TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 04/31 pp55–67
DOI: 10.26599/TST.2024.9010038
Volume 30, Number 1, February 2025

CamDroid: Context-Aware Model-Based Automated GUI Testing for Android Apps

Hongyi Wang, Yang Li*, Jing Yang, Daqiang Hu, and Zhi Liao

Abstract: Recent years have witnessed the widespread adoption of mobile applications (apps for short). For quality-of-service and commercial competitiveness, sufficient Graphical User Interface (GUI) testing is required to verify the robustness of the apps. Given that testing with manual efforts is time-consuming and error-prone, automated GUI testing has been widely studied. However, existing approaches mostly focus on GUI exploration while lacking attention to complex interactions with apps, especially generating appropriate text inputs like real users. In this paper, we introduce CamDroid, a lightweight context-aware automated GUI testing tool, which can efficiently explore app activities through (1) a model-based UI-guided testing strategy informed by the context of previous event-activity transitions and (2) a data-driven text input generation approach regarding the GUI context. We evaluate CamDroid on 20 widely-used apps. The results show that CamDroid outperforms non-trivial baselines in activity coverage, crash detection, and test efficiency.

Key words: Android app; automated Graphical User Interface (GUI) testing; state transition model; text input generation

1 Introduction

With the widespread popularity of mobile phones, mobile applications (apps) have been an indispensable part of our daily life and have increased dramatically in number over recent years[1]. To maintain commercial competitiveness and keep user loyalty, it is crucial to adequately test these apps to guarantee their robustness. In practice, Graphical User Interface (GUI) testing for apps heavily involves human efforts. However, due to the rapid releasing cycle and limited human resources, manual GUI testing is time- and resource-consuming with poor activity coverages in limited test time. App stores also rely on automated testing to detect malicious apps before they are officially released[2–5]. Therefore, automated GUI testing for Android apps has been studied extensively in both academia and industry.

A variety of automated GUI testing approaches have been proposed, including model-based[6], probability-based[7], and deep learning based methods[8] to dynamically explore app activities by injecting actions (like clicking and scrolling) according to the detection and analysis of the current GUI components[9]. However, since quick feedback on the quality of new app features is required whenever a new internal version of the app is built[8], these approaches are mostly inefficient and ineffective in terms of continuous testing. This is because they rerun each version from scratch without learning from the context knowledge of previous testing runs to accelerate the current test. Moreover, inside one testing run, most of them focus on the GUI exploration algorithm

• Hongyi Wang, Yang Li, and Jing Yang are with School of Software, Tsinghua University, Beijing 100084, China. E-mail: hongyi-w21@mails.tsinghua.edu.cn; liyang14thu@gmail.com; yangj23@mails.tsinghua.edu.cn.
• Daqiang Hu and Zhi Liao are with Hangzhou Uusense Technology Inc., Hangzhou 310012, China. E-mail: hudaqiang@uusense.com; liaozhi@uusense.com.
* To whom correspondence should be addressed.
Manuscript received: 2023-10-25; revised: 2024-02-09; accepted: 2024-02-15

© The author(s) 2025. The articles published in this open access journal are distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/).

improvement while lacking attention to complex interactions with apps, especially generating appropriate text inputs like real users, leading to unsatisfactory app activity coverages.

To fill the above gaps, we propose CamDroid, a context-aware model-based automated GUI testing approach for Android apps. To effectively store the context knowledge of previous testing runs, CamDroid abstracts the key identities of feasible widgets with the allowed actions on the GUI page as events, and builds a state transition model to memorize the historical probabilities of the event-activity transitions (each of which represents the probability of an event to reach an app activity). To further leverage the context knowledge, CamDroid combines the one-step guidance of the state transition model with the multi-step guidance of reinforcement learning, aiming to reach deep activities requiring sequential event executions.

Furthermore, CamDroid learns the correlation of user characteristics and app metadata (collected from public datasets[10–20]) with a Generative Adversarial Network (GAN) and generates text inputs for diverse user profiles and app scenarios (each pair of profile and scenario information is encoded as a vector named "input context vector", which is corresponding to one generated text input) in advance. When an event requiring text inputs is selected by the aforementioned state transition model, CamDroid encodes the contexts of current GUI widget identities as a feature vector, and calculates BERTScores[21] of this vector with each of the input context vectors corresponding to the text inputs generated by GAN. Then CamDroid ranks the input context vectors by BERTScores and selects the text input corresponding to the one with the highest score. In this way, CamDroid can generate suitable text inputs like real users, and certain sequential activities requiring specific text inputs to reach can be effectively explored. The architectural overview of CamDroid is depicted in Fig. 1.

We implement the prototype of CamDroid based on the source of Android Monkey. To evaluate the effectiveness of CamDroid, we compare CamDroid with state-of-the-art testing tools (i.e., Monkey[22], APE[6], and Fastbot2[7]) on 20 large, widely-used apps from Google Play. Experiment results show that CamDroid achieves 1.25× (1.27×) higher average (median) activity coverage than the best baseline within one hour. Though the apps have been well tested, CamDroid manages to find 34 unique crashes in one hour, while the best baseline Fastbot2 finds 18. We also conduct ablation study to further demonstrate the effectiveness of our text input generation technique and its adaptability to other testing tools. Results show that the proposed text input generation technique can improve the activity coverages of Monkey, APE, and Fastbot2 by 1.13×, 1.19×, and 1.15×, respectively.

Roadmap. The remainder of the paper is organized as follows. In Section 2, we illustrate the problem and challenges. In Section 3, we present our algorithm CamDroid. In Section 4, we introduce the prototype setting and the evaluation results. We survey the related works in Section 5 and conclude this paper in Section 6.

2 Background and Motivation

In this section, we introduce the problem and challenges of the automated GUI testing for Android apps, especially issues related to text input generation.

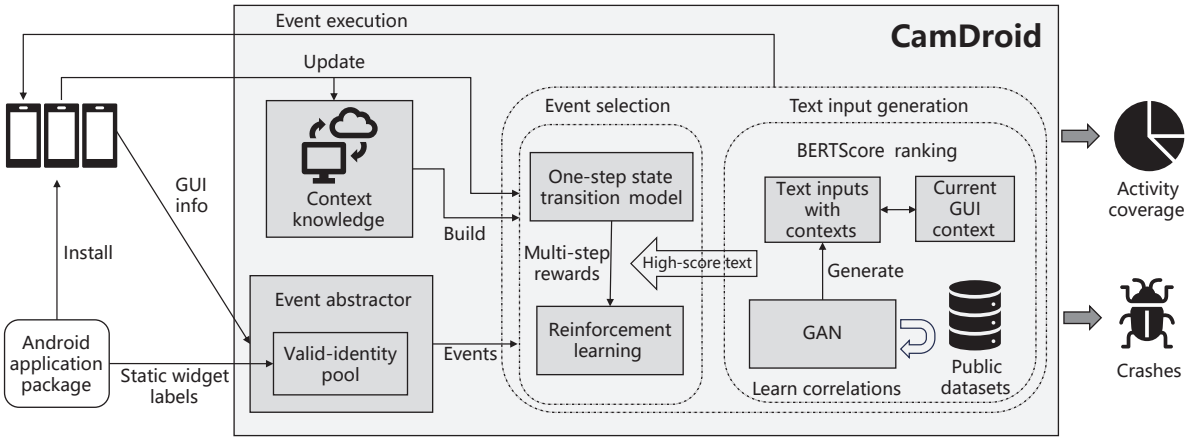


Fig. 1 Architectural overview of CamDroid.

