# IconChecker: Anomaly Detection of Icon-Behaviors for Android Apps

Yuxuan Li[*], Ruitao Feng[†], Sen Chen[*], Qianyu Guo[*], Lingling Fan[‡], and Xiaohong Li[*]
[*]College of Intelligence and Computing, Tianjin University, China
[†]School of Computer Science and Engineering, Nanyang Technological University, Singapore
[‡]College of Cyber Science, Nankai University, China
Email: {yuxli, senchen, qianyuguo, xiaohongli}@tju.edu.cn, rtfeng@ntu.edu.sg, linglingfan@nankai.edu.cn

***Abstract*—As a result of the technical evolution in network technologies and the upper applications, the reliance of mobile apps on the Internet increased heavily on the purpose of excellent service in years. However, the speedy increase brought not only conveniences but also security risks. For instance, it is unveiled that there exists a series of malicious apps, which are aiming to collect users' private data and imperceptibly send them to remote servers under the camouflage of normal users' behaviors. To defend against the threat, although lots of research has been proposed, it is still a challenge to capture the abnormal behaviors more precisely.**

**In this paper, we propose IconChecker, a GUI-based anomaly detection framework, to detect icons that can cause malicious network payloads under the premise of users' normal intentions. IconChecker can detect the abnormal icon-behaviors with the icon's semantics and triggered network traffic in relatively high precision, and further generate a security report for analysis and development. To demonstrate the effectiveness, we evaluate IconChecker from: (1) the accuracy of network traffic sniffing; (2) the accuracy of icon semantics classification; (3) the overall precision of IconChecker towards real apps; (4) comparing IconChecker with the existing tool, i.e., DeepIntent. The detection results show that IconChecker can outperform at the precision of 84% in terms of our summarized 8 categories of icon-behaviors. We remark that IconChecker is the first work, which dynamically detects abnormal icon-behaviors, to identify the malicious network payloads in Android apps.**

***Index Terms*—Android malware, Malicious payload, Network traffic, Anomaly detection, Dynamic analysis**

## I. Introduction

Mobile phones have become indispensable devices in daily lives [1]. On the one hand, the latest statistics [2] unveils that there are more than three million apps on Google Play Store. On the other hand, around 50% of Internet traffic is generated by mobile devices. With the development of new technologies (e.g., 5G), the network traffic generated by mobile devices will increase further [3]. Hence, promising personal privacy security will become more desired and challenging.

Private data leakage is one of the most powerful and typical security threats in Android malware. Android malware has been categorized into various families [4], [5] based on different malicious payloads such as malicious network payloads to steal private data. Consequently, in recent years, it is not surprising that lots of Android malware detection methods have been proposed, such as signature-based methods [6], behavior-based methods [7], data flow analysis-based methods [8], [9], and machine learning-based methods [5], [10]–[17]. However, these methods did not consider whether the payloads especially for the network payloads belong to users' intentions, neither do the existing network traffic classification methods for Android apps [18], [19], which would cause a lot of false positives during detection [20].

AppIntent [21] is the first work to put forward the concept of user-intended data transmission. They leveraged program analysis to obtain the path that triggers the sending behaviors of private data by constructing call graphs. According to the results, the security analysts can simulate the execution of the app and check if it has caused privacy leakage through the user interface (UI) pages. However, the semi-automated method is constrained by the human experience in practice. In recent work, DeepIntent [20] used static program analysis to establish the mapping relations among UI icons, the contextual text of icons, and permissions. They then trained a deep learning model to predict the potentially used permissions of the examined icon (e.g., INTERNET) and find the abnormal behaviors. However, according to our experimental results in Section III-F, it would also cause a lot of false positives.

It is well-known that mobile users use mobile devices based on the understanding of the semantics of UI components, however, the actual behaviors behind icons are transparent to users. In the worst case, there may exist behavioral gaps between the icon semantics and real icon-behaviors on the certain icon. Therefore, it would be dangerous to click such icons with inconsistent behaviors, involving hidden malicious operations to trigger the malicious payloads. Furthermore, for almost all malicious behaviors, they finally need to send the users' private data to the server via network traffic as the endpoint of a successful attack. All in all, it is essential to examine the consistency between the real behaviors of icons via network traffic and the semantics of icons.

To this end, we propose IconChecker to dynamically detect malicious network payloads for Android apps. The significant difference between IconChecker and the previous studies is that IconChecker dynamically performs actual operations on the apps to capture the network traffic of icons being clicked and detects the unexpected network traffic. We remark that interactive UI icons leverage images or texts to show their expected behaviors. We first predefined 8 categories of icons that should not generate network traffic when being operated based on a large-scale analysis of 2,820 icons. Specifically, IconChecker consists of three key modules: *icon-traffic mapping, icon-category mapping, and anomaly icon-behavior detection*. (1) Icon-traffic mapping module establishes a one-to-one mapping between icons and behaviors through network traffic. We propose a novel framework to accurately capture the network traffic when the icon is clicked dynamically. (2) Icon-category mapping module obtains the icon semantics and checks whether it belongs to the predefined 8 categories. In addition, we adopt a progressive strategy to establish the relationship between icon and its semantics through heuristic layout analysis and optical character recognition (OCR) [22] techniques. The detection accuracy of our proposed strategy achieves 87.2%. (3) Anomaly icon-behavior detection module combines the detected traffic and the icon category result to check whether the icon has abnormal behaviors and outputs a security report for each app.

To demonstrate the effectiveness of IconChecker, we first evaluate the effectiveness of the key modules and further compare IconChecker with the existing tool (i.e., DeepIntent) on 1,800 Android apps in terms of our predefined 8 categories of icons. As a result, IconChecker detects 25 icons with abnormal behaviors in 23 Android apps. Among them, 21 icons are true positives (84% precision).

In summary, we make the following main contributions:

- We design IconChecker,[1] an automated tool to detect abnormal icon-behaviors with malicious network payloads in Android apps. IconChecker dynamically operates on apps by triggering icons' behaviors instead of static parsing, which is relatively more accurate and precise.
- To clarify the key issues in icon semantics classification, we perform a study on more than 2,820 icons' behaviors and further summarize 8 icon categories that should not generate traffic in any scenario.
- We propose a novel framework to accurately capture the network traffic when triggering icons' behaviors by clicking the icons. Meanwhile, we propose a progressive strategy to extract icons' semantics under various scenarios.
- Compared with the existing method (i.e., DeepIntent) in anomaly detection of icon-behaviors, IconChecker can detect more icons with abnormal behaviors and achieves a better detection precision on the predefined 8 icon categories.

## II. Approach

### A. Overview

Fig. 1 shows the entire workflow of our approach, namely IconChecker. IconChecker takes an apk as input, and outputs a security report, which shows whether the app under testing has malicious network payloads, as well as the corresponding icon ID and malicious code. Specifically, IconChecker leverages the following three modules to detect the abnormal icon-behaviors: (a) *icon-traffic mapping* extracts the icons presented in UI pages, sniffs the network traffic, and further maps icons to the corresponding network traffic at runtime; (b) *icon-category mapping* obtains the semantics from the extracted icons, and classifies them into 8 predefined categories or "Others"; and (c) *anomaly icon-behavior detection* compares the collected traffic with the categories of the icons and determines whether the icons have abnormal behaviors according to their categories. Finally, IconChecker generates a security report for further analyzing and fixing.

