Humanoid: A Deep Learning-based Approach to Automated Black-box Android App Testing

Yuanchun Li, Ziyue Yang, Yao Guo, Xiangqun Chen

Abstract—Automated input generators are widely used for large-scale dynamic analysis of mobile apps. Such input generators must constantly choose which UI element to interact with and how to interact with it, in order to achieve high coverage with a limited time budget. Currently, most input generators adopt pseudo-random or brute-force searching strategies, which may take very long to find the correct combination of inputs that can drive the app into new and important states. In this paper, we propose Humanoid, a deep learning-based approach to GUI test input generation by learning from human interactions. Our insight is that if we can learn from human-generated interaction traces, it is possible to automatically prioritize test inputs based on their importance as perceived by users. We design and implement a deep neural network model to learn how end-users would interact with an app (specifically, which UI elements to interact with and how). Our experiments showed that the interaction model can successfully prioritize user-preferred inputs for any new UI (with a top-1 accuracy of 51.2% and a top-10 accuracy of 85.2%).

We implemented an input generator for Android apps based on the learned model and evaluated it on both open-source apps and market apps. The results indicated that Humanoid was able to achieve higher coverage than six state-of-the-art test generators. However, further analysis showed that the learned model was not the main reason of coverage improvement. Although the learned interaction pattern could drive the app into some important GUI states with higher probabilities, it had limited effect on the width and depth of GUI state search, which is the key to improve test coverage in the long term. Whether and how human interaction patterns can be used to improve coverage is still an unknown and challenging problem.

Index Terms—Dynamic analysis, automated input generation, graphical user interface, deep learning, mobile application

1 Introduction

Mobile applications (*apps* in short) have seen widespread adoption in recent years, with over three million apps available for download in both Google Play and Apple App Store, while billions of downloads have been accumulated [1], [2]. These apps need to be adequately inspected by the app markets and governments who want to prevent malicious or inappropriate apps being published. Many dynamic analysis methods [3], [4], [5] are proposed for such purpose. These methods usually require running an app in a sandbox, and the completeness of analysis is based on how many functionalities in the app are covered. However, due to the huge amount of apps and limited human resources, it is difficult for auditors to manually run the apps. As a result, automated input generators for mobile apps have been studied extensively in both academia and industry.

The inputs for mobile apps are typically represented by the interactions with the graphical user interface (GUI) of an app. Specifically, an interaction may include clicking, scrolling, or inputting text into a GUI element, such as a button, an image, or a text block. The job of an input generator is to produce a sequence of interactions for the app under analysis, which can be used to detect software-related problems, such as bugs, vulnerabilities, and security issues. The effectiveness of a input generator is often measured by its coverage. Given unlimited time, one can potentially try all

possible interaction sequences and combinations to achieve a perfect coverage. However, in real-world situations where the time budget for analysis is typically limited and the app may contain hundreds of GUI states and dozens of possible interactions in each state, an input generator can only choose a small subset of interaction sequences to explore.

The key to success for an automated input generator is to choose the "best" interactions for a given app, while the definition of "best" is based on the goal of analysis. For example, if the input generator is used by developers to find bugs in their apps, the unexpected or corner-case input would be preferred. While if the purpose of analysis is to audit the apps (e.g. detecting inappropriate content, privacy leakage, accessibility issues, etc.), the best input would be the interactions that end-users are more likely to pick, so that the commonly-used content can get more attention during analysis. Existing work on automated GUI input generation [6], [7], [8], [9], [10], [11] are usually focused on the bug-finding scenario, where random strategies or corner-case-targeted strategies are effective. However, these approaches may not be as effective in the dynamic analysis and auditing scenario, since they ignore the GUI information, thus is impossible to understand which interactions would be preferred by the users. The key question we want to investigate in this paper is: Can we teach an automated input generator to prioritize inputs like a human being?

This paper proposes Humanoid, an automated GUI input generator that is able to learn how humans interact with mobile apps and then use the learned model to guide input generation as a human. With the knowledge and model learned from human interaction traces, Humanoid can prioritize the possible interactions on a GUI page according

Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen are with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing 100871, China.
 E-mail: liyuanchun, ziyue.yang, yaoguo, cherry@pku.edu.cn

Yuanchun Li and Ziyue Yang contributed equally to this paper.



Action Type	UI Element	Probability	
touch	=	0.7	
touch	:	0.15	
touch	HAIDIAN QU	0.1	
touch	S	0.02	
touch	<	0.015	
touch	△	0.002	
touch	instagram	0.001	
touch	HAIDIAN QU	0.001	
swipe_left	32°C	0.001	
swipe_right	Value - Transport - Transport	0.001	
long_touch	=	0.0005	

Fig. 1: An illustration of how Humanoid chooses inputs for a UI state. The left side is a screenshot of the current UI of an Android app under analysis, and the right side enumerates the most possible interactions in the UI state. Humanoid computes a probability for each action based on a model learned from human interaction traces. The probability represents how likely the action will be chosen by Humanoid as the test input.

to their importance from user perspective, thus generating inputs that can lead to important states faster and achieve higher coverage.

We can use the GUI page from an app shown in Figure 1 as a motivating example. There are more than 20 actions that can be performed on that page, while most of them are ineffective or unrelated to the core functionalities of the app, such as swiping left on the current page, which is actually not scrollable, or clicking the advertisement on the bottom. While a random input generator may have to try all possible choices (including those ineffective ones), Humanoid is able to increase the probabilities of clicking the menu buttons, which are more likely to drive the app into additional important GUI states.

The core of Humanoid is a deep neural network (DNN) model that predicts which inputs are more likely to be generated by human users. The input of the model is the current UI state as well as the most recent UI transitions, represented as a stack of images, while the output is a predicted distribution of possible next actions, including the action type and the corresponding location coordinates on the screen. By comparing the predicted distribution with all possible actions on the UI page, the model is able to assign a probability to each action.

We then design a biased random search algorithm to generate inputs based on the probabilities predicted by the DNN model. The algorithm always tries to pick an unexplored action in the current state as the input. If there are multiple unexplored actions, the one with the highest probability is picked. If all actions have already been explored, the algorithm will pick the action that can lead to the state with the most unexplored actions. The reason why this strategy can improve coverage is that the important states that users prefer to visit and with more possible inputs are more frequently and adequately explored. Given a limited time budge, spending time on these important states can improve coverage

more efficiently than on other random states.

We implemented Humanoid and trained the interaction model with 304,976 human interactions extracted from a large-scale crowd-sourced UI interaction dataset Rico [12]. As a generic input generator, the model can be easily integrated with other dynamic analysis tools by simply replacing their input selection logic.

To evaluate the interaction model, we examined whether the model can learn human interaction patterns by using it to prioritize the possible actions for each UI state from a subset of the Rico interaction trace dataset. The results show that, for most UI states in the interaction traces, human-performed actions are ranked in the top 10% across all actions according to Humanoid-predicted probabilities, which was significantly better than a random strategy whose expectation would be around 50%.

