2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 978-1-6654-0337-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/ASE51524.2021.9678778

Deep GUI: Black-box GUI Input Generation with Deep Learning

Faraz YazdaniBanafsheDaragh
School of Information and Computer Sciences
University of California, Irvine, USA
faraz.yazdani@uci.edu

Sam Malek
School of Information and Computer Sciences
University of California, Irvine, USA
malek@uci.edu

***Abstract*—Despite the proliferation of Android testing tools, Google Monkey has remained the de facto standard for practitioners. The popularity of Google Monkey is largely due to the fact that it is a *black-box* testing tool, making it widely applicable to all types of Android apps, regardless of their underlying implementation details. An important drawback of Google Monkey, however, is the fact that it uses the most naive form of test input generation technique, i.e., random testing. In this work, we present *Deep GUI*, an approach that aims to complement the benefits of black-box testing with a more intelligent form of GUI input generation. Given only screenshots of apps, Deep GUI first employs deep learning to construct a model of valid GUI interactions. It then uses this model to generate effective inputs for an app under test without the need to probe its implementation details. Moreover, since the data collection, training, and inference processes are performed independent of the platform, the model inferred by Deep GUI has application for testing apps in other platforms as well. We implemented a prototype of Deep GUI in a tool called *Monkey*++ by extending Google Monkey and evaluated it for its ability to crawl Android apps. We found that Monkey++ achieves significant improvements over Google Monkey in cases where an app's UI is complex, requiring sophisticated inputs. Furthermore, our experimental results demonstrate the model inferred using Deep GUI can be reused for effective GUI input generation across platforms without the need for retraining.**

I. Introduction

Automatic input generation for Android applications (apps) has been a hot topic for the past decade in the software engineering community [1]–[14]. Input generators have a variety of applications. Among others, they are used for verifying functional correctness (e.g., [13], [15], [16]), security (e.g., [17], [18]), energy consumption (e.g., [19], [20]), and accessibility (e.g., [21]) of apps. Depending on the objective at hand, input generators can be very generic, and simply crawl apps to maximize coverage [22]–[24], or can be very specific, looking for certain criteria to be fulfilled, such as reaching activities with specific attributes [2].

Common across the majority of existing input generators is the fact that they are *white-box*, i.e., require access to implementation details of the *app under test (AUT)*. For instance, many tools use static analysis to find the right combination of interactions with the AUT [1]–[4], while other tools depend on the XML-based GUI layout of the AUT to find the GUI widgets and interact with them [5]–[14]. The underlying implementation details of an AUT provide these tools with insights to produce effective inputs, but also pose severe limitations that compromise the applicability of these tools. First, there is a substantial degree of heterogeneity in the implementation details of apps. Consider for instance the fact that many Android apps are non-native, e.g., built out of activities that are just wrappers for web content. In these situations, the majority of existing tools either fail to operate or achieve very poor results. Second, the source code analyses underlying these tools are tightly coupled to the Android platform, and often to specific versions of it, making them extremely fragile when used for testing apps in a new environment.

Black-box input generation tools do not suffer from the same shortcomings. Google Monkey is the most widely used black-box testing tool for Android. Despite being a random input generator, prior studies suggest Google Monkey outperforms many of the existing white- and gray-box tools [25]. This can be attributed to the fact that Google Monkey is significantly more robust than almost all other existing tools, i.e., it works on all types of apps regardless of how they are implemented. However, Google Monkey employs the most basic form of input generation strategy. It blindly interacts with the screen without knowing if its actions are valid. This might work well in apps with a simple GUI, where the probability of randomly choosing a valid action is high, but not in apps with a complex GUI. For instance, take Figure 1. In Figure 1a, since most of the screen contains buttons, almost all of the times that Google Monkey decides to generate a touch action, it touches something valid and therefore tests a functionality. However, in Figure 1b, it is much less probable for Google Monkey to successfully touch the one button that exists on the screen, and therefore it takes much longer than needed for it to test the app's functionality.

This article presents *Deep GUI*, a black-box GUI input generation technique with deep learning that aims to address the above-mentioned shortcoming. Deep GUI is able to filter out the parts of the screen that are irrelevant with respect to a specific action, such as touch, and therefore increases the probability of correctly interacting with the AUT. For example, given the screenshot shown in Figure 1b, Deep GUI first produces the heatmap in Figure 1c, which shows for each pixel the probability of that pixel belonging to a touchable widget. It then uses this heatmap to touch the pixels with a

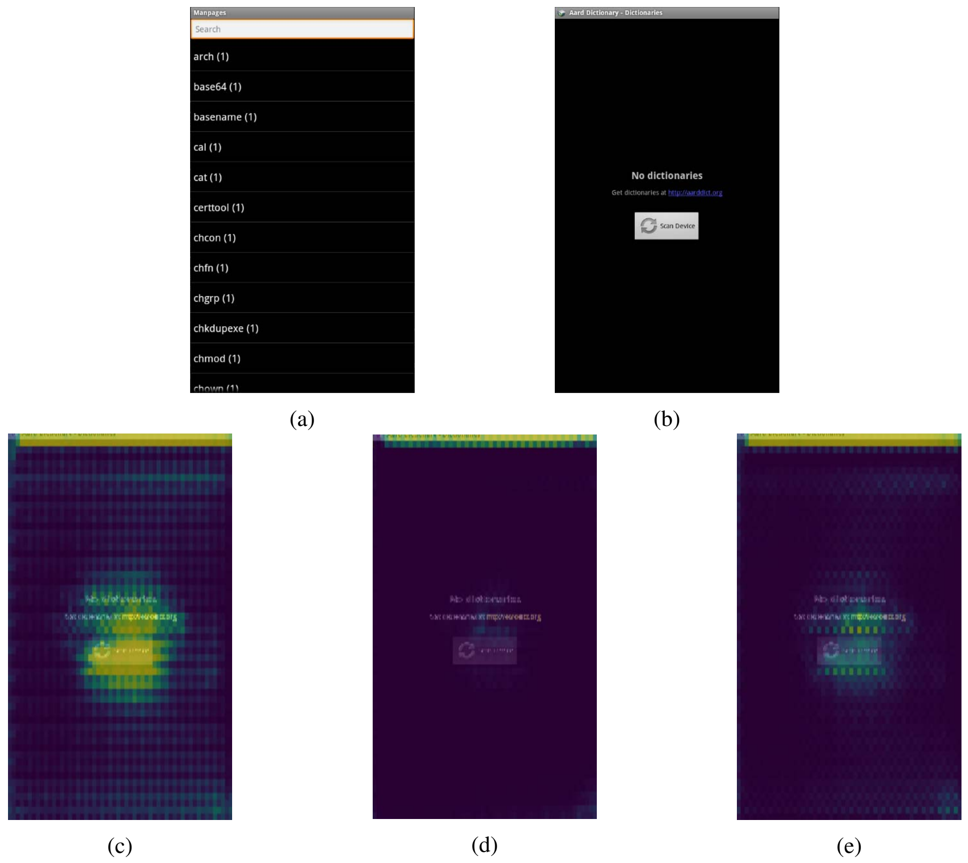


Fig. 1: Two examples where it is respectively easy (a) and difficult (b) for Google Monkey to find a valid action, as well as the heatmaps generated by Deep GUI associated with (b) for touch (c), scroll (d), and swipe (e) actions respectively. Note that in (c) the model correctly identifies both the button and the hyperlink –and not the plain text– as touchable.

probability that is proportionate to their heatmap value, hence increasing the chance of touching the button in this example.

