Hawkeye: Change-targeted Testing for Android Apps based on Deep Reinforcement Learning

Chao Peng
pengchao.x@bytedance.com
ByteDance
Beijing, China

Zhengwei Lv
lvzhengwei.m@bytedance.com
ByteDance
Beijing, China

Jiarong Fu
fujiarong@bytedance.com
ByteDance
Beijing, China

Jiayuan Liang
liangjiayuan.522@bytedance.com
ByteDance
Beijing, China

Zhao Zhang
zhangzhao.a@bytedance.com
ByteDance
Beijing, China

Ajitha Rajan
arajan@ed.ac.uk
University of Edinburgh
Edinburgh, United Kingdom

Ping Yang
yangping.cser@bytedance.com
ByteDance
Beijing, China

ABSTRACT

Android Apps are frequently updated to keep up with changing user, hardware, and business demands. Ensuring the correctness of App updates through extensive testing is crucial to avoid potential bugs reaching the end user. Existing Android testing tools generate GUI events that focus on improving the test coverage of the entire App rather than prioritising updates and impacted elements. Recent research has proposed change-focused testing but relies on random exploration to exercise change-impacted GUI elements that is ineffective and slow for large complex Apps with a huge input exploration space. At ByteDance, our established model-based GUI testing tool, Fastbot2, has been in successful deployment for nearly three years. Fastbot2 leverages event-activity transition models derived from past explorations to achieve enhanced test coverage efficiently. A pivotal insight we gained is that the knowledge of event-activity transitions is equally valuable in effectively targeting changes introduced by updates. This insight propelled our proposal for directed testing of updates with Hawkeye. Hawkeye excels in prioritizing GUI actions associated with code changes through deep reinforcement learning from historical exploration data.

In our empirical evaluation, we rigorously compared Hawkeye with state-of-the-art tools like Fastbot2 and ARES on 10 popular open-source Apps and a commercial App. The results showcased that Hawkeye consistently outperforms Fastbot2 and ARES in generating GUI event sequences that effectively target changed functions, both in open-source and commercial App contexts.

In real-world industrial deployment, Hawkeye is seamlessly integrated into our development pipeline, performing smoke testing for merge requests in a complex commercial App. The positive feedback received from our App development teams further affirmed Hawkeye's ability in testing App updates effectively.

KEYWORDS

Software Testing, Deep Reinforcement Learning, Android, Graphical User Interface

ACM Reference Format:
Chao Peng, Zhengwei Lv, Jiarong Fu, Jiayuan Liang, Zhao Zhang, Ajitha Rajan, and Ping Yang. 2024. Hawkeye: Change-targeted Testing for Android Apps based on Deep Reinforcement Learning. In *46th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3639477.3639749

1 INTRODUCTION

The rapid growth of mobile technology has led to an unprecedented increase in the development and usage of mobile Apps, particularly in the Android ecosystem. These Apps are frequently updated (typically weekly) to keep up with changing user, hardware and business demands. To ensure security and correctness, updates in Apps need to be tested thoroughly to ensure changes and existing functionality work as expected.

The conventional approach to testing Android Apps involves human interaction, which can be time-consuming and error-prone when identifying unexpected behaviors. Automated mobile App testing has been extensively explored in the literature [2, 3, 5, 6, 21, 26, 30–33], focusing primarily on testing a single version of a mobile App and often excluding App updates. Regression test selection techniques have been developed to choose a subset of tests from an existing test suite that exercises updates [5, 9, 10, 14, 15, 28]. QADdroid [28] is a notable tool that considers changes and their impact at the GUI level when selecting regression tests. However, regression test selection techniques, including QADroid, focus on existing tests and do not generate new tests to cover the changes.

To the best of our knowledge, CAT [25] and ATUA [22] are the only tools in the literature that generate GUI events to exercise

ICSE-SEIP '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0501-4/24/04…$15.00
https://doi.org/10.1145/3639477.3639749

changes. However, CAT relies on random exploration before reaching the change-impacted GUI element. If the App under test is complex, the random execution can take hours, and as complexity increases, it may take an impractical amount of time before the target GUI element is reached, diminishing its effectiveness in practice. On the other hand, ATUA relies on static analysis to construct activity transition graphs and perform change impact analysis. However, the core analysis engine, Gator [35], presents scalability challenges with our commercial Apps. Specifically, it fails when analyzing Apps exceeding 20MB in size, making it ineffective for our context.