To evaluate the input generation algorithm, we compared Humanoid with six state-of-the-art input generators in Android. These input generators are primarily designed for testing purpose (e.g. finding bugs), thus may not be very effective in covering apps' functionalities. The apps used for testing include 68 open-source apps obtained from the AndroTest [13] dataset, which is a widely-used benchmark dataset for evaluating Android test input generators. We also tested 200 popular apps from Google Play¹, to see whether Humanoid is also effective for more complicated apps in real-world scenarios. The results show that, Humanoid was able to achieve 43.3% line coverage for opensource apps and 24.1% activity coverage (the percentage of reached activities²) for market apps, which was significantly higher than the best results (38.8% and 19.7%) achieved by other input generators within the same time duration.

We further analyzed Humanoid's input generation algorithm by replacing the learned interaction model with a random policy, in order to see whether the coverage improvement was actually brought by the learned model. The result showed that, given sufficient time, there was no significant difference between the test coverage achieved with and without the learned model. This is because that coverage improvement is primarily due to how many different states are visited, instead of how much or how fast a input generator can navigate into human-preferred states. The former mainly depends on the state space search algorithm, rather than the interaction model learned from human interaction traces.

This paper makes the following main contributions:

- To the best of our knowledge, this is the first work to introduce the idea of prioritizing GUI inputs by learning from human interaction traces, in order to generate effective inputs for automated dynamic mobile app analysis.
- 2) We propose and implement a deep learning model to prioritize test inputs according to their importance from users' perspective. We demonstrate that Humanoid is able to rank the human-preferred interactions higher than others in new apps.
- 3) We design and implement a input generator, named Humanoid, to automatically generate interactions that
- 1. https://play.google.com/store/apps
- 2. https://developer.android.com/guide/components/activities

are more frequently performed by users. We evaluate Humanoid with extensive experiments.

2 BACKGROUND AND RELATED WORK

2.1 Android UI

For a mobile app, *user interface* (UI) is the place where interactions between humans and machines occur. App developers design UI to help users understand the features of their apps, and users interact with the apps through the UI. The graphical user interface (GUI) is the most important type of UI for most mobile apps, where apps present content and actionable widgets on the screen and users interact with the widgets using actions such as clicks, swipes, and text inputs.

The GUI pages (or screenshots) presented in mobile apps typically use a tree-structured layout. For example, in a screenshot of an Android app, all UI elements are built using View and ViewGroup objects and organized as a tree³. A View is a leaf node that draws something on the screen that the user can see and interact with. A ViewGroup is a parent node that holds other nodes in order to define the layout of the interface. A UI state can be identified as a snapshot of the structure and content in the current UI tree, and a node in the UI tree is called a UI element.

An app can be viewed as a combination of many GUI states and the transitions between them. Each GUI state serves different functionalities or renders different content. App users navigate between UI states by interacting with the UI elements.

2.2 Automated GUI Input Generation

Automated GUI input generation has become an active research area since the prevalence of mobile apps. Most of the research work target at the Android platform, partly due to the popularity of Android apps, as well as the fragmentation of Android devices and OS releases.

In Android, input generators interact with apps in the same way as humans: sending simulated gestures to the GUI of an app. Since the acceptable gestures in a UI state are limited, the main difference between different test generators is their strategies used in prioritizing these actions. There are mainly three types of strategies: random, model-based, and targeted.

A typical example using the *random strategy* is Monkey [6], the official tool for automated app testing in Android. Monkey sends random types of input events to random locations on the screen without considering its GUI structure. DynoDroid [7] also uses a random strategy, while the input sent by DynoDroid is smarter than Monkey: A lot of unacceptable events are filtered out based on the GUI structure and registered event listeners in an app. Sapienz [10] makes use of a genetic algorithm to optimize random test sequences. Polariz [14] extracts and reuses "motifs" obtained by human testers to help generate random test sequences.

Several other testing tools build and use *the GUI model* of mobile apps to generate test input. These models are

3. https://developer.android.com/guide/topics/ui/declaring-layout

usually represented as finite state machines that store the transitions between app window states. Such GUI models can be constructed dynamically [8], [9], [11], [15], [16], [17], [18], [19], [20] or statically [21]. Based on the GUI models, testing tools can generate events that can quickly navigate the app to unexplored states. These model-based strategies can be further optimized in various ways. For example, Stoat [11] can iteratively refine the test strategy based on existing explorations, and DroidMate [18] can infer acceptable actions for a UI element by mining from other apps.

The targeted strategy is designed to address the problem that some app behavior can only be revealed with specific test inputs. For example, a malicious app may only send SMS messages upon receiving a certain broadcast [22]. These testing tools [23], [24], [25], [26] usually use sophisticated static analysis techniques such as data flow analysis and symbolic execution to find the interactions that can lead to the target states. However, their effectiveness can be easily affected by the complexity of app code and the difficulty of mapping code to UI elements.

Unlike existing input generators that are mainly designed for developers to test their own apps, our work aim to assist auditors (app markets and governments) in large-scale dynamic app analysis, such as inappropriate content detection [5], privacy leakage detection [3], ad fraud detection [4], etc. In such scenarios, the input generator should prioritize user-preferred interactions over unexpected or corner-case inputs, in order to trigger more functionalities that may be used by users.

2.3 Software GUI Analysis

GUI is an indispensable part of software on most major platforms including Android. Analyzing the app's GUI is of great interest to many researchers and practitioners. There are mainly two lines of research in this area. One is to understand the behavior of apps from the software engineering perspective. Another is from the human-computer interaction perspective to analyze the user interface design.

As mentioned before, many automated testing tools build and use GUI models to guide test input generation. Unlike such models that mainly use the transitions between UI states to abstract the app behavior, there are also some approaches focused on analyzing the information in each individual UI state. For example, Huang *et al.* [27] and Rubin *et al.* [28] proposed to detect stealthy behaviors in Android apps by comparing the actual behaviors with the UI. PERUIM [29] extracted the mapping between an app's permissions to its UI to help users understand why each permission is requested, and AUDACIOUS [30] provided a way to control permission access based on UI components. Chen *et al.* [31] introduced a machine learning-based method to extract UI skeletons from UI images, in order to facilitate GUI development.

In human-computer interaction research, software GUI is mainly used to mine UI design practices [32], [33] and interaction patterns [34] at scale. The mined knowledge can further be used to guide UI and UX (user experience) design. To facilitate mobile app design mining, Deka *et al.* have collected and released a dataset named Rico [12],

which consists of a large number of UI screens and human interactions.

Our work lies in the intersection between software engineering and human-computer interaction: we propose a deep learning approach to mining human interaction patterns from the Rico dataset and use the learned patterns to guide automated input generation.

3 Our Approach

In order to employ human knowledge on mobile apps to augment mobile app testing, this paper proposes Humanoid, a new automated input generator that is able to prioritize inputs based on knowledge learned from humangenerated app interaction traces. Similar to many existing input generators, Humanoid uses a GUI model to understand and explore the behavior of the app under analysis. However, unlike traditional model-based approaches that randomly choose an action to perform when exploring a UI state, Humanoid prioritize the actions that are more likely to be performed by human users. We expect that such strategy can drive the app into important states more frequently and achieve higher coverage than random strategies.

3.1 Approach Overview

Figure 2 shows the overview of Humanoid. The core of Humanoid is a deep learning model (i.e. interaction model) that learns the patterns about how humans interact with apps. Based on the interaction model, the whole system can be separated into two phases, including *an offline phase* for training the model with human-generated interaction traces and *an online phase* in which the model is used to guide test input generation.