In order to produce such heatmaps, Deep GUI undertakes a deep-learning approach. We further show that this approach is a special case of a more general method known as deep reinforcement learning, and we discuss how this method can be used to develop even more intelligent input generation tools. Moreover, what makes Deep GUI unique is that it uses a completely black-box and cross-platform method to collect data, learn from it, and produce the mentioned heatmaps, and hence supports all situations, applications, and platforms. It also uses the power of *transfer learning* to make its training more data-efficient and faster. Our experimental evaluation shows that Deep GUI is able to improve Google Monkey's performance on apps with complex GUIs, where Google Monkey struggles to find valid actions. It also shows that we can take a Deep GUI model that is trained on Android, and use it on other platforms, specifically web in our experiments, for efficient input generation.

In summary, this article makes the following contributions:

1) We propose Deep GUI, a black-box approach for generation of GUI inputs using deep learning. To the best of our knowledge, this is the first approach that uses a completely black-box and cross-platform approach for data collection, training, and inference in the generation of test inputs.
2) We provide an implementation of Deep GUI for Android, called *Monkey*++, by extending Google Monkey. We make this tool available publicly.[1]
3) We present detailed evaluation of Deep GUI using Androtest benchmark [25], consisting of 31 real-world mobile apps, as well as the top 15 websites in the US [26]. Our results corroborate Deep GUI's ability to improve both the code coverage and the speed with which this coverage can be attained.

The remainder of this paper is organized as follows. Section II describes the details of our approach. Section III provides our evaluation results. Section IV reviews the most relevant prior work. Finally, in Section V, the paper concludes with a discussion of our contributions, limitations of our work, and directions for future research.

[1] https://github.com/Feri73/deep-gui

II. Approach

We formally provide our definition of the problem for automatically generating inputs in a test environment. Suppose that at each timestep t, the environment provides us with its state s_t. This can be as simple as the screenshot, or can be a more complicated content such as the UI tree. Also, suppose we define $A = \{\alpha_1, ...\alpha_N\}$ as the set of all possible actions that can be performed in the environment at all timesteps. For instance, in Figure 1b, all of the touch events associated with all pixels on the screen can be included in A. Note that these actions are not necessarily valid. We define a valid action as an action that results in triggering a functionality (like touching the send button) or changing the UI state (like scrolling down a list). Let us define $r_t = r(s_t, a_t)$ to be 1 if a_t is valid when performed on s_t, and 0 otherwise. Our goal is to come up with a function Q that, given s_t, produces the probability of validity for each possible action. That is, $Q(s_t, a_t)$ identifies how probable it is for a_t to be a valid action when performed on s_t. Therefore, Q is essentially a binary classifier (valid vs. non-valid) conditioned on s_t independently for each action in the set A. For simplicity, we also define $Q(s_t)$ as a function that, given an action α, returns $Q(s_t, \alpha)$. That is, $Q(s_t)(\alpha) = Q(s_t, \alpha)$.

In Deep GUI, we consider s_t to be the screenshot of AUT at each timestep. Set A consists of touch, up and down scroll, and right and left swipe events, on all of the pixels of the screen. We also define r_t as follows:

$$r(s_t, a_t) = \begin{cases} 0 & \text{if equals}(s_t, s_{t+1}) \\ 1 & \text{otherwise} \end{cases}$$

That is, if the screenshot undergoes a legitimate change after an action, we consider that action to be a valid one in that screen. We define what a legitimate change means later in this section. Note that we defined s_t, A, and r_t independent of the platform on which AUT operates. Therefore, this approach can be used in almost all existing test environments.

This work consists of four components:

A. Data collection: This component helps in collecting necessary data to learn from.

B. Model: At the core of this component is a deep neural network that processes s_t and produces a heatmap $Q(s_t)$ for all possible actions a_t, such as the ones shown in Figure 1. The neural network is initialized with weights learned from large image classification tasks to provide faster training.

C. Inference: After training, and at the inference time, there are multiple readout mechanisms available for using the produced heatmaps and generating a single action. These mechanisms are used in a hybrid fashion to provide us with the advantages of all of them.

D. Monkey++: This is the only component that is specialized for Android, and its application is to fairly compare Deep GUI with Google Monkey. It also provides a convenient medium to use Deep GUI for testing of Android apps, as it can replace Google Monkey and be used in practically the same way.

Figure 2 shows an overview of these four components and how they interact.

A. Data Collection

Since we reduced the problem to a classification problem, each datapoint in our dataset needs to be in the form of a three-way tuple (s_t, a_t, r_t), where our model tries to classify the pair (s_t, a_t) into one of the two values that r_t represents, i.e. whether performing the action a_t on the state s_t is valid or not. Training a deep neural network requires a large amount of data for training. To that end, we have developed an automatic method to generate this dataset.

As defined above, r_t represents whether the screen has a legitimate change after an action. We here define legitimate change as a change that does not involve an animated part of the screen. In other words, if specific parts of the screen change even in case of no interaction with the app, we filter those parts out when computing r_t. For instance, in Android, when focused on a textbox, a cursor keeps appearing and disappearing every second. We filter out the pixels corresponding to the cursor.

For data collection, we first dedicate a set of apps to be crawled. Then, for each app, we randomly interact with the app with the actions in the set A and record the screenshot, the action, and whether the action resulted in a legitimate change. In order to filter out animated parts of the screen, before each action, we first record the screen for 5 seconds and consider all pixels that change during this period to be animated pixels. While this method does not fully filter all of the illegitimate changes[2], as our experimental results suggest, it is adequate.

