2017 IEEE/ACM 39th International Conference on Software Engineering

Automatic Text Input Generation for Mobile Testing

Peng Liu*, Xiangyu Zhang†, Marco Pistoia*, Yunhui Zheng*, Manoel Marques* and Lingfei Zeng†
*IBM T. J. Watson Research Center, Yorktown Heights, New York, USA
Email: {liup, pistoia, zhengyu, manoel }@us.ibm.com
† Purdue University, West Lafayette, Indiana, USA
Email: xyzhang@cs.purdue.edu, zengl@purdue.edu

***Abstract*—Many designs have been proposed to improve the automated mobile testing. Despite these improvements, providing appropriate *text inputs* remains a prominent obstacle, which hinders the large-scale adoption of automated testing approaches. The key challenge is how to automatically produce the most *relevant* text in a use case context. For example, a valid website address should be entered in the address bar of a mobile browser app to continue the testing of the app; a singer's name should be entered in the search bar of a music recommendation app. Without the proper text inputs, the testing would get stuck. We propose a novel deep learning based approach to address the challenge, which reduces the problem to a minimization problem. Another challenge is how to make the approach generally applicable to both the trained apps and the untrained apps. We leverage the Word2Vec model to address the challenge. We have built our approaches as a tool and evaluated it with 50 iOS mobile apps including Firefox and Wikipedia. The results show that our approach significantly outperforms existing automatic text input generation methods.**

I. Introduction

Mobile devices have become an integral part of our life. Mobile apps are the vehicle to deliver a wide variety of convenient and high-quality services, such as web browsing, entertainment, transportation information assistance, banking, and social networking. In response to the increasing need, the mobile market is growing rapidly in a speed of 1000+ new apps per day [1] [6]. Accordingly, the development teams are constantly in fierce competition, with great pressure of meeting release deadlines. This unfortunately often leads to bugs in mobile apps, such as run-time app crashes, flaws in UI designs and functions that are not fully implemented.

The goal of mobile testing is to find bugs in an app before it is released. There are two mainstream mobile testing methods: *manual testing* and *automated monkey testing*. In manual testing, the testers manually perform actions to exercise as many use cases as possible. The disadvantage of this approach is that it requires substantial human efforts as testers need to closely interact with the app throughout the entire testing period. Moreover, the human testers, who usually focus on demonstrating the functionality in common use cases, may often miss the corner cases which could trigger exceptions.

Automated monkey testing [8], [28], [18], [19], [37], [21], [20], [24], [27] was proposed to reduce human efforts and maximize use case coverage. "Monkey" is a metaphor to describe how this type of testing works; like a monkey, the tool performs random sequences of actions including clicking the buttons on a UI view (or UI screen) and performing random keystrokes. To cover as many action sequences as possible, researchers have proposed various novel search algorithms in contrast to the monkey random search strategy. Despite these improvements, providing appropriate *text inputs* during monkey testing remains a prominent obstacle, which hinders the large-scale adoption of monkey testing approaches. Most existing techniques can hardly provide *meaningful* text inputs in many use cases. As an example, for a movie app, monkey testing can hardly provide meaningful inputs such as `Star Trek`. Instead, it would produce irrelevant inputs such as `4t6`. As a consequence, no results would be found, thereby making Monkey unable to proceed to the screen that displays appropriate results. Manual specification may mitigate this problem, but incur substantial human effort.

In this paper, we present a solution that lets Monkey automatically produce relevant text inputs. With such inputs, Monkey can proceed to UI screens deep in the workflow (for short, *deep UI screens*) through long action sequences, instead of getting stuck at the very beginning. Once it reaches a deep UI screen, Monkey can apply other bug disclosure-oriented testing techniques, such as search-based testing [26], symbolic test generation [22], or even random testing [34] to identify bugs in the complex workflow.

A. Generating Relevant Inputs

A key requirement of input generation is to produce inputs that are *relevant* to the context.

$$Movie \rightarrow Search \rightarrow \underline{Star\ Trek} \quad (1)$$

$$Weather \rightarrow Search \rightarrow \underline{New\ York} \quad (2)$$

In Use Case 1, after the menu item labeled Movie is clicked, and a search bar labeled Search is triggered, the app expects the title of a movie from the user. Hence, the input `Star Trek` is relevant; In Use Case 2, after the menu item labeled Weather is clicked and the search bar labeled Search is triggered, the app expects the name of a city from the user. Therefore the input `New York` is relevant. In contrast, an input that is relevant in one context may not be relevant in a different context. For example, the input `Star Trek` would be inappropriate when menu item Weather is clicked. We also refer the readers to more real world examples in Section VII.

Challenges. The above requirement imposes great challenges to automatic text input generation. First, the relevance is specific to the natural-language semantics which only human

1558-1225/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSE.2017.65

testers can understand. Therefore, traditional automated input generation approaches, such as symbolic execution based testing [22], are not applicable. Second, the action (e.g., clicking menu item Movie or Weather) that contains the information that determines the relevant text inputs may not immediately precede the input action in the action sequence. Therefore, maintaining and looking up the mapping between a text input and the information of immediately precedent action is an approach that will hardly work.

Our Deep Learning Based Approach. We propose a novel deep learning based approach to solve the challenges described above. At the high level, our solution consists of two phases: In the training phase, the monkey learns the testers' manual inputs and statistically associates them with the contexts, such as the action history and the textbox label; In the prediction phase, the monkey automatically predicts text inputs based on the observed contexts. The core of our approach is a recurrent neural network (RNN) model. This type of model has achieved great success in many natural-language processing (NLP) applications, such as machine translation and input method auto-completion. Take the input method auto-completion for example, the RNN model quantifies semantic connections among words as a non-linear function, of which the parameters are trained with a large corpus of texts. Given a word, the non-linear function calculates the probability distribution of the word next to it, then the next word is sampled from the distribution. In addition, the RNN model maintains a memory state to summarize important information from previous words, and uses it as another source of input while recommending the next word. To the best of our knowledge, we are the first ones to apply deep learning to the problem of text input generation for automated mobile testing.

According to the empirical study of Mikolov, et al. [32], the RNN model is significantly more accurate than traditional statistical models such as the n-gram model and the hidden Markov model. Specifically, RNN leads to 18% error reduction in comparison with the traditional models, assuming they are trained on the same data, and leads to 12% error reduction even when the traditional model is trained on 5 times more data than the RNN model.

B. App-independent Input Generation

Another important requirement of input generation is *app independence*. This means that the RNN model trained atop a set of apps should also apply to other apps.