In the offline learning phase, we use a deep neural network model to learn the relation between the GUI contexts and user-performed interactions. A GUI context is represented as the visual information in the current UI state and the latest UI transitions, while an interaction is represented as the action type (touch, swipe, etc.) and the location coordinates of the action. After learning from large-scale human interaction traces, Humanoid is able to predict a probability distribution of the action type and action location for a new UI state. The predicted distribution can then be used to calculate the probability of each UI element being interacted with by humans and how to interact with it.

During the online testing phase, we adopt a biased random search algorithm to generate inputs. The algorithm maintains a GUI model named *UI transition graph* (UTG) to remember the explored states and actions in the app. Both the GUI model and the interaction model are used by Humanoid to decide what input to send. The algorithm picks new actions to explore based on the probabilities predicted by the interaction model and navigate between explored UI states based on the UTG.

3.2 Interaction Trace Preprocessing

First of all, we will need a large dataset (Rico [12]) with human interaction traces to train a user interaction model,

TABLE 1: Rules to extract actions from cursor movements. Note that $|loc_{end} - loc_{start}|$ means the Euclidean distance between loc_{start} and loc_{end} . The numbers are heuristically chosen to accommodate Android default configurations.

Condition	Action type	location
$ loc_{end} - loc_{start} < 50px$ and	touch	loc_{start}
$time_{end} - time_{start} < 500ms$		
$ loc_{end} - loc_{start} < 50px$ and	long_touch	loc_{start}
$time_{end} - time_{start} \ge 500ms$		
$ loc_{end} - loc_{start} \geq 50px$ and	swipe_left	loc_{start}
loc_{end} is on the left / right / top /	swipe_right	
bottom of loc_{start}	swipe_up	
	swipe_down	
Successive interaction sessions	input_text	center of the
where the keyboard is displayed		editable ele-
and an editable element is focused		ment

which is the key component in Humanoid. Because the human interaction traces in Rico are not designed for training for our purpose, we first need to preprocess the interaction traces.

A raw human interaction trace is usually a continuous stream of motion events sent to the screen [12], where each motion event is comprised of when (the timestamp) and where (the x,y coordinate) the cursor (the user's finger) enters, moves, and leaves the screen. The state change is also continuous because of the animations and dynamically loaded content.

The input acceptable to our model is a set of user interaction flows. Each interaction flow consists of a sequence of UI states $< s_1, s_2, s_3, ..., s_n >$ and a sequence of actions $< a_1, a_2, a_3, ..., a_n >$ that are taken in the corresponding UI states. To convert the raw interaction traces to the format acceptable to our model, we need to split cursor movements and identify user actions from them.

We consider seven types of user actions in Humanoid, including touch, long_touch, swipe_up/down/left/right, and input_text. Most use cases in apps can be accomplished with these types of actions. Each action is represented by the action type and the target location on the screen. In order to extract user actions from raw cursor traces, we first aggregate the cursor movements into *interaction sessions*.

An *interaction session* is defined as the period between when the cursor enters the screen and when the cursor leaves the screen. We denote the timestamps of the session start and the session end as $time_{start}$ and $time_{end}$, and the cursor locations as loc_{start} and loc_{end} . Then we map interaction sessions to user actions according to a list of heuristic rules, as shown in Table 1.

Once we have extracted the sequence of actions $< a_1, a_2, a_3, ..., a_n >$, we are able to match UI state changes with the actions based on the action timestamps. We use the UI state captured right before the timestamp of a_i as s_i to form the state sequence $< s_1, s_2, s_3, ..., s_n >$. The state sequence and the action sequence together represent a user interaction flow, which will be used as the training data for our human interaction model.

3.3 Model Training

This section explains in more details on how we use a deep neural network model to learn human interaction patterns

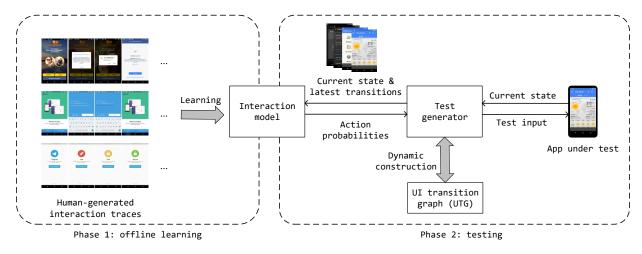


Fig. 2: Overview of our approach.

Raw data		Data processed for model				
Screenshots	Actions	UI skeletons	Heatmaps			
12.003	touch CANCEL	<mark>⊧</mark> o!				
For image Section The Control of	touch		*			
Napar	swipe left		·			
To an	input text "2"		·			

Fig. 3: The representation of UI states and actions in the interaction model.

from human interaction traces.

End-users interact with an app based on what they want to do with the app and what they see on its GUI. Since different apps often share common UI design patterns, it is intuitive that the way how humans interact with GUI is generalizable across different apps. The goal of the interaction model is to capture such generalizable interaction patterns.

We introduce a concept UI context to model what humans reference when they interact with an app. A UI context $context_i$ consists of the current UI state s_i and three latest UI transitions $(s_{i-1}, a_{i-1}), (s_{i-2}, a_{i-2}), (s_{i-3}, a_{i-3})$. The current UI state represents what the users see when they perform the action, while the latest UI transitions are used to model the users' underlying intention during the current interaction session. The reason why we used exactly three historic UI transitions is because most common interaction patterns contain no more than three actions [34].

Figure 3 shows how we represent the UI states and actions in our model. Each UI state is represented as a two-channel UI skeleton image, in which the first channel

(red channel) renders the bounding box regions of text UI elements and the second channel (green channel) renders the bounding box regions of non-text UI elements. The reason why we use the UI skeletons instead of the original screenshots is that most characters on the screenshots do not affect how humans interact with the apps. For example, the UI style (font size, button style, background color, etc.) of the same app may change across different OS and app versions, whereas the way how users use the app remains the same. Some apps even provide functionalities like "night mode" to allow users change the UI styles internally. Such UI style characters may bring noises to our model and affect the model's generalization ability, thus we exclude them from the input representation.

Each action is represented by its action type and target location coordinates. The action type is encoded as a seven-dimensional vector, in which each dimension maps to one of the seven action types as described earlier. The action target location is encoded as a *heatmap*. Each pixel in the heatmap is the probability of the pixel being the action target location. We use a heatmap rather than the raw coordinates to represent the action location because the raw coordinates are highly non-linear and more difficult to learn [35].

In summary, the representation of a *UI context*, i.e. the input feature for our interaction model, is a stack of images including one 2-channel image for the current UI state and three 3-channel images for three latest UI transitions (each transition include one 2-channel image for the UI state and one 1-channel image for the action). All images are scaled to the size of 180×320 pixels. For ease of learning, we also add one channel of zero padding for the current UI state. In the end, a *UI context* is represented as a 4×180×320×3 vector.