Furthermore, we uncover why existing methods struggle to address these challenges.

2.1 Problem

With the rapid growth of the Android app market, ensuring the quality and reliability of Android apps becomes a critical concern for developers. However, mobile apps have short update cycles and encompass a wide range of scenarios. These characteristics make comprehensive testing of mobile apps challenging. Traditional manual testing is not only time-consuming and labor-intensive but also prone to overlooking corner cases. As a result, there is a growing demand for effective and efficient automated testing tools that can streamline the testing process, improve the testing coverage, and enhance the quality of Android apps.

2.2 Challenges

(1) Vast and dynamically changing search space

Considering that a mobile app has hundreds of activities and potentially thousands of widgets, the search space for app testing is very extensive. Random testing tools, like Monkey[22], suffer from getting stuck in loops early and no longer making progress. Additionally, since GUI pages often update dynamically, sometimes even a simple backward action fails to return to the exact previous GUI page. Consequently, the effectiveness of search-based tools based on graph traversal algorithms[23, 24] is not ideal.

Model-based tools[6, 7, 25] abstract GUI trees into different states and regard the page changes caused by events as transitions between states. They adapt to the dynamic search space by adjusting state abstractions or transition relationships. However, simply defining different GUI pages as different states leads to a large state space, thus reducing the testing efficiency.

Some model-based tools leverage coarse-grained abstractions to tackle the problem. Unfortunately, they face low accuracy in modeling app behaviors. Take Fastbot2[7] for example, it may consider two events with different functionalities as the same one, which may mislead event selections. This limitation makes it only suitable for specific apps (i.e., Douyin and Toutiao) and less effective for others. Therefore, balancing the trade-off between the size and precision of states is untrivial for model-based tools.

Worse still, some actions lead to irreversible changes in GUI pages and result in great differences between the search space before and after performing them. For example, a variety of pages cannot be accessed without the login operation. On the other hand, some activities can only be reached when the app is not logged in. However, existing tools fail to handle them properly.

Additionally, a model-based tool needs to initialize its model at the beginning of the testing. Considering the vast search space, the initialization can be time-consuming. For industrial app development, a single version update for an app rarely undergoes heavy-weight changes, which means that the model built in previous testing can be somehow reusable. However, most model-based tools fail to leverage this characteristic and instead rebuild the model from scratch after each version update, which is ineffective and inefficient.

(2) Text input generation

Most existing tools only focus on GUI exploration, while lacking attention to complex interaction with apps like text input generation. They just perform completely random inputs or input a few fixed words[25, 26]. However, most apps contain pages that require meaningful text inputs in the preceding page to proceed. As shown in Fig. 2, the test will be hindered if the tool can not enter meaningful content in the search box of X (former name Twitter). Therefore, testing tools failing to generate valid text inputs struggle to access these pages, resulting in limited activity coverages and difficulties in surpassing the bottleneck.

Automated text input generation is challenging. The algorithm needs to generate text that conforms to the GUI context and satisfies constraints, requiring strong Natural Language Processing (NLP) capabilities. Although the Large Language Model (LLM) demonstrates remarkable performance in the field of NLP, applying the inference of such a model for the input generation of app testing[27] would entail several seconds on generating and encoding the prompt for each text input. Consequently, the whole testing process would be slowed down by the inference overhead of the LLM, resulting in unsatisfactory test efficiency. As a result, maintaining test efficiency without significantly reducing the quality of generated text inputs is difficult.

Additionally, correlations may exist among a real user's multiple text inputs while interacting with an app. For example, when a user purchases a flight ticket and then proceeds to book accommodation in a travel app, the destination of the ticket and the city of the

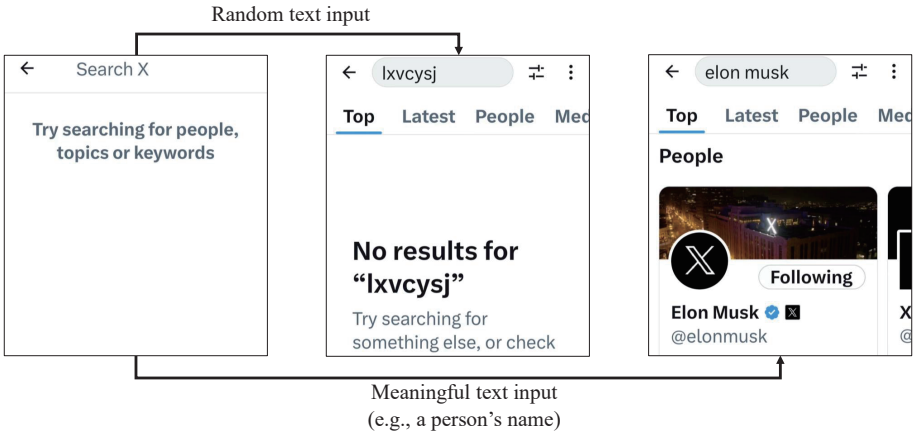


Fig. 2 Example of text input in Android app.

accommodation typically are the same.

However, realizing the above insight in practice is challenging. First, there is no public dataset of real user text inputs while using apps, while collecting data manually is time-consuming, labor-intensive, and lacks diversity. Therefore, it is difficult to conduct a large-scale study on interrelationships among text input scenarios within the same app. Second, further utilizing previous text inputs and user information for real-time text generation can lead to even longer prompt processing time and lower efficiency.

3 Approach

In this section, we present CamDroid, a context-aware model-based automated GUI testing tool that tackles the above challenges. Our efforts lie in two folds. First, we propose a model-based testing tool where we carefully design its state abstraction and enable it to leverage historical context knowledge from previous testing runs. Second, we introduce a lightweight approach to generate text inputs by incorporating GUI widget identities, user characteristics, and app metadata.

3.1 Overview of CamDroid

The workflow of CamDroid is illustrated in Fig. 1, which mainly consists of two components: event selection and text input generation. CamDroid first utilizes the GUI info to extract feasible events. The event selection component then combines the one-step guidance of the state transition model with the multi-step guidance of reinforcement learning to choose the best event. The text input generation component is activated when an event requiring text inputs is selected. This component utilizes a trained GAN to text inputs for various possible scenarios (each is encoded as an input context vector) before the test. During the test, it encodes the current GUI context into a feature vector and calculates BERTScores[21] of this vector with each of the input context vectors. It then selects the text corresponding to the highest-scoring input context vector to fill in.