Challenges. Using different words in different apps to represent the same concept imposes a significant challenge to app-independence.

$$Film \rightarrow Search \rightarrow \underline{?} \tag{3}$$

Suppose that label Film has not been seen in the training phase. A human tester can easily figure out the semantic similarity between labels Film and Movie in Use Case 1. Accordingly, we can predict the relevant text input—for example, `Star Trek`. However, the RNN model is completely oblivious of label Film since it did not appear in the training phase. As a consequence, the RNN model cannot predict the relevant text input from the label Film.

Our Word2Vec based Solution. We address the challenge by statistically learning the human knowledge of synonyms from the Google News corpus. Synonyms are stored in equivalence classes. The representative of each equivalence class is used to replace the words that fall into the class during training and prediction.

Our approach is built upon the Word2Vec model, a statistical model recently proposed by the NLP researchers. This model maps a word to a vector, and the distance between vectors measures the similarity of words. The Word2Vec model learns the vector encoding by solving an optimization problem. Specifically, it minimizes the distance between similar words and maximizes the distance between irrelevant words. Word2Vec is an unsupervised learning algorithm, meaning that it does not require manual labeling of training data.

Word2Vec outperforms other existing models in the quality of results [33]. This is mainly because Word2Vec can train on 2 to 3 orders of magnitude more data than prior work, within just a fraction of the time required by prior work.

We have built a tool and evaluated it with 50 iOS apps, including popular apps such as Firefox and Wikipedia. The evaluation results confirmed the effectiveness of our approach. The RNN prediction improves the coverage of Monkey testing by 21% on average, while the combination of the RNN model and the Word2Vec model improves the coverage by 28%. Besides, if we exclude the simple apps with very few UI screens available from the benchmark suite, we find the RNN prediction improves the coverage by 46% and the combination of both models leads to 60% improvement. Equally important, our prediction is very efficient, which typically completes within 1 ms.

In summary, we make the following contributions in this paper:

- We propose a novel approach based on deep learning to automatically produce the *relevant* text inputs for the mobile UI testing.
- We propose a novel combination of the Word2Vec NLP model with the learning to make the input generation *app-independent*.
- We have implemented our approach as a system and conducted experiments over 50 iOS apps including Firefox and Wikipedia. The results show that our approach significantly increases the coverage of Monkey testing.

9

II. Overview of System Design

The system design is shown in Figure 1. The monkey testing engine explores the mobile app by clicking the buttons in the screen. When it encounters a textbox, it requests the relevant input from the text input server, as denoted by ①②. The server resolves the request by further dispatching it to either a human tester or the RNN running instance, which

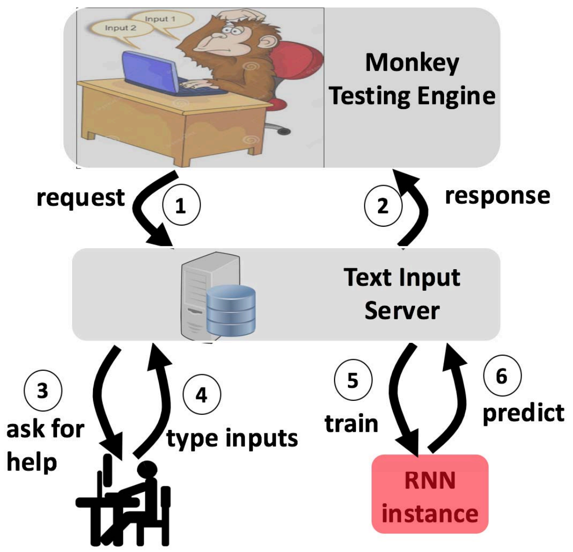


Fig. 1. Workflow of the System

corresponds to two modes: the manual mode and the AI mode. In the manual model, upon the receipt of the request for help ③, the human tester can either enter a text input relevant in the context ④ or ignore it if she does not think the context corresponds to any action sequence in practice. The inputs entered by the tester and the corresponding contexts are then recorded in the database associated with the server. In the AI mode, we first train the RNN model with the training dataset recorded in the database ⑤, after which the RNN instance can predict the text input automatically ⑥.

In the prediction phase ⑥, Monkey leverages the RNN model to predict the text input value in a given context. Basically, given the context maintained by the Monkey testing engine, the model predicts the text input value. However, since the apps that we predict inputs for are different from the apps that we train the model upon, the RNN model may not recognize some context information encountered during the Monkey testing. To address this challenge, we improve the RNN model with a Word2Vec model, which helps recognize the semantically similar contexts despite their different syntactical forms (e.g., "Movie" and "Film"). Through the novel combination of the RNN model and the Word2Vec model, we effectively address the problem.

In the following, we first introduce the background in Section III, and then explain how we apply the deep learning techniques to automatically predict the text input in Section IV. In Section V, we explain the implementation details. In Section VI, we discuss the assumptions of this work. Section VII presents the evaluation results.

III. Background On Deep Learning

This section introduces the background of recurrent neural network (RNN) and Word2Vec. Both RNN and Word2Vec are special forms of neural network. In order to discuss these two, we first have to briefly introduce neural network.

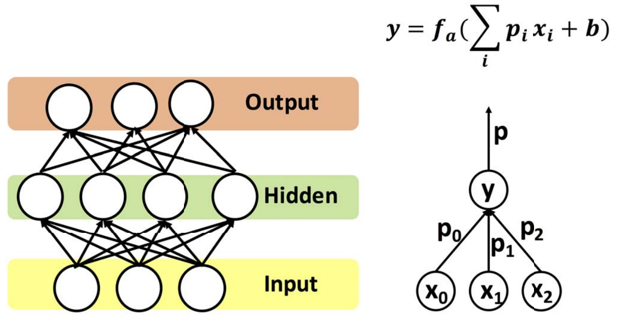


Fig. 2. (a) Neural Network. (b) Neuron

Neural network (NN) has two forms: *graphical form* and *mathematic form*. In the graphical form (Figure 2a), NN has an input layer, an output layer, one or more hidden layers. Each layer (except the input layer) comprises multiple neurons that connect with the neurons in the previous and next layers.

Each neuron, as shown in Figure 2b, is a basic computation unit. It first linearly combines the values passed along the incoming edges as $\sum_i p_i x_i + b$, where p_i and b are parameters to be trained. Then it applies an activation function f_a, e.g., the tanh function [12] or the sigmoid function [11]. The activation function is important in making the neuron non-linear. Without the activation function, the neuron is merely a linear function and the neural network is also a linear function, which cannot characterize many complex models.

In the mathematic form, NN is a composition of the functions represented by the neurons. Let $f : R^n \mapsto R^m$ denote the composite function represented by a neural network. Let the input vector $\vec{I} \in R^n$ denote the inputs to the n neurons in the input layer. Let the output vector $\vec{O} \in R^m$ denote the output by the m neurons in the output layer. The training of a neural network can be viewed as an optimization problem:

$$\underset{p_i, b, \ldots}{\text{minimize}} \sum_{\vec{I}, \vec{O} \in train} loss(f(\vec{I}), \vec{O})$$

The parameters are updated to minimize the total loss between the predicted output and the real output over the training dataset, where the loss function can be defined as distance or other forms. Regardless of the complexity of the loss function, the Gradient Descent algorithm [16] is generally applied to minimize its value.