Given the *UI context* vector, the output of the interaction model is an "action" that is likely to be performed by humans in the current state. Note that the predicted "action" is not an actual acceptable action in the current UI state. Instead, it is a probability distribution of types and locations of the expected human-like actions. Specifically, the goal of the model is to learn two conditional probability distributions:

1) $p_{type}(t \mid context_i)$ where $t \in \{touch, long_touch, swipe_up, ...\}$, meaning the probability distribution of t, the type of the next action a_i , given the current UI context.

2) $p_{loc}(x,y \mid context_i)$ where $0 < x < screen_width$ and $0 < y < screen_height$, meaning the probability distribution of the target location x,y of the next action a_i , given the current UI context.

Figure 4 shows the deep neural network model used to learn the two conditional probability distributions defined above. It accepts the representation of the current $UI\ context\ context_i$ as input, and outputs location and type distributions of a_i . The model consists of five main components: convolutional layers, residual LSTM modules, de-convolutional layers, a fully connected layer and loss functions.

Convolutional layers. Convolutional network structure has become a popular approach for image feature extraction, since it has been proved very powerful in computer vision tasks on large real-world datasets [36]. In our model, we use 5 convolutional layers with ReLU activations to extract features from UI skeleton images and action heatmaps. After each convolutional layer, there is a stride-2 max-pooling layer that reduces the width and height of its input to half. The pooling layers also help the model to identify UI elements having the same shape but different surroundings.

Residual LSTM modules. LSTM (Long-Short-Term Memory) networks are widely adopted in sequence modeling problems, such as machine translation [37], video classification [38], etc. In our model, extracting features from historical transitions is also a sequence modeling problem. We insert residual LSTM modules after each of the last 3 convolutional layers, in order to capture UI transition sequence features on different resolution levels. In a residual LSTM module, the last dimension of input and the output of the normal LSTM are directly added through a residual path.

Such residual structure makes the neural network easier to optimize [39], and gives hint that the location of an action should lie inside a UI element. To decrease model complexity, we also added a 1×1 convolutional layer before each residual LSTM module to reduce the feature dimension.

De-convolutional layers. This component is used to generate high-resolution probability distributions from the low-resolution output of residual LSTM modules. There are several options to accomplish this, such as bilinear interpolation, de-convolution, etc. We use de-convolutional layers, as it is easier to integrate with deep neural networks and more general than the interpolation methods. Features on different resolution levels are combined to improve the quality of generated heatmap [40]. A softmax layer is followed to normalize the generated heatmap so that all pixels in the heatmap sum to 1, which is the probability distribution of action locations.

Fully connected layer. A single fully connected layer with softmax is used to generate the probability distribution of action types.

Loss functions. The model predicts both the action location and action type as probability distributions. Thus their cross-entropy losses against the ground truth (the action

performed by humans) are suitable for model optimization. We use the sum of these two losses and a layer weight regularizer (to avoid overfitting) as the final loss function in the training process.

During training, each action a_i in an interaction flow ($\langle s_1, s_2, s_3, ..., s_n \rangle$, $\langle a_1, a_2, a_3, ..., a_n \rangle$) is converted to the following probability distributions:

$$p_{type}(t) = \begin{cases} 1 & t = a_i.type \\ 0 & \text{otherwise} \end{cases}$$

and

$$p_{loc}(x,y) = f(x - a_i.x, y - a_i.y)$$

where f is the density function of the Gaussian distribution with variance = 20. We use the Gaussian distribution to approximate the probability distribution of actual screen coordinates recognized by a device, when the same UI element is interacted by many people for many times.

Similarly, when applying the model, we feed it with the representation of the current UI state to predict the probability distributions $p_{type}(t)$ and $p_{loc}(x,y)$ for the next action. As the predicted distributions cannot be directly used to guide test generation, we need to further convert them to the probabilities of the actions that can be performed in the current state. In order to do that, we first traverse the UI tree to find all possible actions in the current state, with each action containing the action type (denoted as action.type) and the action target element (denoted as action.element). Then we calculate the probability of each action based on the distribution predicted by the model:

$$p(action) = p_{type}(action.type) * \sum_{x,y \text{ in action.element}} p_{loc}(x,y)$$

The action probabilities can finally be used to guide test input generation in the next step.

3.4 Test Input Generation with Humanoid

In this section, we describe how we apply the learned human interaction model to generate human-like test inputs.

Humanoid generates two types of test inputs, including explorations and navigations. *Exploration inputs* are used to discover the unseen behaviors in an app, while *navigation inputs* drive the app to known states that contain unexplored actions. When choosing from exploration inputs, the test generator does not know about the consequences of each test input, and the decision is made based on the guidance of the human interaction model (traditional test generators usually choose the input randomly). When generating navigation inputs, the test generator knows the target states of the input, as it has saved the memory of the transitions.

Similar to many existing test generators, Humanoid uses a GUI model to save the memory of transitions. The GUI model we use is represented as *UI transition graphs* (UTG in short), which is a directed graph, whose nodes are UI states and edges are the actions that lead to UI state transitions. The UTG is constructed at runtime: each time the test generator observes a new state s_i , it adds a new edge $< s_{i-1}, a_{i-i}, s_i >$ to the UTG, where s_{i-1} is the last observed UI state and a_{i-i} is the action performed in s_{i-1} . Figure 5 shows an example of UTG. With the UTG, the test

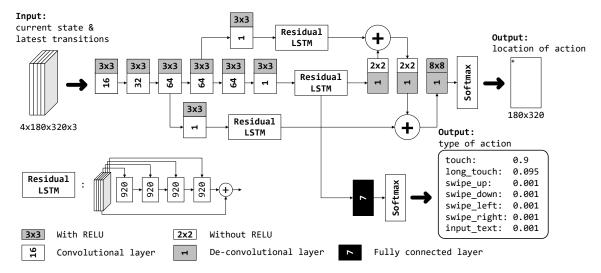


Fig. 4: The architecture of the deep learning model used in Humanoid. The model takes the representation of the current UI context (current UI state and latest UI transitions) as input, and outputs a probability distribution of the next action (including the action type and the action location).

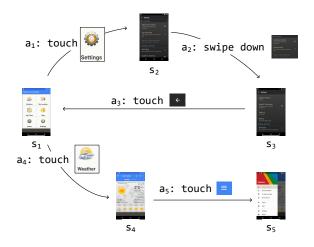


Fig. 5: An example of a UI transition graph (UTG).

generator can navigate to any known state by following the path to the state.

To decide between exploration and navigation and generate the input action, Humanoid adopts a simple but effective strategy, which is shown in Algorithm 1. In each step, Humanoid checks whether there are unexplored actions in the current state. Humanoid chooses exploration if there are unexplored actions (line 8), and chooses navigation if the current state is fully explored (line 10 to 12). The navigation process is straightforward. In the exploration process, Humanoid gets the probabilities of the actions predicted by the interaction model, and makes a weighted choice based on the probabilities. Since the actions that humans would take will be assigned higher probabilities, they get higher chances to be chosen by Humanoid as test inputs. Thus the inputs generated by Humanoid are more human-like than randomly chosen ones.

Compared to the existing testing tools, the main feature of Humanoid (and the main difference between different model-based test generators) is how the exploration input **Algorithm 1** The biased random search strategy used by Humanoid to generate inputs.