Fastbot2 [19], an automated model-based GUI testing technique, has been actively used at ByteDance for three years. It leverages model-based testing (MBT) to store and apply prior knowledge, effectively steering GUI testing. A pivotal element of the model reuse approach in Fastbot2 is a probabilistic model designed to remember event-activity transitions from earlier testing iterations. This stored knowledge serves as a guide during current testing, ensuring efficient coverage of critical App functionalities by selecting events accordingly. Going beyond one-step guidance, Fastbot2 enhances this model with a deep reinforcement learning algorithm. This augmentation empowers the model to offer multi-step guidance, enabling traversal into deeper activities that necessitate a sequence of events for effective testing coverage.

While Fastbot2 stands out as the only model-reuse approach in the literature, it does not specifically address changes introduced in new versions nor prioritize GUI events that are more likely to trigger functions impacted by these changes. In this paper, we introduce Hawkeye, a novel approach that concentrates on generating GUI events with the goal of executing targeted changes. This is achieved through a reinforcement learning-based strategy. Hawkeye is triggered when a merge request is submitted. It first analyses the codebase and the code commit to identify changed functions and initiates a GUI exploration task to perform change-targeted testing. To quickly locate and exercise changed functions, a reinforcement learning model is loaded from the historical exploration data to infer GUI events that are most likely to execute these functions. The task is terminated once all changed functions are covered or the execution times out. Additionally, Hawkeye uses a client-server implementation with the ability to test the same App on multiple devices simultaneously while sharing the deep reinforcement learning agent.

We investigate the effectiveness and practicality of Hawkeye in testing App updates using a diverse dataset of 10 open-source Android Apps from the F-Droid App market. Additionally, we include a widely-used commercial enterprise collaboration App, Feishu, developed by ByteDance, to assess practicality of the technique on a large-scale complex App. Finally, we compare Hawkeye's performance against two reinforcement learning-based GUI testing tools for Android, namely *ARES* [27] and *Fastbot2* [19].

Our experimental findings strongly indicate that Hawkeye outperforms both *Fastbot2* and *ARES* by more effectively and frequently exercising modified functionalities with fewer GUI events. This highlights the efficiency and reliability of Hawkeye in the context of testing App updates. Furthermore, when specifically evaluating Hawkeye's performance on the commercial App, Feishu, our results demonstrate its efficiency in testing changes within this complex application – 85% of the changed functions can be covered within the initial 5 minutes of the 180-minute testing run. This signifies Hawkeye's potential in efficiently testing updates within real-world Apps. This finding is in line with the experience of our product teams who found after a one year review that Hawkeye is able to increase change-related statement coverage by 11.3% and increase the number of uncovered crashes three times.

In summary, the main contributions in this paper are:

(1) A novel deep reinforcement learning-based GUI testing tool that prioritises GUI events related to changed functions.
(2) An optimised client-server implementation that allows testing on multiple devices while sharing the deep reinforcement learning agent.
(3) Empirical evaluation against state-of-the-art reinforcement learning tools using 10 open-source and 1 commercial App.
(4) Implications and lesson learned from our development and deployment experience.

2 BACKGROUND

Before we present our approach in detail, we briefly introduce basic concepts in Android App development and testing. We also discuss reinforcement learning and how it can be used for test generation.

2.1 Android Apps

Android Apps are typically developed using Java or Kotlin, which are compiled and converted to Dalvik bytecode. To enhance performance, native code can also be incorporated. The Dalvik bytecode, native code (if available), and any data or resource files are packed into an Android package (APK). The APK file is the sole requirement for installing the App on Android devices. In order to construct the APK file, an Android project relies on the following components:

(i) source code files containing App's classes and functions,
(ii) layout-XML files outlining the GUI layout for all activities, and
(iii) the Android manifest, which is located in the App's root folder as AndroidManifest.xml and provides crucial details about the App, such as the package name (used to find the source code), component lists, required user permissions, utilized hardware and software features, and necessary API libraries.

