DeepIntent: Deep Icon-Behavior Learning for Detecting Intention-Behavior Discrepancy in Mobile Apps

Shengqu Xi[1,*] Shao Yang[2,*] Xusheng Xiao[2] Yuan Yao[1] Yayuan Xiong[1] Fengyuan Xu[1]
Haoyu Wang[3] Peng Gao[4] Zhuotao Liu[5] Feng Xu[1] Jian Lu[1]
[1]State Key Lab for Novel Software Technology, Nanjing University [2]Case Western Reserve University
[3]Beijing University of Posts and Telecommunications [4]University of California, Berkeley
[5]University of Illinois at Urbana-Champaign
{xsq,yayuan.xiong}@smail.nju.edu.cn, {sxy599, xusheng.xiao}@case.edu, {y.yao, fengyuan.xu, xf, lj}@nju.edu.cn
haoyuwang@bupt.edu.cn, penggao@berkeley.edu, zliu48@illinois.edu

ABSTRACT

Mobile apps have been an indispensable part in our daily life. However, there exist many potentially harmful apps that may exploit users' privacy data, e.g., collecting the user's information or sending messages in the background. Keeping these undesired apps away from the market is an ongoing challenge. While existing work provides techniques to determine what apps do, e.g., leaking information, little work has been done to answer, *are the apps' behaviors compatible with the intentions reflected by the app's UI*?

In this work, we explore the *synergistic cooperation of deep learning and program analysis* as the first step to address this challenge. Specifically, we focus on the UI widgets that respond to user interactions and examine whether the intentions reflected by their UIs justify their permission uses. We present DeepIntent, a framework that uses novel *deep icon-behavior learning* to learn an icon-behavior model from a large number of popular apps and detect *intention-behavior discrepancies*. In particular, DeepIntent provides *program analysis* techniques to associate the intentions (i.e., icons and contextual texts) with UI widgets' program behaviors, and infer the labels (i.e., permission uses) for the UI widgets based on the program behaviors, enabling the construction of a large-scale high-quality training dataset. Based on the results of the static analysis, DeepIntent uses *deep learning* techniques that jointly model icons and their contextual texts to learn an icon-behavior model, and detects intention-behavior discrepancies by computing the outlier scores based on the learned model. We evaluate DeepIntent on a large-scale dataset (9,891 benign apps and 16,262 malicious apps). With 80% of the benign apps for training and the remaining for evaluation, DeepIntent detects discrepancies with AUC scores 0.8656 and 0.8839 on benign apps and malicious apps, achieving 39.9% and 26.1% relative improvements over the state-of-the-art approaches.

*The first two authors contributed equally to this research.

CCS '19, November 11–15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3363193

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**; **Software security engineering**.

KEYWORDS

mobile apps; discrepancy detection; static analysis; deep learning

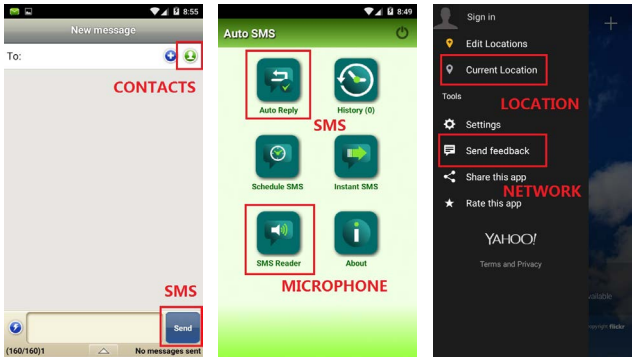
ACM Reference Format:
Shengqu Xi[1,*] Shao Yang[2,*] Xusheng Xiao[2] Yuan Yao[1] Yayuan Xiong[1] Fengyuan Xu[1] and Haoyu Wang[3] Peng Gao[4] Zhuotao Liu[5] Feng Xu[1] Jian Lu[1] . 2019. DeepIntent: Deep Icon-Behavior Learning for Detecting Intention-Behavior Discrepancy in Mobile Apps. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS'19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3319535.3363193

1 INTRODUCTION

Mobile apps are playing an increasingly important role in our daily life, from travel, education, to even business [50, 87]. While these apps use users' personal information to provide better services, certain behaviors of the apps are less desirable or even harmful. Example undesired behaviors include disclosing users' sensitive data such as location [18, 43, 61, 92] without expressing the intentions to use it, and stealthily exploiting users' private resources for advertising [41, 42, 75].

However, detecting such apps is challenging, since *undesired behaviors* appear to be indistinguishable from the behaviors of benign apps. For example, apps recommending restaurants use users' GPS data to suggest the nearby restaurants, and apps providing travel planning services let users make phone calls or send messages. As such, the *permission-based access control mechanism* employed by popular smartphone platforms (i.e., Android and iOS) [68], has shown little success [9, 19, 20]. For example, users can disallow an app to share the GPS data by not granting the GPS-related permissions; however, it is a difficult decision as many benign apps do need to use the GPS data.

To detect the undesired behaviors in mobile apps, we are motivated by the vision: *can the compatibility of an app's intentions and program behaviors be used to determine whether the app will perform within the user's expectation*? In other words, as the user-perceivable information of apps' UIs (i.e., texts and images) represent users' expectation of apps' behaviors [33] (i.e., apps' intentions), we aim to automatically check the compatibility between apps' intentions and their behind-the-scene behaviors, i.e., detecting *intention-behavior*



(a) **(b)** **(c)**

Figure 1: Example icon widgets that access sensitive information.

discrepancies. For example, if a music player app's button shows a "+" icon, it indicates that clicking the button will add a song to the playlist. However, if the app discloses users' GPS data when the button is pressed, red flags should be raised.

In this work, we focus on detecting the intention-behavior discrepancies of interactive UI widgets in Android apps[1], which express their intentions via texts or images and respond to users' interactions (e.g., clicking a button). Specifically, we focus on the interactive UI widgets that use icons to indicate their expected behaviors, referred to as *icon widgets*, since icon widgets are prevalent in apps and many of them access sensitive information [80]. Figure 1 shows the UI screenshots that contain example icon widgets in which their icons and texts express their intentions in performing sensitive behaviors: Figure 1(a) shows icon widgets that use pure icons and icons embedded with texts; Figure 1(b) shows icon widgets that use both icons and texts, but the texts do not explicitly explain their intentions; Figure 1(c) shows icon widgets that use both icons and texts, and the texts help explain their intentions.

Checking the compatibility between the icon widgets' intentions and their behaviors is a challenging task. First, their intentions are expressed mainly via a mixture of icons and texts, and it is difficult to model such correlations using these unstructured artifacts. Existing approaches have either modeled the text semantics to detect undesired disclosures of sensitive user inputs through UIs [4, 31, 51], or classified the icons using computer vision techniques to detect sensitive UI widgets [80]. However, none of them have modeled the joint semantics of both icons and their texts. Second, Android's UI design model and the asynchronous programming model pose challenges to precisely identify sensitive behaviors of an icon widget. Android apps may associate UI handlers[2] with icon widgets via UI layout files or code. Also, UI handlers may invoke sensitive APIs via Android's multi-threading [86] and Inter-Component Communication [37, 54]. Existing approaches either produce high false positives due to enumerating all possible combinations of lifecycle methods [5, 79], or fail to identify certain behaviors due to low coverage [24, 25, 48, 69]. Third, it is difficult to correlate an app's intention and behavior to determine whether the behavior is undesired. Existing research efforts have been put forth to detect undesired disclosures of sensitive user inputs through UIs [4, 31–33, 51]. However, the resulting behavior patterns from these approaches can capture only a fixed set of undesired behaviors. Furthermore, a behavior of a UI widget often uses several permissions. Existing prediction-based approaches [33, 80] mainly focus on predicting a single permission use based on intentions, and such lack of modeling multiple permission uses renders the prediction less effective in detecting intention-behavior discrepancies.

Contributions. Towards realizing the vision, we propose to build a novel framework, DeepIntent, that learns an icon-behavior model from a large number of apps, and uses the model to detect undesired behaviors[3]. In particular, DeepIntent explores the synergistic cooperation of *deep learning* and *program analysis* as the first step to address the above challenges in Android apps: (1) *Deep Intention Modeling*: following the success of deep learning [21, 27, 28, 35, 36] in modeling unstructured artifacts such as texts and images, DeepIntent uses *deep learning* to model apps' intentions that are reflected mainly by the unstructured information (i.e., icons and texts) and predict expected behaviors; (2) *Traceability and Label Inference*: the power of deep learning highly depends on the large-scale high-quality labeled data [1, 21], and simply modeling all the code as part of the features without deeper analysis on the code introduces too much noise into the training data, rendering the deep learning less effective. As such, DeepIntent leverages *program analysis* techniques to associate the intentions with the program behaviors, and infer the labels for the icon widgets based on the program behaviors (e.g., whether the behaviors accessing sensitive data), enabling the construction of a large-scale high-quality training dataset. Such synergy of program analysis and deep learning enables building an icon-behavior model from a large number of apps and exposing intention-behavior discrepancies based on the model.

The design of DeepIntent is based on three key insights. First, mobile apps' UIs are expected to be evident to users, and icons indicating the same type of sensitive behavior should have similar looks. This inspires us to follow the success of CNN [35, 36] in image recognition and model the icons (i.e., pixels of the icons) using CNN to identify similar icons. Second, in different UI contexts, icons may reflect different intentions. For example, a "send" button may mean sending an email or an SMS in different contexts. While it is difficult to differentiate the intentions by just comparing the icons, the contextual texts, such as the nearby text labels and the header of the UI, can be used to help distinguish the contexts of the icons. Third, users expect certain behaviors when interacting with icon widgets that have specific looks, and undesired behaviors usually contradict users' expectations. For example, when users look at the first highlighted icon in Figure 1(a), they are expecting the app to read their contacts, but not disclosing their location information. To capture such general expectation, we propose to develop program analysis techniques that can associate icons to their sensitive behaviors, and apply the techniques to extract the associations from a corpus of popular apps to learn models on expected behaviors for icon widgets with specific looks. Such model

[1]While our work focuses on Android apps due to its popularity, the findings can be generalized to other mobile platforms such as iOS.
[2]A UI handler is the method to be invoked when a user interacts with the icon widget.

[3]DeepIntent is publicly available at https://github.com/deepintent-ccs/DeepIntent.