- 1: Load the pretrained interaction model M
- 2: Create an empty UTG $G = \langle S, E \rangle$
- 3: Start the app under test
- 4: Observe current UI state s and add s to S
- 5: repeat
- 6: Extract all actions in s as A, $A_1 \subset A$ is the set of unexplored actions, $A_2 \subset A$ is the set of explored actions.
- 7: **if** A_1 is not empty **then**
- 8: Choose action $a \in A$ with the highest probabilities predicted by M
- 9: **else**
- 10: Get $s' \in S$ that has the most unexplored actions
- 11: Get the shortest path p from s to s' in G
- 12: Choose the first action in p as a
- 13: **end if**
- 14: Perform action a
- 15: Observe the new UI state s_{new} and add s_{new} to S
- 16: Add the edge $\langle s, a, s_{new} \rangle$ to E
- 17: **until** all actions in all states in S have been explored

is chosen (line 8). Humanoid prioritizes the more valuable actions in exploration based on the interaction model, which has been trained from human interaction traces. This feature makes it faster to discover the correct input sequences, which in turn will drive the app into important UI states, thus leading to higher test coverage.

4 EVALUATION

We evaluate Humanoid by primarily looking at the following aspects:

1) Can Humanoid's deep learning model learn and predict correct human interaction patterns? What is its prediction accuracy? What is the learning and predicting cost? (Section 4.2)

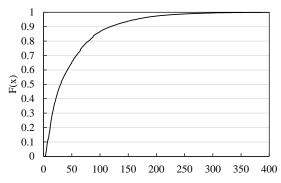


Fig. 6: CDF for the number of actions for each UI state.

- 2) Can Humanoid generate effective inputs for new Android apps? How does it compare with existing testing tools? (Section 4.3)
- 3) How effective is the interaction model learned from human traces? Is the high coverage brought by the interaction model? (Section 4.4)

4.1 Experimental Setup

The dataset we used to train the Humanoid model is processed from Rico [12], a large crowd-sourced dataset of human interactions. We extracted interaction flows from the raw data by identifying action sequences and state sequences. In the end, we obtained 12,278 interaction flows belonging to 10,477 apps. Each interaction flow contains 24.8 states on average. The cumulative distribution function (CDF) for the number of possible actions in each UI state is shown in Figure 6. On average, each UI state has 50.7 possible action candidates, while more than 10% of the UI states include more than 100 possible action candidates.

The machine we used to train and test the interaction model is a workstation with two Intel Xeon E5-2620 CPUs, 64GB RAM and an NVidia GeForce GTX 1080 Ti GPU. The operating system of the machine was Ubuntu 16.04. The model was implemented with Tensorflow [41].

For experiments on testing real Android apps, we used 4 computers with the same hardware and software as above, and ran 4 instances of Android emulators on each machine to test apps in parallel. The Android apps we used during testing evaluation include 68 open-source apps obtained from AndroTest [13], a commonly used dataset for evaluating Android test input generators, and 200 popular commercial apps downloaded from Google Play. We used Emma [42] to measure line coverage when testing the open-source apps. For the market apps without source code available, we used activity coverage (the percentage of reached activities, an activity for an Android app is like a webpage for a website) instead to measure the testing performance.

4.2 Evaluation of the Deep Learning Model

In this experiment, we trained and tested our interaction model on existing human interaction traces to evaluate whether the model is able to learn how humans interact with apps.

TABLE 2: Top-N accuracy of the Humanoid interaction model in prioritizing human-generated actions, as compared with the random strategy.

N	Random top-N accuracy	Humanoid top-N accuracy
1	5.8%	51.2%
3	17.5%	67.6%
5	29.1%	74.8%
10	58.3%	85.2%

4.2.1 Model Accuracy

We randomly selected 100 apps from the dataset and used their interaction traces for testing. The interaction traces for the remaining 10,377 apps were used for training. In total, we had 302,382 UI states for training and 2,594 UI states for testing. For each UI state in the testing set, we used the interaction model to predict the probabilities for all possible actions and sort the actions in the descending order of predicted probabilities. The actions performed by humans were considered as the ground truth.

Table 2 shows the accuracy of the Humanoid interaction model in prioritizing the human-performed actions in each UI state. Specifically, we calculated the probability that the ground truth (the human-performed action) ranks within top N (N=1,3,5,10) in the order of actions predicted by the interaction model. For comparison, we also calculated the top-N accuracies for the random strategy, i.e. the probability that the ground truth ranks top N if the actions are in random order. According to the results, our interaction model can identify and prioritize the human-generated actions with a higher accuracy. In particular, Humanoid was able to assign the highest probability to the human-generated action for more than 50% of the UI states. We also calculated the percentile rank of the human-generated action in each UI state. The mean percentile rank was 20.6% and the median was 9.5%, meaning that Humanoid was able to prioritize the human-like actions into the top 10% for most UI states.

4.2.2 Training and Prediction Cost

We then evaluated the cost of the interaction model. It took about 66 hours to train the interaction model with the dataset that contains 304,976 human-generated actions. It is acceptable since the model only needs to be trained once before being used for testing. The average time spent to predict the probabilities of actions for a UI state was 107.9 milliseconds. Given the fact that it typically takes more than 2 seconds for an Android test generator to send a test input and wait for the new page being loaded, the time overhead that our interaction model would bring to the test generator is minimal.

4.3 Test Coverage

In this experiment, we used Humanoid to generate test inputs for Android apps. We evaluated the test generator by examining whether it can actually improve test coverage.

4.3.1 Testing on Open-source Apps

We test Humanoid with 68 open-source apps (none of them is in the training set of the interaction model) and compare it with six state-of-the-art test generators for Android, including Monkey [6], PUMA [16], Stoat [11], DroidMate [18],

TABLE 3: Line coverage achieved by each testing tool for each open-source app. MO, PU, ST, DM, SA, DB, HU are short for Monkey, PUMA, Stoat, DroidMate, Sapienz, DroidBot, and Humanoid respectively. "-" means that the tool crashed during testing.