A. Recurrent Neural Network

RNN is a special form of neural network that has a feedback loop as shown in Figure 3. For clarity, we use arrows to denote the connections between neurons.

The loop allows the information derived in a step of the network execution to be passed to the next, analogous to human's long-term memory. Figure 4 shows a conceptually unfolded version, where the neural network is conceptually copied to serve a sequence of inputs through multiple steps. Note the connection between the neural network copies.

Consider the application of the input method auto-completion, we apply RNN to generate the sentence "*how are you doing*". The words are fed to the input layer one by one. Each word is encoded in the one-hot vector representation to enable learning, e.g., the word "how" is encoded as [1,0,0,0]. Simply put, each entry of the vector corresponds to a word in the vocabulary (["*how*", "*are*", "*you*", "*doing*"]) and the location of value 1 in the vector indicates the word encoded. From the perspective of probability, the vector represents the word with 100% probability and others with 0% probability.

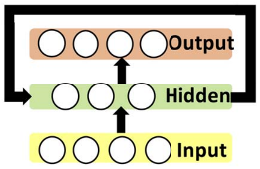


Fig. 3. RNN

With the initial parameter settings, given an input word combined with the information passed from the last step, the neural network outputs a probability vector, where the ith entry estimates the probability that the ith word in the vocabulary will appear next. For instance, based on the input "*how*", assume that it predicts the next word as [0.4, 0.2, 0.1, 0.3], i.e., the most likely next word is "*how*". However, we observe from the training dataset that the next word should be "*are*". Therefore, the training algorithm needs to update the parameters so that the predicted probability of the word "*are*" becomes significantly larger than other words. Similarly, given the word "*are*", the neural network needs to predict "*you*" as the next word with higher probability than others.

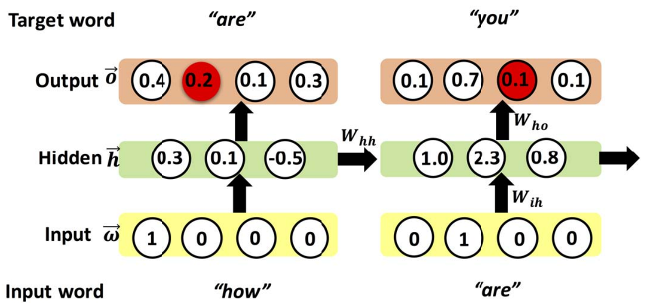


Fig. 4. Recurrent Neural Network Unfolded

B. *Word2Vec*

The Word2Vec algorithm learns the vector representation of each word in a space R^k, where similar words are likely to have similar vectors. Word2Vec is typically applied to very large corpus. For better scalability, Word2Vec adopts the traditional neural net structure.

The vector representation adopted by Word2Vec is significantly different from the one-hot representation used in the aforementioned RNN. Word2Vec maps each word to a real-valued vector in the space R^k, where k is much smaller than the vocabulary size s, while the one-hot representation maps each word to a vector in the space $\{0, 1\}^s$. First, by encoding vectors in a low-dimensional space, Word2Vec reduces the space complexity of computation. More importantly, the vector representation of Word2Vec is in a continuous space, and the distance (i.e., cosine similarity distance) between two vectors effectively measures the similarity of the words. In contrast, the one-hot representation cannot measure similarity. In our work, we treat the Word2Vec as a blackbox.

IV. Applying Deep Learning to Input Generation

At the high level, our approach statistically learns the correlations between a text input value and its context in the training phase (Section IV-B). Then in the prediction phase (Section IV-C), once Monkey needs to provide some value in a textbox, our approach predicts the value based on the context observed by Monkey so far. Section IV-D further explains how we generalize the input generation to make it app-independent. To facilitate the discussion, we first introduce the training dataset (Section IV-A), which is used in the training phase.

A. *Training Dataset*

Without loss of generality, we assume a simplified user action model. In the model, we are only interested in two types of UI elements: button and textbox $\in$ *domain* $\mathcal{T}_{ui}$. A wide range of clickable UI elements such as menu and tableview cell have behaviors similar to button, and hence have the same abstraction in our model. Similarly, a range of text fields that accept user input, such as secure text field and search field, are abstracted as textbox. Note our implementation fully supports all these UI elements. We are only interested in two types of actions, tap and typeText $\in \mathcal{T}_{action}$, which are the most representative behaviors of the UI elements $\in \mathcal{T}_{ui}$.

A user action $\alpha \in \mathcal{A}$ is a tuple that includes the UI element type $\tau_{ui} \in \mathcal{T}_{ui}$, the action type $\tau_{action} \in \mathcal{T}_{action}$, the label ℓ displayed in the UI element (e.g., "Movie" in the button in Figure 5), and optionally the value v involved in the action, e.g., the text input value.

The training dataset essentially records action sequences $\langle \alpha_0, \alpha_1, \ldots, \alpha_n \rangle$ with α_i the user action at the ith step. The subsequence preceding α_n refers to the actions that the user takes to reach the UI screen on which α_n occurs. More details about the action sequence are discussed in Section VI.

We observe that label l is the most prominent information that the user perceives when he/she uses the app, thinking of what to provide as the input value. Based on this observation, we represent every action α in the action sequence with a label ℓ_α, while abstracting away all other information. Besides, we are also interested in the input value v_{α_n} entered in the action α_n. Therefore, the above action sequence is simply represented as $\langle \ell\alpha_0, \ell\alpha_1, \ldots, \ell\alpha_n, v_{\alpha_n} \rangle$. For ease of presentation, we also refer to $\ell\alpha_0, \ell\alpha_1, \ldots, \ell\alpha_n$ as the context of the input value v_{α_n}.

Consider the example in Figure 5, suppose the tester performs actions to trigger the UI screen transitions and to enter the input value. In the first screen, the tester clicks (or taps) the button labeled with "Movie", leading to the second screen with the search bar labeled with "Search". In the third screen, the tester enters the input value "Star Trek" [1] in the search bar.

Following the user action model, we have two actions: clicking the button with the label Movie and entering "Star Trek" in the search bar with the label Search. Accordingly, the sequence recorded is $\langle Movie, Search, Star\ Trek\rangle$.

