

A Deep Learning based Approach to Automated Android App Testing

Yuanchun Li[†], Ziyue Yang[†], Yao Guo[‡], Xiangqun Chen
 Key Laboratory of High Confidence Software Technologies (Ministry of Education)
 Peking University
 Beijing, China
 {liyanchun, ziyue.yang, yaoguo, cherry}@pku.edu.cn

Abstract—Automated input generators are widely used for large-scale dynamic analysis and testing 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 automated Android app testing. Our insight is that if we can learn from human-generated interaction traces, it is possible to generate human-like test inputs based on the visual information in the current UI state and the latest state transitions. 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), and show that we can successfully generate human-like inputs for any new UI based on the learned model. We then apply the model to automated testing of Android apps and demonstrate that it is able to reach higher coverage, and faster as well, than the state-of-the-art test input generators.

Index Terms—Software testing, automated test input generation, graphical user interface, deep learning, mobile application

I. 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 tested before being released, by app developers who want to gain confidence that their apps behave correctly and by the app markets who want to prevent malicious apps being published. However, due to the rapid releasing cycle and limited human resources, it is difficult for both developers and auditors to manually construct test cases in a short time. As a result, automated test input generators for mobile apps have been studied extensively in both academia and industry.

The test 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 test

(AUT), which can be used to detect software problems, such as bugs, vulnerabilities, and security issues. The effectiveness of a test input generator is often measured by its test coverage. Given unlimited time, one can potentially try all possible interaction sequences and combinations to achieve a perfect test coverage. However, in real-world situations where testing time is limited and the AUT may contain hundreds of GUI states and dozens of possible interactions in each state, a test input generator can only choose a small subset of interaction sequences to explore.

The key to success for an automated test input generator is to choose the correct interactions for a given UI (the current UI during testing), such that the chosen interactions may reach new and important UI states, which in turn will lead to additional UI states. Because it is hard for a machine to understand the GUI layout and the content within a GUI element, it is also difficult to determine which button to click or what should be inputted. As a result, most existing test generators [3]–[6] ignore the differences between various types of UI elements and apply a random strategy to choose one to interact with. Even some of them may maintain a GUI model of the app, the model is only used to remember explored states and interactions to avoid duplicated exploration. Although random strategies can also be further optimized, it has inherent limitations that make it difficult to choose the most efficient path to find the interactions that can drive the app into important states within a short time.

In contrary to random input generators, human testers can easily identify the UI elements that are worth interacting with, even for a new app they have never seen before. The reason is that human testers are themselves app users, so they have already gained some experience and knowledge about various mobile apps. Thus human testers know where to click and what to input, in order to achieve higher coverage, and taking less time as well. The key question we want to investigate in this paper is: *Can we teach an automated test input generator to behave like a human being?*

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

[†] co-primary authors, [‡] corresponding author



Action Type	UI Element	Probability
touch		0.7
touch		0.15
touch		0.1
touch		0.02
touch		0.015
touch		0.002
touch		0.001
touch		0.001
swipe_left		0.001
swipe_right		0.001
long_touch		0.0005
...

Fig. 1: An illustration of how Humanoid chooses test inputs for a UI state. The left side is a screenshot of the current UI of the AUT, 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.

and meaningfulness, thus generate test inputs that can lead to important states faster.

We can use the GUI page shown in Fig. 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 AUT, 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 AUT into additional important GUI states.

The core of Humanoid is a deep neural network model that predicts which UI elements are more likely to be interacted with by real users on a UI page and how to interact with it. 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, Humanoid is able to assign a probability to each action and choose the actions with higher probabilities as the next test input.

We implemented Humanoid and trained the interaction model with 304,976 human interactions extracted from a large-scale crowd-sourced UI interaction dataset Rico [7]. The model can be easily integrated with other testing tools by simply replacing the input selection logic.

To evaluate Humanoid, we first examined whether Humanoid can learn human interaction patterns by using it to prioritize the possible actions for each UI state in the 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 effectiveness of the interaction model in app testing, we compared Humanoid with six state-of-the-art test generators. The apps used for testing include 68 open-sourced apps obtained from the AndroTest [8] dataset, which is a widely-used benchmark dataset for evaluating Android test generators. We also tested 200 popular apps from Google Play, to see whether Humanoid is also effective in more complicated apps. According to the experiments, Humanoid was able to achieve 43.1% line coverage for open-source apps and 24.1% activity coverage for market apps, which was significantly higher than the best results (38.8% and 19.7%) achieved by other test generators using the same amount of time.

