
	Top-1 Acc.	Top-3 Acc.	Top-5 Acc.	Top-10 Acc.	Precision	Recall	F1-score	η	α	ϵ	Accuracy	Precision	Recall	F1-score
ResNet50	0.8618	0.8924	0.8993	0.9155	0.9107	0.8618	0.8796	5	10	0.6	0.7849	0.8629	0.7849	0.7986
DenseNet161	0.8699	0.8981	0.9077	0.9181	0.9205	0.8699	0.8884	5	15	0.5	0.8395	0.8783	0.8395	0.8519
DenseNet201	0.8696	0.9000	0.9077	0.9186	0.9193	0.8696	0.8872	5	24	0.5	0.8370	0.8754	0.8370	0.8491
EfficientNetB7	0.8582	0.8965	0.9077	0.9183	0.9080	0.8582	0.8766	5	10	0.5	0.8163	0.8615	0.8163	0.8298

Table 5: The mapping comparison results

Mapping	Data Source	#Apps	#Icons	#Known Icon Classes	#Known Intentions
Liu et al. [39]	Google Play	~9.7k	73,449	80	320
Auto-Icon [29]	Iconfont	-	~100k	40	184
ICONSEER	Google Play	15,571	268,353	179	852

5.1 RQ1: Classifier Performance

The open-set icon classifier is divided into two components: the closed-set classifier and an extra OpenMax layer. To answer RQ1, we first evaluated the performance of the closed-set classifier; then, we conducted experiments to determine the parameters of the OpenMax layer; finally, we evaluated the performance of the open-set classifier as a whole. As introduced in Section 4.2.2, we used DenseNet161 as our closed-set classifier in the detection module of ICONSEER. Thus, we compared its performance with the other three CNN models: ResNet50 [31], DenseNet201 [32], and EfficientNetB7 [49] in our experiments.

5.1.1 Performance of the Closed-Set Icon Classifier. We compared the performances of the four fine-tuned models for closed-set classification. We used the dataset of 57,794 icons constructed in Section 4.2.2 to train and evaluate the classifiers (90% for training and 10% for testing). The image preprocessing, training process and parameters setting are the same as that in Section 4.1.3. For this multiclass classification task, we used accuracy, precision (weighted), recall (weighted), and F1-score (weighted) to measure the performance of the four models. Besides, as there are many icon classes and some icons from different classes have high similarities. To fairly compare the models' performances, we also used the top-3, top-5 and top-10 accuracy to evaluate and compare them. The top-3 accuracy (top-5 accuracy/top-10 accuracy) is the percentage of test icons whose right label is among the predicted top three (five/ten) labels. The metrics used here are standard ones for evaluating the performance of multi-class classifications [30]. Table 4 shows the performance of the four models in terms of seven metrics. DenseNet161 shows the best F1-score, 0.8884, compared with others.

5.1.2 Performance of the Open-Set Icon Classifier. We identified the three parameters of OpenMax, namely, η , α , ϵ , by performing a grid search of 9,000 parameter combinations using the dataset containing 10% training data (5,201) and 500 "unknown" icons (see Section 4.2.2 for more details). The best parameter combinations of the four models are shown in Table 4. We constructed the test dataset of the open-set classifier by combining the 5,779 icons from the test dataset of closed-set classifier and 500 newly sampled "unknown" icons. For the four trained models, the output of their

penultimate layer will be passed to the OpenMax layer to get the scores for each class. The models' performances are shown in Table 4. DenseNet161 performs best with the OpenMax parameters (5, 15, 0.5), achieving an F1-score of 0.8519.

Answer to RQ1: The performance ICONSEER's open-set icon classifier, which is based on DenseNet161, is better than that of other alternatives.

5.2 RQ2: Mapping Comprehensiveness

There are two recent studies that have also built the icon-intentions mapping: (1) the work of Liu et al. [39] and (2) Auto-Icon [29]. Both of them built the mapping on self-constructed dataset following an open coding process. We compared our mapping with the two existing mappings from five perspectives as shown in Table 5. We discuss the comparison results below.