2.2 Android GUI and Testing

The window displayed in the screen is referred to as an `Activity`, encompassing a range of `GUI elements` (also known as `Views` or `Widgets`) such as buttons and text fields. By incorporating suitable callbacks for each life-cycle stage (i.e., created, paused, resumed, and destroyed), developers can manage the behavior of individual Activities. These Activities must be initially specified in the *AndroidManifest.xml* file and are executed as Java classes within the source code directory.

`GUI elements` serve as the fundamental building blocks for user interaction, such as textboxes, buttons, and containers for other `GUI elements`. `GUI elements` play a crucial role in event handling, which may include button clicks, text editing, touch events, and more. To react to a specific event type, it needs to register a suitable event listener and implement the associated callback method (invoked by the Android Framework when the `GUI element` is activated through user interaction). For instance, to enable a button to respond to a click event, it must register the

`View.onClickListener` event listener and implement the relevant `onClick()` callback method. Upon detecting a button click event, the Android framework will call the `onClick()` method for that specific `GUI element`.

An `event sequence` refers to an organized series of input events. In this paper, the term `state` denotes the GUI state, which encompasses the current screen's GUI information and all its elements. Since Android Apps are event-driven, inputs typically take the form of events. Manually writing or recording input events can be laborious and time-intensive [29]. Consequently, automated test input generation for Android Apps is an active research area. The following section discusses existing Android testing techniques.

2.3 Reinforcement Learning

Reinforcement Learning (RL) is a sub-domain of machine learning that focuses on training agents to make intelligent decisions by interacting with an environment. In RL, an agent learns to select actions within a given context to maximize a cumulative reward signal. The underlying idea is that the agent learns an optimal policy through trial-and-error, wherein it continuously updates its knowledge based on the observations and feedback obtained from the environment. The agent's learning process is driven by balancing the exploration of the environment and the exploitation of the acquired knowledge [16]. Reinforcement learning has exhibited impressive results in a wide variety of domains, ranging from robotics and game-playing to recommendation systems and self-driving cars. In the context of automated Android App testing, reinforcement learning can be leveraged as a means to efficiently explore the state-space by formulating testing as a decision-making problem.

3 RELATED WORK

In this Section, we provide an overview of current research on Android GUI test generation, which is divided into random, model-driven and learning-based testing approaches. Additionally, we discuss related studies that focus on choosing regression tests or generating new tests based on App updates.

3.1 Random Android GUI Testing

Android Monkey [8] is a well-known random testing tool that analyses the GUI and arbitrarily selects events to be executed in the current state until the number of executed events surpasses the user-defined limit. DynoDroid [20] employs heuristics for input event selection instead of complete randomness. However, DynoDroid has not been updated for several years and only supports Android version 2.3.5 (with Android 13 being the latest version). Wetzlmaier et al. [34] enhance existing test inputs by incorporating random test inputs, providing users with more control than Monkey. None of the current random testing tools concentrate on App updates.

3.2 Model-based Android Testing

DroidBot [17] and DroidMate [3, 13] emphasise generating test inputs based on GUI models. DroidMate directs test input generation in real-time using the GUI model. DroidBot consults the GUI model of the target App, calculates, and performs possible events within this model. DroidBot also offers a user-friendly interface for App exploration.

In contrast to static GUI model-based test generation, Ape [12] dynamically optimises the GUI model by taking advantage of runtime information during testing. Ape employs a decision tree-based representation while exploring the App and continuously refines the GUI model, aiming to maintain an optimal balance between model size and model accuracy.

3.3 Machine Learning-based Android Testing

Machine learning techniques have been extensively applied in testing Android applications. Some methods [4, 18] use supervised learning to generate test inputs that achieve high coverage. Borges et al. [4] suggest a straightforward yet effective approach that can guide test generation towards UI elements with the highest likelihood of being reactive. Humanoid [18] utilizes a deep neural network to learn about users' interactions with the Apps being tested, enabling it to prioritise user-preferred inputs for new UIs.