3.2 Model-based testing tool

3.2.1 Model abstraction

In our model-based testing tool, we regard each activity as a state, and the transitions between states represent the switchings between activities. The key identities of interactive widgets on each page are extracted and considered as events. In this way, we construct a probability model M to represent the relationships between activities and widgets in an app. Specifically, M stores event-activity transition probabilities, each one of which is denoted as $p(e,a)$, where

- e represents for an event,
- a represents for an app activity, and
- p represents the probability of the event e to reach the app activity a.

During the test, the model M is updated whenever an event, e.g., e_i, is executed. The value of every $p(e_i,a_j)$ that associates with e_i is updated according to the following rule:

$$p(e_i,a_j)=\frac{n(e_i,a_j)}{n(e_i)} \tag{1}$$

where $n(e_i,a_j)$ represents the number of times reaching a_j after executing e_i, and $n(e_i)$ represents the total execution times of e_i. In order to reuse the context information from previous testing runs, CamDroid stores the model M as well as total execution times of

each event. M will be reconstructed from the aforementioned values before the next testing run.

Additionally, if each widget is considered as an individual event, the scale of M would be rather large. Therefore, we carefully design abstract rules for widgets to prevent redundant records of widgets with the same functionalities. In detail, we classify widgets into the same event if they share the following 6 properties: the `activity` to which the widget belongs, `content-description`, `text`, `resource-id`, `class-name`, and `allowed actions` (i.e., `click`, `long click`, `scroll`, and `text input`). According to sampling observations, there are tens of widget attributes that contain meaningful representatives of functionalities. Furthermore, based on statistical calculation results, the heterogeneity is most prominent among the aforementioned 6 properties, resulting in an effective abstraction.

The contents of the widget's `text`, `resource-ID`, and `content-description` sometimes overlap, as they often provide a brief summary of the widget's functionality, such as search or location. The reason for recording all three properties is that some widgets may have empty values for some of them (this is also why there is still significant heterogeneity among them). Imagine a scenario that a widget only has one non-empty property of these three, e.g., `text` with the content "search", while another widget only has one non-empty `resource-ID` with the content "location". If, unfortunately, they are on the same GUI page and we only record `content-description`, then they would be mistakenly classified as the same event. By simultaneously recording these properties, we ensure an accurate abstraction of the GUI event. With the size of M slightly increased, this approach guarantees a high precision for classifying the widgets into events.

3.2.2 One-step guidance for event selection

During testing, model M is used to guide event selection. However, in the early stages of the test, M only has limited information. Therefore, event selection at this stage is relatively random, aiming at rapidly initializing model M. Specifically, when there are still unexecuted events on a page, that is, events not recorded in M, CamDroid randomly selects one and updates M based on the execution result. Once the information in M is sufficient, it can be utilized to further explore unreached activities. When all events on a GUI page are included in M, CamDroid selects events based on the transition probabilities in M, aiming to cover activities that have not been covered in the current testing.

For each event e_i of all events E on the page, we calculate the probability that it can reach a new activity, and select the event according to the probability distribution. The probability (represented as p_r) can be calculated according to M,

$$p_r(e_i) = \sum_{a_j \notin A_t} p(e_i, a_j) \tag{2}$$

where A_t is the set of activities tested in the current testing run. The probability that event e_i is selected (represented as $p_s(e_i)$) is calculated through applying the softmax function to $p_r(e_i)$,

$$p_s(e_i) = \frac{\exp(\alpha_1 \times p_r(e_i))}{\sum_{e_j \in E} \exp(\alpha_1 \times p_r(e_j))} \tag{3}$$

where α_1 is a float larger than 1 and is set as 1.25 in CamDroid. Here we use the scaled probability $\alpha_1 \times p_r(e_i)$ instead of p_r to magnify the gaps among the probabilities assigned to different events. In this way, events with higher p_r values are further preferred, leading to faster exploration of uncovered activities.

Furthermore, we have detailed settings towards the selection of events and actions for improving the test efficiency in CamDroid from the following three aspects. First, if an event corresponds to multiple widgets, CamDroid will randomly select one for execution. Second, if the selected event has multiple executable actions (e.g., `click` and `scroll`), CamDroid will prioritize actions that have not been executed or have been executed only a few times. Third, if an event on the current page has been selected twice under the one-step guidance, it will not be chosen for a time. This continues until all other events on the page have been executed twice. Then their counters of execution times in this stage will be reset to 0.

3.2.3 Multi-step guidance for event selection

Some activities may require sequential events to reach, and for these activities, the aforementioned one-step guidance may not be sufficient. Reinforcement learning, on the other hand, enables multi-step decision-making. Therefore, we have incorporated a typical reinforcement learning algorithm, Q-learning, into CamDroid to provide multi-step guidance.

The core of Q-learning is to utilize a Q-table to store the Q-values of actions (events in CamDroid), where

the Q-value represents the benefits that an action can bring. In our tool, the Q-value suggests the probability of reaching new activities in the future. Then, at each time t CamDroid selects an event e_t (no matter whether e_t is selected under one-step guidance or multi-step guidance), observes a reward r_t, enters the activity a_{t+1}, and updates the $Q(e_t)$,

$$Q^{\text{new}}(e_t) = (1-\beta)\times Q(e_t) + \beta\times(Q_t^s(e_t) + Q_t^r(e_t)) \tag{4}$$

where β is the learning rate and is set as 0.8. $Q_t^s(e_t)$ is the reward earned from this event e_t, including the immediate reward r_t from e_t and the potential future rewards. $Q_t^r(e_t)$ is a bonus reward, indicating whether the selection at t is better than that at $t-1$.

$Q_t^s(e_t)$ is calculated by the N-step Sarsa method[28] for sequential decision guidance,

$$Q_t^s(e_t) = r_t + \gamma r_{t+1} + \cdots + \gamma^{m-1} r_{t+m-1} + \gamma^m Q(e_{t+m}) \tag{5}$$

where γ is the discount factor and is set as 0.5. m is the number of steps taken into account for updating the Q-values and set as 3. r_{t+i} is the immediate reward from event e_{t+i} executed at time $t+i$.

The design of the reward can be approached from two perspectives. From the aspect of events, the benefits are higher when executing events with fewer execution times but higher probabilities of reaching the unvisited activities. From the aspect of activities, the benefits are higher when accessing activities with fewer visit times and more unexecuted events on the page. Therefore, $r_t = r_t^e + r_t^a$, where r_t^e and r_t^a are calculated as follows:

$$r_t^e = \frac{p_r(e_t)}{\sqrt{n(e_t)+1}} \tag{6}$$

$$r_t^a = \frac{n_{\text{ne}} + 0.5\times n_{\text{nt}} + \sum_{e_i\in E_t} p_r(e_i)}{\sqrt{n(a_t)+1}} \tag{7}$$

where n_{ne} denotes the number of events on a_t that have never been executed. n_{nt} denotes the number of events on a_t that have not been executed in the current testing run but executed in previous ones. E_t denotes the set of all events on a_t. $n(a_t)$ denotes the number of times a_t is visited in this testing run.