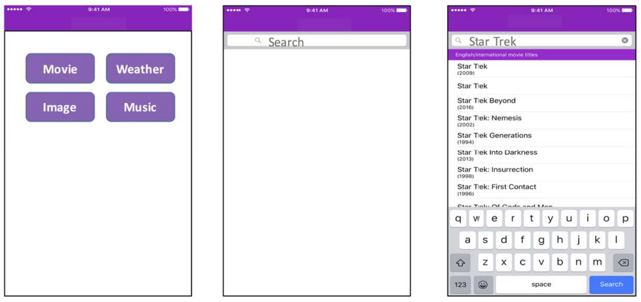

9:41 AM
100%
Movie
Weather
Image
Music
Search
Star Trek
English/international movie titles
Star Trek (2009)
Star Trek
Star Trek Beyond (2016)
Star Trek: Nemesis (2002)
Star Trek Generations (1994)
Star Trek Into Darkness (2013)
Star Trek: Insurrection (1998)
Star Trek: First Contact (1996)
space
Search

Fig. 5. UI Screen Transitions triggered by User Actions

B. Training Phase

Without loss of generality, we assume each label ℓ or input value v is a single word, rather than a phrase. Section VI presents a simple preprocessing that assures the assumption.

Let Vocabulary $\mathcal{V} = \mathcal{L} \cup Val$ denote the list of all the labels and the input values that appear in the training dataset. In the following, we do not distinguish the label and the input value unless otherwise specified. Instead, we refer to them with the general term word.

In order to enable learning, we propose the following vector representation of each word. Given a word ω, its vector form $\vec{\omega}$ has the same size as the vocabulary. And every entry of the vector is defined as,

$$\vec{\omega}[i] = \begin{cases} 1 & \vec{\mathcal{V}}[i] = \omega \\ 0 & otherwise \end{cases}$$

Since only one entry can take the value 1, the vector representation is also referred to as the one-hot representation. Consider Figure 4. At step 2, the input vector is [0,1,0,0], as shown in the input layer.

At each step, the RNN model accepts the vector representation of a word as an input. The input, as well as the previous state of the hidden layer (also encoded as a vector), is used to predict the output, a probability vector which characterizes the probability of each word appearing next in the sequence. The goal of training is to update the parameters of RNN model so that the predicted probability distribution of the next word is close to the true probability distribution observed from the dataset.

[1]Alternatively, the value can be "Star Trek`\n`" where the trailing symbol is equivalent to initializing the search.

We formalize the training as an optimization problem:

$$\underset{x}{\operatorname{argmax}}\ P(v_{\alpha_n} = x | \ell\alpha_0, \ell\alpha_1, \ldots, \ell\alpha_n)$$

Given the context $\ell\alpha_0, \ell\alpha_1, \ldots, \ell\alpha_n$, we find the input value $v_{\alpha_n} = x$ that has the maximum conditional probability. In the training phase, we build a RNN model to predict the conditional probability. The internal parameters of the model are automatically calculated so that the predicted probability distribution approximates the true probability distribution observed from the training dataset. The trained model is then used for prediction under the assumption that the probabilistic association between the context and the input value of a textbox does not change significantly. We explain more details about the formalization in our website [4].

C. Prediction Phase

After the model is trained, it can be used for prediction. In general, the RNN model accepts a sequence of words as the input and outputs a probability vector which characterizes the probability that each word in the vocabulary appears immediately after the sequence (which we refer to as the next word). The sampling of the probability distribution will select a word with the probability described in the distribution.

In our problem settings, when Monkey encounters a textbox, it serializes the action history and the label of the textbox into a sequence, and then it sends the sequence to the trained model to predict the probability distribution of the next word. Lastly, it samples the probability distribution to get the value of next word.

In general, any word in the vocabulary can be the next word. The words in the vocabulary may correspond to action labels, textbox labels or text input values, but we are only interested in predicting the text input values. If a sampled result does not represent a text input value (i.e., it does not belong to the vocabulary of the text input values), we discard it and re-sample the distribution.

The idea of sampling a probability distribution is as follows. Let $\vec{o}$ denote the output vector of the RNN model, which characterizes the probability distribution. We first separate the range between 0 and 1 into $|\vec{o}|$ intervals. Interval i ($0 \le i < |\vec{o}|$) starts at $\sum_{j=0}^{i-1} \vec{o}[j]$ and ends at $\sum_{j=0}^{i} \vec{o}[j]$. A uniform random variable $X \sim U(0,1)$ falls into the interval with the probability $\vec{o}[i]$, i.e., the length of the interval. In other words, the probability that the uniform random variable X falls into interval i is identical to the probability that we select the word $\vec{\mathcal{V}}[i]$. Therefore, if the random variable falls into interval i, we return the word $\vec{\mathcal{V}}[i]$.

Consider the following example. Suppose the output probability vector is [0.1,0.7, 0.1, 0.1], as shown in Figure 4. Figure 6 shows the four intervals. Suppose the uniform random variable is assigned the value 0.5, then it falls into interval 1 (note there is interval 0). Therefore, we return the word $\vec{\mathcal{V}}[1]$, i.e., the word "*are*".

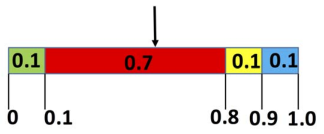


Fig. 6. Sampling the probability distribution.

D. App-independent Input Generation

Our ultimate goal is to predict inputs for apps that we have not trained upon. The main challenge is to make the input generation app-independent.

Our idea is that if we encounter a word that has not been seen in the training phase, we may be able to connect it to some similar word that has been seen in the training phase, leveraging Word2Vec. We use a Word2Vec model trained beforehand using the very large Google News corpus. When encountering a word that is not in the vocabulary, for example, a textbox label that does not occur in any of the apps during training, our system queries the Word2Vec to look for the most similar word in the vocabulary (more precisely, the vocabulary of the action labels and the textbox labels). If the similarity, which can be computed from the vector representations of the two words, is lower than a preset threshold 0.7, we consider that the word does not have the counterpart in the training dataset and simply ignore it during the prediction. In the worst case, if the word is important in determining the text input value, our technique hence degrades to the traditional Monkey testing. These cases happen mostly because the app is in a new category that is very different from the kinds of apps that we have trained on.

V. Implementation

We presented the overview of our system design in Section II. In this section, we explain the implementation details of each component shown in Figure 1.

a) Monkey Testing Engine: Our Monkey adopts a depth-first search strategy in exploring action sequences, i.e., it clicks the available buttons one by one after filling the values for all the textbox elements in the current screen. The buttons inside the same screen can be clicked either in the random order or in some sequential order (e.g., the alphabetic order of the button label). We implemented both options and adopted the random order in our experiment. In addition, to avoid the repeated exploration of the same screen, we constructed the signature of each screen, which consists of the title of the screen and the labels of the buttons in the screen. The screens with the identical signature are treated as the same screen, which are explored only once.