[1]The code is released on https://github.com/IconCheck/IconChecker.

### B. Icon-Traffic Mapping

In the *icon-traffic mapping* module, the system takes the target apk as input. Firstly, we explore the information of defined activities in the target app, which can be used to start them. Secondly, the app is installed on the Android device since the icon traffic sniffing needs to be performed while the app is running dynamically. After the installation, we then dynamically capture the network traffic associated with the icons and finally construct the icon-traffic mapping relations. The output of this module is the explored icons and their corresponding captured network traffic.

*1) **Activity exploration**:* To easily launch more activities implemented in the input app, we determine to start activities directly from the console instead of triggering them with testing tools. Thus, we first disassemble the apk to conduct the corresponding information collection and manipulation. In this step, we set the `android:exported` attribute of each activity to *true* in the *AndroidManifest.xml* file to ensure the activity launching process can be started with commands in the console. In Activity, `android:exported` attribute is used to indicate whether the current activity can be invoked by third-party components. Besides, we also summarize the name of each activity and the package it belongs to in this phase. Secondly, we reassemble the modified contents and repackage the apk with a signature. With these efforts, most of the activities are able to be directly launched from the console with our collected information by IconChecker. We then start each activity with command lines directly to dump the layout and intercept the current page (i.e., UI) for further use.

*2) **Traffic sniffing**:* The second step of the *icon-traffic mapping* module is sniffing the potential network traffic triggered by the icon's operations on each page. Some existing malware detection or app identification work, which applied network traffic as their evaluation criteria, usually filtered the packets to capture the harmful network traffic, according to the port numbers occupied by the app at runtime. However, unlike them, capturing the traffic triggered by an icon is a far more fine-grained issue.

```
private static String downloadFeed(OkHttpClient
    httpClient,...) {
  HttpUrl url = HttpUrl.parse(theURI);
  Request.Builder requestBuilder = new Request.
   Builder().url(url);
  ...
  Request request = requestBuilder.build();
  okhttp3.Response response = httpClient.newCall(
   request).execute();
  ...
}
```

Listing 1. An example of starting an activity to generate network traffic

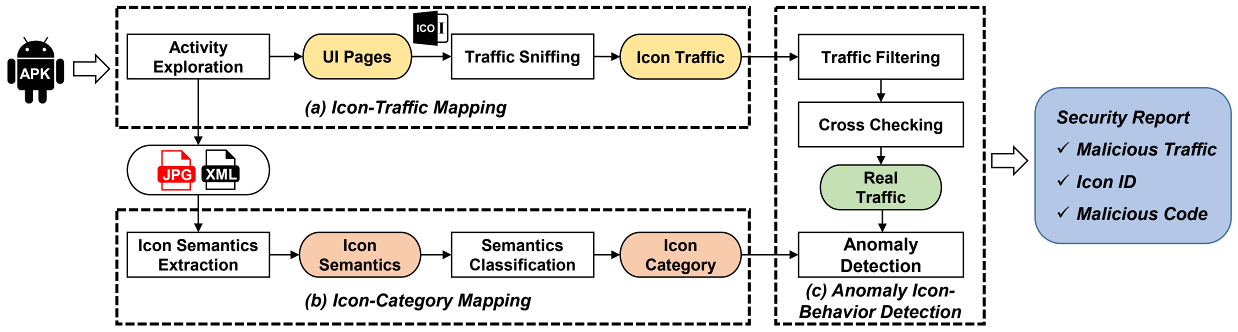

Fig. 1. Overview of IconChecker

In the early stage of our work, we found that some activities might cause network traffic while launching, because some pages may contain advertisements or require Internet access to load resources. Thus, the traffic caused by the activity launching or page loading can be mixed up with the traffic caused by clicking, which further affects the statistical accuracy in experiments. Listing 1 shows a sample code snippet, which can cause network traffic while launching an activity in an open-source app called RadioDroid [23]. Obviously, in line 6, it sends a request to access the Internet. To solve this kind of problem, before monitoring the clicked icon as well as its network traffic, IconChecker sets a *timer* to skip the potential supernumerary traffic after launching the activity.

After launching the explored activities successfully, we then extract the text attributes and resource-ids from the layout of the current page which is displayed on the screen. These text attributes and resource-ids are assigned as the file name of the corresponding network traffic profiling result to establish the *icon-traffic mapping* relationship. In usual cases, each page may have zero or more clickable icons and the current page may be changed after clicking. Therefore, before the next clicking, we need to restart the activity and check whether the next target icon is on the current page. If the target icon is still on the current page, IconChecker first waits for a while to ensure there is no network traffic noise, and then clicks on the icon and records the network traffic in the next 3 seconds at the same time. Moreover, during the process of capturing network traffic, there exists some hidden traffic, which may also be considered as the app's belonging. For instance, the traffic of Network Time Protocol (NTP) [24], which is a protocol for network time synchronization, will inevitably be captured by the *traffic sniffing* module at runtime. Therefore, to enhance the experimental accuracy of the captured traffic, we repeat this process for additional 3 times on each icon. Furthermore, the port numbers occupied by the app will also be obtained and saved together with other information (i.e., icon and network traffic) to the result set and pass to the detection module.

## C. *Icon-Category Mapping*

The *icon-category mapping* module consists of two steps: (1) icon semantics extraction, (2) semantics classification. Firstly, with the retrieved icon information, e.g., icon ID, position coordinates, name of the belonging activity and package from the *activity exploration* in module Fig. 1(a), icons' semantics can be obtained with our proposed *progressive strategy*. Secondly, according to our predefined semantics of icon categories, we then classify the icons by measuring the similarity between the extracted and predefined semantics. At last, the output of *icon-category mapping* module, i.e., categorized icons, is accepted as the input of the final *anomaly detection* step in Fig. 1(c).

*1)* ***Icon semantics extraction****:* To distinguish the normal and abnormal network-access behaviors triggered by clicking the icon, in the first step, we extract semantics information from the icons' definition and implementation. Generally, there are several challenges while working on this task. Firstly, icons defined in various forms and styles may have the same or similar semantics information. Secondly, according to the icons' design, specifically, they can be divided into three types, i.e., *graphical icons, text-based icons, and icons combining graphics and text*. Besides, the icons are usually small, scattered, and partially or completely transparent [25]. Therefore, computer vision techniques such as scale-invariant feature transform (SIFT) [26], are not completely suitable for our task, which classifies icons defined in Android apps. In the second step, we analyze the extracted semantics information to determine the icons that should not cause network traffic, and further maintain a network-independent icon dataset.