$Q_t^r(e_t)$ is designed according to potential-based reward shaping[29]:

$$Q_t^r(e_t) = \gamma\times r_t - r_{t-1} \tag{8}$$

where r_t and r_{t-1} represent instant rewards after executing e_t and e_{t-1}, respectively. CamDroid gets an additional reward if it performs better at t than $t-1$.

Similar to the one-step guidance, multi-step guidance is used only after the Q-table contains enough information,i.e., when all the events on the page have been executed at least twice under the former guidance. We still use the softmax function to calculate p_s,

$$p_s(e_i) = \frac{\exp(\alpha_2\times Q(e_i))}{\sum_{e_j\in E}\exp(\alpha_2\times Q(e_j))} \tag{9}$$

where α_2 is set as 10.

3.2.4 Restructure the testing pipeline

In practice, we observe that there is a non-trivial number of operations which are irreversible. Benchmark experiments are conducted to identify such actions with the help of the previous model-based testing tool. In detail, we record the GUI page before an operation is performed. Then the backward button is clicked right after the operation, and the new GUI page is recorded. We repeat the above steps multiple times to obtain two sets of GUI pages, and calculate the difference between the two sets. Significant difference indicates that the operation is irreversible. The benchmark experiments ultimately yield 10+ irreversible operations.

To cover both GUI pages before and after these operations, CamDroid intelligently select the timing to perform them. When an irreversible action is encountered, CamDroid checks whether the remaining widgets on the activity have been selected during the current test. CamDroid selects the action only after all the other widgets have been executed at least once.

3.3 Text input generation

To better understand the scenarios of text inputs as well as their categories and correlations in real world apps, we conduct a detailed research with one of the largest Android UI datasets Rico[30]. Then we design a lightweight tool for automated text input generation based on the experience from the aforementioned study.

Especially, the research on correlations between text inputs is not straightforward, due to the lack of datasets of real users' continuous text inputs. Fortunately, we notice that there are abundant datasets containing user information from various apps, including basic user information, user preferences, and usage patterns. These pieces of information largely originate from the

text inputs during user interactions. Hence, we utilize open datasets of app user profiles for our research.

In addition, in order to address the problem of low efficiency in real-time text input generation, CamDroid performs the generation before testing. During testing, CamDroid only selects the appropriate text according to the scenario, which greatly improves efficiency. The detailed pipeline is explained in Section 3.3.3.

3.3.1 Text input categories

Rico, the dataset we used for analysis, contains UI screenshots and their view hierarchy files from over 9300 Android apps. We filter out pages that contain text input widgets by checking if the `class-name` of a widget includes the keywords "EditText" and "AutoCompleteTextView"[31]. As widgets related to login are handled specifically in Section 3.2.4, we do not consider them in the subsequent analysis.

Finally, we get 2866 pages with 6846 text inputs from a total of 2241 apps. We classify these apps into 10 categories referring to Google Play, i.e., finance, information, entertainment, video streaming, social, reading, shopping, health, map, and travel, with each category including 87 to 427 apps.

For each text input, we utilize the contexts of the GUI widget it belongs to along with the app metadata to describe the scenario. The contexts include the `hint-text`, `resource-id`, and `text` properties of the widget, while app metadata includes the app category and the activity name. We tokenize and encode these identities with Bert[32], a well-established and widely used model in the field of NLP. As a result, we obtain a vectorized representation of the current text input scenario, which we refer to as a feature vector. Next, we cluster the 6846 feature vectors to explore the distribution of input text scenarios. Specifically, considering our large data scale, we employ the DBSCAN clustering algorithm. The similarity between vectors is measured by BERTScore[21], a metric used to evaluate sentence similarity. We name each text input category by decoding the feature vector of its cluster centroid (termed input context vector). We ultimately identify 81 classes of text input scenarios and list the detailed information for the top-5 classes in Table 1.

3.3.2 Correlations between different categories

According to the text input categories identified in Section 3.3.1, we select 11 datasets[10–20] to cover them. We associate each category with attributes in public datasets based on their meanings. Some categories, such as user age and gender, may have correspondences in multiple public datasets. Attributes in public datasets that do not have a corresponding text input category are removed. Next, we calculate regression models[33] between each pair of attributes in the same dataset, and then perform F-tests[34] on the obtained models. If the p-value of F-statistic is less than 0.05, it indicates a significant correlation between them.

Because not all inputs depend on others, we further divide these text input categories into independent and non-independent ones based on the results of correlation calculations. First, we examine categories with p-values higher than 0.05 for all other columns in the same dataset. In other words, the inputs for these categories are independent of those in other scenarios. We refer to them as independent categories.

Next, we look at combinations of categories with p-values less than 0.05. For such a pair of categories, if one of them has similar text input scenarios in more than 5 app types, it indicates that it is common enough to be regarded as an independent category. The threshold for universality 5 is set empirically based on our manual inspection of representative samples. If both categories are very universal, the one with similar scenarios in more app types is an independent category, while the other one is not. If they have similar scenarios in an equal number ($\geqslant 5$) of app types, then

Table 1 Detailed information for top-5 text input categories.

Name	Brief description	Percentage (%)	Attribute	Independent
Information-search-news	Input news keywords and search	5.9	News headlines[12]	Yes
Health-host-height	Input the user's height on the profile page	4.7	Height[10]	No
Travel-map-destination	Input the destination on the navigation/ booking page	4.3	Neighbourhood_group[20]	No
Social-profile-country	Input the country the user is from	2.8	Tweet_location[15]	Yes
Shopping-product-name-search	Input a product name and search	2.4	Product_name[17]	No

Note: The columns respectively denote the classes' names (Name, i.e., input context vector), a brief description of the scenario of the text input category (Brief description), the percentage it occupies among all text inputs in Rico (Percentage), its corresponding attribute in public datasets (Attribute), and whether it is an independent category or not.

both are considered independent.

This process allows us to identify all the independent text input categories, while the remaining categories are considered non-independent. Finally, the 81 text input categories are classified into 33 independent ones and 48 non-independent ones. The pipeline for this classification is shown in Fig. 3 and the results of the top-5 categories are recorded in Table 1.

After that, we utilize the Residual Sum of Squares (RSS)[35] and Goodness Of Fit (GOF)[36] tests to study the statistical distribution of the content of each independent category. Due to the distribution of an independent category may vary across different types of apps, we perform fitting for each app class. Furthermore, we employ a GAN[37] to learn the relationship between the content of non-independent and independent categories. The GAN takes the content of all independent ones as input and outputs the corresponding content for all non-independent ones. In this method, we obtain the distributions and dependency relationships among various text input categories, which prepares us for the automatic generation.