Note our system can be easily extended to predict next actions, in addition to text inputs. However, we consider this would undesirably limit Monkey's ability to achieve good coverage as it would tend to follow human testers' action sequences, which are limited. Hence, one of the important design principles of our system is to use the random exploration for actions and the RNN prediction for text inputs.

We implemented the iOS Monkey with the test automation framework XCTest shipped with the Xcode IDE. The framework provides a set of APIs [9] that allow us to find the button or textbox elements in the screen. For example, we can use the following API call to easily find all the buttons in the current screen:

```
descendantsMatchingType(.Button)
```

b) Text Input Server: The text input server, implemented using the python web framework Bottle [2], is an abstraction layer that hides the details of how the inputs are generated. Since it is implemented in Python and the Monkey testing engine is implemented in Swift, there is a programming language barrier between the two components. To break the barrier, they communicate with each other by sending the HTTP messages in the JSON format. The input server communicates with the RNN instance through the function calls since both are implemented in Python.

In addition, we maintained a database for the input server to store the inputs entered by the human testers in certain contexts. We adopted the MongoDB [7]. The input server interacts with the database using the Python driver PyMongo [10].

c) RNN instance: We built the RNN model on top of the Python deep learning framework Tensorflow [13], which provides the high-level APIs while hiding many low-level details. With the abstraction of Tensorflow, one can build the RNN model with merely 40 lines of code, as shown in our website [3]. We trained the model by first preprocessing the dataset with Word2Vec and then feeding them into the RNN model. Besides, the trained RNN model can be saved to the disk and loaded for further use at any time.

VI. Discussion

In this section, we discuss the assumptions that we make in this work.

An assumption that we make in training/prediction is that a label or an input value is a single word, rather than a phrase or a sentence. In case the label is indeed a phrase, we break it into words and treat each word as a separate label context. By reasoning at the word level, our approach can handle any phrase as long as the words in it belong to our vocabulary. In contrast, for text input values that are phrases, we treat the phrases as atomic units so that they are not partitioned throughout the training and the prediction.

Another important assumption of our approach is that the apps that we train the Monkey upon and the apps that Monkey will test should share some similarity, e.g., they fall into the same category on the App market. Otherwise, if we train the Monkey upon entertainment apps and apply it to test Tax apps, the predicted text inputs would make no sense. To avoid such situations, we collect apps of as many different types as we can and use them as the training subjects.

In our approach, we assume the context comprises only textual labels. In real world apps, developers may use icons such as 🔍 instead of textual labels. We currently do not support non-textual labels. We note that our approach can be extended to support non-textual labels, for example, leveraging the image2text tools [14].

When we record the action sequence, we maintain a list and append each action to the list. However, if we encounter the action that undoes the previous action, we discard the action and remove the previous action from the list; If we encounter the action that leads to the home screen of the app, we simply clear the list.

Lastly, our work complements the large body of recent advances on automatic action sequence generation [28], [24], [27]. While we anticipate great synergy, we will leave it to our future work.

VII. Evaluation

In our experiments, we are interested in the following research questions:

- How effective is our *automated* approach compared to other automated input generation approaches for mobile testing?
- Does Word2Vec allow better results compared to using the RNN model only?
- What is the performance overhead incurred by our tool?

To address the first question, we compare with the automatic random input generation method, which randomly picks [30] an input value from a pool of commonly used input values.

We measure the effectiveness by computing the screen coverage, i.e., how many different UI screens have been explored within a fixed time window. Monkey testing is usually adopted for exploring as many use cases as possible. However, identification of the unique use cases requires domain knowledge. Instead, we use the screen coverage to objectively approximate the use case coverage.

Note that screen coverage is different from the classic coverage criteria (e.g., statement coverage and path coverage) in functional testing. We argue both are important. In particular, our approach complements functional testing in the sense that by providing the meaningful input values, our technique allows Monkey to reach the interesting UI screens. Starting from such UI screens, existing functional testing work can be applied to expose app crashes. Additionally, we argue that some UI screens are interesting even if they do not correspond to any app crash. We refer the readers to our website [5] for such examples.

Note that our approach is orthogonal to automatic event sequence generation approaches. In this work, we adopt the existing depth-first search strategy. Our contributions mainly lie in producing relevant text inputs based on contexts and tolerating the use of different words in different apps.

To address the second research question, we compare the version with only RNN enabled and the version with both RNN and Word2Vec enabled. Section VII-B addresses the above two questions by comparing three versions together.

Lastly, the performance overhead matters, especially given that the prediction happens interactively during Monkey testing. Ideally, the approach should achieve effectiveness with low performance overhead. The measurement of the performance is presented in Section VII-A.

a) Experiment Settings: We run the experiments on a MacBook Pro with OS X EI Capitan. The iOS apps are run in the simulator of iPhone 6S Plus (iOS 9.2). We collect 200 open-sourced iOS apps from Github, mostly from https://github.com/dkhamsing/open-source-ios-apps. They fall into different categories, including Movie, News, Image, Browser, Travel, Radio, Calendar, Weather and Tasks. They include popular apps such as Firefox and Wikipedia. We trained on 150 apps and tested the capability of prediction on the remaining 50 apps.

Our training dataset contains 14061 words in total. When we collect the training dataset, we try to reuse input values as much as possible. For example, when we need to enter a movie name in different apps, we stick to the same movie name.

A. *Performance Overhead*

The learning/training process entails many iterations. The model that we used for prediction was trained for 6 hours, but we measure the performance within a much shorter period. We measure the computation time for every 1000 iterations and report the average time for an iteration. We also use the model generated after every 1000 iterations to predict the next word using a random sequence from the dataset. This is to sample the accuracy of the model. We also measure the time taken by the RNN model for the training and the prediction. As shown in Figure 7, the training/prediction time remains constant except at the initial steps. This is because each iteration handles a fixed amount of data and the neural network structure does not change dynamically. Another important observation is that each prediction takes 0.7 ms on average, which is negligible compared to other actions in the testing (e.g., entering the text).

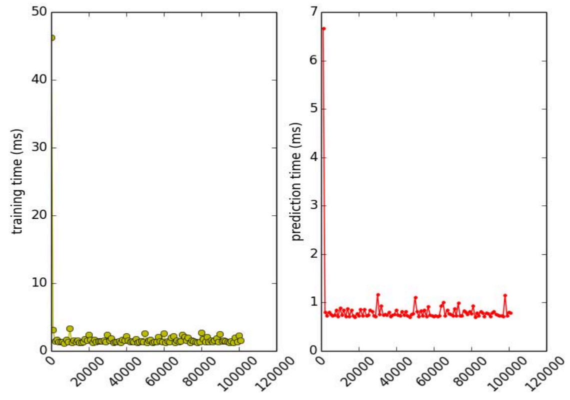


Fig. 7. Running Time of RNN model

We also measure the loading time of the Word2Vec model and the time for each query for a similar word. The loading takes around 54 seconds, while every query takes around 0.7

ms. Due to the space limit, we do not show the figures. The loading time is long because the model is large, which is trained on Google News corpus with 3 billion running words. Our text input server tackles this problem by loading the model during the initialization of the server. Once loaded, the model can be used for all the incoming queries.