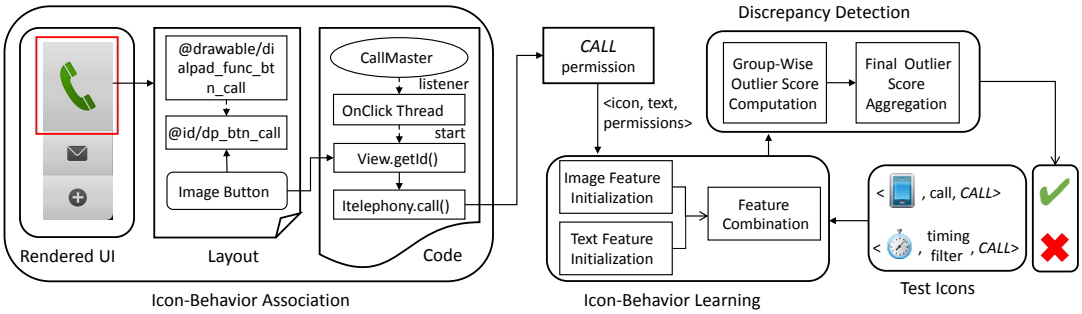


Figure 2: Motivating example of DeepIntent.

can then be used to detect abnormal behaviors as intention-behavior discrepancies. In particular, we use *permission uses* to summarize icon widgets' sensitive behaviors (i.e., sensitive APIs invoked) [7, 17, 83], since undesired behaviors need to request permissions to access sensitive information.

Based on these key insights, DeepIntent provides a novel learning approach, *deep icon-behavior learning*, which consists of three major phases.

Icon Widget Analysis. The input used in our learning model consists of icons, contextual texts, and the permission uses associated with the icons. To extract the icons and their permission uses, DeepIntent provides a static analysis that analyzes APK files to identify icon widgets and extract corresponding icon-permission mappings, i.e., mapping the icons used in the UI widgets to their permission uses. Specifically, the static analysis (1) associates icons with UI widgets by analyzing both UI layout files and code, (2) associates icon widgets with UI handlers, (3) builds call graphs for UI handlers by considering multi-threading and ICCs, and (4) maps method calls in call graphs to permission uses. From the extracted icons, DeepIntent provides a text extraction technique that extracts contextual texts for the icons by analyzing UI layout files, embedded texts in the icons, and icon file names.

Learning Icon-Behavior Model. DeepIntent adopts a parallel co-attention mechanism [47, 90] to jointly model icons and their contextual texts. Specifically, DeepIntent first uses DenseNet [30] and GRUs [10] to extract the initialized features for icon images and contextual texts, respectively. DeepIntent then combines these two features into a joint feature vector via co-attention, whose basic idea is to simultaneously update the image/text features by highlighting the image/text regions that are relevant to each other. Next, DeepIntent learns the joint feature vector for an icon by training the model with the mapped permissions for icons. Since each icon may relate to multiple permission uses, we formulate a multi-label classification problem to learn the joint features.

Detecting Intention-Behavior Discrepancies. With the learned icon-behavior model, given an icon widget, DeepIntent first extracts its joint feature vector, and then detects the intention-behavior discrepancies by computing and aggregating the outlier scores from each permission used by the icon widget. Specifically, we compute the outlier score for each used permission via AutoEncoder [3], and aggregate these scores to form the final outlier score based on the icon-behavior model. The actual permission uses are obtained by the program analysis used for extracting icon-permission mappings.

Results. We collect a set of 9,891 benign apps and 16,262 malicious apps, from which we extract over 10,000 icon widgets that are mapped to sensitive permission uses. We use 80% of the icons from the benign apps as training data, and detect the intention-behavior discrepancies on the remaining icons from the benign apps and all the icons from malicious apps. For the test set, we manually label whether there is an intention-behavior discrepancy to form the ground truth. Finally, DeepIntent returns a ranked list based on the outlier scores for detecting intention-behavior discrepancies.

The results demonstrate the superior performance of the proposed DeepIntent. First, our joint modeling of icons' image and text features is effective in terms of predicting their permission uses. Compared to the state-of-the-art sensitive UI widget identification approach, IconIntent [80], that relies on traditional computer vision techniques, DeepIntent achieves at least 19.3% relative improvement in different permissions. DeepIntent is also better than its sub-variants when only icons' image or text features are used. This result indicates the generalization ability of the proposed deep learning techniques for the joint feature learning. Second, our static analysis is essential to accurately extract icon-permission mappings for the learning of the icon-behavior model. For example, DeepIntent achieves 70.8% relative improvement on average compared to the learning approach without static analysis. Third, DeepIntent can detect discrepancies with AUC values 0.8656 and 0.8839 for benign apps and malicious apps. For malicious apps, DeepIntent can successfully identify over 85% discrepancies in most cases. The state-of-the-art approach, IconIntent [80], is originally proposed for predicting permission uses, and we extend it to the discrepancy detection setting by feeding its features into the proposed outlier detection module. The results show that DeepIntent achieves 39.9% and 26.1% relative improvements in terms of AUC values compared to IconIntent on benign apps and malicious apps, respectively.

2 MOTIVATING EXAMPLE

To motivate DeepIntent, we present an example in Figure 2. The rendered UI with sensitive buttons are from the app *Smart Dialog*. Consider the phone call button as an example. DeepIntent extracts the resource ID from the UI layout file, analyzes the code that handles the button, builds the call graphs, and maps the button to its permission uses (i.e., the CALL permission). The output of the icon-behavior association is a set of $\langle icon, text, permissions \rangle$ triples. Next, DeepIntent learns an icon-behavior model using the set of triples from popular benign apps, with the assumption that most

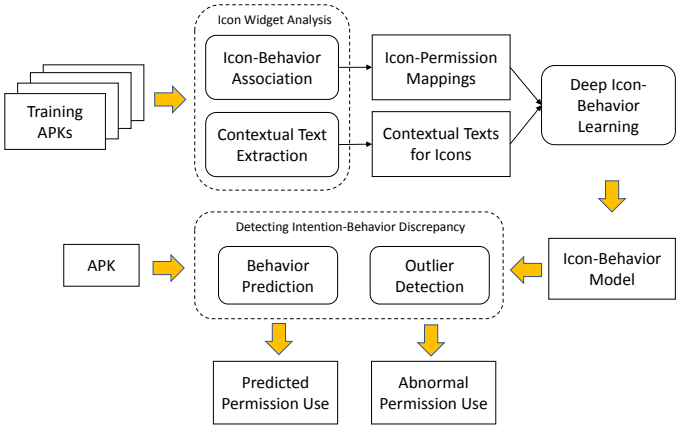


Figure 3: Overview of DeepIntent.

of the icons in these apps use the icons and permissions properly, capturing the general expectation of users. With the learned icon-behavior model, DeepIntent also trains a discrepancy detection model which can be used to compute the outlier scores for test icons. Then, for a button whose intention-behavior discrepancy is to be checked, DeepIntent adopts the same static analysis to extract the $\langle icon, text, permissions \rangle$ triple, and feeds the triple into the icon-behavior models (i.e., icon-behavior learning and discrepancy detection) to determine whether there are any discrepancies between the intentions (represented using icons and contextual texts) and the permission uses. In this example, it is expected for the first button ('call') to use the CALL permission, while there is a discrepancy for the second button ('timing filter') to use the CALL permission.

3 DESIGN OF DEEPINTENT

3.1 Overview

Figure 3 shows the overview of DeepIntent. DeepIntent consists of three phases: (1) icon widget analysis, (2) learning icon-behavior model, and (3) detecting intention-behavior discrepancies.

The first phase accepts a training dataset of Android APK files as input, and extracts features (i.e., icons and texts) and labels (i.e., permission uses) of icon widgets. Specifically, the icon-behavior association module applies static analysis techniques to identify the icons used in UI widgets, associates the icons with UI handlers, and infers the icon-permission mappings based on the API-permission mappings. The contextual text extraction module extracts the contextual texts for the identified icons.

Based on the output of icon widget analysis, the deep icon-behavior learning module uses both icons and their contextual texts as features, and the corresponding behaviors, i.e., permission uses, as labels to train the icon-behavior model. In particular, this module uses a parallel co-attention mechanism that can learn the joint features from both icons and their contextual texts.

In the next phase, DeepIntent extracts the icon and text features for each icon widget, predicts the permission uses for the icon widgets, and detects abnormal permission uses. Specifically, given

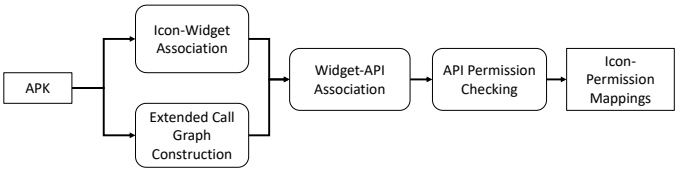


Figure 4: Workflow of icon-behavior association module.

an icon used in a UI widget and its contextual text, the behavior prediction module uses the trained icon-behavior model to predict its permission uses. Also, based on the actual permission uses obtained from our static analysis techniques (the same techniques used for processing training APKs), the outlier detection module uses the trained icon-behavior model to determine whether the permission uses are abnormal, i.e., detecting icon-behavior discrepancies.

3.2 Threat Model

DeepIntent is a UI analysis tool that detects intention-behavior discrepancies for icon widgets. Rather than focusing on malicious behaviors that deliberately evade detection (i.e., camouflaged as normal behaviors), DeepIntent is designed to determine whether the behavior of an icon widget matches the intentions reflected by the user-perceivable information in the UIs, i.e., whether the UIs provide justifications for the behaviors. While some of the underlying data flows may not be intuitively reflected by the UI information, such as disclosing contacts, if many apps with similar UIs have such behaviors, DeepIntent will still be able to capture such compatibility in the model.

Note that most apps in the app markets are legitimate, whose developers design the apps to meet users' requirements, even though some of them may be aggressive on exploiting user privacy for revenue. For certain third-party app markets that may be flooded with malicious apps, the training quality may be affected. In that case, anti-virus techniques and malware detection techniques should be applied to remove such apps from the training dataset. Malicious apps that deliberately evade detection can be detected by special techniques [7, 16, 83, 93], which is out of scope of this paper.

4 ICON-BEHAVIOR ASSOCIATION

This module provides static analysis techniques to identify icon widgets, extracts their icons and texts, and infers the permission uses of the icon widgets. It plays a key role in learning an icon-behavior model, since it enables the construction of a large-scale high-quality training dataset. Our techniques analyze both UIs and source code to associate icons/texts and handlers to icon widgets. Particularly, we build extended call graphs to patch missing calling relationships introduced by the Android environment, and use the extended call graphs to identify APIs invoked by the UI widgets.