3.3.3 Text input generation pipeline

The pipeline is divided into two stages: pre-test and in-test. Before the test, CamDroid utilizes the knowledge from Section 3.3.2 to generate all the potential inputs, i.e., the content of the 81 text input categories.

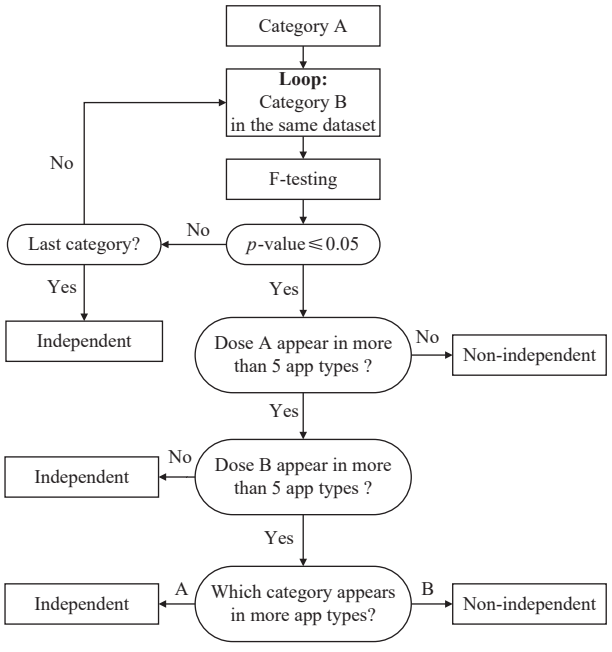


Fig. 3 Pipeline of classifying text input categories into independent and non-independent ones.

CamDroid determines the distribution functions for each independent category based on the app type. It then generates their text inputs using these distribution functions. Then, these text inputs are input into the trained GAN model to produce content of all non-independent categories. During the test, if an event with text input is selected, CamDroid encodes its context into a feature vector with the method described in Section 3.3.1. It calculates the BERTScore between this vector and each of the 81 input context vectors. CamDroid then selects the pre-generated content corresponding to the input context vector with the highest score to fill in.

3.4 Implementation

The prototype of CamDroid is implemented based on the source of Android Monkey and includes server and client components. The client part can be installed and used directly through Android Debug Bridge (ADB) without any additional modifications to the device or the app under test. It is compatible with both physical and virtual devices and supports Android 10–13. The selection of events and the generation of text inputs are carried out on the server side, and the historical context information is stored in an online database. Therefore, CamDroid only occupies a minimal amount of hardware resources on the device.

4 Evaluation

We evaluate CamDroid on 20 widely-used apps from Google Play and compare its performance with 3 state-of-the-art automated testing tools i.e., Monkey[22], APE[6], and Fastbot2[7]. The highlights are as follows:

- Compared with the field-renowned baselines, CamDroid improves the average (median) activity coverage by 1.25× to 3.69× (1.27× to 5.45×).
- Within the time limitation of industrial app testing, CamDroid detects 1.89× to 8.50× more crashes than the baselines.
- The learning-based text input generation approach alone improves the activity coverages of CamDroid and all the baselines by 1.13× to 1.19×.

4.1 Experimental setup

Setup. All tools can run without modifications to apps and devices. We conduct parallel tests on four mobile phones with the same configuration (i.e., Snapdragon 855 2.84 GHz CPU, 6 GB RAM, and Android 11 operating system) to mitigate potential bias. We

choose Android 11 because it had the highest market share when we started our development (Jan. 2023), and it can cover the new functions and performance improvements introduced in Android 10 compared to Android 9[38]. Following previous work[6, 24, 39], the four tools are set to perform a 1-hour test on each app. Each test is repeated 5 times. The average activity coverage and the total number of detected crashes are used to evaluate their performance.

Benchmark collection. We select 20 widely-used apps as our test subjects. Specifically, for each of the 10 app types mentioned in Section 3.3.1, we randomly choose 2 apps with high downloads on Google Play. We filter out apps if (1) they contain too few activities ($\leqslant 30$); (2) they have no text input widgets; (3) their view hierarchy files are inaccessible by UIAutomator[40]; and (4) one or more baselines crash on them. We present the basic information of the 20 selected apps in Table 2.

4.2 Performance on benchmark apps

Figure 4 illustrates the activity coverage of four tools across the 20 apps. It is evident that CamDroid achieves the highest coverage on each app. Its average (median) activity coverage on the 20 apps is 3.69× (5.45×), 1.94× (2.11×), 1.25× (1.27×) higher than Monkey, APE, and Fastbot2, respectively. Furthermore, CamDroid performs well regardless of app's type and number of activities, which indicates its universality.

Table 2 Mobile apps used for testing.

ID	App	Type	Number of downloads on Google Play	Total activities
1	Dianping	Travel	$>1\times10^6$	535
2	Weibo	Social	$>1\times10^7$	778
3	Facebook	Social	$>5\times10^9$	862
4	Netease News	Information	$>1\times10^4$	249
5	Youtube	Video streaming	$>1\times10^{10}$	55
6	Kugou Music	Entertainment	$>5\times10^4$	258
7	iQiyi	Video streaming	$>5\times10^7$	279
8	Resso Music	Entertainment	$>5\times10^7$	53
9	Booking	Travel	$>5\times10^8$	295
10	YahooFinance	Finance	$>1\times10^7$	92
11	Tencent News	Information	$>1\times10^5$	149
12	Vested	Finance	$>5\times10^5$	31
13	Fitbit	Health	$>5\times10^7$	456
14	Kindle	Reading	$>1\times10^8$	142
15	Fizzo Novel	Reading	$>5\times10^4$	118
16	Amazon	Shopping	$>5\times10^8$	85
17	AliExpress	Shopping	$>5\times10^8$	379
18	Google Map	Maps	$>1\times10^{10}$	36
19	Citymapper	Maps	$>1\times10^7$	100
20	Fasting	Health	$>1\times10^7$	94

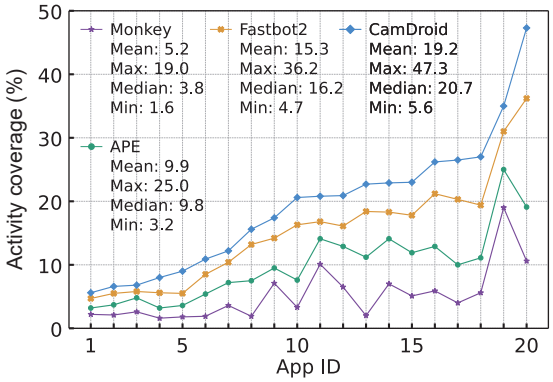


Fig. 4 Results of activity coverage by CamDroid and other tools.