This paper makes the following main contributions:

- 1) To the best of our knowledge, this is the first work to introduce the idea of generating human-like test inputs with deep learning by mining GUI interaction traces, in order to improve automated mobile app testing.
- 2) We propose and implement Humanoid, a deep learning-based method to generate human-like test inputs by learning from human interaction traces.
- 3) We evaluate Humanoid with both open-source apps and popular market apps. The results show that Humanoid is able to achieve higher test coverage, and faster, than the state-of-the-art approaches.

II. BACKGROUND AND RELATED WORK

A. 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 tree¹. 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.

¹<https://developer.android.com/guide/topics/ui/declaring-layout>

B. Automated GUI Test Generation

Automated GUI test 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, testing tools 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 [3], 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 [4] 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 [9] makes use of a genetic algorithm to optimize random test sequences. Polariz [10] extracts and reuses motifs obtained by human testers to help generate random test sequences.

Several other testing tools build and use a *GUI model* of the app to generate test input. These models are usually represented as finite state machines that store the transitions between app window states. Such GUI models can be constructed dynamically [5], [6], [11]–[17] or statically [18]. Based on the GUI models, testing tools can generate events that can quickly navigate the app to unexplored states. Model-based strategies can be optimized in various ways. For example, Stoat [14] can iteratively refine the test strategy based on existing explorations, and DroidMate [15] 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 [19]. These testing tools [20]–[22] usually use sophisticated 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.

A key disadvantage of existing test generators is that they ignored the visual information of UI elements, which is an important reference when human users or testers are exploring an app. In Humanoid, we try to guide test generation by understanding how the GUI of an app may affect the way users interact with it.

C. 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.* [23] and Rubin *et al.* [24] proposed to detect stealthy behaviors in Android apps by comparing the actual behaviors with the UI. PERUIM [25] extracted the mapping between an app's permissions to its UI to help users understand why each permission is requested, and AUDACIOUS [26] provided a way to control permission access based on UI components. Chen *et al.* [27] 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 [28], [29] and interaction patterns [30] 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 [7], 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 testing.

III. OUR APPROACH

In order to employ human knowledge on mobile apps to augment mobile app testing, this paper proposes Humanoid, a new automated test input generator that is able to generate human-like test inputs based on automatically learned knowledge from human-generated app interaction traces. Similar to many existing testing tools, Humanoid uses a GUI model to understand and explore the behavior of the app under test. However, unlike traditional model-based approaches that randomly choose an action to perform when exploring a UI state, Humanoid prioritize the UI elements that are more likely to be interacted with by human users. We expect that such human-like exploration can drive the app into important states faster than random strategies.

A. Approach Overview

Fig. 2 shows the overview of Humanoid. The core of Humanoid is a machine learning 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

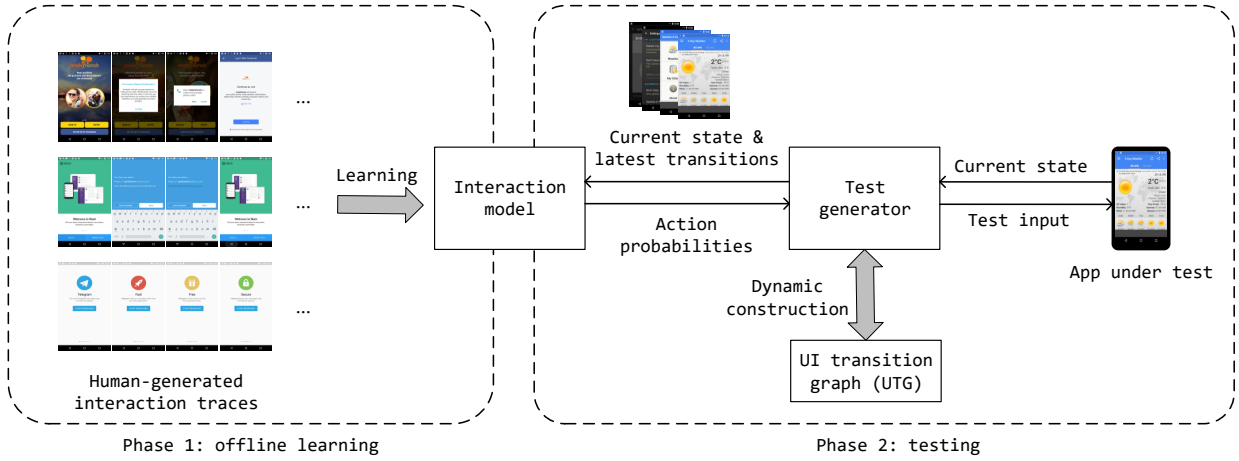


Fig. 2: Humanoid system overview.