- **Data source:** Liu et al. collected apps from Google Play to build the icon-intentions mapping, which is the same as our work. This ensures that the extracted icons are really used by real-world apps. Comparatively, Auto-Icon collected icons from Iconfont [14], a popular icon website, instead of real apps.
- **Number of apps:** We collected 15,571 apps to construct the mapping. For Liu et al.'s work, the number is 9.7k.
- **Number of icons, icon classes, and intentions:** Liu et al.'s mapping contains 73,449 icons from 80 different classes and mapped these icon classes to 320 different intentions. Auto-Icon contains 100k icons from 40 different classes and mapped the icons to 184 different intentions. Compared with the two prior works⁴, our mapping is much more comprehensive. As presented in Section 4.1.3, we successfully assigned 185,980 icons to 179 classes. Among the 185,980 icons, 30,320 (16.30%) cannot be covered by Liu et al.'s mapping, i.e., they do not belong to any of the 80 icon classes mentioned in Liu et al.'s paper, and 78,070 (41.98%) cannot be covered by Auto-Icon. We also mapped the 179 classes to 852 unique intentions, which is significantly more than that of the existing mappings.

Answer to RQ2: The icon-intentions mapping constructed by us is so far the most comprehensive one.

5.3 RQ3: Discrepancy Detection of ICONSEER

In the pilot study, we manually checked 9,075 icons from 1,111 apps and found that 1,576 icons are semantically inconsistent with their corresponding three text properties. In this section, we take the 9,075 icons as the test dataset to evaluate the effectiveness of

⁴The papers only reported partial mappings. We emailed the authors for complete mappings and full datasets, but did not receive responses. The statistics reported in our paper are based on the available data, i.e., classes and intentions listed in the papers.

Table 6: The discrepancies detection results of ICONSEER

Settings	#Discrepancies Reported	Accuracy	Precision	Recall	F1-score
Top-1	2,327	0.8432	0.5329	0.7868	0.6354
Top-3	2,006	0.8636	0.5842	0.7437	0.6544
Top-5	1,862	0.8664	0.5977	0.7062	0.6475
Top-10	1,621	0.8705	0.6237	0.6415	0.6325

Table 7: The misleading icons detection results of ICONSEER

Settings	#Real Discrepancies	#Real Misleading Icons	#Wrong Text
Top-1	1,240	508 (40.97%)	732 (59.03%)
Top-3	1,172	482 (41.13%)	690 (58.87%)
Top-5	1,113	459 (41.24%)	654 (58.76%)
Top-10	1,011	421 (41.64%)	590 (58.36%)

ICONSEER for detecting the semantic inconsistencies between icons and three text properties.

ICONSEER leverages DenseNet161 as the base model for open-set icon classification. An icon’s true label may not be the class with the highest output score of OpenMax, but can be one of the top several predicted classes. To reduce false positives, we configure the classifier to return more than one label for an icon, and then the icon’s intention set is constructed by computing the union of the intentions corresponding to all returned labels. In our experiments, the number of returned labels (*numRtnCls*) of the classifier was set to 1, 3, 5, and 10, separately. We compared the discrepancy detection performances of ICONSEER under these different settings of *numRtnCls*. The results are shown in Table 6⁵. When *numRtnCls* is set to 3, the best F1-score, 0.6544, can be achieved. The best accuracy is 0.8705 when *numRtnCls* is set to 10.

Answer to RQ3: ICONSEER can achieve the best performance (in terms of F1-score) of detecting discrepancies when its open-set classifier is configured to output three predicted classes.

5.4 RQ4: Misleading Icon Detection of ICONSEER

To answer RQ4, we manually checked the discrepancies successfully identified by ICONSEER to verify whether the icon is indeed misleading or not. The results are given in Table 7. As we can see, under different settings of *numRtnCls*, we found different numbers of real misleading icons from the detected discrepancies. When *numRtnCls* was set to one, ICONSEER detected the largest number (508) of real misleading icons, accounting for 40.97% of all the detected 1,240 discrepancies. When *numRtnCls* was set to larger numbers, the number of detected real misleading icons slightly decreased. This is because when *numRtnCls* gets larger, the number of inferred intentions for an icon under test also gets larger, and then the identified discrepancies will be fewer (with more intentions, the probability that the three text properties semantically match the intentions gets higher). Another observation is that many discrepancies are in fact not caused by misleading icons. The reason is that developers

⁵The metrics in the table are standard ones for evaluating the performance of binary classifications [30].

have specified inaccurate or wrong text properties (the statistics are provided in the last column of Table 7).