Overall, CamDroid shows a more pronounced improvement over the baselines in apps with more activities, such as Booking and AliExpress. This is because the baselines face more difficulties in comprehensively exploring these complex apps. In contrast, apps with fewer activities are more prone to reaching saturation, limiting the potential for enhancement. Relatively, all four tools exhibit poorer performance on Dianping, Weibo, and Facebook. This is due to the strong interactivity of these three apps, where automated testing tools still face certain gaps compared to human users.

In terms of crash detection, as shown in Fig. 5, CamDroid also performs the best. Since these apps undergo thorough test before being released, the number of crashes uncovered by these tools is relatively low. Nevertheless, CamDroid, with its high testing efficiency, manages to discover 34 crashes across the 20 apps, whereas Monkey, APE, and Fastbot2 only find 4, 8, and 18 crashes, respectively. All the crashes detected by CamDroid can be reproduced reliably.

4.3 Ablation study

We conduct ablation study to explore the effectiveness of our model-based testing tool and learning-based text input generation method. We integrate the text generation method into Monkey, APE, and Fastbot2, and test them following the approach described in

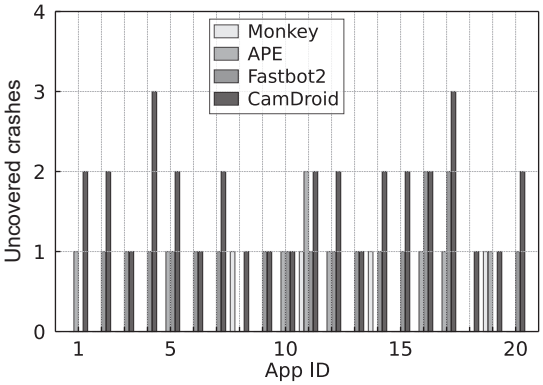


Fig. 5 Results of crash detection by CamDroid and other tools.

Section 4.1. We also evaluate the performance of CamDroid without the text input generation. The results in Table 3 illustrate that both aspects of our efforts contribute to CamDroid's overall performance.

It can be observed that our text input generation method improves the activity coverage of Monkey, APE, and Fastbot2 by 1.13×, 1.19×, and 1.15×, respectively. CamDroid, with text input generation, achieves a coverage increase of 1.15× compared to the one without it. Relatively speaking, Monkey shows the lowest improvement, which can be attributed to its lower activity coverage, thus triggering fewer pages containing text input widgets. On the other hand, APE demonstrates the highest improvement. This is because both Fastbot2 and CamDroid already achieve high activity coverages, and their effectiveness may easily reach saturation. The numbers of crashes discovered by Monkey, APE, and Fastbot2 increase by 1.75× (7 vs. 4), 1.89× (15 vs. 8), and 1.78× (32 vs. 18), respectively.

Additionally, as shown in Table 3, CamDroid without text input generation (referred to as C-T) also performs better than the baselines. Its average activity coverage is 16.7%, which is 3.21× higher than Monkey (5.2%), 1.69× higher than APE (9.9%), and 1.09× higher than Fastbot2 (15.3%). It discovers a total of 21 unique crashes, which are 5.25×, 2.62×, and 1.17× as many as Monkey (4), APE (8), and Fastbot2 (18).

5 Related Work

Our work integrates learning-based text input generation with model-based automated GUI testing

Table 3 Activity coverage and crash detection with automated GUI testing tool with and without text input generation.

ID	Activity coverage (%)								# Uncovered crashes							
	M	M+T	A	A+T	F	F+T	C-T	C	M	M+T	A	A+T	F	F+T	C-T	C
1	2.2	2.6 ↑	3.2	3.6 ↑	4.7	5.2 ↑	4.9	5.6 ↑	0	0 –	1	1 –	0	2 ↑	1	2 ↑
2	2.1	2.3 ↑	3.7	4.5 ↑	5.5	6.4 ↑	5.8	6.6 ↑	0	1 ↑	0	1 ↑	1	1 –	2	2 –
3	2.6	3.0 ↑	4.8	5.0 ↑	5.8	6.7 ↑	6.4	8.0 ↑	0	0 –	0	0 –	1	1 –	1	1 –
4	1.6	1.6 –	3.2	3.6 ↑	5.6	7.6 ↑	6.4	8.0 ↑	0	0 –	0	0 –	1	2 ↑	1	3 ↑
5	1.8	1.8 –	3.6	3.6 –	5.5	7.3 ↑	7.3	9.0 ↑	0	0 –	1	1 –	1	2 ↑	1	2 ↑
6	1.9	2.3 ↑	5.4	5.8 ↑	8.5	9.3 ↑	9.7	10.9 ↑	0	0 –	0	0 –	1	1 –	1	1 –
7	3.6	3.9 ↑	7.2	8.2 ↑	10.4	11.1 ↑	10.8	12.2 ↑	0	0 –	0	0 –	1	1 –	1	2 ↑
8	1.9	3.8 ↑	7.5	7.5 –	13.2	13.2 –	13.2	15.6 ↑	1	1 –	0	1 ↑	0	2 ↑	1	1 –
9	7.1	7.1 –	9.5	11.9 ↑	14.2	17.3 ↑	15.9	17.4 ↑	0	0 –	0	0 –	1	1 –	0	1 ↑
10	3.3	5.4 ↑	7.6	9.8 ↑	16.3	18.5 ↑	18.5	20.6 ↑	0	1 ↑	1	1 –	1	1 –	1	1 –
11	10.1	10.1 –	14.1	16.8 ↑	16.8	19.5 ↑	19.5	20.8 ↑	1	1 –	1	2 ↑	1	2 ↑	2	2 –
12	6.5	6.5 –	12.9	12.9 –	16.1	19.4 ↑	16.1	20.9 ↑	0	0 –	1	2 ↑	1	2 ↑	1	2 ↑
13	2.0	3.1 ↑	11.2	15.4 ↑	18.4	23.5 ↑	19.5	22.7 ↑	0	1 ↑	0	1 ↑	1	2 ↑	0	1 ↑
14	7.0	9.2 ↑	14.1	16.9 ↑	18.3	21.8 ↑	18.3	22.9 ↑	1	1 –	0	0 –	1	1 –	2	2 –
15	5.1	5.1 –	11.9	15.3 ↑	17.8	19.5 ↑	20.3	23.0 ↑	0	0 –	0	0 –	1	1 –	1	2 ↑
16	5.9	5.9 –	12.9	15.3 ↑	21.2	22.4 ↑	22.4	26.2 ↑	0	0 –	1	1 –	2	2 ↑	1	2 ↑
17	4.0	5.0 ↑	10.0	14.2 ↑	20.3	21.9 ↑	23.5	26.5 ↑	0	0 –	1	1 –	2	2 –	2	3 ↑
18	5.6	8.3 ↑	11.1	13.9 ↑	19.4	22.2 ↑	22.2	27.0 ↑	0	0 –	0	1 ↑	0	2 ↑	0	1 ↑
19	19.0	21.0 ↑	25.0	26.0 ↑	31.0	35.0 ↑	31.0	35.0 ↑	1	1 –	1	1 –	0	2 ↑	0	1 ↑
20	10.6	10.6 –	19.1	26.6 ↑	36.2	44.7 ↑	42.6	47.3 ↑	0	0 –	0	1 ↑	1	2 ↑	2	1 –