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, Humanoid constructs a GUI model named *UI transition graph* (UTG) for the app under test (AUT). Both the GUI model and the interaction model are used by Humanoid to decide what test input to send. The UTG is responsible for guiding Humanoid to navigate between explored UI states, while the interaction model guides the exploration of new UI states.

B. Interaction Trace Preprocessing

First of all, we will need a large dataset (Rico [7]) with human interaction traces to train a user interaction model, 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 [7], 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 $\langle s_1, s_2, s_3, \dots, s_n \rangle$ and a sequence of actions $\langle a_1, a_2, a_3, \dots, a_n \rangle$ 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`. Each action is represented by the action type and the target location

TABLE I: Heuristic rules to extract actions from cursor movements. Note that $|loc_{end} - loc_{start}|$ means the Euclidean distance between loc_{start} and loc_{end} .

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

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 I.

Once we have extracted the sequence of actions $\langle a_1, a_2, a_3, \dots, a_n \rangle$, 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 $\langle s_1, s_2, s_3, \dots, s_n \rangle$. 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.

C. Model Training

This section explains in more details on how we use a machine learning model to learn human interaction patterns 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

Raw data		Data processed for model	
Screenshots	Actions	UI skeletons	Heatmaps
	touch CANCEL		
	touch ←		
	swipe left 		
	input text "2" 		

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

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.

Fig. 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 [31].

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 $4 \times 180 \times 320 \times 3$ 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) $p_{type}(t | context_i)$
where $t \in \{touch, long_touch, swipe_up, \dots\}$, meaning the probability distribution of t , the type of the next action a_i , given the current *UI context*.
- 2) $p_{loc}(x, y | context_i)$
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*.

Fig. 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 [32]. 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 [33], video classification [34], 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 [35], and gives hint that the location of an action should lie inside a UI element. To decrease model complexity,

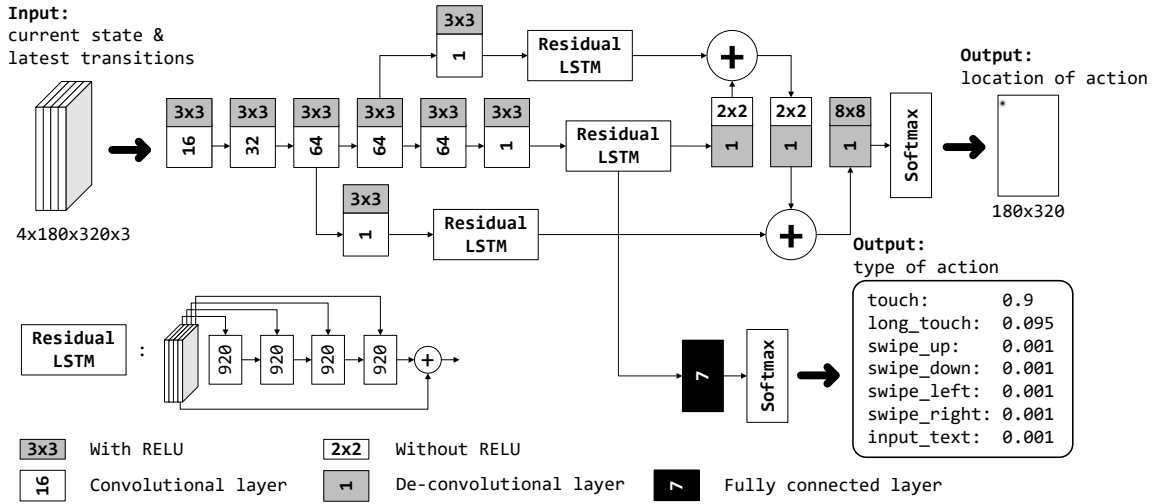


Fig. 4: The architecture of the interaction model in Humanoid. The model takes the representation of the current UI context (current UI state and latest UI transitions) as input, and outputs a probability distribution of the next action (including the action type and the action location).