4.1 Static Analysis Overview

This module contains four major components: 1) Icon-Widget Association, 2) Extended Call Graph Construction, 3) Widget-API Association, and 4) API Permission Checking, as shown in Figure 4. The first two components take an Android APK file as input. The

```
<LinearLayout android:orientation="vertical" ...>
  <RelativeLayout android:id="@+id/RelativeSearch" >
    <ImageView android:id="@+id/ImageViewLocation" android:src="
      @drawable/ic_location" ... />
    <EditText android:id="@+id/TxtCity" ... />
    <Button android:text="@string/search" .../>
  </RelativeLayout>
  <LinearLayout android:id="@+id/LinearLayout02" ...>
    <TextView android:text="*" />
    <TextView  android:text="@string/select_city" />
  </LinearLayout>
  <ListView android:id="@+id/TxtList"  />
</LinearLayout>
```

Figure 5: Simplied UI layout file (search.xml) for Animated Weather App.

```
1  public class SearchForm extends Activity {
2    public void onCreate(Bundle savedInstanceState) {
3      setContentView(R.layout.search); // bound to Figure 3
4      ((ImageView) findViewById(R.id.ImageViewLocation)).
         setOnClickListener(new OnClickListener {
5          public void onClick(View v) {startAsincSearch;} });
6      ... } // bound to OnClick handler
7
8    private void startAsincSearch {
9      ...
10     searchThread = new LocationThread;
11     searchThread.start; // bound to LocationThread.run
12     ... } } // end of class SearchForm
13
14 class LocationThread extends Thread {
15   ...
16   public void run {
17     ManagerOfLocation.findPosition; // use GPS data
18     ... } }
19
```

Figure 6: Example sensitive API in multi-threading.

output of the Icon-Widget Association component is a mapping between icons and their corresponding UI widgets. The output of the Extended Call Graph Construction component is an extended call graph of the entire app. Then these outputs are used as the inputs for the Widget-API Association component, which associates the UI handlers with the UI widgets and constructs the corresponding call graphs for each UI handler. At last, each method call in the call graphs is associated with the corresponding permission uses based on the PScout [6] Android API-permission mappings. The output of the icon-behavior association module is the icon-permission mappings that map all the extracted icons to their corresponding permission uses. We next present the details of each component.

4.2 Icon-Widget Association

In Android apps, icons can be associated with UI widgets by specifying in the UI layout files or in the source code. Each UI layout, widget, and icon has its own unique ID. UI layout files are loaded through API calls like `setContentView` or `inflate` in activities at runtime. Then UI widgets in the layout can be bound to variables via API calls such as `findViewByID` using UI widgets' IDs. Icons can be associated with UI widgets directly in UI layout files as well. Figure 5 shows a simplified UI layout file for the Animated Weather app. The UI widget `ImageView` at Line 3 associates an icon to the widget via the attribute `android:src`.

We adopt the static analysis of IconIntent [80], the state-of-the-art sensitive UI widget identification approach, to associate icons to the UI widgets. IconIntent performs static analysis on both the UI layout files and the source code to infer the associations between icons and UI widgets. The static analysis on UI layout files parses the UI layout files and identifies the names of the app's icons (such as `@drawable/ic_location`) and the UI widgets with IDs, such as `ImageView` in Figure 5. The analysis on app's code provides a data flow analysis to overapproximate the associations between variables and UI widget IDs, and the associations between variables and icon IDs. Then IconIntent combines the analysis results on both the UI layout files and the code to determine which UI widgets are associated with which icons (many-to-many mappings).

4.3 Extended Call Graph Construction

Android app executions follow the event-driven execution model. When a user navigates through an app, the Android framework triggers a set of lifecycle events and invokes lifecycle methods such as `onCreate` and `onResume`; when a user interacts with the app's UIs, the Android framework triggers a set of UI interaction events and invokes the corresponding callback methods such as `onClick`. Furthermore, multi-threaded communications split the execution into executions in both foreground and background. Thus, to determine which behavior is triggered (i.e., which APIs are invoked), DeepIntent builds a static call graph for each UI handler.

Figure 6 shows an example app that requests a user's GPS location via multi-threading. When the UI widget (`ImageViewLocation`) is clicked at Line 5, `startAsincSearch` is invoked. Then a new thread is initiated and started for `startAsincSearch` at Line 10 and Line 11. At last, a sensitive API that requires the location permission is invoked in `LocationThread.run` at Line 17. In other words, in addition to the explicit calling relationships established via calling statements, there exist implicit calling relationships between `setOnClickListener` and `onClick`, as well as `LocationThread.start` and `LocationThread.run`. In addition to multi-threading, sensitive APIs may be invoked via a service or a broadcast receiver using Inter-Component Communications (ICC) [37, 54].

These implicit calling relationships pose challenges for inferring the APIs invoked when the UI handler is triggered. Existing work [5, 79] has proposed techniques to address these challenges in building call graphs. However, they often assume every possible combination of lifecycle methods (e.g., `onCreate` and `OnResume`), multi-threading methods, and ICC methods, resulting in exhaustive calling contexts. To find APIs invoked by a UI handler, such exhaustive calling contexts result in a large number of false associations between UI widget and sensitive APIs.

To address this problem, we propose a static analysis technique to patch these missing calling relationships without exhausting the lifecycle method calls. Specifically, our static analysis first leverages existing static call graph techniques to build a call graph based on calling statements, and then expands the static call graph with edges representing implicit calling relationships. In particular, our analysis includes four types of implicit calling relationships that are most commonly used in Android apps, including multi-threading, lifecycle method, event-driven method, and inter-component communication (ICC). Table 1 shows the implicit calling relationships

Caller	Callee
`setOnClickListener`	`onClick`
`Thread.start`	`Thread.run`
`AsyncTask.execute`	`doInBackground`
	`onPreExecute`
	`onPostExecute`
`sendMessage`	`handleMessage`

Table 1: Implicit caller and callee pairs captured.

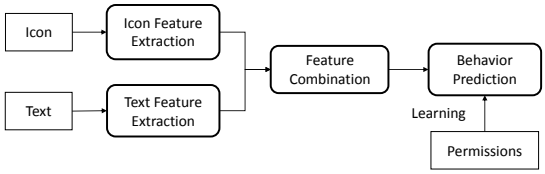


Figure 7: Workflow of Deep Icon-Behavior Learning.

used by our analysis, except for ICC methods. Our analysis leverages existing ICC analysis [37, 54] to identify the implicit calling relationships for ICCs and create edges in the call graph for them.

4.4 Widget-API Association

This component aims to associate the UI widgets with their UI handlers and construct the call graphs for the UI handlers. We adapt the existing Android static analysis tool, GATOR [63, 81, 82], to associate the UI handlers with UI widgets. GATOR applies static analysis to identify UI widgets and UI handlers, and provides an over-approximation dataflow algorithm to infer the associations between the event handler methods and the UI widgets. Our approach then combines the output from GATOR with the output from the Icon-Widget Association component to build the associations among icons, layout files, UI widgets, and UI handlers. For example, in Figure 6, the UI widget `ImageViewLocation` is bound to the layout file in Figure 5, its icon is *lc_location* (Line 3 in Figure 5), and it is associated to the UI handler `SearchForm.onClick`. Note that there can be multiple handlers for one UI widget.

Once each widget is associated to icons and UI handlers, our approach then generates a call graph for each of its UI handlers. A call graph of a UI handler is a subgraph of the extended call graph for the entire app, which contains the nodes that are reachable from the UI handler. DeepIntent then finds API uses for the UI widget by searching the API calls inside the call graph.

4.5 API Permission Checking

This component maps the APIs found in the extended call graph of each icon widget to permission uses based on PScout [6], a widely-used permission mapping from Android APIs to Android permissions. This component outputs the associations between each icon and a set of permissions. Note that an icon widget can be mapped to one or more permission uses since it may invoke multiple sensitive APIs or some sensitive APIs are mapped to multiple permissions.

4.6 Contextual Texts Extraction for Icons

As mentioned in introduction, similar icons may reflect different intentions in different UI contexts. While it is difficult to differentiate them by just comparing the icons, the contextual texts, such as the nearby text labels and the header of the UI, can be used to help distinguish the contexts of the icons. Thus, DeepIntent further provides a contextual text extraction component that extracts the contextual texts for each icon. Specifically, DeepIntent extracts three types of contextual texts: (1) layout texts that are contained in the XML layout files, (2) icon-embedded texts which can be extracted by Optical Character Recognition (OCR) techniques, and (3) resource names split by variable naming conventions. The final output of our static analysis is a set of $\langle icon, text, permissions \rangle$ triples.

5 DEEP ICON-BEHAVIOR LEARNING

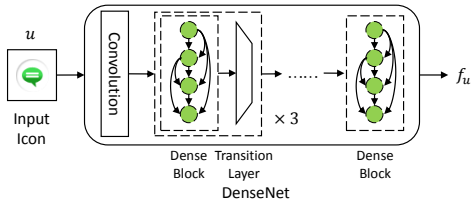
Using the output from the previous module (i.e., $\langle icon, text, permissions \rangle$ triples), DeepIntent leverages the co-attention mechanism to jointly model icons and texts, and trains an icon-behavior model that predicts the permission uses of icon widgets.

5.1 Model Overview

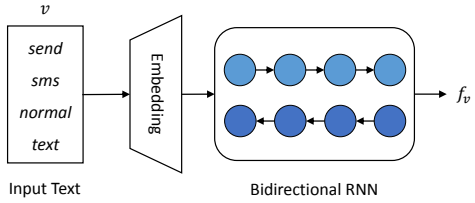
The overview of the this module is shown in Figure 7. Each piece of input consists of an icon and its text. As an initialization step to simultaneously learning their joint features, we need to first feed icons and texts into their respective feature extraction components. More specifically, we adapt DenseNet layers to extract icon features and bidirectional RNN layers to extract text features. We then combine them into a joint feature vector by using co-attention layers. The intuition is as follows. On one hand, the icon and its text could be semantically correlated; consequently, although we can simply concatenate the icon features and text features, the correlation between these two sources would be ignored, making the final representations of the input icon-text pair sub-optimal. On the other hand, the recently proposed co-attention layers can simultaneously update the icon features and the text features with the guidance of each other, and thus can capture the correlations between icons and texts. Next, since each icon may relate to multiple permissions, we formulate a multi-label classification problem [72] to learn the joint features. With the mapped permissions for icons, DeepIntent trains an icon-behavior model in an end-to-end manner.