Answer to RQ4: ICONSEER can successfully detect real misleading icons. Over 40% detected discrepancies are caused by misleading icons. This shows that ICONSEER has good potential to help Android developers or app companies to find misleading icons in their apps.

6 DISCUSSIONS AND FUTURE WORK

Misleading icons identification. In the pilot study, we used icon widgets’ text properties, which typically describe the widgets’ functionality, to pinpoint misleading icons. With this process, we may miss some real misleading icons, which do not have text properties. We chose to use text properties because: (1) we found that 89% icon widgets contain non-empty text properties; (2) many commercial apps adopt code obfuscation/hardening techniques for intellectual property protection, it is hard to precisely analyze the functionality of icon widgets for such apps via code analysis. In other words, it is hard to obtain the perfect ground truth for a large set of commercial apps but leveraging text properties is a feasible way to achieve the goal. In the future, we will combine automatic and manual analysis, to construct a better misleading icons dataset.

Dynamically icon extraction. Dynamic analysis only worked on partial apps during data collection of the pilot study. The main reasons are as follows: (1) the top commercial apps often adopt well-developed mechanisms for code protection and emulator prevention; (2) many apps have a login page that requires the username and password, which could not be skipped by the dynamic analysis tool. However, if ICONSEER is used by developers during testing, they can provide unprotected test versions of their apps and help ICONSEER log into the apps. In such scenarios, the success rate of icon extraction can be largely improved.

False positives and negatives of ICONSEER. We observed three causes of false positives and negatives. First, some developers may not specify precise text properties of icons, bringing false positives. Besides, not all icons have the text properties, resulting in false negatives. In the future, we plan to consider more features (e.g., code-level features such as API calls and permissions) of icon widgets and address the technical challenges of leveraging such features to improve ICONSEER. Second, the icon-intentions mapping we constructed is incomplete. In some situations, icon widgets may have very specific functionalities. If such functionalities are not covered by our mapping, ICONSEER may report false positives. Besides, given a rarely used icon, ICONSEER may not be able to find similar ones from our constructed icon classes. In such cases, ICONSEER may fail to detect real misleading icons (i.e., false negatives). In the future, we plan to further improve our mapping. Third, open-set classification is a challenging problem and currently there are no perfect algorithms. ICONSEER may report false positives if an “unknown” icon is wrongly classified by our adopted open-set classifier to a known class. Whereas, an icon of a known class being wrongly classified as “unknown” will cause false negatives. In the future, we will try more state-of-the-art open-set algorithms.

The reported issues are identified manually. The purpose of our pilot study is to investigate the feasibility of our approach and

build a dataset with ground truth to evaluate our tool. Although the reported issues are identified manually, many of them can indeed be detected by our tool. In our future work, we will apply ICONSEER to a wider range of apps to further evaluate its detection capability.

Practical usefulness of ICONSEER. In Section 3, we studied developers' actions to misleading icons. We found that the majority of our detected misleading icons have not been fixed. One possible reason is that developers are not aware of such issues. This is understandable because spotting misleading icons during app testing may not be an easy task. It requires testers to have a good understanding of the commonly perceived intentions of each icon and the actual functionality of the icon widget. Due to the variety and multiple intentions of icons, it can be hard for testers to get a complete picture of the common intentions of icons (in our dataset, many icons have more than 10 common intentions). This might be a reason why misleading icons are widespread in real-world apps. ICONSEER can be used by developers/testers to help identify the misleading icons in their apps during testing. It can also be used by app markets to assess the UI design of apps and supervise developers to improve their products. In order to have a deeper understanding of the usefulness of ICONSEER, in the future, we plan to conduct surveys and interviews to further investigate the testers' practices, difficulties of identifying misleading icons, and their perceptions on misleading icon detection tools.