B. Effectiveness of Our Approach

To measure the effectiveness, we compare three versions: (1) Random, which randomly picks a value from the pool of commonly used input values. Note that Random version is oblivious of the context information. (2) RNN, which applies the RNN prediction model and hence is aware of the context information, (3) RNN + Word2Vec, which enables both the RNN prediction and the Word2Vec model.

We count the number of screens explored by different versions within 5 minutes. For each version, we repeated the experiment three times and reported the maximal number. The results over the 50 apps are reported in Figure 8. The RNN version detected 21.1% more screens than the Random version, while the RNN + Word2Vec version detected 28.6% more screens than the Random version. As there are a number of apps that have simple functionalities and hence a small number of screens, the three versions explore the same number of screens for those apps. By excluding those cases, we observe that the RNN version outperforms the Random version by 46% whereas the RNN + Word2Vec version outperforms it by 60%. Note the difference between the RNN version and the RNN + Word2Vec version highlights the effectiveness of using the Word2Vec.

We manually inspect the detailed results and have a few interesting observations. First, the three versions produce the same results for 27 apps. The reason is that these apps either do not require any user input or work with any kind of input values. For instance, in PropertyFinder, even when Monkey enters in the search bar a meaningless input value which should lead to no matches, the app still retrieves and displays a list of real estate properties for sale. In another app Chats, the app is for receiving/sending messages, it works regardless of what the message is.

Second, we observe that the RNN version can explore many UI screens that the Random version cannot explore. Intuitively, when the textbox requires the input value of special format or special meanings, the RNN version is aware of the context and produces the proper input value based on the experience it learned from the training dataset. In contrast, the Random version, which is oblivious of the context, usually produces the input values that do not meet with the special requirements. For instance, in the sip-calculator app, there is a textbox with the label "amount", meaning that it expects some numeric input (i.e., the app will perform some arithmetic computation over the input number). Monkey with our RNN model is aware of the context information and has learned that the context is strongly correlated with a numeric input value. As a result, Monkey enters a numeric input value and proceeds to a new screen. In contrast, the Random version does not produce any numeric input and hence cannot proceed.

In an extremal case, the AlzPrevent app (i.e., a research lab's survey app) requires the users to fill in a registration form before they can proceed to a survey. AlzPrevent has 10 screens for the registration process, including the user name, height, weight and some other information. The Random version got stuck in the first page, while our RNN version (and also the RNN + Word2Vec version) knows how to fill in the input values since it has been trained over multiple apps that need registration. As a result, our RNN version outperforms the Random version by 112%, and the RNN + Word2Vec version outperforms it by 212%.

Third, we find the RNN + Word2Vec version performs better than the RNN version when the contexts are slightly different from those in the trained apps. With the Word2Vec model, the version recognizes the semantic similarity among the contexts from different apps. Based on the similarity information, the version predicts the relevant input values based on the knowledge associated with the contexts from the trained apps. The improvement over the RNN model shows the importance and effectiveness of combining the RNN model and the Word2Vec model.

We also present case studies in Section VII-C to demonstrate the strength of our technique.

C. Case Studies

a) Firefox: The first case is from the official Firefox iOS app. This case illustrates the importance of producing relevant input values and demonstrates how our approach produces them. As shown in Figure 9, on Screen 1, the home button is missing (by default) so that the functionality associated with the button cannot be tested. To make the home button appear, one has to tap the setting button to change the setting. However, a valid webpage URL is required as the input value on Screen 2. Otherwise the Home button will not show up. Our Monkey can predict a valid URL for the textbox based on its label "`Enter a webpage`" and the action history Setting $\rightarrow$ Homepage (on Screen 2), and hence enables the Home button, as shown on Screen 3.

The Random version, which is oblivious of the label "`Enter a webpage`" in the textbox, produces a random input value "`New York`". The Firefox app does not accept the input and accordingly does not enable the Home button. In fact, Firefox does not record invalid inputs in its database. After Monkey navigates to another screen and then back, the input value entered by the Random version is lost.

By manually searching for the predicted webpage URL in the training dataset, we identify some interesting connections. The URL was used by a web crawler app and was associated with the label "`website`" in the app. Our approach first recognizes based on Word2Vec the similarity between "`website`" and "`webpage`" (Screen 2), then it predicts based on the RNN model the most likely input value. This example demonstrates the effectiveness of our technique in