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 [36]. 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 as the final loss function in the training process.

During training, each action a_i in an interaction flow ($\langle s_1, s_2, s_3, \dots, s_n \rangle$, $\langle a_1, a_2, a_3, \dots, a_n \rangle$) 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.

D. Guided Test Generation

In this section, we describe how we apply the 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

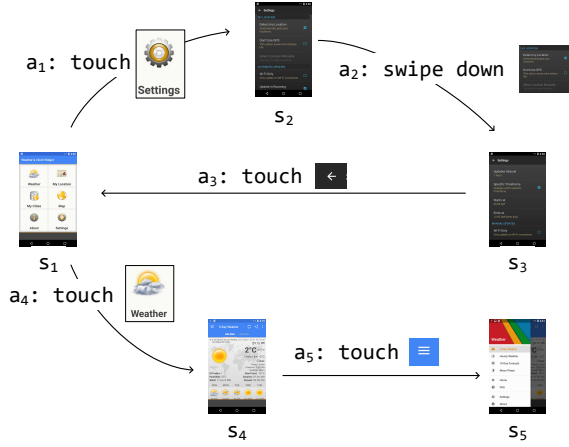


Fig. 5: An example of a UI transition graph (UTG).

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 $\langle s_{i-1}, a_{i-1}, s_i \rangle$ to the UTG, where s_{i-1} is the last observed UI state and a_{i-1} is the action performed in s_{i-1} . Fig. 5 shows an example of UTG. With the UTG, the test 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 input. 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 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 testing coverage.

IV. EVALUATION

We evaluated Humanoid by primarily looking at the following aspects:

- 1) Can Humanoid learn human interaction patterns? Specifically, can the interaction model predict user actions in a UI state with high accuracy?
- 2) How much extra time does Humanoid require to use the interaction model? Specifically, how long does it take to train the model and predict action probabilities?

Algorithm 1 The test generation strategy of Humanoid.

```

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 unexplored actions in  $s$  as  $A$ 
7:   if  $A$  is not empty then
8:     Choose an action  $a$  from  $A$  based on the probabilities
       predicted by  $M$ 
9:   else
10:    Get a state  $s'$  in  $S$  that has 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

```

- 3) Can Humanoid actually achieve higher and faster coverage when the trained interaction model is used to guide test generation?

We conducted two experiments to answer these questions. First, we used a dataset of human interaction traces to train and test the interaction model. We looked at the model accuracy and time efficiency in this experiment. Second, we integrated the model trained on the dataset to a test generator and used the test generator to conduct testing of two different sets of Android apps. We measured the test coverage and test progress of Humanoid and compared the results with several state-of-the-art testing tools.

A. Experimental Setup

The dataset we used to train and test the Humanoid model is processed from Rico [7], 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 contained 24.8 states on average. The cumulative distribution function (CDF) for the number of possible actions in each UI state is shown in Fig. 6. On average, each UI state has 50.7 possible action candidates, while more than 10% of UI states include more than 100 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 [37].

In the experiment of test coverage evaluation, we used 4 computers with the same hardware and software as the above one. We ran 4 instances of Android emulators on each machine to test apps in parallel. The apps we used to test include 68 open-source apps obtained from AndroTest [8], a commonly

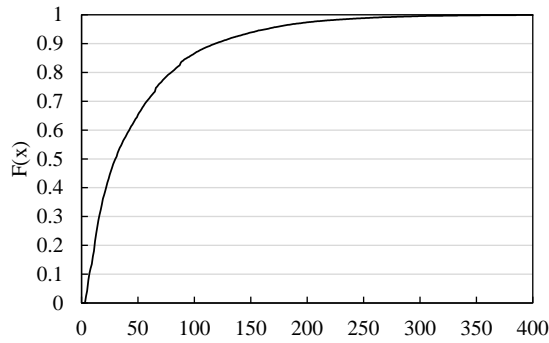


Fig. 6: CDF for the number of actions for each UI state.
TABLE II: Top-N accuracy of the Humanoid interaction model in prioritizing human-generated actions, as compared with the random strategy.

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%

used dataset for evaluating Android test input generators, and 200 popular commercial apps downloaded from Google Play. We used Emma [38] to measure line coverage when testing the open-source apps. For the commercial apps without source code available, we used activity coverage (the percentage of reached activities) instead to measure the testing performance.

B. Accuracy of the Interaction Model

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

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 action performed by humans was considered as the ground truth.