A keen observer would realize that this method of data collection is a very natural choice in the realm of Android. For years, Google Monkey has been used to crawl Android apps for different purposes, but the valuable data that it produces has never been leveraged to improve its effectiveness. That is, *even if a particular app has already been crawled by Google Monkey thousands of times before, when Google Monkey is used to crawl that app, it still crawls randomly and makes all of the mistakes that it has already made thousands of times before.* The collection method described here is an attempt to share these experiences by training a model and exploiting such model to improve the effectiveness of testing, as we discuss next.

B. Model

While, as discussed above, the problem is to classify the validity of a single action a_t when performed on s_t, it does not mean that each datapoint (s_t, a_t, r_t) cannot be informative about actions other than a_t. For instance, if touching a point results in a valid action, touching its adjacent points may also result in a valid action with a high probability. This can

[2] For instance, if an accumulative progress bar is being shown, this method may not work.

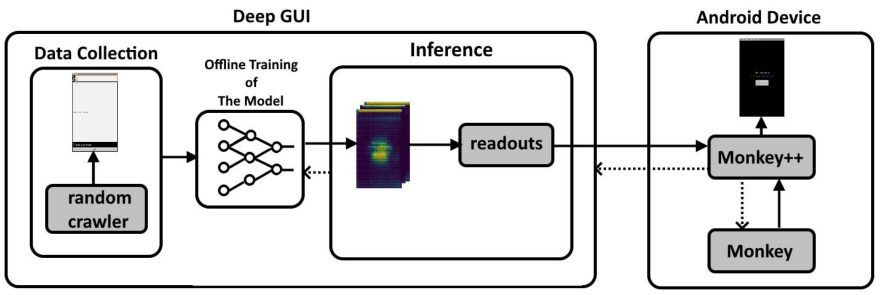


Fig. 2: Overview of the components comprising Deep GUI & Monkey++.

make our training process much faster and more data-efficient. Therefore, we need a model that can capture such logic.

1) Input and Output: As the first step toward this goal, in our model, we define input and output as follows. Input is a 3-channel image that represents s_t, the screenshot of the AUT at time t. For output, we require our model to perform the classification task for all the actions of all types (i.e., touch, scroll, swipe, etc.), and not just a_t. While we do not directly use the prediction for other actions to generate gradients when training, this enables us to (1) use a more intuitive model, and (2) use the model at inference time by choosing the action that is most confidently classified to be valid. We use a T-channel heatmap to represent our output; T being the number of action types, i.e. touch, scroll, swipe. Note that we do not differentiate between up/down scroll or left/right swipe at this stage. Each channel is a heatmap for the action type it represents. For each action type, the value at (i, j) of the heatmap associated with that action type represents the probability that the model assigns to the validity of performing that action type at location (i, j). For instance, in Figure 1, the three heatmaps 1c, 1d, and 1e show the model's confidence in performing touch, scroll, and swipe, respectively, at different locations on the screen.

2) UNet: We also would need a model that can intuitively relate the input and output, as defined above. We use a *UNet architecture*, since it has shown to be effective in applications such as image segmentation, where the output is an altered version of the input image [27]. In this architecture, the input image is first processed in a sequence of *convolutional layers* known as the *contracting path*. Each of these layers reduces the dimensionality of the data while potentially encoding different parts of the information relevant to the task at hand. The contracting path is followed by the *expansive path*, where various pieces of information at different layers are combined using *transposed convolutional layers*[3] to expand the dimensionality to the suitable format required by the problem. In our case, the output would be a 3-channel heatmap. In order for this heatmap to produce values between 0 and 1 (as explained above), it is processed by a *sigmoid* function in the last step of the model. As one can notice, because of the nature of convolutional and transposed convolutional layers, adjacent coordinate pairs are processed more similarly than other pairs.

[3] In some references these are referred to as deconvolutional layers.

This makes it easier for the network to make deductions about all actions, and not just a_t. Moreover, the entire model seems to have an intuitive design: First, the relevant parts of information are extracted and grouped in different layers, and then combined to form the output. This is similar to how the UI elements are usually represented in software applications as a GUI tree.

3) Transfer Learning: While Google Monkey might struggle in finding valid actions when crawling an app, and other tools might need to use other information such as GUI tree or source code to detect such actions, humans find the logic behind a valid action to be pretty intuitive, and can learn it within minutes of encountering a new environment. The reason behind this "intuition" lies in the much more elaborate visual experience that humans have that goes beyond the Android environments. Since birth, we see a myriad of objects in a myriad of contexts, and we learn to distinguish objects from their backgrounds. This information helps us a lot to distinguish a button in the background of an app, even if the background itself is a complicated image. Because of this, we humans do not need thousands of examples to learn to interact with an environment.

How can we use this fact to get the same training performance with fewer data in our tool? Research in machine learning has shown the possibility of achieving this through *transfer learning* [29]. In transfer learning, instead of a randomly initialized network, an existing model previously trained on a dataset for a potentially different but related problem is used as the starting point of all or some part of the network. This way, we "transfer" all the experience related to that dataset (as summarized in the trained weights), without having invested time to actually process it. Therefore, training is more data-efficient. This is in particular important for us because, as discussed, the data collection process is very time-consuming given that the tool needs to monitor the screen for animations before collecting each datapoint.

The contracting path of the UNet seems like a perfect candidate for transfer learning because, unlike the expansive path, it is more related to how the network processes the input, rather than how it produces the output. This means that any trained model that exists for processing an image can be a candidate for us to use its weights.

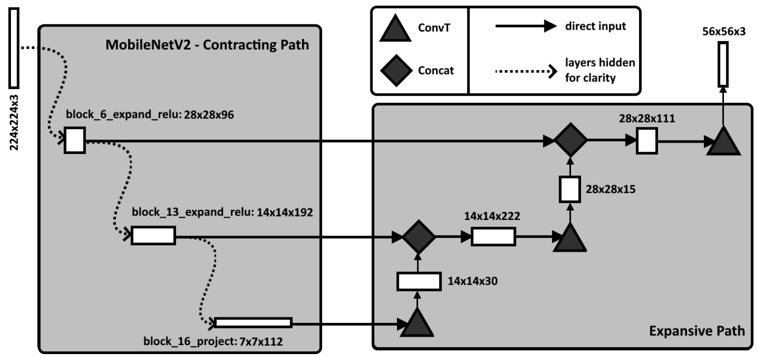

MobileNetV2 - Contracting Path
224x224x3
block_6_expand_relu: 28x28x96
block_13_expand_relu: 14x14x192
block_16_project: 7x7x112
ConvT
Concat
direct input
layers hidden for clarity
56x56x3
28x28x111
28x28x15
14x14x222
14x14x30
Expansive Path

Fig. 3: The deep neural network architecture used in Deep GUI. The layers' names shown in MobileNetV2 are from Tensorflow [28] implementation of the architecture. `ConvT` is a transpose convolutional layer.