In addition, many researchers have tried to adopt reinforcement learning in automatic test input generation. Q-testing [24] is a Q-learning based approach for automated testing. Q-testing leverages Q-table as a lightweight model while exploring unfamiliar functionalities with a curiosity-driven strategy. ARES [27] and ATAC [11] are both testing tools that utilise deep reinforcement learning algorithms. ARES employs algorithms such as DDPG, SAC and TD3 as agents to learn both the state similarity and action-value functions, while ATAC leverages the Advantage Actor-Critic algorithm to generate test cases for Android GUI testing. DeepGUI [7] is a Deep Q-Network-based testing tool. It can generate diverse test cases by exploring the App using trial-and-error, with the aim of achieving code coverage and revealing failures and crashes. All three tools — ARES, ATAC and DeepGUI — do not consider changes in Android Apps which is the focus of our approach. We compare performance against ARES in Section 6. We were unable to evaluate against ATAC as it is not publicly available. We attempted to use DeepGUI in our experiments but the test generation was extremely time consuming making it impractical to use.

3.4 Regression Test Selection and Generation

Numerous studies have investigated the selection of regression tests based on App updates and their effects. However, Hawkeye's focus is different - it aims at *generating inputs* to exercise changed functions unlike existing regression test selection contributions that focus on test selection.

Redroid [9, 10] and ReTestDroid [14] are regression test selection techniques that compare Java source files from the original and updated App versions to identify changes and calculate change impact at the source code level. The tools select regression tests that exercise change-impacted code. ReTestDroid handles more Java features than Redroid, such as fragments, native code, and asynchronous tasks. Both tools perform change impact analysis at the source code level, not considering GUI elements, and are used for test selection rather than generation.

QADroid [28] and ATOM [15] also carry out test selection for regression versions of Apps. QADroid analyses the impact of App updates on code and GUI elements. QADroid constructs call graphs

based on FlowDroid [1] and links events to function calls using event-function bindings defined in the source code. QADroid does not support change impact analysis for dynamic GUI elements, as it does not support Java reflection. ATOM [15] creates an event-flow graph for each App version, with nodes representing activities and edges representing events that trigger activity transitions. It then computes a delta graph using event-flow graphs of the updated and original App versions. Only events present in the delta graph are chosen for regression test selection.

CAT [25] conducts change impact analysis at both the source code and GUI levels, and prioritises change-related GUI events. However, CAT does not provide a way to localise these GUI events and relies on purely random GUI exploration before these events are available on the screen. More importantly, the static analysis is not applicable to large Apps. CAT is not applicable to large industrial Apps that we used in our initial experiment on a devbox with 128 GB memory. As a result, we do not use this tool in our evaluation.

ATUA [22] is a state-of-the-art update-driven App testing tool that achieves high coverage of the updated code with a minimal number of test inputs. It employs a model-based approach that combines static and dynamic program analysis to select test inputs that exercise the updated methods. Similar to CAT, the static analysis phase employed by ATUA to perform change impact analysis and widget transition graph construction fail on Apps larger than 20 MB and we have to exclude this tool in our evaluation.

4 OUR APPROACH

In this section, we present Hawkeye based on deep reinforcement learning model to guide change-targeted GUI exploration. The workflow of Hawkeye is illustrated in Figure 1. The input to Hawkeye is the list of changed functions and the instrumented App with real-time function coverage monitoring and the output is GUI event sequences to be executed.

Hawkeye includes two major phases for each testing run. The first phase (with Prefix A in Figure 1) is setup *before* testing:

A1. The list of changed functions (target functions) is generated by analyzing the merge request from the codebase. The user can also provide the list of target functions manually in the tool configuration file.
A2. Hawkeye instruments the App on the source code level to enable the compiled APK file to record and report real-time function coverage. The coverage information is used to train the reinforcement learning model and determine whether any of the target functions are covered.
A3. The instrumented App is installed on multiple devices to perform GUI exploration. During testing, the device continuously sends GUI information to Hawkeye and executes GUI events that it selected.
A4. The list of target functions is sent to the decision engine to infer GUI events to be executed.

The second phase (with prefix B in Figure 1) performs guided GUI exploration.

B1. Hawkeye captures the current GUI screen from the App under test (AUT). Hawkeye interacts with the device using the Android Debug Bridge, allowing it to query GUI information and collect executable GUI events present on the current screen.
B2. Hawkeye selects a GUI event with the highest likelihood of exercising one of the target functions.
B3. The selected GUI event is executed on the device.
B4. The GUI state change from event execution in B3 is sent to the decision engine to update the RL model.

Steps B1 to B4 are iteratively executed in a loop until either the predetermined threshold number of GUI events is reached or the designated time budget is exhausted. In the rest of this section, we discuss each step in detail.