During semantics information extraction, we observe that: (1) the text shown on some icons can accurately represent their semantics; (2) some icons do not have text attribute; (3) developers usually name the icon's resource-id attribute according to its functionality based on our investigation on 1,601 icons from 100 apps. In other words, the resource-id of the icon represents the corresponding semantics to some extent. *Therefore, we propose a progressive strategy to extract the semantics from icons according to the above three points.* Specifically, we first extract all the icon information (i.e., text, resource-id, and position coordinates) from the layout file of the current page, and analyze the icon's text attribute, which can represent most of its semantics, if it exists. If the icon does not have a text attribute, we then leverage OCR techniques to identify the text displayed on the icon from the

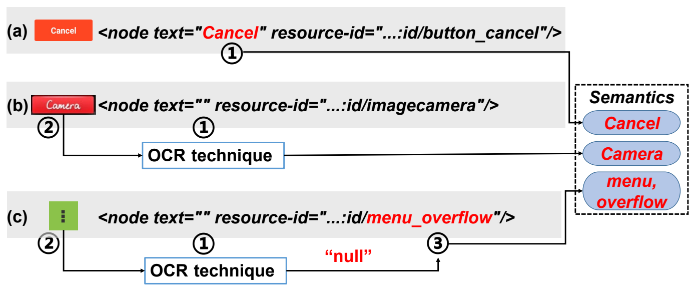
(a)
Cancel
<node text="Cancel" resource-id="...:id/button_cancel"/>
(b)
Camera
<node text="" resource-id="...:id/imagecamera"/>
OCR technique
(c)
<node text="" resource-id="...:id/menu_overflow"/>
OCR technique
"null"
Semantics
Cancel
Camera
menu, overflow

Fig. 2. Progressive strategy for icon semantics extraction

current page. To avoid the interference from contents other than icon pictures, we crop icons from their belonging pages, which are saved in the *activity exploration* step according to the position coordinates (e.g., [(566, 1301), (894, 1419)]) of icons. Otherwise, as the worst case, there exists no text-based information, we then directly extract the semantics from the resource-id attribute. Three detailed examples are provided in Fig. 2 to illustrate the proposed progressive strategy.

- **Use text-layout as icon's semantics.** In Fig. 2(a), we can directly use the icon's text attribute value "Cancel" to represent the semantics of this icon.
- **Use text-OCR as icon's semantics.** In Fig. 2(b), we can not obtain any related information from the text attribute in layout. Thus, we apply the OCR technique to recognize whether there is embedded text on the icon. In this case, we obtain and use "Camera" as this icon's semantics.
- **Use resource-id as icon's semantics.** In Fig. 2(c), we can not obtain any related information from both text attribute and icon itself. Hence, we analyze the resource-id attribute of the icon to retrieve its semantics alternatively from resource-id.

*2) **Semantics classification**:* Existing work [25] summarizes the icon categories according to the icons' popularity and the access to sensitive resources, but these categories will generate traffic under some circumstances which are not suitable for our work. Therefore, before *semantics classification* step, we first empirically perform an investigation on 2,820 icons randomly extracted from real apps. With our in-depth analysis of the icons' usage scenarios and implemented functionalities (Section III-C), we summarize 8 network-independent icon categories, which do not generate any network traffic under any circumstances, as well as their contained semantics. The details are provided in Table I.

To determine whether the extracted icon semantics belongs to the defined 8 categories, IconChecker leverages Levenshtein distance algorithm [27] to compute the distance between the extracted semantics of the target icon and predefined semantics of icon categories in Table I, and further calculates the similarity between them with the following equation:

$$Similarity = max(1 - \frac{Ld_i}{len(semantics)}),$$

TABLE I
ICON CATEGORIES AND SEMANTICS

| Icon categories | Corresponding semantics |
|---|---|
| Audio | audio, sound, volume, ring |
| Camera | camera, photo, album |
| Cancel | cancel |
| Close | close |
| Menu | menu, list |
| Phone | phone, call |
| SMS | sms, message |
| ToolsSettings | tool, setting, tools, settings, set up |

where $Ld_i$ is the Levenshtein distance, $len(semantics)$ refers to the number of alphabets in predefined semantics term.

For each predefined semantics term in Table I, we first compute its similarity to the extracted semantics of the target icon. Then, we consider the category, which contains a term with maximal similarity value, as the potential classification result. With the similarity value, we further set a threshold value based on the ground truth. Specifically, we collect the similarity values between the semantics of all the icons in the ground truth dataset and the predefined semantics terms, and then find the best threshold value, which can achieve the highest overall classification accuracy.

### *D. Anomaly Icon-Behavior Detection*

In this module, we focus on detecting abnormal traffic based on the results from the previous two modules, i.e., *icon-traffic mapping* and *icon-category mapping*, and finally generating a security report, which contains the detected malicious traffic, icon ID, and malicious code, to help developers and security analysts with further analyzing and fixing. Firstly, to obtain the traffic generated by the current app among overall network traffic, there is a first step, namely *traffic filtering*, that intends to filter out the unrelated traffic based on the port number assigned to the app according to its PID (i.e., process ID). Secondly, we apply cross-checking to find the real traffic among a large number of results. The rule used to determine the existence of traffic, is that if there is no traffic captured in any of 4 repeating tests, we consider there is no traffic; while only if there is traffic in all 4 tests, we regard the network traffic exists. Thirdly, to determine the icons, which can trigger abnormal traffic, we match the captured semantics of the icons, which belong to the 8 predefined categories, from the *icon-category mapping* module with the real traffic and output the detection results. Finally, based on the results, IconChecker generates a security report for each tested app, including detected malicious traffic, icon ID, and the corresponding malicious code, which is parsed based on the abnormal icon ID and name of its belonging activity and package.

## III. Evaluation

In this section, to evaluate the effectiveness of IconChecker, we conduct in-depth experiments from different perspectives to answer the following 5 research questions (RQs):

- **RQ1:** How effective is IconChecker in sniffing icon's traffic?
- **RQ2:** How are the 8 categories decided in our study?
- **RQ3:** How effective is IconChecker in icon-category mapping?
- **RQ4:** How effective is IconChecker in detecting abnormal icon-behaviors?
- **RQ5:** How does IconChecker perform compared to the existing tool (i.e., DeepIntent)?

### *A. Dataset and Environment*

To support the evaluations towards IconChecker, we collected 1,800 apps from StormDroid [11], [28] as the app dataset. For each raised RQ, we constructed a specific ground truth dataset, which is obtained by analyzing our app dataset, to support the IconChecker and our conclusions.

- **RQ1:** To demonstrate the effectiveness of icon's traffic sniffing, we select 15 apps, which consist of 10 apps from Google Play Store and 5 apps developed by ourselves, as a result of the limited occurrence of malicious network traffic in real-world apps.
- **RQ2:** To ensure the validity of our study, we analyze 2,820 icons in 100 apps for various use, such as video, education, and tools.
- **RQ3:** To investigate the effectiveness of the icon-category mapping, we randomly select 2,000 icons from the screenshots dumped by IconChecker, and manually labelled these icons as groud truth according to the icon's appearance.
- **RQ4 & RQ5:** We randomly select 1,800 apps from StormDroid [11], [28] to evaluate IconChecker as an entire system, and further compare IconChecker with the existing tool. Note that, since the applicable domain and strategy used to obtain icons in DeepIntent is a little different from IconChecker, after experimental testing (details in III-F), we obtain 147 apps, which can be successfully used by DeepIntent, in our predefined 8 categories.