In this work, as the contracting path, we used part of the network architecture MobileNetV2 [30] trained on the ImageNet dataset [31].[4] We chose MobileNetV2 because it is powerful and yet lightweight enough to be used inside mobile phones if necessary. Figure 3 shows how MobileNetV2 interacts with our expansive path to build the model used in Deep GUI. Note that in order for the screenshot to be compatible with the already trained MobileNetV2 model, we first resize it to 224×224. Also, because of computational reasons, the produced output is 56×56, and is later upsampled linearly to the true screen size.

4) Training: At the training time, for each datapoint (s_t, a_t, r_t), the network first produces $Q(s_t)$ as the described heatmaps. Then, using the information about the performed action a_t, it indexes the network's prediction for the action to get $Q(s_t)(a_t) = Q(s_t, a_t)$. Finally, since this is a classification task, we use a binary crossentropy loss between r_t and $Q(s_t, a_t)$ to generate gradients and train the network.

C. Inference

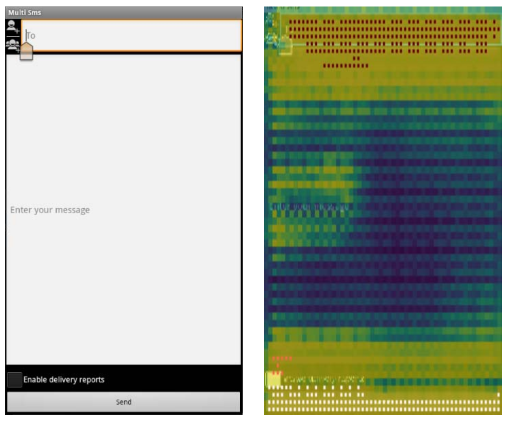
Once we have the trained model, we would like to be able to use it to pick an action given a screenshot of an app at a specific state. Therefore, we require a readout function that can sample an action from the produced heatmaps. Here, we propose two readouts, and we explain how we use both in Deep GUI.

The simplest possible readout is one that samples actions based on their relative prediction. That is, the more probable the network thinks it is for the action to be a valid one, the more probable it is for the action to be sampled. For this to happen, we need to normalize the heatmaps to a probability distribution over all actions of all types. Formally:

$$p(a_t = \alpha | s_t) = \frac{f(Q(s_t, \alpha))}{\sum_{\alpha' \in A} f(Q(s_t, \alpha'))}$$

where f identifies the kernel function. For instance if $f(x) = \exp(x)$, we have a softmax normalization. In our work, we

[4] Please note that we used this existing trained model as the initialization of the contracting path. In the training step, we do train the weights on the contracting path.



(a) (b)

Fig. 4: (a) An example of a screen with equally important widgets of different sizes. (b) The touch channel of the produced heatmap. The pixels belonging to different clusters that the `cluster_sampling` readout detects are colored with maroon, red, and white, depending on the cluster they belong to.

chose to use the linear kernel $f(x) = x$. Using the probability distribution that the linear kernel produces, we then sample an action. We call this method the `weighted_sampling` readout.

However, humans usually interact with apps differently. We see widgets rather than pixels, and interact with those widgets as a whole. The `weighted_sampling` readout does not take this into account as it treats each pixel independently. Take Figure 4a as an example. The "Enable delivery reports" checkbox is potentially as important as the send button, because if it is checked a new functionality can be tested. However, because the button is larger than the checkbox, it takes the `weighted_sampling` readout longer to finally toggle the checkbox and test the new functionality.

To address this issue, we use the `cluster_sampling` readout. In this approach, we first filter out all the actions α for which the predicted $Q(s_t, \alpha)$ is less than a certain threshold. This way, we ensure only the actions that are highly probable to be valid are considered. In Deep GUI this threshold is 0.99. Then, for each channel in $Q(s_t)$, we use agglomerative clustering as implemented in python library `scikit-learn` [32] to cluster the pixels into widgets. Figure 4b shows the clustering result for the touch channel of the heatmap corresponding to Figure 4a. After detecting the clusters, we first randomly choose one of the action types, and then randomly choose one of the clusters (i.e. widgets) in the channel associated with that action type. Finally, we choose a random pixel that belongs to that cluster and generate a_t.

While configurable, in our experiments we used a hybrid readout that uses `weighted_sampling` in 30% of the times, and `cluster_sampling` in 70% of the times. This way, we exploit the benefits that `cluster_sampling` offers, while we make sure we do not completely abandon certain valid actions because of the imperfections of the tool.

The discussed readouts identify the action type and the location of it on the screen. However, scroll and swipe also require other parameters such as direction or length. Deep GUI chooses these parameters randomly. Also, because swipe and scroll are mostly used to discover other buttons, while touch is actually the action that triggers the functionality of the buttons, we configure the described readouts so that they are more biased towards choosing the touch action.[5]

D. Monkey++

While touch, swipe, and scroll are the most used action types when interacting with an environment, there are other actions that may affect the ability of a tool to crawl Android apps. In order to cover those actions as well, and also in order to be able to compare Google Monkey with our solution fairly in the Android environment, we introduce Monkey++, which is an extension to Google Monkey. Monkey++ consists of a server side, which responds to queries with Deep GUI, and a client side, which is implemented inside Google Monkey.

Google Monkey works as follows. First, it randomly chooses an action type (based on the probabilities provided to it when starting it), and then randomly chooses the parameters (such as the location to touch). Monkey++ works the same as Google Monkey with one exception. If the chosen action type is touch or gesture (which represents all types of movement, including scroll and swipe), instead of proceeding with the standard random procedure in Google Monkey, it sends a query to the server side. Using the inference procedure described above, Deep GUI samples an action and returns to the client, which is then performed on the device. Algorithm 1 shows how Monkey++ works.

Algorithm 1: Monkey++ algorithm

```
while Google Monkey is running do
    get action type $t$ from Google Monkey;
    if $t$ is touch or gesture then
        get action $a$ from Deep GUI server
    else
        continue with Google Monkey and get action $a$
    end
    perform $a$
end
```

III. Evaluation

We evaluated Deep GUI with respect to the following research questions:

RQ1. How does Monkey++ compare to Google Monkey?
RQ2. Can Deep GUI be used to generate effective test inputs across platforms?
RQ3. How much is transfer learning helping Deep GUI in learning better and faster?