5.2 Icon Feature Extraction

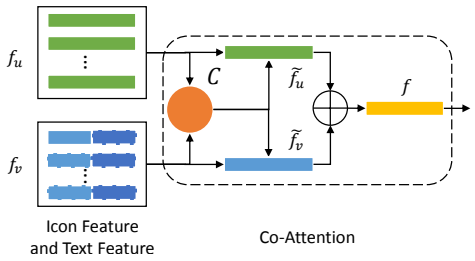
Each icon can be treated as a color tensor with fixed width, height, and color channels. In this work, we adapt DenseNet [30], the state-of-the-art image feature extraction model, to initialize the icon features. Typically, DenseNet contains several dense blocks and transition layers. We do not directly use the pre-trained DenseNet model for two reasons. First, the pre-trained model is trained on datasets such as CIFAR [34] and ImageNet [15], which contain natural images taken from cameras, while icons are mostly artifacts. Second, the pre-trained model considers images with 3 channels (RGB) and ignores the alpha channel that describes the opacity of the image. However, many icons use the alpha channel, and ignoring this channel might mislead the learned features [80].



(a) Icon feature extraction.



(b) Text feature extraction.



(c) Feature combination.

Figure 8: Structure of the icon-behavior model.

Therefore, we design our own DenseNet with 4 channels (RGBA) to extract icon features, as shown in Figure 8 (a). Our DenseNet starts with a convolutional layer, followed by four dense blocks and three transition layers between these dense blocks. For all the icons, we resize them to 128×128 (most icons are within this size). For an icon u, the output of our DenseNet is a $16 \times 16 \times 68$ tensor, which means that there are $M = 16 \times 16$ regions of the icon each of which is represented via a $d_u = 68$ dimensional vector, i.e.,

$$f_u = DenseNet(u), \tag{1}$$

where $u \in R^{128 \times 128 \times 4}$ is the color tensor of the input icon (with width 128, height 128, and channel number 4), and $f_u \in R^{d_u \times M}$ contains M regional feature vectors each of which is of d_u dimension. In practice, we further add a fully connected layer after the DenseNet to convert each regional feature vector into a new vector that has the same dimension with the text feature vectors.

5.3 Text Feature Extraction

To initialize the text features, we employ the state-of-the-art bidirectional RNNs to extract the text features, whose structure is shown in Figure 8 (b). For an input text v, i.e., a sequence of words, we first embed each word into a vector v_i, and then feed these vectors into the RNNs with GRU neurons [10],

$$\begin{aligned} \vec{h}_i &= GRU(v_i, \vec{h}_{i-1}) \\ \overleftarrow{h}_i &= GRU(v_i, \overleftarrow{h}_{i+1}) \end{aligned} \tag{2}$$

where $\vec{h}_i, \overleftarrow{h}_i \in R^{d/2}$ are the forward and backward features of v_i, respectively, and $d/2$ is dimension of the forward/backward feature vector. We concatenate forward and backward feature vectors into the overall feature vector for an input word, i.e., $h_i = [\vec{h}_i; \overleftarrow{h}_i]$. By setting the maximum length of text to N, we can obtain N feature vectors of the text as $f_v = [h_1, .., h_N]$ where $h_i \in R^d$. For text feature initialization, we can also directly adopt word embedding models [49, 57]. However, such models tend to ignore the order of the words in the sentence. In this work, we set $d = 100$ and $N = 20$ as the length of most surrounding texts are within this limitation.

5.4 Feature Combination

With the extracted icon features and text features, we next combine them to obtain the joint feature vectors. The key idea is to apply the parallel co-attention mechanism [47, 90] to bidirectionally update the icon features and text features with the guidance of each other, as shown in Figure 8 (c). Take the direction from icon features to text features as an example. Here, the attention improves the text feature vector by highlighting the words that are relevant to the image.

Specifically, with the extracted icon features f_u and text features f_v, we first define the correlation matrix $C \in R^{N \times M}$ as follows,

$$C = tanh(f_v^T W_c f_u) \tag{3}$$

where $W_c \in R^{d \times d}$ contains the parameters to be learned. Note that, there are M regional feature vectors for an icon and N feature vectors for its text. Consequently, this C matrix contains the similarities/correlations between $M \times N$ pairs of feature vectors.

Based on the above correlation matrix, we can connect the icon features and text features by transferring the features for each other. In particular, we use the following equations to update the icon features with the guidance of text features,

$$\begin{aligned} H_u &= tanh(W_u f_u + (W_v f_v)C) \\ a_u &= softmax(W_h H_u) \\ \tilde{f}_u &= \sum_{i=0}^{M} (a_u^{(i)} f_u^{(i)}) \end{aligned} \tag{4}$$

where $\tilde{f}_u \in R^d$ contains the updated icon feature vector, $W_v, W_u \in R^{k \times d}, W_h \in R^{1 \times k}$ are parameters, and k is the dimension size of these parameters. In the above equations, $H_u \in R^{k \times M}$ stands for the feature matrix obtained by transforming the text features (i.e., f_v) into icon features through the correlation matrix C, and a_u stands for the importance/weights of each regional feature vector to the final icon feature vector. Analogously, we can update the text features from f_v to $\tilde{f}_v$ and the equations are omitted for brevity. In practice, we update the icon features and text features in parallel.

With the updated icon features and text features, the combined feature vector $f \in R^d$ for an icon and its text is computed as follows.

$$f = \tilde{f}_u + \tilde{f}_v \tag{5}$$

5.5 Training

Given the above design, the next step is to train the icon-behavior model to connect the features to the used permissions. We formulate it as a multi-label prediction problem to allow one icon to match multiple permissions. Since the prediction of each permission can be

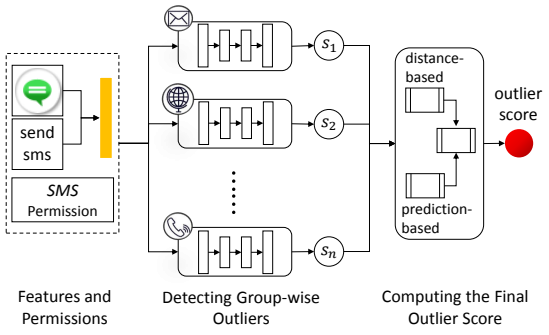


Figure 9: Workflow of detecting intention-behavior discrepancy.

formulated as a binary classification problem, we use the *sigmoid* function in logistic regression [29] to predict whether the icon matches each of the permissions. Next, since multiple permissions used by an icon can be treated as a probability distribution, we employ the *binary cross entropy* [21] as our loss function to measure the differences between the predicted permissions and the real ones obtained by our static analysis. The detailed equations are omitted for brevity. With the trained icon-behavior model, we can easily obtain the joint feature vector f of a test icon by feeding its icon image and contextual text into the model.

6 DETECTING INTENTION-BEHAVIOR DISCREPANCY

Based on the learned icon-behavior model, we next detect the icon-behavior discrepancies via identifying the permission-based outliers. For example, if the feature vector of an icon is far away from that of normal icons with the same permissions, there might be a mismatch between the icon intention and its behavior. Although we can directly use the icon-behavior model to predict the permission uses for each icon and then detect the outliers based on the prediction results, we deliberately add an outlier detection module for the following two reasons. First, directly using the prediction results would be less accurate as neural networks are inherently probabilistic, especially considering the fact that there might exist some intention-behavior discrepancies in the training data. Instead, we try to make use of the learned low-dimensional features as these features tend to be more robust [52, 94], which is also verified by our experimental results. Second, our outlier detection module can effectively make use of the prediction results from the learned icon-behavior model. For example, if the icon-behavior model predicts that the test icon should use a certain permission, the test icon is less likely to be an outlier in this permission group.

6.1 Outlier Detection Overview

The overview of the outlier detection module is shown in Figure 9. Given a test icon and its contextual text from a new APK file, we first extract its low-dimensional feature vector f based on the learned icon-behavior model, and obtain its actual permission uses by static analysis (Section 4). We organize the sensitive permissions into permission groups based on the Android dangerous permission groups [22] (see Table 2), and learn an outlier detector for each permission group. Note that we exclude certain permissions (e.g., `READ_PHONE_STATE` and permissions in the `SENSORS` and `CALL_LOG` groups) since these permissions are not evident to the users through UIs (e.g., `READ_PHONE_STATE`) or rarely appear in the collected apps (e.g., `ANSWER_PHONE_CALL`). We also add the `NETWORK` group for network communications as some apps may disclose users' data. Then, we use group-wise outlier detectors to compute the *group-based outlier scores* through each used permission. For example, the example icon in Figure 9 uses only the `SMS` permission. Then, the outlier detector corresponds to this permission group will be activated. Next, as one icon may be related to multiple permission groups, we aggregate the group-based outlier scores to form the the *final outlier score*. Here, a key step is to compute the weights of each group-based outlier score. The final outlier score reflects the overall likelikood of intention-behavior discrepancy for the input test icon.

6.2 Computing Group-Wise Outlier Score

There exist several choices for the group-wise outlier detector, including KNN [59], OCSVM [65], IForest [44], and AutoEncoders [3]. We empirically found that the performances of these detectors are relatively close to each other (Section 7.3.3). Therefore, we adopt the AutoEncoder structure for simplicity, which is known to be an effective replacement of traditional methods for the outlier detection problem [3]. The key idea of AutoEncoder is to first reduce the dimension of the original feature and then reconstruct it, i.e.,

$$\begin{aligned} g &= reduce(f) \\ f' &= reconstruct(g) \end{aligned} \tag{6}$$

where $f \in R^d$ is the original feature vector from the icon-behavior model, g is the reduced feature vector, and $f' \in R^d$ is the reconstructed feature vector for f. Specially, we implement *reduce* and *reconstruct* with two fully-connected layers, respectively. The learning process is then guided by minimizing the reconstruction error, i.e.,

$$\min \sum_{j=1}^{d} \left(f^{(j)} - f'^{(j)} \right)^2 . \tag{7}$$

We train one AutoEncoder for each permission group, with the icons from benign apps (the same input with the icon-behavior learning). Then, for a test icon, the reconstruction error of the corresponding AutoEncoder is used to indicate the outlier score for each permission group. The intuition is that normal icons in the same permission group can be easily reconstructed while the reconstruction of anomalies would be relatively difficult. To be specific, suppose the test icon uses a permission in the i-th permission group, and f_i' is the reconstructed feature vector in the i-th corresponding AutoEncoder. Then, $s_i = (f - f_i')^2$ is used as the outlier score for this permission group.

6.3 Computing Final Outlier Score

Given that there are n permission groups with scores $[s_1, s_2, ..., s_n]$, the remaining problem is to aggregate these scores into a final outlier score s. In this work, we consider the following three aggregation methods. Note that the computations of all the following aggregation methods are based on the output of the learned icon-behavior model.