ID	App package name	MO	PU	ST	DM	SA	DB	HU
1	com.example.amazed	67%	50%	57%	14%	60%	63%	67%
2	com.example.anycut	61%	61%	11%	48%	59%	62%	63%
3	com.google.android.divideandconquer	80%	-	45%	45%	71%	52%	58%
4	com.android.lolcat	24%	4.40/	23%	17%	22%	23%	27%
5 6	info.bpace.munchlife	58% 47%	44% 29%	40% 35%	44% 39%	56% 46%	41% 43%	65% 58%
7	org.passwordmaker.android	19%	22%	24%	12%	27%	22%	27%
8	com.google.android.photostream com.bwx.bequick	40%	33%	30%	35%	38%	37%	46%
9	com.example.android.musicplayer	53%	51%	42%	52%	52%	52%	53%
10	com.google.android.opengles.spritetext	61%	58%	60%	2%	58%	59%	59%
11	com.nloko.android.syncmypix	20%	-	11%	4%	19%	20%	21%
12	com.google.android.opengles.triangle	76%	72%	76%	19%	72%	75%	75%
13	a2dp.Vol	33%	25%	19%	16%	35%	32%	39%
14	org.jtb.alogcat	60%	64%	57%	66%	63%	65%	62%
15 16	aarddict.android	13% 64%	12% 15%	12% 29%	12% 54%	13% 61%	12% 48%	13% 62%
17	net.sf.andbatdog.batterydog be.ppareit.swiftp_free	15%	13/6	13%	14%	12%	16%	17%
18	caldwell.ben.bites	30%	15%	19%	16%	28%	16%	36%
19	ch.blinkenlights.battery	74%	42%	50%	13%	72%	45%	75%
20	com.addi	18%	17%	16%	17%	18%	17%	18%
21	com.angrydoughnuts.android.alarmclock	64%	-	36%	41%	65%	61%	66%
22	com.chmod0.manpages	42%	47%	50%	57%	-	52%	42%
23	com.everysoft.autoanswer	14%	13%	11%	10%	13%	13%	13%
24	com.gluegadget.hndroid	15%	9%	7%	3%	9%	6%	9%
25	com.hectorone.multismssender	42%	23%	25%	24%	55%	26%	51%
26 27	com.irahul.worldclock com.kvance.Nectroid	90% 32%	85% 27%	86% 29%	87% 27%	87% 42%	87% 27%	93% 53%
28	com.morphoss.acal	14%	27 /0	13%	27 /0	23%	16%	22%
29	com.teleca.jamendo	37%	42%	14%	3%	21%	22%	43%
30	com.templaro.opsiz.aka	53%	42%	46%	61%		54%	82%
31	com.tum.yahtzee	44%	8%	10%	7%	8%	10%	59%
32	com.zoffcc.applications.aagtl	15%	-	15%	12%	16%	14%	19%
33	de.homac.Mirrored	38%	29%	36%	8%	29%	22%	59%
34	org.dnaq.dialer2	36%	31%	37%	35%	38%	39%	40%
35	edu.killerud.fileexplorer	24%	31%	20%	14%	40%	14%	16%
36 37	demo.killerud.gestures	36%	22%	28% 29%	22% 48%	22% 53%	32% 62%	32%
38	com.smorgasbork.hotdeath	62% 31%	41% 28%	27%	24%	29%	31%	54% 29%
39	hu.vsza.adsdroid i4nc4mp.myLock	29%	23%	28%	26%	25%	27%	27%
40	in.shick.lockpatterngenerator	69%	52%	46%	57%	71%	56%	76%
41	jp.sblo.pandora.aGrep	40%	25%	20%	23%	45%	26%	47%
42	com.fsck.k9	4%	3%	3%	3%	4%	3%	6%
43	net.jaqpot.netcounter	19%	33%	18%	16%	43%	16%	22%
44	org.beide.bomber	73%	47%	55%	24%	64%	50%	68%
45	org.jfedor.frozenbubble	71%	40%	50%	9%	52%	52%	61%
46	org.liberty.android.fantastischmemo	19%	22%	15%	4%	27%	30%	31%
47 48	org.scoutant.blokish	51% 20%	32% 15%	35% 23%	21% 16%	42% 14%	44% 19%	46% 24%
49	org.smerty.zooborns org.waxworlds.edam.importcontacts	32%	4%	6%	38%	23%	36%	41%
50	org.wikipedia	23%	20%	23%	11%	29%	20%	30%
51	com.android.keepass	6%	2%	4%	4%	7%	5%	9%
52	hiof.enigma.android.soundboard	42%	31%	31%	42%	31%	42%	42%
53	net.everythingandroid.timer	73%	51%	50%	49%	58%	54%	75%
54	com.ringdroid	16%	21%	1%	17%	15%	19%	17%
55	com.android.spritemethodtest	54%	34%	28%	67%	79%	60%	81%
56	com.beust.android.translate	41%	29%	31%	27%	44%	29%	48%
57	com.eleybourn.bookcatalogue	19%	3%	4%	3%	10%	11%	20%
58 59	org.tomdroid org.wordpress.android	42% 4%	38% 4%	32% 3%	18% 3%	40% 3%	38% 4%	49% 5%
60	com.evancharlton.mileage	21%	-17.0	15%	27%	34%	30%	29%
61	cri.sanity	16%	11%	12%	14%	15%	16%	19%
62	org.jessies.dalvikexplorer	17%	46%	33%	50%	69%	49%	33%
63	jp.gr.java_conf.hatalab.mnv	25%	24%	18%	16%	28%	19%	35%
64	org.totschnig.myexpenses	33%	39%	24%	9%	44%	42%	44%
65	net.fercanet.LNM	38%		28%	32%	36%	33%	32%
66	net.mandaria.tippytipper	64%	46%	35%	8%	77%	47%	85%
67	es.senselesssolutions.gpl.weightchart	47%	19%	25%	21%	39%	52%	52%
68	de.freewarepoint.whohasmystuff	53%	53%	36%	47%	60%	56%	67%
	Overall	39%	31%	28%	26%	39%	35%	43%

Sapienz [10] and DroidBot [15]. All tools were used with their default configurations. The input speeds of PUMA, Stoat, DroidMate and DroidBot were close to 600 events/hour, as they all need to read the UI state before performing an action and wait for the state transition after sending input, while Monkey and Sapienz could send input events with a very high speed (about 6000 events/hour in our experiments).

We used each testing tool to run each open-source app for one hour. In order to accommodate the recent market apps, most of the tools were evaluated on Android 6.0, as it was supported by most of the tools (some with minor modification). However, as Sapienz is close-sourced and only supports Android 4.4, so it was evaluated on Android 4.4 instead. For each app and tool, we recorded the final coverage and the progressive coverage after each action was performed. We repeated this process three times and used the average as the final results.

The overall comparison of the line coverage achieved by

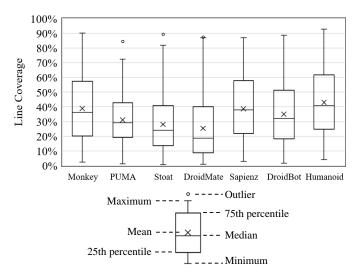


Fig. 7: Line coverage comparison of different testing tools over open-source apps.

the testing tools on open-source apps is shown in Figure 7. On average, Humanoid achieved a line coverage of 43.3%, which is the highest across all test input generators.

It is interesting to see that Monkey, which adopts a random exploration strategy, achieved higher coverage than all other model-based testing tools except Humanoid. The fact that Monkey performs better than most other testing tools has been also confirmed by other researchers [43]. Because Monkey is able to generate much more inputs than other tools in the same amount of time. However, our work demonstrated the benefits of the extensibility of model-based approaches. Model-based testing tools have great potential to achieve better test performance if the GUI information is properly used.

The detailed line coverage for each app is shown in Table 3. For some apps such as #3, #36, and #45, Monkey's coverage was significantly higher. The reason was that Monkey can generate many types of inputs (such as intents and broadcasts) that were not supported by other tools. For most of the other apps, Humanoid achieved the best results, especially for apps such as #6, #18, #30.

We further investigated why Humanoid was able to outperform other testing tools by examining the detailed test traces. We carefully inspected five apps on which Humanoid achieved significantly higher coverage. We found several cases where Humanoid behaved better than others, as illustrated in Table 4. To sum up, the high coverage of Humanoid was mainly due to two reasons: First, Humanoid was able to identify and prioritize the critical UI elements when there were plenty of UI elements to choose from. Second, Humanoid had a higher chance to perform a meaningful sequence of actions, which can drive the app into new and unexplored core functionalities.