Note: “M” denotes Monkey, “A” denotes APE, and “F” denotes Fastbot2. “M+T”, “A+T”, and “F+T” denote Monkey, APE, and Fastbot2 with our text input generation method. “C-T” denotes CamDroid without text input generation. “C” denotes CamDroid. “↑” means performance increase of automated testing tools after integrating text input generation and “–” means no growth.

for Android apps. CamDroid automates the GUI testing of Android apps using a combinatorial model with the one-step guidance of event-activity transitions and the multi-step guidance of reinforcement learning, and generates text inputs like real users with a GAN based on public datasets. We review related literature in this section.

Monkey[22] is the most classical and lightweight tool to perform black box testing. Given that the exploration strategy of Monkey is completely random, Monkey lacks extensibility and is easy to bypass intentionally. To strategically automate the GUI testing for Android apps, learning-based approaches have been widely studied[8, 41, 42]. Humanoid[8] is a representative one that uses a deep neural network model to learn how users choose UI actions from human interaction traces. Such approaches require specific sequential trace data and in the early stage of the test, they have similar performance with the random strategy.

Since the GUI switches of apps can be modeled as state transitions via UI actions, a plethora of model-based testing approaches[6, 7, 25, 43] emerge. Recently, APE, a practical model-based approach via dynamic model abstraction, has significantly advanced the state-of-the-art model-based techniques[6]. Furthermore, based on the implementation of APE, Fastbot2[7] is proposed by ByteDance and achieves outstanding industrial success. However, when encountering scenarios with text input requirements, these methods just randomly input and thus possibly miss the hidden activities with specific inputs to reach.

To fill the above gap, many efforts have been devoted to generating text inputs like real users[27, 44, 45]. Existing text input generation methods either rely on large amounts of manual text input samples and thus lack generalizability[44], or adopt heavy-weight language models requiring undesirable prompt processing time[27]. Different from them, CamDroid achieves high input efficiency, as well as good generation effects with the help of the generation-and-selection pipeline. In addition, CamDroid considers the correlations of multiple text inputs during a single test, resulting in more realistic behaviors.

6 Conclusion

This paper presents a context-aware model-based GUI testing approach for Android apps, named CamDroid. Through combining the one-step guidance of event-activity transitions and the multi-step guidance of reinforcement learning, CamDroid leverages the context knowledge of previous tests to efficiently and effectively explore app activities. Moreover, CamDroid efficiently generates text inputs regarding the GUI context like real users through pre-training a GAN based on public datasets when encountering GUI widgets requiring text inputs. Experiments with 20 widely-used apps from Google Play show that CamDroid outperforms 3 state-of-the-art testing tools in terms of both activity coverage and number of detected crashes within industrial testing time limitation.

Acknowledgment

This work was supported by the National Key R&D Program of China (No. 2022YFB4500703), the National Natural Science Foundation of China (Nos. 61902211 and 62202266), the China Postdoctoral Science Foundation (No. 2022M721831), and Microsoft Research Asia (No. 100336949).

References

[1] Number of android apps, https://www.appbrain.com/stats/number-of-android-apps, 2022.

[2] L.i Gong, Z. Li, H. Wang, H. Lin, X. Ma, and Y. Liu, Overlay-based android malware detection at market scales: Systematically adapting to the new technological landscape, *IEEE Transactions on Mobile Computing*, vol. 21, no.12, pp. 4488–4501, 2021.

[3] L. Gong, H. Lin, Z. Li, F. Qian, Y. Li, X. Ma, and Y. Liu, Systematically landing machine learning onto market-scale mobile malware detection, *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 7, pp. 1615–1628, 2020.

[4] L. Gong, Z. Li, F. Qian, Z. Zhang, Q. Chen, Z. Qian, H. Lin, and Y. Liu, Experiences of landing machine learning onto market-scale mobile malware detection, in *Proc. 15th European Conference on Computer Systems*, Bordeaux, France, 2020, pp. 1–14.

[5] Y. Yan, Z. Li, Q. Chen, C. Wilson, T. Xu, E. Zhai, Y. Li, and Y. Liu, Understanding and detecting overlay-based android malware at market scales, in *Proc. 17th Annual International Conference on Mobile Systems, Applications, and Services*, Seoul, Republic of Korea, 2019, pp. 168–179.

[6] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su, Practical GUI testing of android applications via model abstraction and refinement, in *Proc. 41st International Conference on Software Engineering*, Montreal, Canada, 2019, pp. 269–280.

[7] Z. Lv, C. Peng, Z. Zhang, T. Su, K. Liu, and P. Yang, Fastbot2: Reusable automated model-based GUI testing for android enhanced by reinforcement learning, in *Proc.*

37th International Conference on Automated Software Engineering, Rochester, MI, USA, 2022, pp. 1–5.

[8] Y. Li, Z. Yang, Y. Guo, and X. Chen, Humanoid: A deep learning-based approach to automated black-box android app testing, in *Proc. 34th International Conference on Automated Software Engineering*, San Diego, CA, USA, 2019, pp. 1070–1073.

[9] H. Lin, J. Qiu, H. Wang, Z. Li, L.i Gong, D. Gao, Y. Liu, F. Qian, Z. Zhang, P. Yang, et al., Virtual device farms for mobile app testing at scale, in *Proc. 29th ACM International Conference on Mobile Computing and Networking*, Madrid, Spain, 2023, pp. 1–17.

[10] Body fat prediction dataset, https://www.kaggle.com/datasets/fedesoriano/body-fat-prediction-dataset, 2021.

[11] Bank customers churn, https://www.kaggle.com/datasets/santoshd3/bank-customers, 2018.

[12] A million news headlines, https://www.kaggle.com/datasets/therohk/million-headlines, 2022.

[13] Movielens 20m dataset, https://www.kaggle.com/datasets/grouplens/movielens-20m-dataset, 2018.

[14] Dataset for chatbot, https://www.kaggle.com/datasets/grafstor/simple-dialogs-for-chatbot, 2020.

[15] Twitter friends, https://www.kaggle.com/datasets/hwassner/TwitterFriends, 2016.

[16] Goodreads books, https://www.kaggle.com/datasets/jealousleopard/goodreadsbooks, 2019.