Table II 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.

C. Overhead of the Interaction Model

We then evaluated the overhead 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.

D. Coverage of Guided Testing

In this experiment, we used the interaction model trained in the previous experiment to guide test generation. We evaluated the guided test generator by examining whether it can actually improve test coverage.

We tested two sets of apps, including 68 open-source apps and 200 popular market apps. We compared Humanoid with six state-of-the-art test generators for Android, including Monkey [3], PUMA [12], Stoa [14], DroidMate [15], Sapienz [9] and DroidBot [11]. All tools were used with their default configurations. The input speeds of PUMA, Stoa, 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 1 hour and each market app for 3 hours. 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.

1) *Line Coverage for Open-Source Apps*: The overall comparison of the line coverage achieved by the testing tools on open-source apps is shown in Fig. 7. On average, Humanoid achieved a line coverage of 43.1%, 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 [39]. 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.

TABLE III: 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%	-	23%	17%	22%	23%	27%
5	info.bpace.munchlife	58%	44%	40%	44%	56%	41%	65%
6	org.passwordmaker.android	47%	29%	35%	39%	46%	43%	58%
7	com.google.android.photostream	19%	22%	24%	12%	27%	22%	27%
8	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.spritestext	61%	58%	60%	2%	58%	59%	59%
11	com.nloko.android.syncmypyx	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	aarddict.android	13%	12%	12%	12%	13%	12%	13%
16	net.sf.andbatdog.batterydog	64%	15%	29%	54%	61%	48%	62%
17	be.ppareit.swiftfp_free	15%	-	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.eversoft.autoanswer	14%	13%	11%	10%	13%	13%	13%
24	com.gluegadget.hndroid	15%	9%	7%	3%	9%	6%	9%
25	com.hectorone.multissmessenger	42%	23%	25%	24%	55%	26%	51%
26	com.irahul.worldclock	90%	85%	86%	87%	87%	87%	93%
27	com.kvance.Nectroid	32%	27%	29%	27%	42%	27%	53%
28	com.morphoss.acal	14%	-	13%	-	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.zoffce.applications.aagtl	15%	-	15%	12%	16%	14%	19%
33	de.homac.Mirrorred	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	demo.killerud.gestures	36%	22%	28%	22%	32%	32%	32%
37	com.smorgasbork.hotdeath	62%	41%	29%	48%	53%	62%	54%
38	hu.vszs.adsdroid	31%	28%	27%	24%	29%	31%	29%
39	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.fscck.k9	4%	3%	3%	3%	4%	3%	5%
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	org.scoutant.blokish	51%	32%	35%	21%	42%	44%	46%
48	org.smetry.zooborns	20%	15%	23%	16%	14%	19%	24%
49	org.waxworlds.edam.importcontacts	32%	4%	6%	38%	23%	36%	41%
50	org.wikipedia	23%	20%	23%	11%	29%	20%	29%
51	com.android.keeppass	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	org.tomdroid	42%	38%	32%	18%	40%	38%	49%
59	org.wordpress.android	4%	4%	3%	3%	3%	4%	5%
60	com.evancharlon.mileage	21%	-	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.senselessolutions.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%

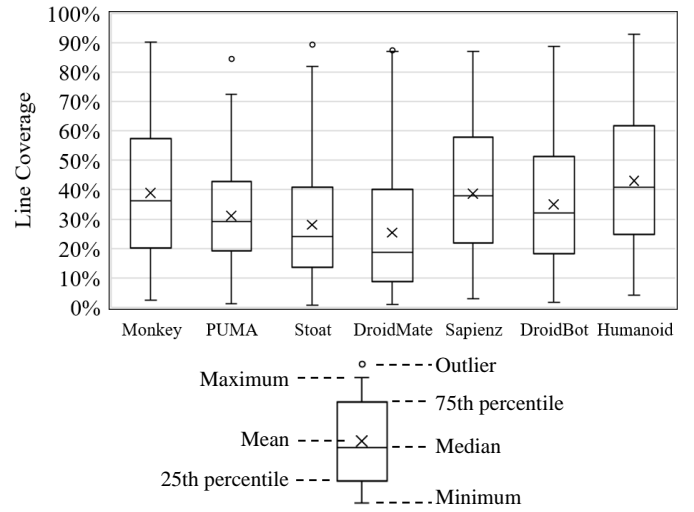


Fig. 7: Line coverage comparison of different testing tools over open-source apps.