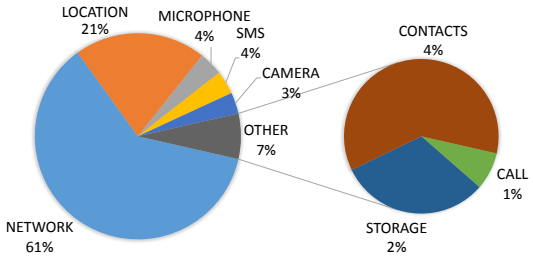


Figure 10: Distribution of sensitive permission groups.

Distance-based aggregation. The first aggregation method is based on the clustering degree of the neighbor icons near the test icon. Here, the distance is computed based on the features learned from the icon-behavior model in the vector space. The intuition is that if the local neighborhood of a test icon is closely clustered, the AutoEncoder should have been sufficiently trained in the neighborhood; therefore, the ourlier score is more reliable. Here, we compute the average distance among the T nearest neighbors of the test icon, and use it to weight the group-based outlier scores as

$$s = s_1/AvgDis_1 + s_2/AvgDis_2 + \ldots + s_n/AvgDis_n,$$

where $AvgDis_i$ means the average distance among the neighbors in i-th permission group. We normalize $1/AvgDis_i$ to compute s.

Prediction-based aggregation. In the second aggregation method, we integrate the predicted probabilities over the permissions for a test icon from the icon-behavior model. The intuition is that if the icon-behavior model predicts that the icon widget should use a certain permission, the test icon tends not to be an outlier in this permission group. Therefore, we have

$$s = s_1 * (1 - p_1) + s_2 * (1 - p_2) + \ldots + s_n * (1 - p_n),$$

where p_i is the probability of using the i-th permission (group) predicted by the icon-behavior model.

Combined aggregation. Based on the above two aggregation methods, we also define a combined method by adding the two weights, i.e.,

$$s = s_1 * (1 - p_1 + 1/AvgDis_1) + \ldots + s_n * (1 - p_n + 1/AvgDis_n).$$

7 EVALUATION

We evaluate DeepIntent on a large number of real world apps. Specifically, we aim to answer the following research questions:

- *RQ1: How effective is the co-attention mechanism for icons and texts in improving icon-behavior learning?*
- *RQ2: How effective is icon-behavior association based on static analysis in improving icon-behavior learning?*
- *RQ3: How effective is DeepIntent in detecting intention-behavior discrepancies?*

7.1 Evaluation Setup

To evaluate DeepIntent, we collected 9,891 benign apps and 16,262 malicious apps. The benign apps are downloaded from Google Play, and we further send them to VirusTotal to ensure that no anti-virus engines flag them as positive. We resort to Wang et al. [77] and RmvDroid [2, 78] to collect the up-to-date malicious apps. These

Permission Groups	Sensitive Permissions
NETWORK	INTERNET
	CHANGE_WIFI_STATE
LOCATION	ACCESS_COARSE_LOCATION
	ACCESS_FINE_LOCATION
	ACCESS_MOCK_LOCATION
MICROPHONE	RECORD_AUDIO
SMS	SEND_SMS
	READ_SMS
	WRITE_SMS
	RECEIVE_SMS
CAMERA	CAMERA
CALL	CALL_PHONE
STORAGE	WRITE_EXTERNAL_STORAGE
	READ_EXTERNAL_STORAGE
CONTACTS	READ_CONTACTS
	WRITE_CONTACTS
	GET_ACCOUNTS
	MANAGE_ACCOUNTS
	AUTHENTICATE_ACCOUNTS

Table 2: Sensitive permissions and permission groups.

apps are flagged by at least 20 anti-virus engines on VirusTotal and are further removed by app markets. For all the apps, we apply our icon-behavior association techniques (Section 4) to obtain the triples of $\langle icon, text, permissions \rangle$. We then divide these permissions into 8 permission groups as shown in Table 2. The distribution of the 8 permission groups is shown in Figure 10.

Next, to train the icon-behavior model, we randomly select 80% of the triples from benign apps as training data, and use the remaining 20% of the triples from benign apps as a test set, i.e., *benign test set*. The benign test set is used to evaluate the effectiveness of DeepIntent in predicting permission uses based on the icons and texts. To evaluate the effectiveness of DeepIntent in detecting intention-behavior discrepancies, we further use all the triples from malicious apps as the second test set, i.e., *malicious test set*. We also apply DeepIntent on the benign test set to see whether there are intention-behavior discrepancies in these benign apps.

Obtaining Ground-Truths. Since not all the permission uses in malicious apps are abnormal, we need to collect the ground-truth of intention-behavior discrepancies for icon widgets. We recruited 10 volunteers who are graduate students from computer science and have been using Android phones for at least three years. We then ask these volunteers to manually mark if there are intention-behavior discrepancies based on the $\langle icon, text, permissions \rangle$ triples. We define a discrepancy as *an icon widget uses the permissions that cannot be justified by the icons and texts in the UIs*. Specifically, we randomly selected 1500 triples from both test sets. Each triple is assigned to two volunteers for them to mark independently. In cases the two volunteers disagree with each other, we will ask another volunteer to discuss with these two volunteers. If a consensus cannot be reached, we will exclude this triple.

Overall, we obtain 7691, 1274, and 1362 unique triples that contain sensitive permission uses in the training set, the benign test

Metric	Method	NETWORK	LOCATION	MICROPHONE	SMS	CAMERA	CALL	STORAGE	CONTACTS
Precision	IconIntent	0.9054	0.3702	0.2334	0.3221	0.2917	0.2264	0.1286	0.2644
	icon_only	0.9478	0.7006	0.7447	0.7817	0.7793	0.9138	0.8273	0.7725
	text_only	0.9775	0.8505	0.8515	0.8651	0.8763	0.8028	0.7971	0.8489
	concatenate	0.9665	0.8684	0.7827	0.8747	0.8955	1.000	0.8526	0.8809
	add	0.9617	0.8588	0.8117	0.9076	0.9378	1.000	0.8915	0.8528
	co-attention	0.9674	0.8675	0.8531	0.9389	0.9322	1.000	0.9137	0.9073
Recall	IconIntent	0.7488	0.7897	0.6985	0.6825	0.6885	0.7059	0.5454	0.5576
	icon_only	0.987	0.6021	0.4355	0.5209	0.5558	0.6306	0.4548	0.5326
	text_only	0.9807	0.8784	0.7473	0.7874	0.7512	0.7315	0.6802	0.7926
	concatenate	0.9841	0.8426	0.5839	0.72	0.7258	0.6134	0.4054	0.7177
	add	0.9858	0.8549	0.5606	0.6605	0.7214	0.65	0.468	0.7532
	co-attention	0.9883	0.8325	0.6487	0.7763	0.7274	0.6111	0.5554	0.7705
F1	IconIntent	0.8197	0.5041	0.3499	0.4377	0.4098	0.3429	0.2081	0.3587
	icon_only	0.967	0.6476	0.5496	0.6252	0.6488	0.7462	0.5869	0.6305
	text_only	0.9791	0.8642	0.796	0.8244	0.8089	0.7655	0.734	0.8198
	concatenate	0.9752	0.8553	0.6688	0.7898	0.8018	0.7604	0.5495	0.791
	add	0.9736	0.8568	0.6632	0.7646	0.8155	0.7879	0.6138	0.7999
	co-attention	0.9777	0.8496	**0.737**	**0.8499**	0.8172	0.7586	**0.6909**	**0.8333**

Table 3: Evaluation results for icon-permission prediction. Our co-attention mechanism in DeepIntent generally performs the best, especially in four permission groups that are relatively difficult to predict.

set, and the malicious test set, respectively. In the two test sets, we have manually found 432 and 865 intention-behavior discrepancies.

7.2 Implementation

DeepIntent contains three key steps: icon-behavior association, icon-behavior modeling, and outlier detection. For icon-behavior association, we implement it upon Gator [64] and Soot [74] for static analysis. We decode apps using ApkTool [73] and map API methods to permissions using PScout [6]. We use Pillow [12] to process icons and Google Tesseract OCR [60] to extract embedded texts from icons. We followed the standard steps as the previous work [80] including tuning image colors and rotating images to fit the text angles before extracting embedded texts from icons. For the icon-behavior model, we implement it using Keras [11]. We embed each word with the dimension of 100, and set feature dimension d to 100. The model is trained using stochastic gradient descent with the Adam optimizer. We also adopt dropout [21] with the dropout rate being 0.5. For outlier detection, we set the hidden dimensions of AutoEncoder to [64, 32, 64], where the first two dimensions are to compress and the last two are to reconstruct. The neighborhood size for the distance-based aggregation method is set to 5 (i.e., $T = 5$).

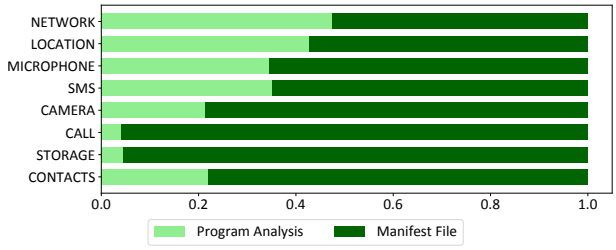
7.3 Evaluation Results

7.3.1 RQ1: Effectiveness of joint feature learning. To demonstrate the effectiveness of the co-attention mechanism that jointly models icons and texts in icon-behavior learning, we first compare DeepIntent with IconIntent [80], the state-of-the-art sensitive UI widget identification approach that adapts computer vision techniques (SIFT [46] and FAST [62]) to predict sensitive categories of UI widgets. As IconIntent is designed to predict each single permission group while DeepIntent targets at a multi-label prediction problem, for fair comparison, we run IconIntent multiple times over each permission group to obtain the predicted labels. Next, to demonstrate the improvement brought by modeling both icons and texts, we compare with two variants of DeepIntent: 'icon_only' and 'text_only', which consider either only image features or only text features. Finally, we further compare the co-attention mechanism used in DeepIntent with two variants: 'add' and 'concatenate', which adds or concatenates the image and text features to substitute our feature combination in Figure 8 (c).