We used the apps in the Androtest benchmark [25] as our pool of apps. Out of 66 apps available[6], we randomly chose 28 for training, 6 for validation, and 31 for testing purposes. We also eliminated one of the apps because of its incompatibility with our data collection procedure.[7]

To support a variety of screen sizes, we collected data from virtual devices of size 240×320 and also 480×854, and trained a single model that is used in the experiments explained in Sections RQ1 and RQ2. We collected an overall amount of $210,000$ data points. Virtual devices, both for data collection and the Android experiments, were equipped with a $200MB$ virtual SD card, as well as $4GB$ of RAM. For data collection, training, and the experiments, we used an Ubuntu 18.04 LTS workstation with 24 Intel Xenon CPUs and $150GB$ RAM. We did not use GPUs at any stage of this work. The entire source code for this work, the experiments, and the analysis is available at https://github.com/Feri73/deep-gui.

RQ1. Line Coverage

In order to test the ability of Monkey++ in exploring Android apps, we ran both Monkey++ and Google Monkey on each app in the test set for one hour, and monitored line coverage of the AUT every 60 seconds using Emma [33]. We ran 9 instances of this experiment in parallel, and calculated the average across different executions of each tool. Table I shows the final line coverage for the apps in the test set. While in some apps Monkey++ and Google Monkey perform similarly, in other apps, such as `com.kvance.Nectroid`, Monkey++ significantly outperforms Google Monkey. We believe this is directly related to an attribute of apps, referred to as *Crawling Complexity (CC)* in this paper.

CC is a measure of the complexity of exploring an app. Different factors can affect this value. For instance, if the majority of the app's code is executed at the startup, there

[5] In `weighted_sampling`, we multiply each heatmap belonging to touch, scroll, and swipe with 1, 0.3, and 0.1 respectively. In `cluster_sampling`, when randomly choosing an action type from the available ones, we use the same three numbers to bias the probability.

[6] Three apps caused crashes in the emulators and hence were not used.

[7] Application `org.jtb.alogcat` keeps updating the screen with new logs from the logcat regardless of the interactions with it, which highly deviates from the behavior of a normal Android app.

TABLE I: The results of running Monkey++ and Google Monkey on the test set, sorted by Crawling Complexity. The shading indicates the tool that achieved the best result.

Application	Crawling Complexity	Monkey++ Line Coverage	G Monkey Line Coverage
`es.senselesssolutions.gpl.weightchart`	2.8	67%	65%
`com.hectorone.multismssender`	2.6	64%	67%
`com.templaro.opsiz.aka`	2.4	72%	66%
`com.kvance.Nectroid`	2.3	65%	50%
`com.tum.yahtzee`	2.3	67%	61%
`in.shick.lockpatterngenerator`	2.2	86%	84%
`net.jaqpot.netcounter`	2.2	71%	69%
`org.waxworlds.edam.importcontacts`	2.0	41%	34%
`cri.sanity`	1.8	25%	23%
`com.chmod0.manpages`	1.7	72%	63%
`com.google.android.divideandconquer`	1.5	85%	88%
`com.example.android.musicplayer`	1.3	71%	71%
`ch.blinkenlights.battery`	1.3	91%	93%
`org.smerty.zooborns`	1.2	34%	33%
`com.android.spritemethodtest`	1.2	71%	87%
`com.android.keepass`	1.1	7%	8%
`org.dnaq.dialer2`	1.0	39%	39%
`hu.vsza.adsdroid`	1.0	24%	24%
`com.example.anycut`	0.9	71%	71%
`org.scoutant.blokish`	0.9	45%	46%
`org.beide.bomber`	0.8	89%	88%
`com.beust.android.translate`	0.7	48%	48%
`com.addi`	0.6	18%	18%
`org.wordpress.android`	0.5	5%	5%
`com.example.amazed`	0.3	82%	81%
`net.everythingandroid.timer`	0.2	65%	65%
`com.google.android.opengles.spritetext`	0.1	59%	59%
`aarddict.android`	0.0	14%	14%
`com.angrydoughnuts.android.alarmclock`	0.0	6%	6%
`com.everysoft.autoanswer`	0.0	9%	9%
`hiof.enigma.android.soundboard`	0.0	100%	100%

`com.tum.yahtzee`: This is a dice game with fairly complicated logic and several buttons, each activating different scenarios over time.

`org.waxworlds.edam.importcontacts`: This app imports contacts from the SD card. There are multiple steps to reach to the final activity, and each contains multiple options that change the course of actions that the app finally takes.

`hu.vsza.adsdroid`: The only functionality of this app is to search for and list the data-sheets of electronic items. The search activity contains one drop-down list for search criteria, and a search button.

`org.wordpress.android`: This app is for management of WordPress websites. At the startup, it either requires a login or opens a web container, which does not affect the line coverage.

is not much code left to be explored. As another example, consider apps that require signing in to an account to access their functionality. Unless it is explicitly supported by the tools (which is not in this study), not much can be explored within the app.

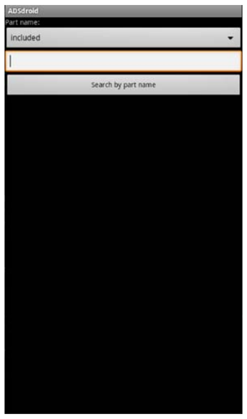
We hypothesize that Monkey++ outperforms Google Monkey in apps with high CC. In order to test this, we define CC as the uncertainty in coverage when randomly interacting with an app. That is, if random interactions with an app always result in a similar trace of coverage, it means that the available parts of the app are trivial to reach and will always be executed, and therefore, not much is offered by the app to be explored. To compute uncertainty (and hence CC) for an app, we use the concept of entropy.

The entropy of a random variable is a measure of the uncertainty of the values that this variable can get. For instance, if a random variable only gets one value (i.e., it is not random), the entropy would be zero. On the other hand, a random variable that samples its values from a uniform distribution has a large entropy, because it is more difficult to predict its exact value. The formula for calculating entropy H of a discrete random variable X is as follows:

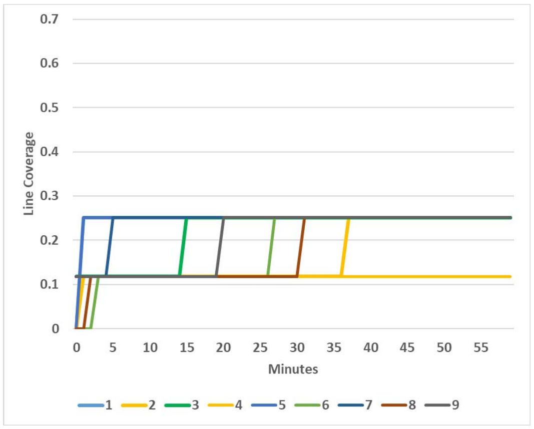
$$H(X) = -\sum_{i=1}^{n} p(x_i) \log_2(p(x_i))$$

where x_i represents the values that X can get, and $p(x_i)$ is the probability distribution for X. To calculate CC of an app using entropy, we take all line coverage information for that app in all timesteps of all experiments involving Google Monkey (as a random interaction tool), and calculate the entropy of the distribution of these coverage values using the above formula. The coverage values for two apps with low and high CC are shown in Figure 5.