[17] Amazon sales dataset, https://www.kaggle.com/datasets/karkavelrajaj/amazon-sales-dataset, 2019.

[18] Medical transcriptions, https://www.kaggle.com/datasets/tboyle10/medicaltranscriptions, 2018.

[19] China city dataset, https://github.com/brightgems/china_city_dataset, 2017.

[20] New york city airbnb 2023, public data, https://www.kaggle.com/datasets/godofoutcasts/new-york-city-airbnb-2023-public-data, 2023.

[21] T. Zhang, V. Kishore, F. Wu, K. Weinberger, and Y. Artzi, BERTScore: Evaluating text generation with bert, arXiv preprint arXiv: 1904.09675, 2019.

[22] Google, Ui/application exerciser monkey, https://developer.android.com/studio/test/monkey.html, 2018.

[23] R. Mahmood, N. Mirzaei, and S. Malek, Evodroid: Segmented evolutionary testing of android apps, in *Proc. 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Hong Kong, China, 2014, pp. 599–609.

[24] K. Mao, M. Harman, and Y. Jia, Sapienz: Multi-objective automated testing for android applications, in *Proc. 25th International Symposium on Software Testing and Analysis*, Saarbrucken, Germany, 2016, pp. 94–105.

[25] 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 *Proc. 11th Joint Meeting on Foundations of Software Engineering*, Paderborn, Germany, 2017, pp. 245–256.

[26] Y. Li, Z. Yang, Y. Guo, and X. Chen, Droidbot: A lightweight UI-guided test input generator for android, in *Proc. 39th International Conference on Software Engineering Companion*, Buenos, Aires, 2017, pp. 23–26.

[27] Z. Liu, C. Chen, J. Wang, X. Che, Y. Huang, J. Hu, and Q. Wang, Fill in the blank: Context-aware automated text input generation for mobile GUI testing, in *Proc. 45th International Conference on Software Engineering*, Melbourne, Australia, 2023, pp. 1355–1367.

[28] K. De Asis, J Hernandez-Garcia, G Holland, and R. Sutton, Multi-step reinforcement learning: A unifying algorithm, in *Proc. 32nd AAAI Conference on Artificial Intelligence*, New Orleans, LA, USA, 2018, pp. 2902–2909.

[29] Y. Gao and F. Toni, Potential based reward shaping for hierarchical reinforcement learning, in *Proc. 24th International Joint Conference on Artificial Intelligence*, Buenos Aires, Argentina, 2015, pp 3504–3510.

[30] 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 *Proc. 30th Annual ACM Symposium on User Interface Software and Technology*, Quebec City, Canada, 2017, pp. 845–854.

[31] Google, Introduction of text input, https://developer.android.com/reference/android/widget/EditText?hl=en, 2023.

[32] J. Devlin, M. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, arXiv preprint arXiv: 1810.04805, 2018.

[33] T. Amemiya, Non-linear regression models, *Handbook of Econometrics*, vol. 1, pp. 333–389, 1983.

[34] L. M. Lix, J. C. Keselman, and H. J. Keselman, Consequences of assumption violations revisited: A quantitative review of alternatives to the one-way analysis of variance f test, *Review of Educational Research*, vol. 66, no. 4, pp. 579–619, 1996.

[35] J. A. Morgan and J. F. Tatar, Calculation of the residual sum of squares for all possible regressions, *Technometrics*, vol. 14, no. 2, pp. 317–325, 1972.

[36] E. B. Andersen, A goodness of fit test for the rasch model. *Psychometrika*, vol. 38, pp. 123–140, 1973.

[37] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, Generative adversarial nets, in *Proc. 28th Conference on Neural Information Processing Systems*, Montreal, Canada, 2014, pp. 2672–2680.

[38] Y. Li, H. Lin, Z. Li, Y. Liu, F. Qian, L. Gong, X. Xin, and T. Xu, A nationwide study on cellular reliability: Measurement, analysis, and enhancements, in *Proc. of 2021 ACM International Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, Virtual Event, 2021, pp. 597–609.

[39] S. Choudhary, A. Gorla, and A. Orso, Automated test input generation for android: Are we there yet? in *Proc. 30th International Conference on Automated Software Engineering*, Lincoln, NE, USA, 2015, pp. 429–440.

[40] Google, Ui automator, https://developer.android.com/training/testing/other-components/ui-automator, 2021.

[41] W. Choi, G. Necula, and K. Sen, Guided GUI testing of android apps with minimal restart and approximate learning, *ACM Sigplan Notices*, vol. 48, no. 10, pp. 623–640, 2013.

[42] C. Degott, N. B. Jr, and A. Zeller, Learning user interface element interactions, in *Proc. 28th ACM SIGSOFT*

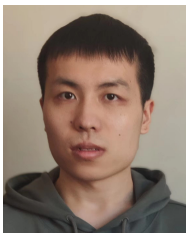
International Symposium on Software Testing and Analysis, Beijing, China, 2019, pp. 296–306.

[43] S. Hao, B. Liu, S. Nath, W. Halfond, and R. Govindan, Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps, in *Proc. 12th Annual International Conference on Mobile Aystems, Applications, and Services*, Bretton Woods, NH, USA, 2014, pp. 204–217.

[44] P. Liu, X. Zhang, M. Pistoia, Y. Zheng, M. Marques, and L. Zeng, Automatic text input generation for mobile testing, in *Proc. 39th International Conference on Software Engineering*, Buenos, Aires, 2017, pp. 643–653.

[45] Y. He, L. Zhang, Z. Yang, Y. Cao, K. Lian, S. Li, W. Yang, Z. Zhang, M. Yang, Y. Zhang, et al., Textexerciser: Feedback-driven text input exercising for android applications, in *Proc. 41st IEEE Symposium on Security and Privacy*, San Francisco, CA, USA, 2020, pp. 1071–1087.

Hongyi Wang received the BEng degree from Tsinghua University, China in 2021. She is working towards the PhD degree at School of Software, Tsinghua University, Beijing, China. Her research areas mainly include network measurement and machine learning.



Yang Li received the BEng and MEng degrees from Tsinghua University, China in 2018 and 2021, respectively. He is working towards the PhD degree at School of Software, Tsinghua University, Beijing, China. His research areas mainly include big data analysis, network measurement, and machine learning.

Jing Yang received the BEng degree from Tongji University, China in 2023. She is currently a master student at School of Software, Tsinghua University, Beijing, China. Her research areas mainly include network measurement and machine learning.

Daqiang Hu received the BEng degree from University of Electronic Science and Technology of China in 1993, and the MEng degree from Chongqing University of Posts and Telecommunications, China in 1996. He is the CEO of Hangzhou Uusense Technology Inc., China. His research areas mainly include automated software testing and network measurement.

Zhi Liao received the BEng degree from Central China Normal University in 2001. He is the CTO of Hangzhou Uusense Technology Inc., China. His research areas mainly include automated software testing and network measurement.