All experiments are conducted on a high-performance workstation, equipped with 64-bit Ubuntu 18.04 LTS OS, 16GB RAM, and two Intel 4-core i5-5200U CPUs.

### *B. RQ1: How effective is IconChecker in sniffing icon's traffic?*

*1) **Dataset**:* In this experiment, we adopt 5 apps[2] developed by ourselves. In each app, we define 10 icons for each behavior with and without network access, respectively. Besides, we also randomly select 10 real apps from Google Play Store. For each app, we randomly choose 5 UI pages and extract all icons. Since we do not know the real behavior of each icon (i.e., whether it requires access to the Internet), we manually analyze the implementation of each icon to build the ground truth with the help of reverse engineering and static code analysis. Totally, 50 out of 100 and 214 out of 458 icons are identified as containing network access behaviors, and the rest as network-independent.

[2]https://github.com/IconCheck/IconChecker/apkfile.

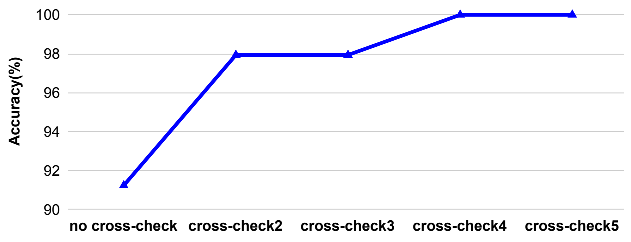

Fig. 3. Network sniffing accuracy under different cross-check times

*2) **Setup**:* Based on the above dataset, we conduct experiments to demonstrate the effectiveness of IconChecker in precisely capturing the icon traffic. Specifically, we run IconChecker on the 100 icons from 5 self-developed apps and the 458 icons from 10 real apps, for the purpose of seeing whether network traffic can be captured once the icons are clicked. To avoid potential noise, we repeat the experiment 4 times for each app and obtain an overall result by cross-checking on the results from 4 repetitions.

*3) **Results**:* As shown in Fig. 3, the accuracy can reach 100% after the 4 experimental results are cross-checked. Although the accuracy can reach 90% after the cross-check of the 2 experimental results, the random potential network traffic noise will interfere with the results. Thus, we set the number of cross-checks to 4. The interference in sniffing come from two aspects:

**NTP (Network Time Protocol) packets.** As addressed in II-B2, NTP is a build-in functionality for time synchronization in computer operation systems. Unlike network traffic caused by other reasons, e.g., background process, the packets generated by NTP are unable to be distinguished with process-level system information, i.e., PID. Hence, the corresponding network traffic may inevitably be captured by the *traffic sniffing* module and further considered as the network traffic produced by the icon being clicked at runtime.

**"generate_204" packets.** Another type of noise comes from the packet, namely "generate_204", which is a network status evaluation mechanism in Android 8.0+ [29]. Specifically, it allows Android devices to send network access requests and listen to the response with the specific status code, i.e., 204, for confirming potential network connection. Similarly, such network packets, which are generated by an OS mechanism, can not be distinguished with process-level system information, i.e., PID, either.

***Answer to RQ1:*** *IconChecker can effectively sniff icon's network traffic at runtime. With cross-checking method, IconChecker can accurately determine whether the network traffic is generated by real icon operations with accuracy at 100%.*

### C. *RQ2: How are the 8 categories decided in our study?*

*1) **Dataset**:* To summarize the icon categories that should not generate payloads, we collect 2,820 icons and grab the payloads generated by each icon through the first module of IconChecker. It is found that 705 icons generate traffic payloads and 2,115 icons do not.

*2) **Setup**:* After using the first module of IconChecker to sniff traffic payloads generated by clicking each icon, we manually classify the 2,820 icons into 23 categories based on their behaviors.

*3) **Results**:* As shown in Table II, icons are divided into 2 categories according to whether they access the Internet or not: 28 categories of icons access the Internet, 10 categories of icons do not. According to the usage scenarios, icon's semantics, and functionalities, these 28 categories that access the Internet can be divided into three situations: (1) Definitely generate traffic: "Search", "Pay", "Link", "Download", etc. The percentage of these icons accessing the Internet is not 100% because when we automatically capture the traffic, some dependent contents, such as keywords in the search box, the information of credit card, valid links, are missing during triggering; (2) Possibly generate traffic: "Send", "Edit", "About", etc. For example, in a scenario where a message is sent using SMS, clicking the "Send" button will not generate traffic, but if the "Send" is in WeChat/WhatsApp, it will generate traffic properly; (3) Must not generate traffic. 4 categories of icons are unreasonable to access the Internet (displayed in bold).

Furthermore, we analyze the 10 categories of icons that do not access the Internet. Among them, 6 categories of icons may need to access the Internet in specific scenarios. However, these situations do not appear in our analyzed cases due to dataset limitation. Take the "Help" category as an example, by extending to more other apps out of our dataset, we find that the help information of some apps can be stored either locally or on the server, so such icons may interact with Internet. Moreover, an icon belongs to the "Save" category will generate traffic when the saved information needs to be transmitted to the server-end database. In total, we summarize 8 icon categories that should not generate traffic by our study (displayed in bold).

***Answer to RQ2:*** *Based on the manual study towards 2,820 icons, 8 categories of icons that should not generate traffic are defined.*

### D. *RQ3: How effective is IconChecker in mapping icon to its category?*

*1) **Dataset**:* In this experiment, we randomly select 2,000 icons from the 1,800 apps in Section III-E (RQ4). The reason why we do not directly use the icons extracted in Section III-B (RQ1) as ground truth is that the diversity of their semantics is quite limited, which can not ensure the credibility of the classification results through a broader dataset. We manually analyze the 2,000 icons to identify their semantics (i.e., text or resource-id) and further build the ground truth, which consists of 4 labels: (1) 600 icons, using resource-id as its semantics and belonging to the 8 predefined categories; (2) 400 icons, using text as its semantics and belonging to the 8 categories; (3) 600 icons, using resource-id as its semantics and not belonging to the 8 categories, which is also denoted by "Others" category; (4) 400 icons, using text as its semantics and belonging to "Others".