Figure 8 shows the progressive coverage *w.r.t.* the number of input events sent by each testing tool. Note that we did not include Sapienz in the progressive coverage figures because it sends events too fast and we could not slow it down as it was close-sourced. In the first few steps, the line coverage of all testing tools increased rapidly, as the

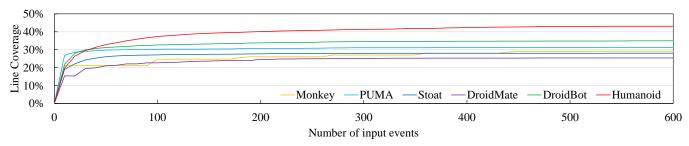


Fig. 8: Progressive line coverage for open-source apps.

TABLE 4: Some app examples where Humanoid outperforms other test generators.

	Coverage				
App Package Name	Humanoid	Best of Others			
org.passwordmaker.android	58%	47%			
- In this app, users can genera	ate a password	l with a hash algo-			
rithm by selecting text, hash le	vel, and hash r	nethod one by one.			
Humanoid predicted higher probabilities for these actions. Thus					
it was able to try more hash als	gorithms in a fi	ixed length of time.			
com.kvance.Nectroid	53%	32%			
- This app is a music player ir	which users o	can create a custom			
playlist before playing music.	Humanoid cor	npleted the process			
of creating a playlist, while oth	ers failed to do) so.			
com.templaro.opsiz.aka	82%	61%			
 A core functionality of this ap 	p is converting	a text message into			
morse code audio. To use this	functionality,	users have to input			
some text, click an Option but	tton, and click	a Create button in			
the order. Humanoid could generate the correct action sequence					
with higher probability as compared with other tools.					
com.tum.yzahtzee	59%	44%			
- This app serves a number game with two text boxes and a play					
button displayed on the screen. Users need to input two numbers					
and then click play to start the game. Since Humanoid would					
raise the probability of inputting text and touching the play					
button, it got higher chance to start the game. Other tools kept					
interacting with the keyboard because it contains many clickable					
UI elements.		•			

85%

- There were many buttons on each UI state in this app. The most important ones include a small OK button and a Split Bill button. Humanoid was able to prioritize these two buttons based

net.mandaria.tippytipper

on the spatial information of UI.

apps were just started and all UI states were new. PUMA achieved the highest coverage in the first 10 steps because it had a strategy to restart the app at the beginning, which led to the coverage of resource recycling code in many apps. Humanoid started to lead after about 20 events. That was because the easy-reachable code was already covered at that point, and the other states were hidden behind specific interactions that can hardly be produced by other testing tools.

At the 600th event point, the line coverage of most testing tools had almost converged, except for Monkey. This was because that the random strategy of Monkey produced a lot of ineffective and duplicated input events, which was not helpful for fast coverage improvement with regard to the number of events. However, Monkey was capable of generating much more events during the same amount of time. Its coverage continued to increase after the 600th step and finally reached about 39% at the end of the one-hour testing duration.

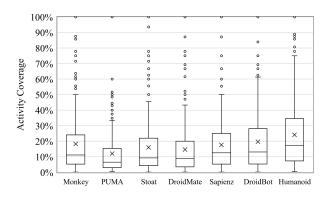


Fig. 9: Activity coverage comparison of the testing tools over popular market apps.

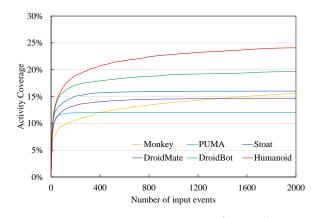


Fig. 10: Progressive activity coverage for market apps.

4.3.2 Testing on Market Apps

Finally, we evaluated Humanoid with 200 popular Android apps. The experimental setup was the same as for open-source apps, however, we ran the testing process for 3 hours for each market app, as these apps are typically much larger and can reach much more UI states. Compared with the open-source apps, market apps usually have more complicated functionalities and UI structures. Thus we use this experiment to check whether Humanoid is effective for real-world scenarios. As mentioned earlier, because we do not have source code of these apps, we use activity coverage to evaluate the testing efficiency.

The final activity coverage achieved by the testing tools and the progressive coverage are shown in Figure 9 and Figure 10, respectively. Similar to open-source apps, Humanoid also achieved the highest coverage (24.1%) as compared

with other tools. Due to the complexity of market apps, the coverage for some apps was not converged at the end of testing. However, we believe that Humanoid will keep the advantage even with longer testing time.

4.4 Effectiveness of the Learned Model

Due to the complexity of Humanoid's input generation algorithm, it is unclear whether the high coverage of Humanoid is brought by the interaction model learned from human traces.

To check the effectiveness of the learned interaction model in improving coverage, we created a baseline algorithm by replacing Humanoid's action selection logic (line 8 in Algorithm 1) to a random policy. Then we used the baseline algorithm to test the same set of apps.

We found that the baseline algorithm without the learned interaction model almost achieved the same coverage as Humanoid, which means the human interaction model was not the main reason of coverage improvement.

5 LIMITATIONS AND FUTURE WORK

More types of inputs. There are some types of inputs, such as system broadcasts and sensor events, that are not considered in this paper. This is a limitation of Humanoid because these inputs are difficult to collect from human interactions and they are also hard to represent in our deep learning model. However, it is not a severe problem as most apps can be well-tested without these actions. Humanoid also does not predict the text when sending text input actions, which can be fixed in the future by extending the interaction model or integrating other text input generation techniques [44].

Further improvement in testing coverage. Although Humanoid has been able to improve the coverage significantly from existing testing tools, the testing coverage is still far from perfect. In particular, the coverage is less than 10% for some apps that require specific inputs such as email addresses and account/passwords. A possible solution is to design better ways of semi-automated testing, in which human testers can provide necessary guidance to the automatic tool with minimal efforts.

Making use of textual information. When learning the human interaction patterns, we use the UI skeleton to represent each UI state in our model, while the text in each UI element is not used. The textual information is very important for humans when using mobile apps. We believe the performance of Humanoid can be further improved if the text information can be properly represented and trained with the help of some natural language processing (NLP) techniques.

More detailed testing on market apps. As we do not have source code on real market apps, we cannot testing them in more details as we did for the open-source apps. However, it might be possible to reverse engineer or instrument these apps (may need to consider obfuscation as well), so that we can evaluate the testing results in more detail, for example what kind of activities are more difficult to cover, and identifying opportunities to improve Humanoid. We will consider this in our future work.

6 CONCLUDING REMARKS

This paper introduces Humanoid, a new GUI test input generator for Android apps that is able to predict user-preferred inputs through deep learning. Humanoid adopts a deep neural network model to learn which UI elements are more likely to be interacted by end-users and how to interact with it, from a large set of human-generated interaction traces. Experiments show that the learned model is able to accurately predict real human interaction for an Android app, but whether the learned model is effective in improving test coverage is unclear.