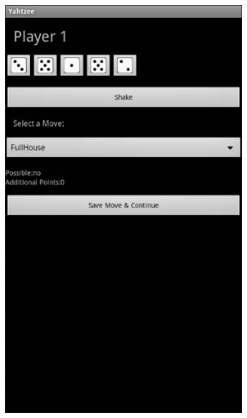
Table I shows the CC value for each app, and discusses some examples of apps with high and low CC, including the examples in Figure 5. As one can notice, most of the apps in which Monkey++ achieves better coverage have higher CC.



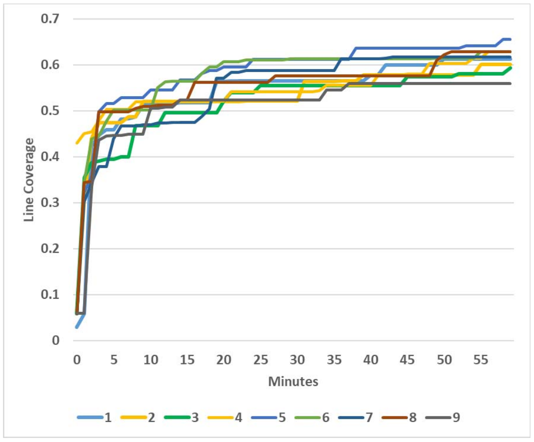
(a)



(b)



(c)



(d)

Fig. 5: The results of exploring two apps *randomly* in 9 independent runs: (a) An example of an app with low CC (`hu.vsza.adsdroid`). (b) We obtain only 3 distinct coverage values for the entire 60 minutes of randomly testing the `adsdroid` app across all 9 agents. This means the portion of the app that is accessible to be explored is very limited. (c) An example of an app with high CC (`com.tum.yahtzee`). (d) Here, the coverage values that we obtain by randomly exploring the `yahtzee` app span a much more *uncertain* space than the `adsdroid` app, which means more is offered by the app to be explored and therefore it is more meaningful to compare the testing tools on this app.

To further evaluate the ability of Monkey++ in crawling complex apps with high CC, we analyzed the progressive coverage of the top 10 apps with the highest CC. Figure 6 shows that Monkey++ achieves better results compared to Google Monkey, and does so faster. This superiority is statistically significant in all timesteps, as calculated by a one-tail Kolmogorov–Smirnov (KS) test (p-value < 0.05).[8]

The improvement over Google Monkey is valuable, since it is currently the most widely used testing tool that does not require the AUT to implement any specific API. For instance, most of the mainstream white-box testing tools fail on non-native applications, because these applications are essentially web content wrapped in an android web viewer, and lack standard UI elements that white-box tools depend on. In these scenarios, practitioners are bound to use random testing tools such as Google Monkey. Monkey++ provides a more intelligent alternative in these situations that, as the results suggest, provide better coverage faster.

[8] To calculate the error bars in Figure 6 and the p-value for KS-test, first for each app, the mean performance of Google Monkey on that app is subtracted from the performance of both Google Monkey and Monkey++, and then the error bars and the significance are computed with regards to this value across all apps.

RQ2. Cross-Platform Ability

Since Deep GUI is completely blind with regards to the app's implementation or the platform it runs on, we hypothesize it is applicable not only in Android but in other platforms such as web or iOS. Moreover, we claim that since UI design across different platforms is very similar (e.g. buttons are very similar in Android and web), we can take a model trained on one platform and use it in other platforms. This is particularly useful when developers want to test different implementations of the same app in different platforms.

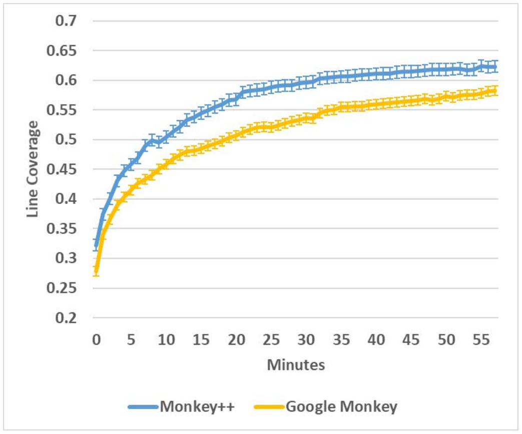


Fig. 6: The progressive line coverage of Monkey++ and Google Monkey on the top 10 Android apps with the highest CC. Error bars indicate standard error of the mean.

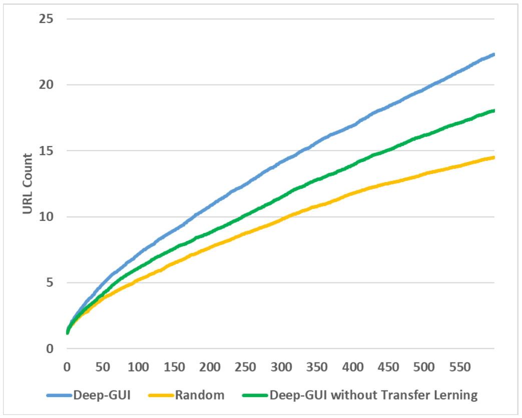


Fig. 7: The progressive performance of Deep GUI and random agent in web crawling. The difference between the three tools is statistically significant in all timesteps, as calculated by one tail KS-tests between all pairs (similar to the procedure described in footnote 8).

To test whether our approach is truly cross-platform, we implemented an interface to use Deep GUI for interacting with Mozilla Firefox browser[9] using Selenium web driver [34], and compared it against a random agent[10]. Note that we did not re-train our model, and used the exact same hyper-parameters and weights we used for the experiments in RQ1, which are learned from Android apps.

For the web experiments, we used the top 15 websites in the United States [26] as our test set, and ran each tool on each website 20 times, each time for 600 steps. To measure the performance, we counted the number of distinct URLs visited in each website, and averaged this value for each tool.

[9] We used Responsive Design Mode in Mozilla Firefox with the resolution of 480×640.

[10] The random agent uses the same bias for action types that is explained in footnote 5 of Section II.