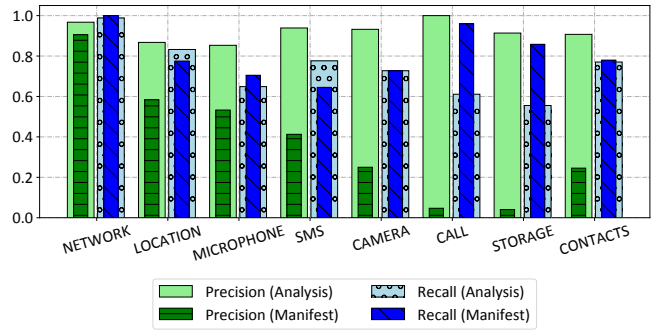
We measure the prediction accuracy of DeepIntent and the compared approaches. The test set of this experiment is a subset of the benign test set with icons marked as intention-behavior discrepancies deleted. Since we model the permission prediction as a multi-label prediction problem, we adopt the average precision, recall, and F1-score over each icon as evaluation metrics. The results are shown in Table 3.

We have several major observations. First, all the three DeepIntent variants significantly outperform IconIntent in terms of precision and F1-score. For example, DeepIntent achieves at least 19.3% relative improvement in different permission groups. IconIntent yields higher recall values. The reason is that the extracted features from IconIntent are less accurate, and it tends to predict a larger number of permissions for each icon, resulting in higher recall and lower precision. This result indicates the superior performance of the used deep learning techniques clearly over the computer vision techniques in IconIntent for large-scale datasets.

Second, compared to 'text_only' and 'icon_only', DeepIntent performs the best in most cases on the F1-score metric. The only exception is from the `Network` group when comparing with 'text_only'. The possible reason is as follows. Compared to the other permissions, various icons with more different appearances may use the `Network` permission; therefore, adding icon features may mislead the predictions when there are insufficient similar icons in the training data. Moreover, we can observe that 'text_only' generally performs better than 'icon_only'. The reasons are three-fold: 1) text



(a) The distributions of icon permissions. Directly using manifest files introduces many unused permissions.



(b) The precision/recall results for the icon-behavior model. Training with permissions in manifest files performs poorly in precision.

Figure 11: The necessity of icon-behavior association.

descriptions are intuitively discriminative in our prediction task, 2) we extract text not only in apps' UIs but also from the layout and resource names, and 3) icon images are relatively noisy and require more data for training.

Third, the co-attention mechanism in DeepIntent performs especially well in 4 out of 8 permission groups (the bold cases in the table). One common place in these 4 cases is that the accuracy of all the feature combination methods is relatively low. This means that our co-attention mechanism helps improve the learned model for the permission groups that are relatively more difficult to predict.

Overall, this experiment shows the effectiveness of our co-attention mechanism for learning the icon-behavior model, which performs especially well in four out of eight permission groups that are relatively difficult to predict.

7.3.2 RQ2: Effectiveness of Icon-Behavior Association. To demonstrate the necessity of icon-behavior association in icon-behavior learning, we compare DeepIntent with an approach *Manifest* that uses the permissions defined in an app's manifest file for all the icon widgets in this app. Obviously, the permission set from the manifest file for each icon widget is a super set of that from our icon-behavior association module. We then use these permissions for each icon as the training set, and evaluate *Manifest*'s performance on the benign test set without icons marked as intention-behavior discrepancies. The resulting permission distributions and the prediction accuracy results are shown in Figure 11.

Type	Permission	Precision/Recall	AUC
benign	NETWORK	0.693	0.8638
	LOCATION	0.7857	0.746
	MICROPHONE	0.7613	0.8221
	SMS	0.7685	0.7841
	CAMERA	0.7647	0.7851
	CALL	0.8697	0.9382
	STORAGE	0.8571	0.9722
	CONTACTS	0.7849	0.787
malicious	NETWORK	0.7568	0.8712
	LOCATION	0.856	0.756
	MICROPHONE	-	-
	SMS	0.939	0.8034
	CAMERA	0.9167	0.8472
	CALL	-	-
	STORAGE	0.9231	0.8462
	CONTACTS	0.9412	0.8412

Table 4: Detection accuracy results over permission groups. AUC = 0.5 means random guess. DeepIntent can accurately detect the intention-behavior discrepancies.

Figure 11 (a) shows the permission distributions where we compute the percentage of icons under each permission group. As we can see, directly using manifest files introduces many unused permissions. For example, there are three times more icons with CAMERA and CONTACTS permissions by using the manifest files. This ratio is even larger for STORAGE and CALL.

Figure 11 (b) shows the precision and recall results of the re-trained model with permissions from manifest files. The results of DeepIntent are also plotted with a wider rectangle and lighter color. As we can see, the precision results of the re-trained model decrease dramatically in many cases while the recall results stay high. This is consistent with our intuition that using more permissions for training tends to predict more permissions on the test set, which could dramatically degenerate the precision performance. On average, the precision result of DeepIntent is 0.9419 while that of the re-trained model is 0.5515.

Overall, the results show that our icon-behavior association module is essential to accurately extract icon-permission mappings for the learning of the icon-behavior model.

7.3.3 RQ3: Detecting Intention-Behavior Discrepancies. For RQ3, we evaluate the effectiveness of DeepIntent in terms of detecting the intention-behavior discrepancies. DeepIntent returns a ranked list based on the outlier score of each test icon widget. We adopt the top-K precision/recall and AUC metrics based on the manually labeled benign and malicious test sets. To choose a best group-wise detector, we also evaluate the effectiveness of different group-wise detectors and different aggregation methods with DeepIntent. Finally, to show the superiority of DeepIntent in detecting discrepancies, we further compare DeepIntent with other outlier detectors based on IconIntent and part of the features used by DeepIntent. The results are shown in Tables 4 - 7 and Figure 12.

Detection accuracy over permission groups. Table 4 shows the results for each permission group. The first column is the type of test set. The precision/recall column is based on the top-K results

Type	Method	AUC
benign	KNN	0.8614
	OCSVM	0.8493
	IForest	0.8345
	AutoEncoder	0.8656
malicious	KNN	0.8922
	OCSVM	0.8539
	IForest	0.8640
	AutoEncoder	0.8839

Table 5: Detection accuracy results using different group-wise outlier detectors. These detectors perform relatively close to each other, and we adopt AutoEncoder due to its efficiency and simplicity.

Type	Method	AUC
benign	mean	0.7368
	distance-based	0.8211
	prediction-based	0.8313
	combined	**0.8656**
malicious	mean	0.7914
	distance-based	0.8484
	prediction-based	0.8605
	combined	**0.8839**

Table 6: Detection accuracy results using different aggregation methods. AUC = 0.5 means random guess. The combined aggregation performs best.

when K is set to the real number of labeled intention-behavior discrepancies; therefore, the precision and recall have the same value in this case. The AUC column stands for the AUC value when we vary K from zero to the test set size. The permission groups MICROPHONE and CALL in the malicious test set are marked as '-'. This is because the malicious apps we collected rarely contain these two permissions.

We can first observe from the table that, in general, DEEPINTENT can accurately detect the intention-behavior discrepancies. For example, the precision and recall results are all above 0.9 for SMS, CAMERA, STORAGE, and CONTACTS in the malicious test set. These permissions are widely adopted by malicious apps to steal user information and perform monetized actions. Second, although still effective, the detection accuracy is relatively lower in LOCATION. The probable reason is that many icons such as refreshing the restaurant recommendations will update the screen using the current location; however, this intention is hardly to observe from the UI. Third, the performance in the NETWORK group is relatively low. This is due to the fact that many icons with many different looks and purposes are related to the network permissions, making it difficult to recognize the discrepancies.

Group-wise Detectors and Aggregation Methods. In our discrepancy detection, we have four group-wise detectors and three aggregation methods. Table 5 and Table 6 show the results of these choices. In Table 5, we use each outlier detector to substitute the AutoEncoder as described in Section 6.2, followed by the combined aggregation method. We can observe that all the four detectors perform relatively close to each other, and we adopt AutoEncoder in this work due to its simplicity and efficiency. For Table 6, we report the results of different aggregation methods. We also show the results when we simply compute the mean outlier score from group-wise detectors (referred as 'mean' in the table). We can see that, compared with 'mean' aggregation, both distance-based aggregation and prediction-based aggregation can achieve much higher accuracy in terms of correctly identifying the intention-behavior discrepancies. Furthermore, combining these two aggregation methods can achieve further improvement. In this work, we adopt the combined aggregation method.

Type	Method	AUC
benign	IconIntent	0.6188
	icon_only	0.7618
	text_only	0.7739
	prediction	0.7991
	DeepIntent	**0.8656**
malicious	IconIntent	0.7009
	icon_only	0.7537
	text_only	0.7752
	prediction	0.8122
	DeepIntent	**0.8839**

Table 7: Detection accuracy comparisons. DEEPINTENT significantly outperforms the competitors.

Comparisons for Outlier Detection. Finally, we compare DEEPINTENT with several competitors in terms of identifying the intention-behavior discrepancies. The average AUC results are shown in Table 7. We include 4 competitors. For ICONINTENT, we input the extracted features into our outlier detection module. For 'icon_only' and 'text_only', we use only the image features and text features, respectively. For 'prediction', we directly use the predicted results from our icon-behavior model to detect the discrepancies.

The results show that DEEPINTENT significantly outperforms the competitors. Compared with ICONINTENT, DEEPINTENT achieves 39.9% and 26.1% improvements on the benign apps and the malicious apps, respectively. This result, again, indicates that the extracted features by DEEPINTENT are more accurate. DEEPINTENT is also better than 'icon_only' and 'text_only', which means that combining icon and text features are also useful for discrepancy detection. Finally, DEEPINTENT outperforms the 'prediction' method. This result is consistent with our motivation of evolving an outlier detection module after the behavior prediction module (Section 6).

To further inspect the performance of DEEPINTENT, we show its precision and recall curves in Figure 12. In the figure, the y-axis means precision/recall, the x-axis is the K (i.e., choosing top K candidates based on the final outlier scores), and the dashed vertical line means the real number of labeled outliers. Intuitively, the larger the area under the curve, the better the method. In the figure, we also plot the theory results of the 'random' method which randomly identifies the outliers for comparison. We can observe from the figure that both curves of DEEPINTENT are significantly better than the 'random' method. Take the malicious test set (the right part of Figure 12) as an example. When K is less than the real number of outliers (i.e., $K < 865$), the precision results are

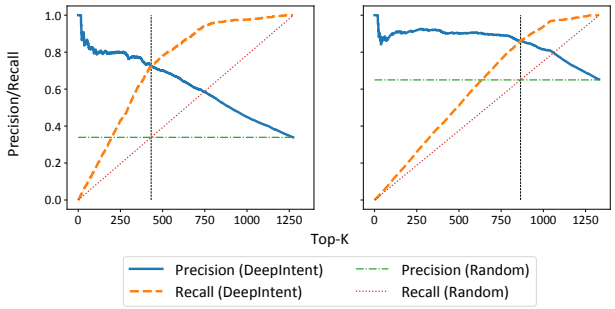


Figure 12: Detection precision and recall for benign apps (left) and malicious apps (right).

always above 0.85. In other words, when the returned number of icons is less than 865, over 85% of them have intention-behavior discrepancies.