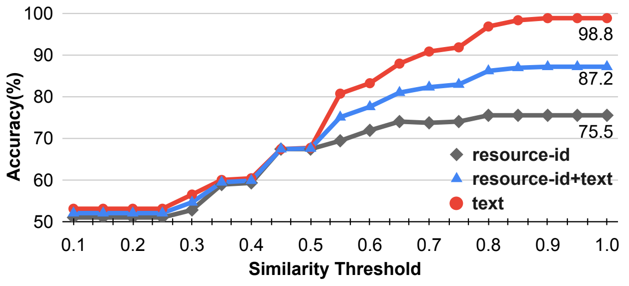

Fig. 4. Accuracy of semantics classification in terms of similarity threshold

*2) **Setup**:* As mentioned in Section II-C, the accuracy of semantics classification highly depends on the determined similarity threshold. Thus, we first conduct an experiment to reveal the detailed procedures in determining the best threshold, i.e., the value between 0.1$\sim$1.0, 0.05 as the step size. Fig. 4 shows the changing trend of the accuracy, when using resource-id, text, and resource-id+text (i.e., the proposed progressive strategy) as icon's semantics under different thresholds, respectively. Note that, the accuracy of using individual resource-id is evaluated on the above dataset (1) and (3); the accuracy of using text is evaluated on the dataset (2) and (4); and the accuracy of using both resource-id and text is evaluated on all the 2,000 icons, which can best indicate the effectiveness of IconChecker on semantics classification.

*3) **Results**:* As shown in Fig. 4, classification on the icons, whose semantics can be extracted from text, achieves the highest accuracy at 98.8%. For the icons without available text, the accuracy of classification based on their resource-id is 75.5%. Additionally, we also evaluate the overall effectiveness of the *icon-category mapping* module by performing classification using our randomly selected 2,000 icons, whose semantics are defined in either text (i.e., text-layout and text-OCR) or resource-id. The overall accuracy of classification using the entire dataset achieves 87.2%, by observing the blue polyline titled "resource-id+text". As a result of the stronger superiority of text-based semantics, it is obvious that this result is a compromise between the accuracy of classification using half dataset selected based on the two semantics extraction methods. Furthermore, we can observe that the accuracy trend of using the icon's resource-id as its semantics continues to rise before the threshold value reaches 0.8, and does not change with the increase of the threshold afterward. For the accuracy trend of using text as semantics, the best threshold value is 0.9. Finally, to ensure a relatively high classification accuracy for all potential cases, the threshold is set to 0.9 in Section III-E.

Based on the comprehensive analysis of the experimental results, it is obvious that both text and resource-id are neces-

TABLE II
THE STUDY OF ICON CATEGORIES WITH/WITHOUT ACCESSING TO THE INTERNET

| **Categories** | **Pos[1](%)** | **Neg[2](%)** | **Categories** | **Pos[1](%)** | **Neg[2](%)** | **Categories** | **Pos[1](%)** | **Neg[2](%)** | **Categories** | **Pos[1](%)** | **Neg[2](%)** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Back | 5.67 | 94.33 | Home | 7.14 | 92.86 | Send | 7.14 | 92.86 | About | 9.09 | 90.91 |
| Finish | 11.11 | 88.89 | Submit | 11.76 | 88.24 | Add | 15.79 | 84.21 | User | 19.57 | 80.43 |
| Share | 20.00 | 80.00 | Eidt | 22.22 | 77.88 | Register/Login | 24.00 | 76.00 | Refresh | 25.00 | 75.00 |
| Logo | 33.71 | 66.29 | Location | 37.5 | 62.5 | Ok | 41.94 | 58.06 | Agree/Confirm | 42.11 | 57.89 |
| Exit | 50.00 | 50.00 | Search | 60.66 | 39.34 | Music | 66.67 | 33.33 | Update | 66.67 | 33.33 |
| Pay | 73.17 | 26.83 | Download/Reload/Install | 74.48 | 25.52 | Link | 93.42 | 6.58 | Retry | 100.00 | 0.00 |
| **Menu** | 3.57 | 96.43 | **Close** | 8.11 | 91.89 | **ToolsSettings** | 8.33 | 91.67 | **SMS** | 25.00 | 75.00 |
| Clean/Clear | 0.00 | 100.00 | Save | 0.00 | 100.00 | Help | 0.00 | 100.00 | Continue | 0.00 | 100.00 |
| Delete | 0.00 | 100.00 | Record | 0.00 | 100.00 | | | | | | |
| **Audio** | 0.00 | 100.00 | **Phone** | 0.00 | 100.00 | **Cancel** | 0.00 | 100.00 | **Camera** | 0.00 | 100.00 |

1. Percentage of accessing to the Internet. 2. Percentage of not accessing to the Internet.

sary for understanding the semantics of icons and further classifying the icons into our predefined 8 categories. Moreover, the results of the three different semantics extraction methods can also support the effectiveness of our progressive strategy, which prioritizes the extraction of semantics from text. In other words, comparing to randomly select semantics from text or resource-id, it definitely pushes the compromised result to a better one on any dataset which consists of icons with and without available text.

***Answer to RQ3:*** *By testing 2,000 icons in Android apps, compared with randomly using icon's text or resource-id to represent icon's semantics, the proposed progressive strategy performs well in practice, achieving an accuracy of 87.2%.*

## *E. RQ4: How effective is IconChecker in detecting abnormal icon-behaviors?*

*1)* ***Dataset****:* In this experiment, we randomly select 1,800 apps from StormDroid [11], [28]. By analyzing the activities (5417/8820) successfully started in these 1,800 apps, we obtain 18,370 icons as well as their information (e.g., icon ID, name of the belonging activity, and package) in total. In terms of our summarized 8 network-independent icon categories, we manually collect 1,012 icons according to their semantics. However, there is a big challenge that all icons are unlabelled for our research purpose. In other words, before using them as the ground truth, we need to confirm their real program behaviors to figure out whether there are illegal network requests inside the implementation. To build a ground truth dataset, we manually review the corresponding source code to each icons. Finally, we construct a labelled dataset, which has 1,012 icons belonging to the aforementioned 8 network-independent icon categories, and use it as our ground truth.

*2)* ***Setup****:* To demonstrate the overall effectiveness of IconChecker, we conduct an experiment on 1,800 apps, which contain 1,012 icons belonging to 8 categories, and then review the corresponding source code of each detected icon to confirm it with the ground truth. Furthermore, according to the generated security reports, we manually analyze the detected icons, which can trigger abnormal network traffic, to provide an in-depth case study for other potential relevant research.

TABLE III
DETECTION RESULTS OF ICONCHECKER

| **Category** | **Dataset[1]** | **ICONCHECKER[2]** | **Confirm** | **Precision (%)** |
|---|---|---|---|---|
| Audio | 80 | 0 | 0 | N/A |
| Camera | 61 | 2 | 2 | 100.0 |
| Cancel | 256 | **2** | 1 | 50.0 |
| Close | 240 | 10 | 10 | 100.0 |
| Menu | 193 | **2** | 1 | 50.0 |
| Phone | 56 | **1** | 0 | 0.0 |
| SMS | 28 | **3** | 2 | 66.7 |
| ToolsSettings | 98 | 5 | 5 | 100.0 |
| **Total** | 1,012 | 25 | 21 | **84.0** |

1. The icons belong to 8 categories.
2. The icons with abnormal icon-behaviors detected by IconChecker.

*3)* ***Results****:* The detection results of IconChecker are provided in Table III. Totally, IconChecker detects 25 icons from 1,012 icons in the dataset. By comparing this result with our ground truth, which is obtained by manual analysis, 21 out of 25 icons are confirmed as abnormal icons with an overall precision at 84.0%. Among 8 categories, none of the detected icons are located in the "Audio" category. Hence, the detection precision of "Audio" is not applicable. The "Close" category, which contains 240 icons in the dataset, has the largest number of confirmed abnormal icons, which is 10 out of 21. For the "Cancel" category, it has the largest number of icons in the dataset. However, the result in the "Confirm" column shows that only 1 out of 256 icons will cause abnormal network traffic. Hence, we can see that the distribution of abnormal icon-behaviors has strong randomness.