REFERENCES

- [1] Wikipedia, "App store," https://en.wikipedia.org/wiki/App_ Store_(iOS), 2018, accessed: 2018-08-06.
- [2] —, "Google play," https://en.wikipedia.org/wiki/Google_ Play, 2018, accessed: 2018-08-06.
- [3] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an informationflow tracking system for realtime privacy monitoring on smartphones," ACM Transactions on Computer Systems (TOCS), vol. 32, no. 2, p. 5, 2014.
- [4] F. Dong, H. Wang, L. Li, Y. Guo, T. F. Bissyandé, T. Liu, G. Xu, and J. Klein, "Frauddroid: Automated ad fraud detection for android apps," in Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, 2018, pp. 257–268.
- [5] B. Hu, B. Liu, N. Z. Gong, D. Kong, and H. Jin, "Protecting your children from inappropriate content in mobile apps: An automatic maturity rating framework," in *Proceedings of the 24th ACM Inter*national on Conference on Information and Knowledge Management. ACM, 2015, pp. 1111–1120.
- [6] A. Developers, "Ui/application exerciser monkey," 2012.
- [7] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013, 2013, pp. 224–234.
- [8] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012, 2012, pp. 258–261.
- [9] Y.-M. Baek and D.-H. Bae, "Automated model-based android gui testing using multi-level gui comparison criteria," in Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ser. ASE 2016, 2016, pp. 238–249.
- Engineering, ser. ASE 2016, 2016, pp. 238–249.

 [10] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016, pp. 94–105. [Online]. Available: http://doi.acm.org/10.1145/2931037.2931054
- [11] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 245–256.
- of Software Engineering. ACM, 2017, pp. 245–256.

 [12] B. Deka, Z. Huang, C. Franzen, J. Hibschman, D. Afergan, Y. Li, J. Nichols, and R. Kumar, "Rico: A mobile app dataset for building data-driven design applications," in Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology. ACM, 2017, pp. 845–854.
- [13] S. R. Choudhary, A. Gorla, and A. Orso, "Androtest," http://bear. cc.gatech.edu/~shauvik/androtest/, 2018, accessed: 2018-08-06.
- [14] K. Mao, M. Harman, and Y. Jia, "Crowd intelligence enhances automated mobile testing," in *Automated Software Engineering (ASE)*, 2017 32nd IEEE/ACM International Conference on. IEEE, 2017, pp. 16–26.
- [15] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Droidbot: a lightweight ui-guided test input generator for android," in Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on. IEEE, 2017, pp. 23–26.

- [16] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps," in *Proceedings of the 12th Annual International Confer*ence on Mobile Systems, Applications, and Services, ser. MobiSys '14, 2014, pp. 204–217.
- [17] K. Jamrozik and A. Zeller, "Droidmate: A robust and extensible test generator for android," in *Proceedings of the International Con*ference on Mobile Software Engineering and Systems, ser. MOBILESoft '16, 2016, pp. 293–294.
- [18] N. P. Borges Jr, M. Gómez, and A. Zeller, "Guiding app testing with mined interaction models," in *Proceedings of the 5th Interna*tional Conference on Mobile Software Engineering and Systems. ACM, 2018, pp. 133–143.
- [19] W. Choi, G. Necula, and K. Sen, "Guided gui testing of android apps with minimal restart and approximate learning," in Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, ser. OOPSLA '13, 2013, pp. 623–640.
- [20] W. Wang, D. Li, W. Yang, Y. Cao, Z. Zhang, Y. Deng, and T. Xie, "An empirical study of android test generation tools in industrial cases," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: ACM, 2018, pp. 738–748. [Online]. Available: http://doi.acm.org/10.1145/3238147.3240465
- [21] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev, "Static window transition graphs for android (t)," in *Automated Software Engineering (ASE)*, 2015 30th IEEE/ACM International Conference on. IEEE, 2015, pp. 658–668.
- [22] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, "Appcontext: Differentiating malicious and benign mobile app behaviors using context," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 303–313.
- [23] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 59:1–59:11.
- [24] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, ser. OOPSLA '13, 2013, pp. 641–660.
- [25] R. Bhoraskar, S. Han, J. Jeon, T. Azim, S. Chen, J. Jung, S. Nath, R. Wang, and D. Wetherall, "Brahmastra: Driving apps to test the security of third-party components," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14, 2014, pp. 1021–1036.
- [26] M. Y. Wong and D. Lie, "Intellidroid: A targeted input generator for the dynamic analysis of android malware." in *The Network and Distributed System Security Symposium (NDSS)*, vol. 16, 2016, pp. 21–24.
- [27] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction," in *Proceedings of the* 36th International Conference on Software Engineering, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 1036–1046.
- [28] J. Rubin, M. I. Gordon, N. Nguyen, and M. Rinard, "Covert communication in mobile applications," in 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2015, pp. 647–657.
- [29] Y. Li, Y. Guo, and X. Chen, "Peruim: understanding mobile application privacy with permission-ui mapping," in *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. ACM, 2016, pp. 682–693.
- [30] T. Ringer, D. Grossman, and F. Roesner, "Audacious: User-driven access control with unmodified operating systems," in *Proceedings* of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2016, pp. 204–216.
- [31] C. Chen, T. Su, G. Meng, Z. Xing, and Y. Liu, "From ui design image to gui skeleton: a neural machine translator to bootstrap mobile gui implementation," in *Proceedings of the 40th International* Conference on Software Engineering. ACM, 2018, pp. 665–676.
- [32] R. Kumar, A. Satyanarayan, C. Torres, M. Lim, S. Ahmad, S. R. Klemmer, and J. O. Talton, "Webzeitgeist: design mining the web," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2013, pp. 3083–3092.

[33] K. Alharbi and T. Yeh, "Collect, decompile, extract, stats, and diff: Mining design pattern changes in android apps," in Proceedings of the 17th International Conference on Human-Computer Interaction with Mobile Devices and Services. ACM, 2015, pp. 515–524.

- [34] B. Deka, Z. Huang, and R. Kumar, "Erica: Interaction mining mobile apps," in *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*. ACM, 2016, pp. 767–776.
- [35] J. J. Tompson, A. Jain, Y. LeCun, and C. Bregler, "Joint training of a convolutional network and a graphical model for human pose estimation," in *Advances in neural information processing systems*, 2014, pp. 1799–1807.
- [36] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in Advances in neural information processing systems, 2012, pp. 1097–1105.
- [37] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in Advances in neural information processing systems, 2014, pp. 3104–3112.
- [38] J. Donahue, L. Anne Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell, "Long-term recurrent convolutional networks for visual recognition and description," in *Proceedings of the IEEE conference on computer vision and pattern* recognition, 2015, pp. 2625–2634.
- recognition, 2015, pp. 2625–2634.
 [39] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.
- [40] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proceedings of the IEEE* conference on computer vision and pattern recognition, 2015, pp. 3431– 3440.
- [41] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard et al., "Tensorflow: a system for large-scale machine learning." in OSDI, vol. 16, 2016, pp. 265–283.
- [42] V. Roubtsov, "Emma: a free java code coverage tool," 2006.
- [43] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet?" in *Proceedings of the* 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), ser. ASE '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 429–440.
- [44] P. Liu, X. Zhang, M. Pistoia, Y. Zheng, M. Marques, and L. Zeng, "Automatic text input generation for mobile testing," in *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 2017, pp. 643–653.