Overall, the results show that DeepIntent can accurately identify the intention-behavior discrepancies in mobile apps.

8 DISCUSSION

Icon-Behavior Association. DeepIntent adapts static analysis to extract the icon-permission mappings. While the static analysis takes into account the major factors introduced by Android's complex environment (i.e., multi-threading, lifecycle methods, and ICCs), apps may evade our analysis by invoking sensitive APIs via reflections and native libraries. In the future, we plan to incorporate more advanced instrumentation techniques and dynamic analysis techniques to deal with reflections and native libraries [38, 39]. Moreover, the static analysis can produce incorrect associations between UI handlers and UI widgets due to its overapproximations. We plan to mitigate such issues using dynamic exploration techniques to filter out false associations [25, 48, 70].

Deep Icon-Behavior Learning. We consider the limitations of our learning within two aspects. First, DeepIntent is trained with certain data and could only react with similar icons within the training set. When DeepIntent meets new icons or noisy icons, the performance is non-deterministic. Furthermore, icons generated by recent deep learning attack models may also compromise DeepIntent. Second, we use contextual texts to enhance the icon-behavior learning process, but the vocabulary of DeepIntent is limited. This problem also troubles other natural language processing approaches, which is known as OOV (i.e., out of vocabulary) problem. Although we could use special characters like 'UNK' to indicate these words, the overall performance will drop.

Adversarial Setting. Our model is learned from the behaviors collected from a large set of popular apps in Google Play, representing the expected behaviors of apps with specific UIs. To avoid our detection, adversary apps may camouflage their undesired behaviors in apps with legitimate UIs and features for the undesired behaviors. For example, an eavesdropping app may pretend to be a voice recording app that has a UI widget with a microphone icon to use the microphone legitimately. With DeepIntent, if the app tries to send out the recorded audio, the extra permission on `NETWORK` will reveal its differences and can be detected. If the app tries to use non-UI events for using the microphone, existing malware detection techniques [83] can be leveraged to detect non-UI permission misuses. Finally, we admit that, as a potential evasion technique, an attacker may collect the data as we did from the whole app market and try to hide their malicious behaviors in benign behaviors with specific UIs. However, summarizing these benign behavior patterns and extracting them from the learned model are non-trivial, especially given the model is trained using deep learning. As a result, our technique significantly raises the bar for potential attacks.

9 RELATED WORK

UI Analysis of Mobile Apps. AsDroid [33] checks the compatibility of the UI widgets' descriptive texts and the pre-defined intentions represented by its sensitive APIs. SUPOR, UIPicker, Uiref [4, 31, 51] analyze the descriptive texts in apps' UI for identifying sensitive UI widgets that accept user inputs. PERUIM [40] extracts the permission-UI mappings from an app based on both dynamic and static analysis, helping users understand the requested permissions. Liu et al. [45] propose an automatic approach for annotating mobile UI elements with both structural semantics such as buttons or toolbars and functional semantics such as add or search. AppIntent [85] presents the sequence of UI screenshots that can lead to sensitive data transmissions of an app for human analysts to review. None of them model both icons and texts to detect abnormal behaviors.

Textual Analysis on Mobile Apps. WHYPER [56] and Autocog [58] adapt Natural Language Processing (NLP) techniques to identify sentences in app descriptions that explain the permission uses. BidText [32] detects sensitive data disclosures by performing bi-directional data flow analysis to detect variables that are at the sink points and are correlated with sensitive text labels. Pluto [14] analyzes app files to identify user data exposure to ad libraries. There are also existing techniques that leverage the textual information from code to infer the purposes of permission uses [76], and synthesize natural language descriptions for the data flow behavior in using users' sensitive data [88]. Unlike these approaches that map text to sensitive data directly, our text analysis uses the contextual texts for icon widgets as part of the features to learn the intention-behavior model.

Android Static Analysis. Existing work [5, 79] has provided approaches to build call graphs that consider Android's complex environment. However, their focus is to enumerate all possible combinations of lifecycle methods for improving the precision of static analysis, which will cause lots of false positives if used for building a UI handler's call graph. AppAudit [13] proposes a solution for handling Android multi-threading, system/GUI callbacks, and lifecycle methods to build extended call graph, and it combines dynamic analysis on filtering false positive call edges. DeepIntent can further benefit from its techniques to improve the call graph construction. There also exists a line of related work [37, 53–55, 91] that leverages program analysis and machine learning techniques to identify the ICCs in Android apps. The ICC analysis of DeepIntent is built on these work.

Modeling Image and Text. Images and texts could be modeled separately or jointly. For example, CNNs (e.g., VGG [67], ResNet [26], and DenseNet [30]) are widely used in modeling images; RNNs as

well as attention mechanisms [84], sequence to sequence structures [8], and memory networks [71] are employed to handle text-related tasks. Recently, Lu et al. [47] and Zhang et al. [89] present a co-attention mechanism to jointly model images and texts. DeepIntent trains its own DenseNet to learn the features of icons, and further integrates the state-of-the-art network structures to co-train icons and texts into a better feature representation.

Outlier Detection. There exist various outlier detection techniques such as KNN [59], PCA [66], and AutoEncoder [3], which are also applied in Android analyzing scenarios. For example, CHABADA [23] groups apps based on their topics, and identifies outliers in each group that use abnormal APIs. MUDFLOW [7] extracts data flows from Android apps and flags malicious apps due to their abnormal data flows. These proposals detect app-level outliers, while DeepIntent considers the icon-level outliers. Moreover, one difficulty of DeepIntent is the absence of icon-level outlier labels, and thus we propose static analysis to obtain these labels.

10 CONCLUSION

We have presented a novel framework, DeepIntent, that synergistically combines program analysis and deep learning to detect intention-behavior discrepancies in mobile apps. In particular, DeepIntent includes a static analysis that handles the complex Android environment to identify icon widgets and associate the widgets to permission uses, applies the parallel co-attention mechanism to jointly model icons and their contextual texts of the icon widgets, and detects the intention-behavior discrepancies by computing and aggregating the outlier scores from each permission used by the icon widget. Our evaluation on a large number of real apps demonstrates that DeepIntent effectively detects intention-behavior discrepancies by checking the intentions reflected by apps' UIs against the behind-the-scene program behaviors.

ACKNOWLEDGMENTS

This work is supported in part by the Natural Science Foundation of China (No. 61690204, 61932021, 61672274), the National Science Foundation (CNS-1755772), and the Collaborative Innovation Center of Novel Software Technology and Industrialization. Fengyuan Xu is partly supported by the National Science Foundation of China (No. 61872180) and Jiangsu "Shuang-Chuang". Haoyu Wang is partly supported by the National Science Foundation of China (No. 61702045). Peng Gao is supported in part by the CLTC (Center for Long-Term Cybersecurity). Yuan Yao is the corresponding author.

REFERENCES

[1] 2017. How Big Data Is Empowering AI and Machine Learning at Scale. (2017). https://sloanreview.mit.edu/article/how-big-data-is-empowering-ai-and-machine-learning-at-scale/.
[2] 2019. RmvDroid Dataset. (2019). https://zenodo.org/record/2593596#.XNtieI4zZyw.
[3] Charu C Aggarwal. 2015. Outlier analysis. In *Data mining*.
[4] Benjamin Andow, Akhil Acharya, Dengfeng Li, William Enck, Kapil Singh, and Tao Xie. 2017. UiRef: Analysis of Sensitive User Inputs in Android Applications. In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*.
[5] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
[6] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS)*.
[7] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2015. Mining apps for abnormal usage of sensitive data. In *Proceedings of the International Conference on Software Engineering (ICSE)*.
[8] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.
[9] David Barrera, H. Güneş Kayacik, Paul C. van Oorschot, and Anil Somayaji. 2010. A Methodology for Empirical Analysis of Permission-based Security Models and Its Application to Android. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*.
[10] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv*.
[11] François Chollet et al. 2015. Keras. https://keras.io. (2015).
[12] Alex Clark. 2010. Pillow. https://github.com/python-pillow/Pillow. (2010).
[13] Octeau Damien, McDaniel Patrick, Jha Somesh, Bartel Alexandre, Bodden Eric, Klein Jacques, and Traon Yves, Le. 2013. Effective inter- component communication mapping in Android with EPIC: An essential step towards holistic security analysis. In *Proceedings of the USENIX Security Symposium ((USENIX Security))*.
[14] Soteris Demetriou, Whitney Merrill, Wei Yang, Aston Zhang, and Carl A. Gunter. 2016. Free for All! Assessing User Data Exposure to Advertising Libraries on Android. In *23rd Annual Network and Distributed System Security Symposium (NDSS)*.
[15] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition (CVPR)*.
[16] Michael D. Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros, Ravi Bhoraskar, Seungyeop Han, Paul Vines, and Edward XueJun Wu. 2014. Collaborative Verification of Information Flow for a High-Assurance App Store. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[17] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android Permissions Demystified *(Proceedings of ACM Conference on Computer and Communications Security (CCS))*.
[18] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. 2011. A survey of mobile malware in the wild. In *Proceedings of ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*.
[19] Adrienne Porter Felt, Kate Greenwood, and David Wagner. 2011. The Effectiveness of Application Permissions. In *USENIX Conference on Web Application Development (WebApps)*.
[20] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. 2012. Android Permissions: User Attention, Comprehension, and Behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security (SOUPS)*.
[21] Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org/
[22] Google. 2019. Android Permission Overview. (2019). https://developer.android.com/guide/topics/permissions/overview.
[23] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. 2014. Checking app behavior against app descriptions. In *Proceedings of the International Conference on Software Engineering (ICSE)*.
[24] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*.
[25] Shuai Hao, Bin Liu, Suman Nath, William G. J. Halfond, and Ramesh Govindan. 2014. PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps. In *The 12th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*.
[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*.
[27] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. 2006. A Fast Learning Algorithm for Deep Belief Nets. *Neural Computing*.
[28] Geoffrey E Hinton and Ruslan R Salakhutdinov. 2006. Reducing the dimensionality of data with neural networks. *Science*.
[29] David W Hosmer Jr, Stanley Lemeshow, and Rodney X Sturdivant. 2013. *Applied logistic regression*. Vol. 398. John Wiley & Sons.
[30] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*.
[31] Jianjun Huang, Zhichun Li, Xusheng Xiao, Zhenyu Wu, Kangjie Lu, Xiangyu Zhang, and Guofei Jiang. 2015. SUPOR: Precise and Scalable Sensitive User Input