7 RELATED WORK

Icon semantics inference. Many works have been made for inferring the underlining semantics of icons in mobile apps' GUIs. Auto-Icon [29] (and its extended version Auto-Icon+ [28]) applies computer vision methods to generate typeface font for an icon and generates descriptive labels for it with CNN. Their work simplifies UI development by reducing manual effort in app prototyping. Liu et al. [39] also adopts text properties, like resource ID, to denote the intentions of icons. Chen et al. [26] developed LabelDroid, a CNN- and RNN-based model, to predict missing text labels for icons. These labels can be used by screen readers to improve accessibility for visually impaired users. Mehralian et al. [42] re-evaluated LabelDroid and found that its effectiveness heavily relies on the imbalanced dataset, which introduces a bias in predicting labels for uncommon icons. They proposed to consider more sources of information for predicting context-aware icon labels. The semantics of icons can also raise security concerns. For instance, Xiao et al. proposed IconIntent [54] to detect mismatches between eight icon categories that demand sensitive data and their required permissions. Unlike IconIntent's mapping-based icon classification, DeepIntent [53] uses deep neural networks for classifying icons and performs the same task as IconIntent. IconChecker [35] can detect unusual network traffic resulting from button clicks that do not match the user's intention due to the button's icon. It can identify such malicious behaviors from eight classes of non-traffic-triggering icons. In ICONSEER's framework, we constructed 179 icon classes based on 40 classes from Auto-Icon [29] and the 80 most frequent classes from [39].

Icon design analysis. Researchers have studied icon design factors that affect app users' experiences. Lu et al. [40] discovered that icon lightness impacts users' visual de-confusion. Similarly, Smythwood

et al. [47] showed that the visual complexity, compared to aesthetic appeal, of icons is a primary factor that affects user's icon search speed on the GUI. Luo et al. [41] found two factors of icon shape that affect visual cognition and created a formula to measure visual complexity in icons. There were also studies investigated how icons affect user preference towards the app. Wang et al. [50] found that icon appearance can influence user's download decision on Android market. Lin et al. [37] also explored how different icon compositions and backgrounds combinations could affect users' preference levels. Through controlled user experiment, they found that the best mode is plane-composed icon displaying on negative background. The above studies mostly focus on the impact of design on apps and aimed to give suggestions to app developers and UI designers. Considering that misleading icons can cause poor user experiences, we proposed ICONSEER to detect such bad designs.

UI design violation detection. Bad GUI design and implementation not only appear on interactive widgets, but also on the whole layouts. Yang et al. [55] built a gallery with real-world bad UI designs against the don't-guidelines provided by Material Design. Also, they proposed a detector, UIS-Hunter, to validate multi-modal UI information (metadata, typography, color, etc.) and detect the violation of these design guidelines. Zhao et al. [56] proposed an unsupervised and CV-based adversarial autoencoder to detect the violations of ten GUI animations design guidelines. Besides, app developers often implement their GUI programs according to a static image depicting the layout and style of widgets. However, there may be gaps between the GUI design and its real implementation. To address this problem, Moran et al. [43] introduced an automated approach that compares GUI-related information from an implemented apps and the initial design image to verify whether the GUI of the app was implemented according to the intended design. These studies aim to detect the bad design and implementation of UI elements based on the certain guidelines. ICONSEER, however, concerns whether an icon's commonly perceived intentions mismatch the actual functionality of its belonging widget.

8 CONCLUSION

In this paper, we studied the misleading icon issues in Android apps. We found that these issues are prevalent in popular apps. We further proposed the first framework, ICONSEER, to automatically detect misleading icon issues by using deep learning and natural language processing techniques. Experiments on real-world apps show that the framework can achieve good performance.

9 DATA AVAILABILITY

We have released our mapping, experiment data, source code and models at our project website [15].

ACKNOWLEDGMENTS

The authors would like to thank ISSTA 2023 reviewers. This work is supported by the Guangdong Basic and Applied Basic Research Fund (Grant No. 2021A1515011562), the National Natural Science Foundation of China (Grant Nos. 61932021, 62141210, 61802164), the Natural Science Foundation of Guangdong Province (Grant No. 2023A1515011959), and the Fundamental Research Funds for the Central Universities (Grant No. N2217005).