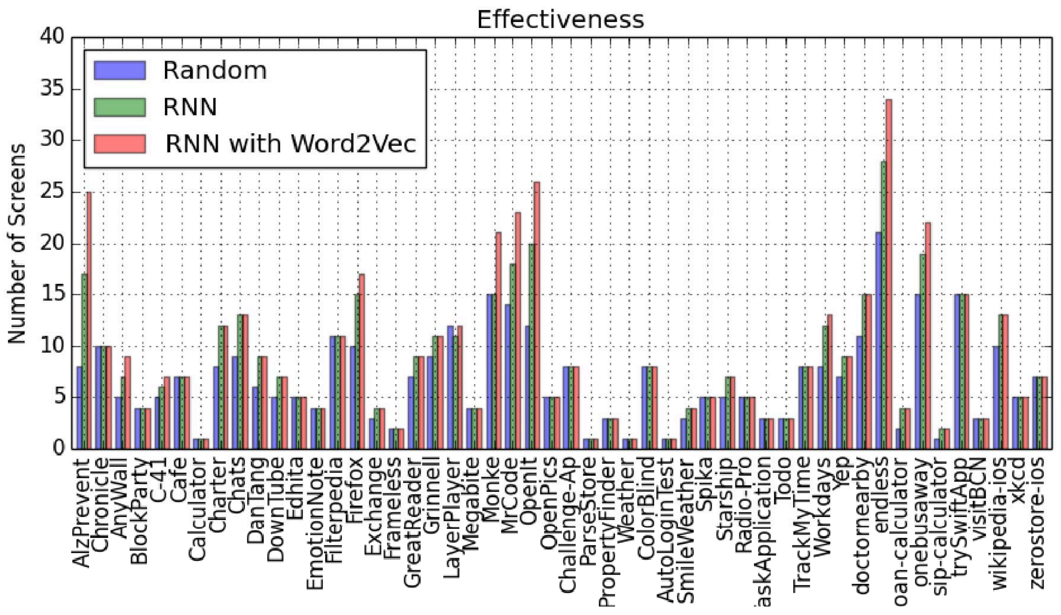


Fig. 8. The measurement of effectiveness

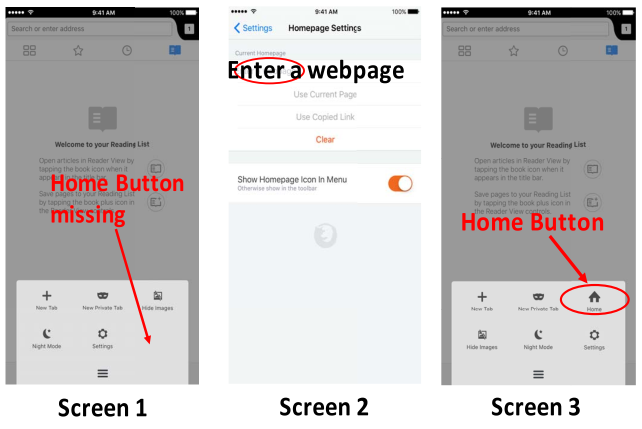


Fig. 9. The Official Firefox App

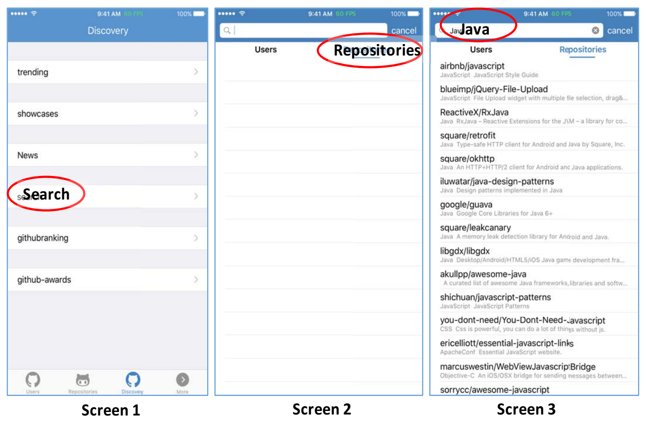
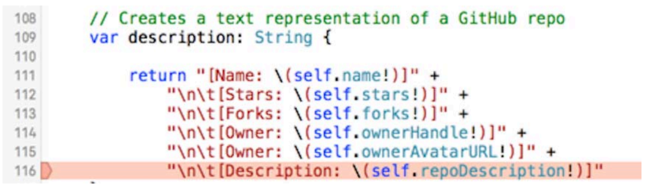


Fig. 10. The Third-party Github App Monkey

exploring new UI screens, even when the two apps have different use cases or business logic.

b) Third-party Github App : Figure 10 shows the case from a third-party app for Github. In this case, our tool taps the `search` menu item (screen 1) and then selects the `Repositories` category (Screen 2) . It also successfully predicts the input value `Java` for the search bar, which leads to further progress with a few matching repositories returned (Screen 3). Our tool further clicks each of the repositories, which leads to the discovery of an exception.

As shown in Figure 11, the exception occurs at line 116. By using the exclamation mark in Swift language [15], the developer assumes that the variable `repoDescription`, which is the description of the repository, cannot be `nil` (i.e., no value). However, in practice, some repository does not have any description, which breaks the developer's assumption. As a consequence, the mobile app crashes when it attempts to unwrap the optional value that is `nil`.

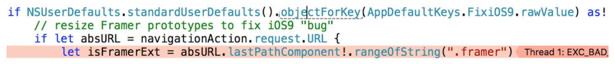


```
108     // Creates a text representation of a GitHub repo
109     var description: String {
110
111         return "[Name: \(self.name!)]" +
112             "\n\t[Stars: \(self.stars!)]" +
113             "\n\t[Forks: \(self.forks!)]" +
114             "\n\t[Owner: \(self.ownerHandle!)]" +
115             "\n\t[Owner: \(self.ownerAvatarURL!)]" +
116             "\n\t[Description: \(self.repoDescription!)]"
```

Fig. 11. The Bug Found in the Github App

In contrast, the Random version produces an input value Benjamin Franklin which is inappropriate in the current context. Accordingly, this version fails to find the above bug. This case clearly shows the usefulness of our tool.

c) Frameless: Frameless is a full-screen web browser which adopts the minimalist UI design.



```
if NSUserDefaults.standardUserDefaults().objectForKey(AppDefaultKeys.FixiOS9.rawValue) as!
    // resize Framer prototypes to fix iOS9 "bug"
    if let absURL = navigationAction.request.URL {
        let isFramerExt = absURL.lastPathComponent!.rangeOfString(".framer")  Thread 1: EXC_BAD
```

Fig. 12. The Bug Found in Frameless

Our tool found a bug similar to the bug found in the Github app, as shown in Figure 12. Finding the bug requires Monkey testing to produce valid inputs. Consider Figure 13, in the first screen, the search bar shows `Website url or search`. Given such a context, our tool produces a valid website in the second screen based on the statistical correlation learned from the training dataset. The mobile app then transits to the third screen. By clicking the `Sign in` button, our tool exposed the exception shown in Figure 12. By inspecting the code, we found that clicking the `Sign in` button leads to the internal execution of some bug-fixing code which unfortunately mishandles the content of the URL. In this case, the context consists of four words, all of which are sent to the RNN model to produce a valid input in the search bar. In contrast, the Random version did not produce the valid URL and therefore cannot find the above bug.

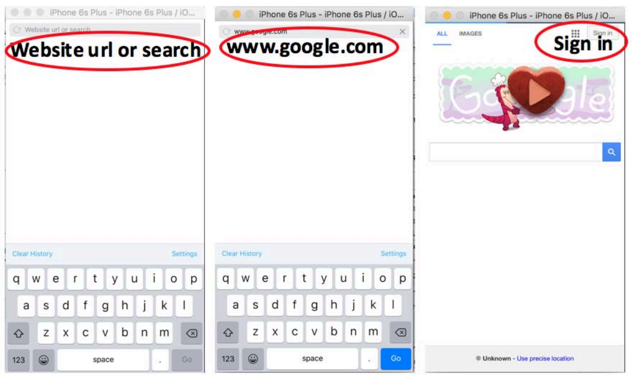

iPhone 6s Plus - iPhone 6s Plus / iO...
Website url or search
iPhone 6s Plus - iPhone 6s Plus / iO...
www.google.com
iPhone 6s Plus - iPhone 6s Plus / iO...
ALL
IMAGES
Sign in
Clear History
Settings
space
Go
Unknown - Use precise location

Fig. 13. The Bug Found in Frameless

VIII. Related work

Our work is closely related to automated test generation for Android apps. Monkey [8] and Dynodroid [28] are random exploration based UI events generation tools. GUIRipper [18] (MobiGUITAR [19]), ORBIT [37], A^{3}E [21], SwiftHand [23] and PUMA [25] build finite state models for UI and generate events to systematically explore states in the model. Contest [20] generates events based on a concolic execution approach and prunes search space by checking conditions among event sequences. Ermuth and Pradel [24] introduced the macro event that summarizes recurring sequences of low-level UI events for a single step. By combining macro events with random testings, they leverage recorded user interaction sequences and automatically generate new tests. Compared to our approach, most of automated test generation work focuses on generating event sequences (or action sequences).