Detection for Android Apps. In *USENIX Security Symposium ((USENIX Security))*.
[32] Jianjun Huang, Xiangyu Zhang, and Lin Tan. 2016. Detecting Sensitive Data Disclosure via Bi-directional Text Correlation Analysis. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*.
[33] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. 2014. AsDroid: Detecting Stealthy Behaviors in Android Applications by User Interface and Program Behavior Contradiction. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*.
[34] Alex Krizhevsky and Geoffrey Hinton. 2009. *Learning multiple layers of features from tiny images*. Technical Report. Citeseer.
[35] Yann LeCun, Yoshua Bengio, et al. 1995. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*.
[36] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature*.
[37] Li Li, Alexandre Bartel, Tegawendé François D Assise Bissyande, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. IccTA: detecting inter-component privacy leaks in android apps. In *Proceedings of the International Conference on Software Engineering (ICSE)*.
[38] Li Li, Tegawendé F. Bissyandé, Damien Octeau, and Jacques Klein. 2016. DroidRA: taming reflection to support whole-program analysis of Android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*.
[39] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. 2017. Static analysis of android apps: A systematic literature review. *Information & Software Technology (IST)*.
[40] Yuanchun Li, Yao Guo, and Xiangqun Chen. 2016. PERUIM: Understanding Mobile Application Privacy with permission-UI Mapping. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp)*.
[41] Jialiu Lin, Shahriyar Amini, Jason I. Hong, Norman Sadeh, Janne Lindqvist, and Joy Zhang. 2012. Expectation and Purpose: Understanding Users' Mental Models of Mobile App Privacy Through Crowdsourcing. In *Proceedings of the ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp)*.
[42] Bin Liu, Bin Liu, Hongxia Jin, and Ramesh Govindan. 2015. Efficient Privilege De-Escalation for Ad Libraries in Mobile Apps. In *Proceedings of the Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*.
[43] Bin Liu, Bin Liu, Hongxia Jin, and Ramesh Govindan. 2015. Medusa: A programming framework for crowd-sensing applications. In *Proceedings of the Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*.
[44] Fei Tony Liu, Ming Ting Kai, and Zhi Hua Zhou. 2009. Isolation Forest. In *Eighth IEEE International Conference on Data Mining (ICDM)*.
[45] Thomas F. Liu, Mark Craft, Jason Situ, Ersin Yumer, Radomír Mech, and Ranjitha Kumar. 2018. Learning Design Semantics for Mobile Apps. In *ACM Symposium on User Interface Software and Technology (UIST)*.
[46] David G. Lowe. 1999. Object Recognition from Local Scale-Invariant Features. In *Proceedings of the International Conference on Computer Vision (ICCV)*.
[47] Jiasen Lu, Jianwei Yang, Dhruv Batra, and Devi Parikh. 2016. Hierarchical question-image co-attention for visual question answering. In *Advances In Neural Information Processing Systems (NeurIPS)*.
[48] Yun Ma, Yangyang Huang, Ziniu Hu, Xusheng Xiao, and Xuanzhe Liu. 2019. Paladin: Automated Generation of Reproducible Test Cases for Android Apps. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications, HotMobile*.
[49] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems (NIPS)*.
[50] K. W. Miller, J. Voas, and G. F. Hurlburt. 2012. BYOD: Security and Privacy Considerations. *IT Professional*.
[51] Yuhong Nan, Min Yang, Zhemin Yang, Shunfan Zhou, Guofei Gu, and Xiaofeng Wang. 2015. Uipicker: User-input privacy identification in mobile applications. In *Proceedings of the USENIX Security Symposium (USENIX Security)*.
[52] Feiping Nie, Heng Huang, Xiao Cai, and Chris H Ding. 2010. Efficient and robust feature selection via joint âĎŞ2, 1-norms minimization. In *NIPS*.
[53] Damien Octeau, Somesh Jha, Matthew Dering, Patrick D. McDaniel, Alexandre Bartel, Li Li, Jacques Klein, and Yves Le Traon. 2016. Combining static analysis with probabilistic models to enable market-scale Android inter-component analysis. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016), St. Petersburg, FL, USA, January 20 - 22, 2016*.
[54] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. 2015. Composite Constant Propagation: Application to Android Inter-component Communication Analysis. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*
[55] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. 2013. Effective Inter-component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis. In *Proceedings of the USENIX Conference on Security (USENIX Security)*.
[56] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. 2013. WHYPER: Towards Automating Risk Assessment of Mobile Applications. In *USENIX Security Symposium*.
[57] Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*.
[58] Zhengyang Qu, Vaibhav Rastogi, Xinyi Zhang, Yan Chen, Tiantian Zhu, and Zhong Chen. 2014. AutoCog: Measuring the Description-to-permission Fidelity in Android Applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[59] Sridhar Ramaswamy, Rajeev Rastogi, and Kyuseok Shim. 2000. Efficient algorithms for mining outliers from large data sets. In *ACM Sigmod Record*.
[60] Zdenko Podobny Ray Smith et al. 2006. Tesseract. https://github.com/tesseract-ocr/tesseract. (2006).
[61] Sanae Rosen, Zhiyun Qian, and Z. Morely Mao. 2013. AppProfiler: A Flexible Method of Exposing Privacy-related Behavior in Android Applications to End Users. In *Proceedings of ACM Conference on Data and Application Security and Privacy (CODASPY)*.
[62] E. Rosten, R. Porter, and T. Drummond. 2010. Faster and Better: A Machine Learning Approach to Corner Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*.
[63] Atanas Rountev and Dacong Yan. 2014. Static Reference Analysis for GUI Objects in Android Software. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*.
[64] Atanas Rountev and Dacong Yan. 2014. Static Reference Analysis for GUI Objects in Android Software. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*.
[65] Bernhard Sch?lkopf, John C. Platt, John Shawe-Taylor, Alex J. Smola, and Robert C. Williamson. 2014. Estimating the Support of a High-Dimensional Distribution. *Neural Computation*.
[66] Mei-Ling Shyu, Shu-Ching Chen, Kanoksri Sarinnapakorn, and LiWu Chang. 2003. *A novel anomaly detection scheme based on principal component classifier*. Technical Report. MIAMI UNIV CORAL GABLES FL DEPT OF ELECTRICAL AND COMPUTER ENGINEERING.
[67] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.
[68] statista. 2017. (2017). https://www.statista.com/statistics/266572/market-share-held-by-smartphone-platforms-in-the-united-states/.
[69] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*.
[70] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*.
[71] Sainbayar Sukhbaatar, Jason Weston, Rob Fergus, et al. 2015. End-to-end memory networks. In *Advances in neural information processing systems (NeurIPS)*.
[72] Grigorios Tsoumakas and Ioannis Katakis. 2007. Multi-label classification: An overview. *International Journal of Data Warehousing and Mining (IJDWM)*.
[73] Connor Tumbleson and Ryszard Wisniewski. 2017. Apktool. (2017). https://ibotpeaches.github.io/Apktool/.
[74] Raja Vallee-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. 2000. Optimizing Java Bytecode using the Soot Framework: Is it Feasible?. In *Proceedings of the International Conference on Compiler Construction (CC)*.
[75] Nicolas Viennot, Edward Garcia, and Jason Nieh. 2014. A Measurement Study of Google Play. In *Proceedings of ACM SIGMETRICS conference (SIGMETRICS)*.
[76] Haoyu Wang, Jason Hong, and Yao Guo. 2015. Using Text Mining to Infer the Purpose of Permission Use in Mobile Apps. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp)*.
[77] Haoyu Wang, Zhe Liu, Jingyue Liang, Narseo Vallina-Rodriguez, Yao Guo, Li Li, Juan Tapiador, Jingcun Cao, and Guoai Xu. 2018. Beyond Google Play: A Large-Scale Comparative Study of Chinese Android App Markets. In *2018 Internet Measurement Conference (IMC)*.
[78] Haoyu Wang, Junjun Si, Hao Li, and Yao Guo. 2019. RmvDroid: Towards A Reliable Android Malware Dataset with App Metadata. In *The 16th International Conference on Mining Software Repositories (MSR 2019), Data Showcase Track*.
[79] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2014. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[80] 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 International Conference on Software Engineering (ICSE)*.
[81] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. 2015. Static Control-Flow Analysis of User-Driven Callbacks in Android Applications.

In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*.

[82] Shengqian Yang, Hailong Zhang, Haowei Wu, Yan Wang, Dacong Yan, and Atanas Rountev. 2015. Static Window Transition Graphs for Android (T). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[83] Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie, and William Enck. 2015. AppContext: Differentiating Malicious and Benign Mobile App Behaviors Using Context. In *International Conference on Software Engineering (ICSE)*.

[84] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. 2016. Hierarchical attention networks for document classification. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*.

[85] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X. Sean Wang. 2013. AppIntent: analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security (CCS)*.

[86] Yujie Yuan, Lihua Xu, Xusheng Xiao, Andy Podgurski, and Huibiao Zhu. 2017. RunDroid: recovering execution call graphs for Android applications. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*.

[87] ZDNet. 2015. Research: 74 percent using or adopting BYOD. (2015). http://www.zdnet.com/article/research-74-percent-using-or-adopting-byod/

[88] Mu Zhang, Yue Duan, Qian Feng, and Heng Yin. 2015. Towards Automatic Generation of Security-Centric Descriptions for Android Apps. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[89] Qi Zhang, Jiawen Wang, Haoran Huang, Xuanjing Huang, and Yeyun Gong. 2017. Hashtag Recommendation for Multimodal Microblog Using Co-Attention Network.. In *IJCAI*.

[90] Suwei Zhang, Yuan Yao, Fent Xu, Hanghang Tong, Xiaohui Yan, and Jian Lu. 2019. Hashtag Recommendation for Photo Sharing Services. In *AAAI*.

[91] Jinman Zhao, Aws Albarghouthi, Vaibhav Rastogi, Somesh Jha, and Damien Octeau. 2018. Neural-augmented static analysis of Android communication. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[92] Yajin Zhou and Xuxian Jiang. 2012. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S & P)*.

[93] Yajin Zhou and Xuxian Jiang. 2012. Dissecting Android Malware: Characterization and Evolution. In *IEEE Symposium on Security and Privacy (IEEE S & P)*.

[94] Xiaofeng Zhu, Cong Lei, Hao Yu, Yonggang Li, Jiangzhang Gan, and Shichao Zhang. 2018. Robust Graph Dimensionality Reduction. In *IJCAI*.