TABLE II: The performance of Deep GUI and random agent on each web site

Website	Deep GUI	Random
google.com	17.4	12.9
youtube.com	94.3	12.1
amazon.com	13.2	15.2
yahoo.com	15.4	21.8
facebook.com	3.2	7.1
reddit.com	5.3	5.1
zoom.us	4.6	6.9
wikipedia.org	41.1	40.6
myshopify.com	3.6	6.0
ebay.com	13.4	11.4
netflix.com	5.1	4.8
bing.com	32.5	25.5
office.com	16.9	15
live.com	2.7	2.5
twitch.tv	65.6	30.1
average	22.2	14.4

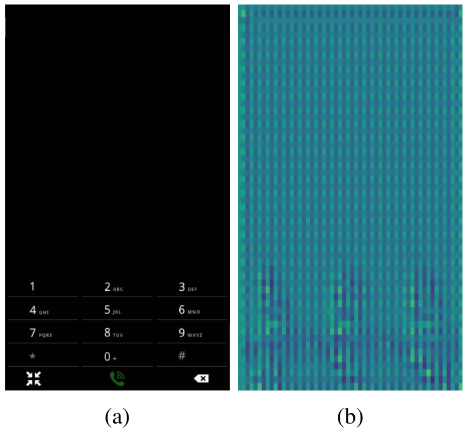


Fig. 8: A screenshot and its corresponding heatmap generated by the model before training.

Figure 7 and table II show that our model outperforms random agent, and confirms that our model has learned the rules of UI design, which is indeed independent of the platform.

The results of the web experiment demonstrate the power of a black-box technique capable of understanding the dynamics of GUI-based applications without relying on any sort of platform-dependent information. Such techniques infer generalized rules about GUI-based environments instead of relying on specific APIs or implementation-choices in the construction of an application, and hence enable users to apply the tools on different applications and on different platforms without being constrained by the compatibility issues.

RQ3. Transfer Learning Effect

As described, we used transfer learning to make the training process more data-efficient, i.e. we crawl fewer data and train faster. To study if using transfer learning was actually helpful, we repeated the web experiments, with the only difference that instead of using the model trained with transfer learning, we trained another model with random initial weights. Figure 7 shows that without transfer learning, the model's performance significantly decreases.

To gain an intuitive understanding of the reason behind this, consider Figure 8b. This figure shows the initial output of the neural network for the screen of Figure 8a before training, when initialized with the ImageNet weights. As one can see, even without training, the buttons stand out from the background in the heatmap, which gives the model a significant head-start compared to the randomly initialized model, and makes it possible for us to train it with a small amount of data.

IV. Related Work

Many different input generation techniques with different paradigms have been proposed in the past decade. Several techniques [35], [36] rely on a model of the GUI, usually constructed dynamically and non-systematically, leading to unexplored program states. Sapienz [15], EvoDroid [16], and time-travel testing [37] employ an evolutionary algorithm. ACTEve [38], and Collider [39] utilize symbolic execution. AppFlow [40] leverages machine learning to automatically recognize common screens and widgets and generate tests accordingly. Dynodroid [23] and Monkey [22] generate test inputs using random input values. Another group of techniques focus on testing for specific defects [20], [41], [42].

These approaches can be classified into two broad categories: *context blind* and *context aware*. The tools in the former category process information in each action independent of other actions. That is, when choosing a new action, they do not consider the previous actions performed, and do not plan for future actions. Tools such as Google Monkey [22] and DynoDroid [23] are in this category. These tools are fast and require very simple pre-processing, but may miss entire activities or functionalities, as this requires maintaining a model of the app and visited states. Tools in the latter category incorporate various sources of information to construct a model of an app, which is then used to plan for context-aware input generation. Most of the existing input generation tools are in this category. For instance, Sapienz [15] uses a genetic algorithm to learn a generic model of app, representing how certain sequences of actions can be more effective than others. Tools that use different types of static analysis of the source code or GUI to model the information flow globally also belong to this category.

Not many tools have explored black-box and/or cross-platform options for gathering information to be used for input generation, either with a context-aware or a context-blind approach. Google Monkey is the only widely used tool in Android that does not depend on any app-specific information. However, it follows the simplest form of testing, i.e., random testing. Humanoid [43] is an effort towards becoming less platform-dependent, while also generating more intelligent inputs. However, it is still largely dependent on the UI transition graph of AUT and the GUI tree extracted from the operating system. Moreover, since it depends on an existing dataset for Android, it would not be easy to train it for a new platform. The study of White et al. [44] is the most similar to our work. They study the effect of machine-learning-powered processing of screenshots in generating inputs with random strategy. However, because they generate artificial apps for training their model, their data collection method is limited in expressing the variety of screens that the tool might encounter. Furthermore, their approach is platform dependent.

Deep GUI uses deep learning to improve context-blind input generation, while also limiting the processed information to be black-box and platform independent. This allows it to be as versatile as Google Monkey in the Android platform, while being more effective by intelligently generating the inputs for crawling of apps.

V. Discussion and Future Work

Deep GUI is the first attempt towards making a fully black-box and cross-platform test input generation tool. However, there are multiple areas in which this tool can be improved. The first limitation of the approach described here is the time-consuming nature of its data collection process, which limits the number of collected data points and may compromise the dataset's expressiveness. By using transfer learning, we managed to mitigate this limitation to some degree. In addition, the complex set of hyperparameters (such as hybrid readout probabilities) and the time-consuming nature of validating the model on apps make it difficult to fine-tune all the hyperparameters systematically, which is required for optimizing the performance to its maximum potential.

Deep GUI limits itself to context-blind information processing, in that it does not consider the previous interactions with AUT when generating new actions. However, it uses a paradigm that can easily be extended to take context into account as well. We believe this paradigm should be explored more in the future of the field of automated input generation.

Take our definition of the problem. If we call s_t the state of the environment, a_t the action performed on the environment in that state, r_t the reward that the environment provides in response to that action, and $Q(s_t, a_t)$ the predictions of the model about the long-term reward that the environment provides when performing a_t in s_t (also known as the quality matrix), then this work can essentially be viewed to propose a single-step deep Q-Learning [45] solution to the problem of test input generation. Looking at the problem this way enables researchers in the area of automatic input generation to benefit from the rich and active research in the Q-Learning and reinforcement learning (RL) community, and explore different directions in the future such as the following:

- **Multi-Step Cross-Platform Input Generation.** Deep GUI uses Q-Learning in a context-blind and single-step manner. However, by redefining s_t to include more context (such as previous screenshots, as tried in Humanoid [43]) and expanding the definition of r_t to express a multi-step sense of reward, one can use the same idea to utilize the full power of Q-Learning to train models that not only limit their actions to only the valid ones (as this tool does), but also plan ahead and perform complex and meaningful sequence of actions.