Besides, by comparing the detection result of IconChecker with ground truth, there exist 4 false positives in 4 categories, i.e., “Cancel”, “Menu”, “Phone”, and “SMS”. With an in-depth analysis on them, we find that the reasons can be summarized as the following two points:

**Continuously network communication.** We note that some malicious apps can continuously generate network traffic without any operations at runtime. For instance, the app, which contains a false positive icon in the package *com.automatic.call.recorder*, can generate continuous network traffic as long as it is started. To validate this problem, we monitor the network traffic generated by this app in continuous 10 seconds after starting it. According to the statistics, it generates 484 packets every 10 seconds on average.

**Semantics misclassification.** A false positive error detected as an abnormal icon by IconChecker is an icon with text “www.radicallabs.com”. Obviously, the functionality of this icon is redirecting to the URL link, namely “www.radicallabs.com”. Thus, it is reasonable to generate network traffic after clicking this icon. However, in *icon-category mapping* module, the substring, “call”, in this URL matches the semantics term “call” in the “Phone” category with a similarity value at 100%. Hence, IconChecker determines that this icon belongs to the “Phone” category which should not trigger any network traffic at runtime.

In summary, according to the results shown in Table III, the overall precision of IconChecker is 84.00% in anomaly detection of icon-behaviors for Android apps. For each icon, the average time consumed for each test is 15 seconds, which is relatively long, since we applied multiple mechanisms to maintain the fairness and accuracy of results at runtime.

*4) **Case Study:*** After detection, IconChecker creates a security report including icons with abnormal behaviors, malicious traffic, and malicious code for each detected app. In this case study, we provide a detailed analysis of some representative detection results. In Table IV, we select and show 4 samples with different attributes on various aspects, which contain semantics, category, and used method. Specifically, for the first icon, the semantics is extracted by the OCR technique and translated into English. For the second icon, the semantics is extracted by the OCR technique, too. For the third icon, since we cannot obtain text-layout from its text attribute and text-OCR through the OCR technique, the value of its resource-id attribute is used to represent its semantics. For the fourth icon, text-layout can be obtained from the icon’s text attribute, so we indicate its semantics with the value of its text attribute.

Listing 2 shows the corresponding malicious code of the “Close” icon in Table IV. The click event of this icon gets the device’s IMEI (Line 4), APP_ID (Line 5) and sends the information to the remote server (Line 6). However, according to the semantics of this icon, which is closing the current dialog, it should not generate any network traffic. Therefore, we confirm that this icon has malicious behaviors, which can leak the user’s private information through network payloads.

```
1 AsyncTaskCompleteListener<String>
        asyncTaskCompleteListener = new
        AsyncTaskCompleteListener<String >() {
2       public void lauchNewHttpTask() {
3           List<NameValuePair> list = new ArrayList<>();
4           list.add(new BasicNameValuePair(IConstants.IMEI
        ,"" + Util.getImei()));
5           list.add(new BasicNameValuePair(IConstants.
        APP_ID, Util.getAppID()));
6           new HttpPostDataTask(OptinActivity.this, list,
        IConstants.URL_OPT_IN, this).execute(new Void[0]);
7       }
8 }
```

Listing 2. An example containing malicious payloads detected by IconChecker

***Answer to RQ4:*** *IconChecker can discover the inconsistency between actual behaviors of the icon and user-intended behaviors by capturing the network traffic generated by the icon operation, achieving 84.00% precision on the experimental dataset.*

## *F. RQ5: How does IconChecker perform compared to the existing tool (i.e., DeepIntent)?*

*1) **Dataset:*** The original dataset used in this experiment is the same as the one in III-E. DeepIntent has a released pre-trained model as well as an open-source project on the GitHub[3], so we directly set up their project to perform the experiment with the same dataset, which has 1,800 apps. However, we met lots of uncertain problems (e.g., timeout, decode error), which failed the detection. Hence, we obtain usable 1,855 icons, which contains 153 icons in the 8 categories, in total, and only few of them are the same as the usable icons in Section III-E. To further gain the ground truth of these icons, we manually analyze the implementation of each icon as well.

*2) **Setup:*** To further demonstrate the performance of our approach, we compare IconChecker with DeepIntent [20], which combines static analysis with deep learning for intention-behavior discrepancies in apps. DeepIntent leverages 8 types of permissions used by the icon to characterize the actual behaviors. Since our task is not exactly the same as DeepIntent’s, we only focus on the NETWORK permission when evaluating DeepIntent.

*3) **Results:*** Table V shows the comparison of results between IconChecker and DeepIntent. We can see the detection result of DeepIntent indicates 150 out of 153 icons in 8 categories are detected as abnormal cases. The ground truth reveals that only 10 out of 150 icons, which belong to 4 apps, actually have malicious payloads. Therefore, in terms of our domain, the false positive rate of DeepIntent is very high. As shown in Table V, the precision of IconChecker and DeepIntent is 84.00% and 6.67% on 1,800 apps, respectively.

We further perform a step-by-step investigation on the distribution of used icon data in this experiment, especially for the DeepIntent. For the icons belonging to the predefined 8 categories, the number of icons tested by IconChecker is six times more than DeepIntent’s (1,012 vs 153). The possible reasons, which caused this problem, are explained as follows:

[3]https://github.com/deepintent-ccs/DeepIntent.