REFERENCES

- [1] 2019. Android Accessibility Guideline. <https://developer.android.com/guide/>
- [2] 2022. 10times. <https://play.google.com/store/apps/details?id=com.tentimes>
- [3] 2022. ABC. <https://play.google.com/store/apps/details?id=com.disney.datg.videoplatforms.android.abc>
- [4] 2022. Accessibility. <https://developer.android.com/guide/topics/ui/accessibility/apps>
- [5] 2022. Android Styles and Themes. <https://developer.android.com/develop/ui/views/theming/themes>
- [6] 2022. AntennaPod. <https://play.google.com/store/apps/details?id=de.danoeh.antennapod>
- [7] 2022. BabyCam. <https://play.google.com/store/apps/details?id=com.arjonasoftware.babycam>
- [8] 2022. Canny edge detector. https://en.wikipedia.org/wiki/Canny_edge_detector
- [9] 2022. Content Labels. <https://support.google.com/accessibility/android/answer/7158690>
- [10] 2022. Cool DJ Club Theme. <https://play.google.com/store/apps/details?id=com.ikkeyboard.theme.cool.dj.club>
- [11] 2022. Cross-entropy loss. <https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html>
- [12] 2022. HD Wallpapers. <https://play.google.com/store/apps/details?id=info.androidstation.hdwallpaper>
- [13] 2022. HP Print Service Plugin. <https://play.google.com/store/apps/details?id=com.hp.android.printservice>
- [14] 2022. IconFont. <https://www.iconfont.cn/>
- [15] 2022. IconSeer. <https://sites.google.com/view/iconseer>
- [16] 2022. Layouts. <https://developer.android.com/develop/ui/views/layout/declaring-layout>
- [17] 2022. nltk. <https://www.nltk.org/>
- [18] 2022. PaddleOCR. <https://github.com/PaddlePaddle/PaddleOCR>
- [19] 2022. Paint and Drawing Fun. <https://play.google.com/store/apps/details?id=com.kidspaint.kaushalmehra.drawingfun>
- [20] 2022. Speak and Translate Languages. <https://play.google.com/store/apps/details?id=com.speakandtranslate.voicetranslator.alllanguages>
- [21] 2022. wordninja. <https://github.com/keredson/wordninja>
- [22] Ian Alexander. 2000. An introduction to qualitative research. *Eur. J. Inf. Syst.* 9, 2 (2000), 127–128. <https://doi.org/10.1057/palgrave.ejis.3000350>
- [23] Abdulaziz Alshayban, Iftekar Ahmed, and Sam Malek. 2020. Accessibility Issues in Android Apps: State of Affairs, Sentiments, and Ways Forward. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 1323–1334.
- [24] Abhijit Bendale and Terrance E. Boult. 2016. Towards Open Set Deep Networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27–30, 2016*. IEEE Computer Society, 1563–1572. <https://doi.org/10.1109/CVPR.2016.173>
- [25] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. 2018. From UI design image to GUI skeleton: a neural machine translator to bootstrap mobile GUI implementation. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 665–676. <https://doi.org/10.1145/3180155.3180240>
- [26] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. 2020. Unblind your apps: predicting natural-language labels for mobile GUI components by deep learning. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 322–334. <https://doi.org/10.1145/3377811.3380327>
- [27] Sen Chen, Chunyang Chen, Lingling Fan, Mingming Fan, Xian Zhan, and Yang Liu. 2021. Accessible or Not An Empirical Investigation of Android App Accessibility. *IEEE Transactions on Software Engineering* (2021), 1–1. <https://doi.org/10.1109/TSE.2021.3108162>
- [28] Sidong Feng, Minmin Jiang, Tingting Zhou, Yankun Zhen, and Chunyang Chen. 2022. Auto-Icon+: An Automated End-to-End Code Generation Tool for Icon Designs in UI Development. *ACM Trans. Interact. Intell. Syst.* (apr 2022). <https://doi.org/10.1145/3531065> Just Accepted.
- [29] Sidong Feng, Suyu Ma, Jinzhong Yu, Chunyang Chen, Tingting Zhou, and Yankun Zhen. 2021. Auto-Icon: An Automated Code Generation Tool for Icon Designs Assisting in UI Development. In *IUI '21: 26th International Conference on Intelligent User Interfaces, College Station, TX, USA, April 13–17, 2021*, Tracy Hammond, Katrien Verbert, Dennis Parra, Bart P. Knijnenburg, John O'Donovan, and Paul Teale (Eds.). ACM, 59–69. <https://doi.org/10.1145/3397481.3450671>
- [30] Margherita Grandini, Enrico Bagli, and Giorgio Visani. 2020. Metrics for Multi-Class Classification: an Overview. *CoRR abs/2008.05756* (2020). <https://arxiv.org/abs/2008.05756>
- [31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27–30, 2016*. IEEE Computer Society, 770–778. <https://doi.org/10.1109/CVPR.2016.90>
- [32] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. 2016. Densely Connected Convolutional Networks. *CoRR abs/1608.06993* (2016). [arXiv:1608.06993](https://arxiv.org/abs/1608.06993) <http://arxiv.org/abs/1608.06993>