- **Smarter Processing of Information.** Even if a tool does not want to limit itself to only platform-independent information, it can still benefit from using a Q-Learning solution. For instance, one can define s_t to include the GUI tree or the memory content to provide the model with more information, but also use Q-Learning to process this information more intelligently.
- **Regression Testing and Test Transfer.** While this work presents a trained model that targets all apps, it is not limited to this. Developers can take a Q-Learning model such as the one described in this work, collect data from the app (or a family of related apps) they are developing, and train the model extensively so that it learns what actions are valid, what sequences of actions are more probable to test an important functionality, etc. This way, when new updates of the app are available, or when the app becomes available in new platforms, developers can quickly test for any fault in that update without having to rewrite the tests.

Acknowledgment

This work was supported in part by award numbers 2106306 and 1823262 from the National Science Foundation and a Google Cloud Platform gift. We would like to thank the anonymous reviewers of this paper for their detailed feedback, which helped us improve the work.

References

[1] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, (New York, NY, USA), p. 641–660, Association for Computing Machinery, 2013.

[2] R. Bhoraskar, S. Han, J. Jeon, T. Azim, S. Chen, J. Jung, S. Nath, R. Wang, and D. Wetherall, "Brahmastra: Driving apps to test the security of third-party components," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, (USA), p. 1021–1036, USENIX Association, 2014.

[3] M. Linares-Vásquez, M. White, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk, "Mining android app usages for generating actionable gui-based execution scenarios," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, p. 111–122, IEEE Press, 2015.

[4] S. Yang, H. Wu, H. Zhang, Y. Wang, C. Swaminathan, D. Yan, and A. Rountev, "Static window transition graphs for android," *Automated Software Engg.*, vol. 25, p. 833–873, Dec. 2018.

[5] D. Amalfitano, A. R. Fasolino, and P. Tramontana, "A gui crawling-based technique for android mobile application testing," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pp. 252–261, 2011.

[6] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pp. 258–261, 2012.

[7] Y. Baek and D. Bae, "Automated model-based android gui testing using multi-level gui comparison criteria," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 238–249, 2016.

[8] N. P. Borges, M. Gómez, and A. Zeller, "Guiding app testing with mined interaction models," in *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, MOBILESoft '18, (New York, NY, USA), p. 133–143, Association for Computing Machinery, 2018.

[9] W. Choi, G. Necula, and K. Sen, "Guided gui testing of android apps with minimal restart and approximate learning," *SIGPLAN Not.*, vol. 48, p. 623–640, Oct. 2013.

[10] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, (New York, NY, USA), p. 204–217, Association for Computing Machinery, 2014.

[11] K. Jamrozik and A. Zeller, "Droidmate: A robust and extensible test generator for android," in *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pp. 293–294, 2016.

[12] L. Mariani, M. Pezze, O. Riganelli, and M. Santoro, "Autoblacktest: Automatic black-box testing of interactive applications," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pp. 81–90, 2012.

[13] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, (New York, NY, USA), p. 245–256, Association for Computing Machinery, 2017.

[14] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen, "Droidbot: a lightweight ui-guided test input generator for android," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pp. 23–26, 2017.

[15] 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, (New York, NY, USA), p. 94–105, Association for Computing Machinery, 2016.

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

[17] J. Garcia, M. Hammad, N. Ghorbani, and S. Malek, "Automatic generation of inter-component communication exploits for android applications," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, (New York, NY, USA), p. 661–671, Association for Computing Machinery, 2017.

[18] C. Cao, N. Gao, P. Liu, and J. Xiang, "Towards analyzing the input validation vulnerabilities associated with android system services," in *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, (New York, NY, USA), p. 361–370, Association for Computing Machinery, 2015.

[19] Y. Liu, C. Xu, S. Cheung, and J. Lü, "Greendroid: Automated diagnosis of energy inefficiency for smartphone applications," *IEEE Transactions on Software Engineering*, vol. 40, no. 9, pp. 911–940, 2014.

[20] R. Jabbarvand, J.-W. Lin, and S. Malek, "Search-based energy testing of android," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 1119–1130, 2019.

[21] A. Alshayban, I. Ahmed, and S. Malek, "Accessibility issues in android apps: State of affairs, sentiments, and ways forward," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, (New York, NY, USA), p. 1323–1334, Association for Computing Machinery, 2020.

[22] "Ui/application exerciser monkey." https://developer.android.com/studio/test/monkey, 2020.

[23] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, (New York, NY, USA), p. 224–234, Association for Computing Machinery, 2013.

[24] K. Mao, M. Harman, and Y. Jia, "Crowd intelligence enhances automated mobile testing," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, p. 16–26, IEEE Press, 2017.

[25] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet? (e)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 429–440, 2015.

[26] A. Internet, "Top sites in united states," 2020.

[27] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015* (N. Navab, J. Hornegger, W. M. Wells, and A. F. Frangi, eds.), (Cham), pp. 234–241, Springer International Publishing, 2015.

[28] 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 *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, (USA), p. 265–283, USENIX Association, 2016.

[29] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 10, pp. 1345–1359, 2010.

[30] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 4510–4520, 2018.

[31] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[32] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[33] V. Roubtsov, "Emma: a free java code coverage tool," 2006.

[34] Selenium, "The selenium browser automation project." https://www.selenium.dev/.

[35] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 245–256, 2017.

[36] 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 *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 269–280, IEEE, 2019.

[37] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury, "Time-travel testing of android apps," in *Proceedings of the 42nd International Conference on Software Engineering (ICSE'20)*, pp. 1–12, 2020.

[38] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pp. 1–11, 2012.

[39] C. S. Jensen, M. R. Prasad, and A. Møller, "Automated testing with targeted event sequence generation," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pp. 67–77, 2013.

[40] G. Hu, L. Zhu, and J. Yang, "Appflow: using machine learning to synthesize robust, reusable ui tests," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 269–282, 2018.

[41] R. Hay, O. Tripp, and M. Pistoia, "Dynamic detection of inter-application communication vulnerabilities in android," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pp. 118–128, 2015.

[42] L. L. Zhang, C.-J. M. Liang, Y. Liu, and E. Chen, "Systematically testing background services of mobile apps," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 4–15, IEEE, 2017.

[43] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Humanoid: A deep learning-based approach to automated black-box android app testing," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1070–1073, 2019.

[44] T. D. White, G. Fraser, and G. J. Brown, "Improving random gui testing with image-based widget detection," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, (New York, NY, USA), p. 307–317, Association for Computing Machinery, 2019.

[45] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *ArXiv*, vol. abs/1312.5602, 2013.