TABLE IV
CASE STUDY OF PARTIAL RESULTS

| Icon_ID | Icon | Identified semantics | Real semantics | Identified category | Real category | Semantics identification method |
|---|---|---|---|---|---|---|
| 1 | 🔧 手机工具 | mobile tools | mobile tools | ToolsSettings | ToolsSettings | OCR, Translate |
| 2 | Scan SMS | scan sms | scan sms | SMS | SMS | OCR |
| 3 | ✕ | cancel, btn | cancel | Cancel | Cancel | Resource-id |
| 4 | Close | close | close | Close | Close | Text |

TABLE V
COMPARISON BETWEEN ICONCHECKER AND DEEPINTENT

| Tool | Tested apps | Tested icons | Detected icons | Abnormal icon-behaviors | Precision (%) |
|---|---|---|---|---|---|
| IconChecker | 1,800 | **1,012** | **25** | **21** | **84.00** |
| DeepIntent | 1,800 | 153 | 150 | 10 | 6.67 |
| IconChecker | 147 | **444** | **6** | **6** | **100.00** |
| DeepIntent | 147 | 153 | 150 | 10 | 6.67 |

TABLE VI
THE REASONS OF FAILURES IN DEEPINTENT

| Step | Reason for failure | Apps |
|---|---|---|
| icon-widget association | Timeout | 148 |
| | Gator_Error | 175 |
| | Decode_Error | 23 |
| icon-permission | None of the icons have targeted permissions | 1,307 |
| remainder | - | **147** |

**From DeepIntent's perspective.** As shown in Table VI, the failures that occur in DeepIntent generally locate in 2 steps, i.e., icon-widget association and icon-permission. Totally, 148, 175, and 23 apps are discarded due to "Timeout", "Gator_Error", and "Decode_Error", respectively. There are 1,307 apps, whose icon-permission relationship is failed to be established and discarded from the dataset. Finally, 147 out of 1,800 apps are tested to be successfully used by DeepIntent.

**From IconChecker's perspective.** IconChecker dynamically captures icons from the UI pages and layout files from real mobile phones at runtime. Thus, there is a problem that some activities fail to start due to their data dependency, but 61.42% of the activities can be started successfully to ensure covering most icons in apps. Moreover, we also observe that some icons will only be loaded at specific conditions. In other words, these icons are occasionally unable to be displayed on the UI page.

In addition, for the tested 1,800 apps, IconChecker can successfully detect 21 abnormal icons, which generate network traffic, in the 8 categories. By comparing the detection result of DeepIntent, which is 10 icons, there are only 4 shared ones out of 21 icons. Moreover, 6 malicious icons detected by DeepIntent can't be found by IconChecker. With a step-by-step analysis, we find that 2 out of 6 icons can be mapped with the sniffed illegal network traffic, and the other 4 icons can not be obtained by IconChecker. For the 2 icons with traffic, we observe that they are misclassified into the "Others" category instead of the predefined 8 categories in the *icon-category mapping* module. For example, the appearance of an icon is "Close" with the resource-id called "btngo_back". Normally, IconChecker can obtain the semantics with text-OCR. However, it fails and uses the resource-id attribute as its semantics. Thus, these 2 icons are all considered to belong to "Others".

Besides, to provide a fairer comparison between IconChecker and DeepIntent, we further evaluate IconChecker with the 147 apps, which can be successfully used by DeepIntent. The results displayed in Table V show that IconChecker explores more icons at three-fold than DeepIntent in these apps and successfully detects 6 icons with abnormal traffic. With an in-depth analysis towards the implementation of these 6 icons, we confirm that all of them are true positives. Compared with the result of DeepIntent, IconChecker outperforms a much more excellent precision (100% vs. 6.67%).

Overall, IconChecker shows a much better capability on detecting the potential malicious icons, which can trigger illegal network traffic, than DeepIntent, in terms of our predefined 8 categories. Besides, please note that the task focused by IconChecker is a little different from DeepIntent's. IconChecker focuses on a more specific (i.e., 8 categories of icons, access the Internet) task. On the contrary, DeepIntent focuses on a wider variety of icon behaviors (e.g., access SD Card, send SMS).

***Answer to RQ5:*** *In terms of the 8 predefined categories, compared with DeepIntent, IconChecker can find more icons with abnormal behaviors (21 vs. 10) and produce fewer false positives (4 vs. 140) on 1,800 Android apps, achieving 84% precision.*

## IV. LIMITATIONS AND FUTURE WORK

This work aims to sniff the network traffic generated by the icon operation at runtime. The limitations of our work and future work include the following points: (1) We modify the properties of each activity to successfully launch more activities, but for activities that rely on data to launch, we can not obtain and test the icons in such activities. (2) Since the icons that can possibly generate traffic have more complex usage scenarios, it's impossible to determine whether their triggered program behaviors are normal or not with only the icon's semantics. Moreover, since the content of encrypted traffic can not be parsed, it's impossible to combine the icon semantics with traffic content. Hence, to solve this problem, we plan to involve more information, such as program semantics or UI layout context, in the future. (3) Our proposed work shows that the static and dynamic hybrid methods have an advantage in their precision of Android anomaly detection. We believe if the bottleneck on its usage scope can be extended to other runtime information except for network transmission, such as sending SMS, calling, it can become a usable and reliable systematic system in near future.

## V. Related Work

### A. Traffic Analysis of Android Apps

There has been a great deal of work to detect malware based on the characteristics of network traffic [19], [30]–[32]. Most of the work [19], [31] focused on using machine learning algorithms (e.g., BayesNet and C4.5) to detect malware. The work in [33] regarded traffic data as images and abnormal patterns, and then classified the abnormal patterns displayed by malware traffic through representation learning.

Traffic features also can be used to identify apps. Rao et al. [34] presented a system leveraging HTTP features to identify apps. AntMonitor [35] and AppScanner [36] leveraged TCP/IP headers for app identification. Recent work [37] built an APP-ID dataset by capturing actual network traffic. Furthermore, in other respects, Oulehla et al. [38] proposed to use a feed-forward neural network for mobile botnet detection.

However, none of them model traffic and icons to detect abnormal behaviors. Our work aims to detect icons' abnormal behaviors in Android apps. Specifically, we use the network traffic to represent the actual behaviors of the icon.

### B. GUI-Based Analysis of Android Apps

GUI testing checks whether the app's behavior is correct by executing events on the GUI. Existing works [39]–[41] by analyzing the event handler to detect crashes in apps. Moreover, Borges Jr and Nataniel P [42] implemented a tool to test the functionality of Android apps using the association of data flows and UI elements. Ki et al. [43] designed Mimic to compare the UI behaviors across different versions or environments. Recently, instead of testing apps, IconIntent [25] has been proposed to identify sensitive UI widgets in mobile apps. DeepIntent [20] adopts deep learning techniques to train a model which uses triples like $\langle image, text, permissions \rangle$ as input and then uses this pre-trained model to detect intention-behavior discrepancies. Different from the previous work, our purpose is to identify abnormal icon behaviors by establishing icon-traffic mapping. We conduct a comprehensive comparison between IconChecker and DeepIntent in (Section III-F).

## VI. Conclusion

In this paper, we design, implement, and evaluate an abnormal icon-behaviors detector (i.e., IconChecker) for Android apps. Specifically, we first propose a framework to accurately captures traffic generated by clicking the icon with 100% accuracy. Then we propose a progressive strategy to extract icon's semantics in Android apps for icon classification, achieving 87.2% accuracy. Overall, IconChecker achieves 84% precision.

## VII. Acknowledgement

This work has partially been sponsored by the National Science Foundation of China (No. 62102284, 61872262, 62102197). Yuxuan Li and Ruitao Feng are the co-first authors. Sen Chen and Xiaohong Li are the corresponding authors.

## References

[1] S. Chen, L. Fan, C. Chen, T. Su, W. Li, Y. Liu, and L. Xu, "Storydroid: Automated generation of storyboard for Android apps," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 596–607.