- [33] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7–9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). <http://arxiv.org/abs/1412.6980>
- [34] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3–6, 2012, Lake Tahoe, Nevada, United States*, Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, Léon Bottou, and Kilian Q. Weinberger (Eds.), 1106–1114. <https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html>
- [35] Yuxuan Li, Ruitao Feng, Sen Chen, Qianyu Guo, Lingling Fan, and Xiaohong Li. 2021. IconChecker: Anomaly Detection of Icon-Behaviors for Android Apps. In *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*. 202–212. <https://doi.org/10.1109/APSEC53868.2021.00028>
- [36] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. DroidBot: a lightweight UI-guided test input generator for Android. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20–28, 2017 - Companion Volume*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE Computer Society, 23–26. <https://doi.org/10.1109/ICSE-C.2017.8>
- [37] Hsuan Lin, Yu-Chen Hsieh, and Wei Lin. 2016. A Preliminary Study on How the Icon Composition and Background of Graphical Icons Affect Users' Preference Levels. In *Human Aspects of IT for the Aged Population. Design for Aging - Second International Conference, ITAP 2016, Held as Part of HCI International 2016, Toronto, ON, Canada, July 17–22, 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9754)*, Jia Zhou and Gavriel Salvendy (Eds.). Springer, 360–370. https://doi.org/10.1007/978-3-319-39943-0_35
- [38] Hsuan Lin, Yu-Chen Hsieh, and Wei Lin. 2017. Shape Design and Exploration of 2D and 3D Graphical Icons. In *Human Aspects of IT for the Aged Population. Applications, Services and Contexts - Third International Conference, ITAP 2017, Held as Part of HCI International 2017, Vancouver, BC, Canada, July 9–14, 2017, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10298)*, Jia Zhou and Gavriel Salvendy (Eds.). Springer, 79–91. https://doi.org/10.1007/978-3-319-58536-9_7
- [39] Thomas F. Liu, Mark Craft, Jason Situ, Ersin Yumer, Radomir Mech, and Ranjitha Kumar. 2018. Learning Design Semantics for Mobile Apps. In *The 31st Annual ACM Symposium on User Interface Software and Technology (Berlin, Germany) (UIST '18)*. ACM, New York, NY, USA, 569–579. <https://doi.org/10.1145/3242587.3242650>
- [40] Aiguo Lu and Chengqi Xue. 2020. A Study on Search Performance and Threshold Range of Icons. In *Engineering Psychology and Cognitive Ergonomics. Mental Workload, Human Physiology, and Human Energy - 17th International Conference, EPCE 2020, Held as Part of the 22nd HCI International Conference, HCII 2020, Copenhagen, Denmark, July 19–24, 2020, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12186)*, Don Harris and Wen-Chin Li (Eds.). Springer, 62–68. https://doi.org/10.1007/978-3-030-49044-7_6
- [41] Zijing Luo, Chengqi Xue, Yafeng Niu, Xinyue Wang, Bingzheng Shi, Lingcun Qiu, and Yi Xie. 2019. An Evaluation Method of the Influence of Icon Shape Complexity on Visual Search Based on Eye Tracking. In *Design, User Experience, and Usability. User Experience in Advanced Technological Environments - 8th International Conference, DUXU 2019, Held as Part of the 21st HCI International Conference, HCII 2019, Orlando, FL, USA, July 26–31, 2019, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 11584)*, Aaron Marcus and Wentao Wang (Eds.). Springer, 44–55. https://doi.org/10.1007/978-3-030-23541-3_4
- [42] Forough Mehralian, Navid Salehnamadi, and Sam Malek. 2021. Data-Driven Accessibility Repair Revisited: On the Effectiveness of Generating Labels for Icons in Android Apps. In *Proc. ESEC/FSE (Athens, Greece) (ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 107â€118. <https://doi.org/10.1145/3468264.3468604>
- [43] Kevin Moran, Boyang Li, Carlos Bernal-Cárdenas, Dan Jelf, and Denys Poshyvanyk. 2018. Automated reporting of GUI design violations for mobile apps. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 165–175. <https://doi.org/10.1145/3180155.3180246>
- [44] Simone Mutti, Yanick Fratantonio, Antonio Bianchi, Luca Invernizzi, Jacopo Corbetta, Dhilung Kirat, Christopher Kruegel, and Giovanni Vigna. 2015. Baredroid: Large-scale analysis of android apps on real devices. In *Proceedings of the 31st Annual Computer Security Applications Conference*. 71–80.
- [45] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25–29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*