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

App Package Name	Coverage	
	Humanoid	Best of Others
org.passwordmaker.android	58%	47%
- In this app, users can generate a password with a hash algorithm by selecting text, hash level, and hash method one by one. Humanoid predicted higher probabilities for these actions. Thus it was able to try more hash algorithms in a fixed length of time.		
com.kvance.Nectroid	53%	32%
- This app is a music player in which users can create a custom playlist before playing music. Humanoid completed the process of creating a playlist, while others failed to do so.		
com.templaro.opsiz.aka	82%	61%
- A core functionality of this app is converting a text message into morse code audio. To use this functionality, users have to input some text, click an <code>Option</code> button, and click a <code>Create</code> button in the order. Humanoid could generate the correct action sequence with higher probability as compared with other tools.		
com.tum.yahtzee	59%	44%
- This app serves a number game with two text boxes and a <code>play</code> button displayed on the screen. Users need to input two numbers and then click <code>play</code> to start the game. Since Humanoid would raise the probability of inputting text and touching the <code>play</code> button, it got higher chance to start the game. Other tools kept interacting with the keyboard because it contains many clickable UI elements.		
net.mandaria.tippytipper	85%	77%
- There were many buttons on each UI state in this app. The most important ones include a small <code>OK</code> button and a <code>Split Bill</code> button. Humanoid was able to prioritize these two buttons based on the spatial information of UI.		

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 III. 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 IV. 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 unexplored core functionalities.

Fig. 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 apps were just started and

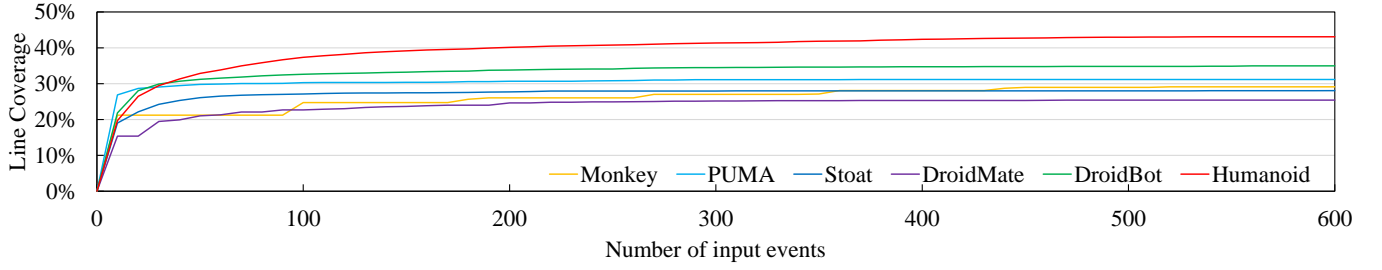


Fig. 8: Progressive line coverage for open-source apps.

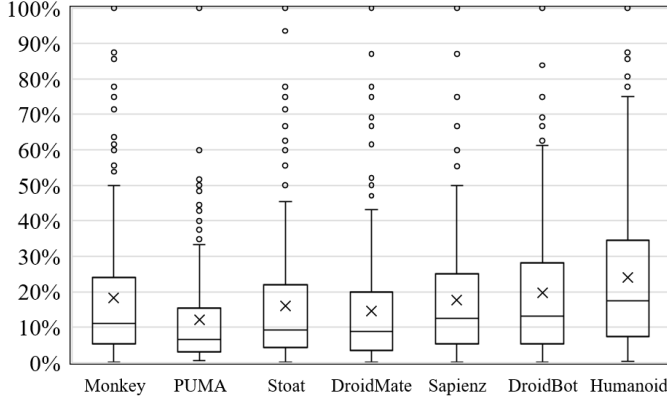


Fig. 9: Activity coverage comparison of the testing tools over popular market apps.

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 been 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 coverage improvement when we count for the number of events. However, Monkey was able to generate a lot of more events during the same amount of time. Its coverage will continue to increase after the 600th step and finally reached about 39% at the end of testing (for the same one hour testing duration).

2) *Activity Coverage for Market Apps*: As compared with the open-source apps, market apps usually have different and more complicated functionalities and UI structures. Thus we further conducted experiments on the market apps to see whether Humanoid is still more effective.