[2] A. Group, "Appbrain." http://www.appbrain.com/stats/, [Online; accessed 08-November-2020].

[3] A. Bhatiaa, A. A. Bahugunaa, K. Tiwaria, K. Haribabua, and D. Vishwakarmab, "A survey on analyzing encrypted network traffic of mobile devices," *arXiv preprint arXiv:2006.12352*, 2020.

[4] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *S&P*. IEEE, 2012.

[5] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket." in *NDSS*. ISOC, 2014.

[6] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: detecting malicious apps in official and alternative Android markets." in *NDSS*. ISOC, 2012.

[7] L. K. Yan and H. Yin, "Droidscope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic Android malware analysis," in *USENIX Security*. USENIX Association, 2012.

[8] 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 *PLDI*. ACM, 2014.

[9] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in Android apps," in *ICSE*. ACM, 2015.

[10] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras, "Droidminer: Automated mining and characterization of fine-grained malicious behaviors in Android applications," in *ESORICS*. Springer, 2014.

[11] S. Chen, M. Xue, Z. Tang, L. Xu, and H. Zhu, "Stormdroid: A streaminglized machine learning-based system for detecting Android malware," in *AsiaCCS*. ACM, 2016.

[12] E. Mariconti, L. Onwuzurike, P. Andriotis, E. D. Cristofaro, G. Ross, and G. Stringhini, "Mamadroid: Detecting Android malware by building markov chains of behavioral models," ISOC, 2016.

[13] S. Chen, M. Xue, L. Fan, S. Hao, L. Xu, H. Zhu, and B. Li, "Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach," *Computers & Security*, 2018.

[14] R. Feng, S. Chen, X. Xie, L. Ma, G. Meng, Y. Liu, and S.-W. Lin, "MobiDroid: A Performance-Sensitive Malware Detection System on Mobile Platform," in *ICECCS*. IEEE, 2019.

[15] R. Feng, S. Chen, X. Xie, G. Meng, S.-W. Lin, and Y. Liu, "A Performance-Sensitive Malware Detection System Using Deep Learning on Mobile Devices," *IEEE Trans.Inform.Forensic Secur.*, 2020.

[16] R. Feng, J. Q. Lim, S. Chen, S.-W. Lin, and Y. Liu, "SeqMobile: An Efficient Sequence-Based Malware Detection System Using RNN on Mobile Devices," in *ICECCS*. IEEE, 2020.

[17] B. Wu, S. Chen, C. Gao, L. Fan, Y. Liu, W. Wen, and M. R. Lyu, "Why an Android app is classified as malware? towards malware classification interpretation," *arXiv preprint arXiv:2004.11516*, 2020.

[18] M. Lotfollahi, M. J. Siavoshani, R. S. H. Zade, and M. Saberian, "Deep packet: A novel approach for encrypted traffic classification using deep learning," *Soft Computing*, 2020.

[19] I. J. Sanz, M. A. Lopez, E. K. Viegas, and V. R. Sanches, "A lightweight network-based android malware detection system," in *IFIP Networking*. IEEE, 2020.

[20] S. Xi, S. Yang, X. Xiao, Y. Yao, Y. Xiong, F. Xu, H. Wang, P. Gao, Z. Liu, F. Xu *et al.*, "Deepintent: Deep icon-behavior learning for detecting intention-behavior discrepancy in mobile apps," in *CCS*, 2019.
[21] 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 *CCS*. ACM, 2013.
[22] S. Impedovo, L. Ottaviano, and S. Occhinegro, "Optical character recognition—a survey," *IJPRAI*, 1991.
[23] segler.alex, "Radiodroid," https://f-droid.org/zh_Hans/packages/net.programmierecke.radiodroid2/, [Online; accessed 12-August-2020].
[24] "Network time protocol." https://en.wikipedia.org/wiki/Network_Time_Protocol, [Online; accessed 25-April-2021].
[25] X. Xiao, X. Wang, Z. Cao, H. Wang, and P. Gao, "Iconintent: automatic identification of sensitive ui widgets based on icon classification for android apps," in *ICSE*. IEEE, 2019.
[26] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *IJCV*, 2004.
[27] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, 1966.
[28] S. Chen, M. Xue, and L. Xu, "Towards adversarial detection of mobile malware: poster," in *MobiCom*. ACM, 2016.
[29] "Androidxref," http://androidxref.com/8.0.0_r4/xref/frameworks/base/services/core/java/com/android/server/connectivity/, [Online; accessed 19-January-2021].
[30] A. Arora and S. K. Peddoju, "Ntpdroid: a hybrid android malware detector using network traffic and system permissions," in *TrustCom*. IEEE, 2018.
[31] J. G. de la Puerta, I. Pastor-López, B. Sanz, and P. G. Bringas, "Network traffic analysis for android malware detection," in *HAIS*. Springer, 2019.
[32] S. Wang, Z. Chen, Q. Yan, K. Ji, L. Peng, B. Yang, and M. Conti, "Deep and broad url feature mining for android malware detection," *Information Sciences*, 2020.
[33] W. Wang, M. Zhu, X. Zeng, X. Ye, and Y. Sheng, "Malware traffic classification using convolutional neural network for representation learning," in *ICOIN*. IEEE, 2017.
[34] A. Rao, A. M. Kakhki, A. Razaghpanah, A. Tang, S. Wang, J. Sherry, P. Gill, A. Krishnamurthy, A. Legout, A. Mislove *et al.*, "Using the middle to meddle with mobile," *CCIS*, 2013.
[35] A. Le, J. Varmarken, S. Langhoff, A. Shuba, M. Gjoka, and A. Markopoulou, "Antmonitor: A system for monitoring from mobile devices," in *C2B(I)D*. ACM, 2015.
[36] V. F. Taylor, R. Spolaor, M. Conti, and I. Martinovic, "Appscanner: Automatic fingerprinting of smartphone apps from encrypted network traffic," in *EuroS&P*. IEEE, 2016.
[37] X. Wang, S. Chen, and J. Su, "Real network traffic collection and deep learning for mobile app identification," *WCMC*, 2020.
[38] M. Oulehla, Z. K. Oplatková, and D. Malanik, "Detection of mobile botnets using neural networks," in *FTC*. IEEE, 2016.
[39] W. Song, X. Qian, and J. Huang, "Ehbdroid: beyond gui testing for android applications," in *ASE*. IEEE, 2017.
[40] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of android apps," in *FSE*. ACM, 2017.
[41] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, G. Pu, and Z. Su, "Large-scale analysis of framework-specific exceptions in android apps," in *ICSE*. IEEE, 2018.
[42] N. P. Borges Jr, "Data flow oriented ui testing: exploiting data flows and ui elements to test android applications," in *ISSTA*. ACM, 2017.
[43] T. Ki, C. M. Park, K. Dantu, S. Y. Ko, and L. Ziarek, "Mimic: Ui compatibility testing system for android apps," in *ICSE*. ACM, 2019.