- Alessandro Moschitti, Bo Pang, and Walter Daelemans (Eds.). ACL, 1532–1543. <https://doi.org/10.3115/v1/d14-1162>
- [46] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7–9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). <http://arxiv.org/abs/1409.1556>
- [47] Mick Smythwood, Siné McDougall, and Mirsad Hadzikadic. 2019. Search-Efficacy of Modern Icons Varying in Appeal and Visual Complexity. In *Design, User Experience, and Usability: User Experience in Advanced Technological Environments - 8th International Conference, DUXU 2019, Held as Part of the 21st HCI International Conference, HCII 2019, Orlando, FL, USA, July 26–31, 2019, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 11584)*, Aaron Marcus and Wentao Wang (Eds.). Springer, 94–104. https://doi.org/10.1007/978-3-030-23541-3_8
- [48] Niko Sünderhauf, Oliver Brock, Walter J. Scheirer, Raia Hadsell, Dieter Fox, Jürgen Leitner, Ben Upcroft, Pieter Abbeel, Wolfram Burgard, Michael Milford, and Peter Corke. 2018. The Limits and Potentials of Deep Learning for Robotics. *CoRR* abs/1804.06557 (2018). arXiv:1804.06557 <http://arxiv.org/abs/1804.06557>
- [49] Mingxing Tan and Quoc V. Le. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9–15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 6105–6114. <http://proceedings.mlrpress/v97/tan19a.html>
- [50] Mengyue Wang and Xin Li. 2017. Effects of the aesthetic design of icons on app downloads: evidence from an android market. *Electron. Commer. Res.* 17, 1 (2017), 83–102. <https://doi.org/10.1007/s10660-016-9245-4>
- [51] Zhou Wang, Alan C. Bovik, Hamid R. Sheikh, and Eero P. Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE Trans. Image Process.* 13, 4 (2004), 600–612. <https://doi.org/10.1109/TIP.2003.819861>
- [52] Michael J. Wilber, Walter J. Scheirer, Phil Leitner, Brian Heflin, James Zott, Daniel Reinke, David K. Delaney, and Terrance E. Boulton. 2013. Animal recognition in the Mojave Desert: Vision tools for field biologists. In *2013 IEEE Workshop on Applications of Computer Vision, WACV 2013, Clearwater Beach, FL, USA, January 15–17, 2013*. IEEE Computer Society, 206–213. <https://doi.org/10.1109/WACV.2013.6475020>
- [53] Shengqu Xi, Shao Yang, Xusheng Xiao, Yuan Yao, Yayuan Xiong, Fengyuan Xu, Haoyu Wang, Peng Gao, Zhuotao Liu, Feng Xu, and Jian Lu. 2019. DeepIntent: Deep Icon-Behavior Learning for Detecting Intention-Behavior Discrepancy in Mobile Apps. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11–15, 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM, 2421–2436. <https://doi.org/10.1145/3319535.3363193>
- [54] Xusheng Xiao, Xiaoyin Wang, Zhihao Cao, Hanlin Wang, and Peng Gao. 2019. IconIntent: automatic identification of sensitive UI widgets based on icon classification for Android apps. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25–31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 257–268. <https://doi.org/10.1109/ICSE.2019.00041>
- [55] Bo Yang, Zhenchang Xing, Xin Xia, Chunyang Chen, Deheng Ye, and Shanping Li. 2021. Don't Do That! Hunting Down Visual Design Smells in Complex UIs against Design Guidelines. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22–30 May 2021*. IEEE, 761–772. <https://doi.org/10.1109/ICSE43902.2021.00075>
- [56] Dehai Zhao, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. 2020. Seenomaly: vision-based linting of GUI animation effects against design-don't guidelines. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1286–1297. <https://doi.org/10.1145/3377811.3380411>

Received 2023-02-16; accepted 2023-05-03