The final activity coverage achieved by the testing tools and the progressive coverage are shown in Fig. 9 and Fig. 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,

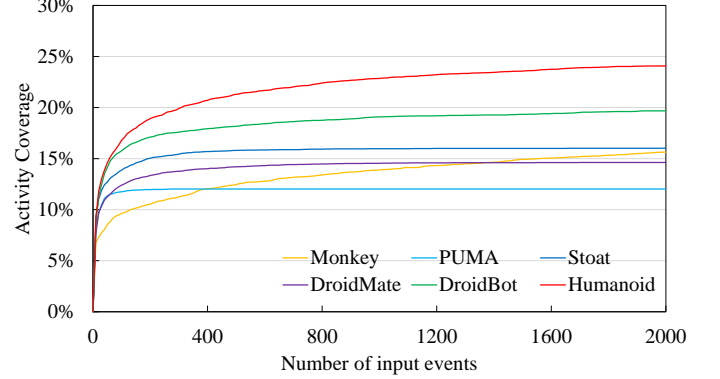


Fig. 10: Progressive activity coverage for market apps.

we believe that Humanoid will keep the advantage even with longer testing time. (Note the difference in the coverage of Monkey in these two figures due to the same reason as described above.)

V. LIMITATIONS AND FUTURE WORK

More types of inputs. There are some types of inputs, such as system broadcasts, sensor events, etc. 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 interaction model. However, it is not a huge problem currently as most apps can be well-tested without these actions. Humanoid also does not predict the text when sending text input actions, which could be fixed in the future by extending the model to support text prediction or integrating other text input generation techniques [40].

Further improvement in coverage. Although Humanoid has been able to improve the coverage significantly from existing testing tools, the test coverage is still relatively low (much worse than perfect coverage). In particular, the coverage is less than 10% for some apps. This is because many apps require specific inputs such as emails and passwords that are hard even impossible to generate automatically. 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 to use the app. We believe the performance of Humanoid can

be further improved if the text information can be properly represented and learned in the interaction model.

Learning from non-human interaction traces. Our method heavily relies on the human interaction traces, which might be hard to scale if we want to learn more interaction patterns from a larger dataset. Since what the model actually learned from the human interactions is the importance of actions in each UI, it is possible to directly train on machine-generated traces as long as the importance of actions can be analyzed.

VI. CONCLUDING REMARKS

This paper introduces Humanoid, a new GUI test input generator for Android apps that is able to generate human-like test 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. With the guidance of the learned model, Humanoid is able to accurately predict real human interaction with an Android app. According to experiments on a large number of open-source apps and popular commercial apps, Humanoid is able to achieve higher test coverage, and faster, than six state-of-the-art testing tools.

REFERENCES

- [1] Wikipedia, “App store,” [https://en.wikipedia.org/wiki/App_Store_\(iOS\)](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] A. Developers, “Ui/application exerciser monkey,” 2012.
- [4] 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.
- [5] 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.
- [6] 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.
- [7] B. Deka, Z. Huang, C. Franzen, J. Hibschan, D. Afegan, 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.
- [8] S. R. Choudhary, A. Gorla, and A. Orso, “Androtest,” <http://bear.cc.gatech.edu/~shauvik/androtest/>, 2018, accessed: 2018-08-06.
- [9] 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>
- [10] —, “Crowd intelligence enhances automated mobile testing,” in *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*. IEEE, 2017, pp. 16–26.
- [11] 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.
- [12] 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 Conference on Mobile Systems, Applications, and Services*, ser. MobiSys ’14, 2014, pp. 204–217.
- [13] K. Jamrozik and A. Zeller, “Droidmate: A robust and extensible test generator for android,” in *Proceedings of the International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft ’16, 2016, pp. 293–294.
- [14] 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.
- [15] N. P. Borges Jr, M. Gómez, and A. Zeller, “Guiding app testing with mined interaction models,” in *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. ACM, 2018, pp. 133–143.
- [16] 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.
- [17] 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>
- [18] 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.
- [19] 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.
- [20] 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.
- [21] 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.
- [22] 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.
- [23] 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.
- [24] 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.
- [25] 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.
- [26] 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.
- [27] 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.
- [28] 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.

- [29] 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.
- [30] 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.
- [31] 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.
- [32] 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.
- [33] 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.
- [34] 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.
- [35] 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.
- [36] 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.
- [37] 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.
- [38] V. Roubtsov, "Emma: a free java code coverage tool," 2006.
- [39] 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.
- [40] 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.