Besides events and intents, generating test input values is also important as some behaviors can only be exposed if the predicates on input values are satisfied. Symbolic execution and evolutionary algorithm based techniques have also been applied. JPF-Android [36] extends Java PathFinder (JPF) and is a model checking tool to explore all paths and identify runtime failures. EvoDroid [29] generates tests (both events and inputs) based on an evolutionary algorithm framework. It uses a random approach to generate inputs. Sapienz [30] is a multi-objective search based testing tool for Android apps. It combines random fuzzing, systematic and search-based exploration, exploiting seeding and multi-level instrumentation. Although it can provide strings as test inputs, these strings are extracted from the app by reverse-engineering the APK and randomly seeded into the text fields. The randomly selected inputs are unlikely to be relevant in a specific context. Afshan et al. [17] apply the guided search based on the n-gram language model to produce the readable string inputs rather than random character sequences. However, the approach is not designed to produce the string inputs in a use case context.

Our work is also related to machine learning based text modeling and generation techniques. Sutskever et al. [35] demonstrated the power of large trained RNNs by applying them to the task of predicting the next character in a stream of text. Melicher et al. [31] proposed a neural network based approach to model human-chosen password and measure its resistances to guessing attacks.

IX. Conclusion

We have developed a deep learning based approach to automatically generate the text inputs for the mobile testing. It produces the most relevant input values in a context. Besides, we have leveraged the Word2Vec model to achieve the app-independence. The evaluation over 50 iOS apps confirms the effectiveness and efficiency of our designs.

Acknowledgment

We thank the anonymous reviewers for their constructive comments. This research was supported, in part, by DARPA under contract FA8650-15-C-7562, NSF under awards 1409668, 1320444, and 1320306, ONR under contract N000141410468, and Cisco Systems under an unrestricted gift. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

References

[1] Apple app store growing by over 1,000 apps per day. http://www.ibtimes.co.uk/apple-app-store-growing-by-over-1000-apps-per-day-1504801. June 6, 2015.
[2] Bottle. https://bottlepy.org/.
[3] Code snippet for building the rnn model. https://sites.google.com/site/monkeyislearning/rnn.

[4] Formalizing the training as an optimization problem. https://sites.google.com/site/monkeyislearning/home/formal-model.

[5] Functions that are not implemented. https://sites.google.com/site/monkeyislearning/non-crash-cases.

[6] How many new apps are added to google play everyday? https://www.quora.com/How-many-new-apps-are-added-to-Google-Play-everyday.

[7] Mongodb. https://www.mongodb.com/.

[8] The monkey ui android testing tool. http://developer.android.com/tools/help/monkey.html.

[9] Official documentation of xctest apis. https://developer.apple.com/reference/xctest/xcuielementquery.

[10] Pymongo. https://docs.mongodb.com/getting-started/python/client/.

[11] Sigmoid function. https://en.wikipedia.org/wiki/Sigmoid_function.

[12] Tanh. https://reference.wolfram.com/language/ref/Tanh.html.

[13] Tensorflow. https://www.tensorflow.org.

[14] Toronto deep learning demos. http://deeplearning.cs.toronto.edu/.

[15] Unwrap an optional value. http://www.mokacoding.com/blog/why-implicitly-unwrapping-swift-optionals-is-dangerous/.

[16] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *OSDI*, 2016.

[17] S. Afshan, P. McMinn, and M. Stevenson. Evolving readable string test inputs using a natural language model to reduce human oracle cost. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, 2013.

[18] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using gui ripping for automated testing of android applications. In *ASE*, pages 258–261, New York, NY, USA, 2012. ACM.

[19] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon. Mobiguitar: Automated model-based testing of mobile apps. *IEEE Software*, 32(5):53–59, Sept 2015.

[20] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *FSE*, pages 59:1–59:11, New York, NY, USA, 2012. ACM.

[21] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *OOPSLA*, pages 641–660, New York, NY, USA, 2013. ACM.

[22] C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, Feb. 2013.

[23] W. Choi, G. Necula, and K. Sen. Guided gui testing of android apps with minimal restart and approximate learning. In *OOPSLA*, pages 623–640, New York, NY, USA, 2013. ACM.

[24] M. Ermuth and M. Pradel. Monkey see, monkey do: Effective generation of gui tests with inferred macro events. In *ISSTA*, pages 82–93, New York, NY, USA, 2016. ACM.

[25] 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 *MobiSys*, pages 204–217, New York, NY, USA, 2014. ACM.

[26] M. Harman and P. McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Trans. Softw. Eng.*, 36(2):226–247, Mar. 2010.

[27] C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *ISSTA*, pages 67–77, New York, NY, USA, 2013. ACM.

[28] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *ESEC/FSE*, pages 224–234, New York, NY, USA, 2013. ACM.

[29] R. Mahmood, N. Mirzaei, and S. Malek. Evodroid: Segmented evolutionary testing of android apps. In *FSE*, pages 599–609, New York, NY, USA, 2014. ACM.

[30] 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*, ISSTA 2016, pages 94–105, New York, NY, USA, 2016. ACM.

[31] W. Melicher, B. Ur, S. M. Segreti, S. Komanduri, L. Bauer, N. Christin, and L. F. Cranor. Fast, lean, and accurate: Modeling password guessability using neural networks. In *USENIX Security*, pages 175–191, Austin, TX, Aug. 2016. USENIX Association.

[32] T. Mikolov, M. Karafit, L. Burget, J. Cernock, and S. Khudanpur. Recurrent neural network based language model. In *INTERSPEECH*, 2010.

[33] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. *CoRR*, 2013.

[34] C. Pacheco and M. D. Ernst. Randoop: Feedback-directed random testing for java. In *OOPSLA*, pages 815–816, New York, NY, USA, 2007. ACM.

[35] I. Sutskever, J. Martens, and G. E. Hinton. Generating text with recurrent neural networks. In *ICML*, pages 1017–1024, 2011.

[36] H. van der Merwe, B. van der Merwe, and W. Visser. Execution and property specifications for jpf-android. *SIGSOFT Softw. Eng. Notes*, 39(1):1–5, Feb. 2014.

[37] W. Yang, M. R. Prasad, and T. Xie. A grey-box approach for automated gui-model generation of mobile applications. In *FASE*, 2013.