4.1 Input Preprocessing

Mapping between GUI actions and source code functions is essential to train the deep reinforcement learning model. We obtain this mapping by instrumenting the Apps to monitor functions executed. Different from the transition graph of events and activities that can be recorded during testing (Android provides a tool to collect events and activities during testing), Hawkeye requires instrumentation to recorded which functions are exercised after an event execution. We build the instrumentation tool as a Gradle plugin based on ASM[1], a Java bytecode manipulation and analysis framework. Gradle is the default build automation tool used by the official Android development environment, Android Studio. As a gradle plugin, code instrumentation can be integrated into existing Android projects seamlessly by only adding the name of the tool to the plugin list of the project configuration file.

Listing 1: Example Instrumented Methods

```
1   void foo() {
2       // function ID = 1;
3       Hawkeye.logMethod(1);
4       ...
5   }
6
7   void bar() {
8       // function ID = 2;
9       Hawkeye.logMethod(2);
10      ...
11  }
```

As shown in Listing 1, the code instrumentation inserts a function call to `logMethod` at the beginning of all methods. The implementation of this function sends a socket message to Hawkeye with the method ID. After code instrumentation, the mapping between function signature and function ID is stored in the database for Hawkeye to query. As a result, during testing, after each GUI event is exercised, Hawkeye is aware of the functions that get triggered.

4.2 RL-based Guided GUI Exploration

We train a deep RL model to select the GUI event that maximises the likelihood of reaching the target changed functions. The overall architecture of the deep RL model is shown in Figure 2. It consists of two modules, *DRL Service* (the server) and *GUI Interaction Layer* (the client) that are described below.

DRL Service. The DRL service is responsible for model training and Q-value (measure of overall expected reward for a given state and action) calculation. It comprises the following components:

[1] https://asm.ow2.io/

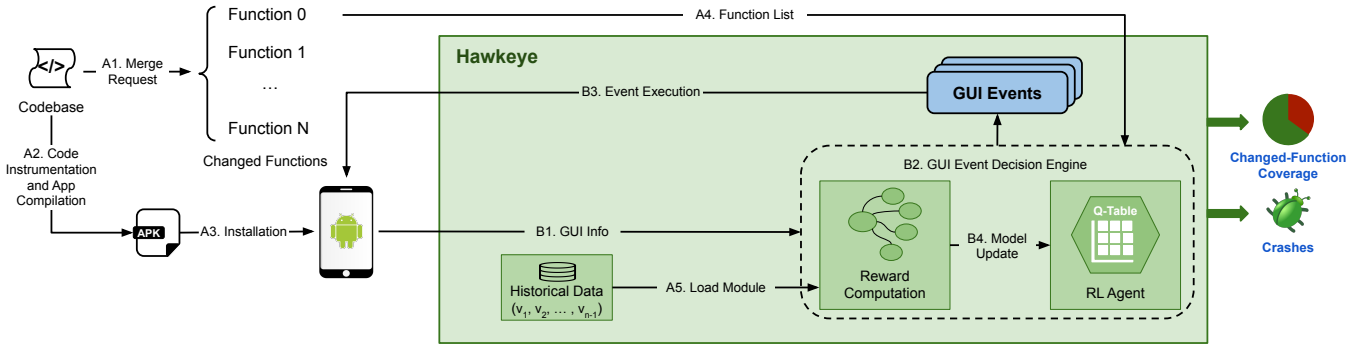

Figure 1: Hawkeye Workflow

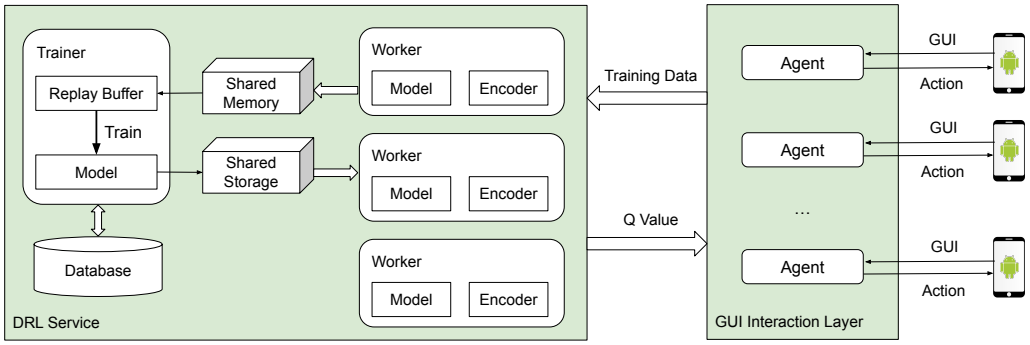

Figure 2: Deep Reinforcement Learning Workflow

- **Shared Memory** transfers the training data to the replay buffer of the model trainer.
- **Shared Storage** stores shared variables across processes.
- **Trainer** trains the deep reinforcement learning model. It stores the training data using the replay buffer and continuously samples to train the model. After each training, it saves model parameters to the `Shared Storage` and database.
- **Worker** component, depicted within the DRL Service module in Figure 2, comprises of two crucial elements: the `Model` and the `Encoder`. The `Model` encapsulates the most current trained model (from the `Trainer`), retrieved at regular intervals from the `Shared Storage`. The `Encoder` is responsible for translating raw GUI information into vectors, a format that can be effectively processed and understood by the model. Additionally, the `Worker` component provides two key interfaces to the GUI Interaction Layer, `Add Training Data` and `Get Q-value`. When `Add training data` is invoked, the `Worker` stores encoded GUI action and associated functions to the `Shared Memory`. When `Get Q-value` is called, the `Worker` provides the encoded GUI information to the `Model`, representing the latest trained version, which then processes it and generates the inferred GUI action to be performed. The inferred GUI action is then transferred to the GUI Interaction Layer (shown as the arrow connecting the two modules labelled with `Q Value`) for execution by the agent.

GUI Interaction Layer. The GUI interaction layer is responsible for device interactions and supports multiple device training and testing. For each available device, the GUI interaction layer instantiates a corresponding `Agent` to communicate with it using the Android Debugging Bridget that provides capabilities including GUI screen query and event execution, depicted in Figure 2 with arrows between devices and `Agents` within the GUI Interaction Layer. The GUI screen is represented in an XML format where each node stands for a GUI element with its class (type of the widget), resource ID (identifier of the GUI element), text, position, etc.

The `Agent` continuously queries the device screen to get available GUI elements and sends this GUI information to the `Encoder` element within `Worker`. When a decision is made by the DRL service, it executes the inferred GUI event with the highest Q-value.

4.2.1 Model Training. The training phase adopts a centralized training and distributed execution architecture, with each device corresponding to a `Worker` and multiple `Workers` corresponding to a `Trainer`. The specific process is as follows:

i. Each device installs the App under test, randomly selects a function from the input list of target functions as the target function, continuously obtains the current screen's GUI information and the list of functions covered by the previous action, and sends them to the corresponding `Worker`. The `Worker` parses the available action list from the current screen and selects the action generation method using the $\epsilon - greedy$ method [23]. With a

probability of ϵ, a random action is selected and returned to the client for execution. With a probability of $1-\epsilon$, the local model is called to obtain the Q-value of each action, and the action with the highest Q-value is returned to the client for execution.

ii. As the `Worker` continuously receives states and selects actions, it stores the test sequence information in the `Shared Memory`. The stored content is of the form

$$< s_0, a_0, F_0, s_1, a_1, F_1, ..., s_t, a_t, F_t, s_{t+1} >$$

with a maximum sequence length of N, where s_t is the current state's GUI information, including the current screen's activity and XML; a_t is the action executed on the current screen, F_t is the list of functions that can be covered after executing a_t in s_t, and s_{t+1} is the next screen after executing a_t in s_t.

iii. The `Trainer` actively monitors and reads the GUI interaction sequences from the `Shared Memory` in real-time and subsequently, stores it as training data in the Replay Buffer. The training data takes the form,

$$(g_t, s_t, a_t, r_t, s_{t+1}, d_t)$$

where g_t is a specific function that is the directed target, r_t is the reward value, and d_t is the termination flag. The training data is generated using the stored sequence by traversing the entire sequence. For each (s_t, a_t) pair in the sequence, the following steps are repeated K times, where K is a configurable model hyperparameter:

1. Randomly select a function from the list of target functions (after executing the action) represented as

$$g_{t,i} \in F_t \cup F_{t+1} \cup ... \cup F_N$$

2. Calculate the reward based on the selected target:

- The executed Action triggers the target Function, $r_t = 1$, $d_t = 1$
- The executed Action does not cause a change in the screen, $r_t = -0.001$, $d_t = 0$
- The target Function can be triggered within n steps after executing the Action, $r_t = 0.01 * \gamma^n$, $d_t = 0$
- Others, $r_t = -0.0001$, $d_t = 0$

where γ is the discount factor.

iv. The `Trainer` continuously extracts training data from the Replay Buffer and updates the model parameters

$$r_t + \gamma(1-d_t)Q'(s_{t+1}, argmax_a Q(s_{t+1}, a, g_t; \theta), g_t; \theta')$$

$$\rightarrow Q(s_t, a_t, g_t; \theta)$$

where Q' is the target network, its parameters are θ', and the parameters are periodically copied from the prediction network.

v. During training, the model is saved to the cloud and `Shared Storage` simultaneously. The `Worker` regularly obtains the latest model from `Shared Storage` and replaces the local model.

4.2.2 Guided GUI Exploration. At the beginning of the exploration, the `Worker` loads the optimal model from the storage and assigns target functions to each device. The client sends the current GUI information to the `Worker`. The `Worker` parses all the available actions in the current state. The `Worker` inputs this information into the local model, obtains the Q-value for each action, and selects the action with the highest Q-value to return to the client for execution. This continues until the target is covered or the maximum number of steps is reached. It then proceeds to select the next target function.

4.3 Implementation

Hawkeye is designed as a comprehensive automated testing framework for Android, comprising code instrumentation, client and server modules. The code instrumentation tool is developed using Java based on the ASM framework to insert function coverage loggers and send socket messages of real time incremental function coverage to the client. A Gradle plugin is developed using Groovy to perform code instrumentation automatically in existing Android projects. The client module is written in Java and leverages the GUI tree retrieving and action execution features of Fastbot2 [19] to interact with the App being tested. The server module, developed in GoLang, facilitates event selection and enables multi-device collaboration (which permits numerous clients to simultaneously test the same App on multiple devices while sharing the same deep reinforcement learning agent). The server provides APIs to receive GUI trees and returns events for execution by the clients.

5 EXPERIMENT

In this Section, we evaluate the feasibility and effectiveness of Hawkeye in generating GUI actions that exercise changed functions. We use 10 open source Android Apps from the F-Droid App market and one commercial App developed by ByteDance. A description of the Apps used is provided below,

Open source - We use 10 Android applications from the F-Droid App market [2] that has a catalogue of free and open source Android applications. We selected top rated Apps in F-Droid with a well documented commit history across multiple versions. Table 1 lists the names and versions for the Android Apps used in our experiment. For each open source App, we chose five commits that followed the base version, each leading to updates in a Java or Kotlin method of the respective App.

Commercial - We use a prominent enterprise collaboration software, Feishu, developed by ByteDance (#11 in bold in Table 1). The code instrumentation and APK building is carried out by the App development team. Changes used for this evaluation are subsequent sub-versions since the base version.

Index	App Name	Base Version	APK Size
1	Amaze File Manager	3.8.1	15.43 MB
2	Diary	1.77	0.58 MB
3	OpenTasks	1.2.3	0.30 MB
4	Simple Draw	6.8.0	5.56 MB
5	Simple File Manager	6.14.0	3.73 MB
6	WiFiAnalyzer	3.0.1	3.48 MB
7	Hibi	1.3.1	11.98 MB
8	Suntimes	0.14.12	40.12 MB
9	Nani	0.3.0	8.20 MB
10	Currency	1.31	4.43 MB
11	**Feishu**	5.15.1	183.7 MB

Table 1: Subject Apps: 10 open source Apps and 1 commercial App (listed at #11)

[2] https://f-droid.org/

We compare Hawkeye's performance with state-of-the-art (SOTA) Android GUI testing tools, *ARES*[27] and an in-house model-based testing tool, *Fastbot2* [19]. We investigate the following research questions:

Q1. Hawkeye versus *Fastbot2* exploration: *Is Hawkeye able to exercise the selected target functions better than Fastbot2?* To answer this question, we first randomly selected 50 functions within each open source App as our target functions. Then, we used Hawkeye and *Fastbot2* to each generate 1000 events to exercise the App Under Test (AUT). For each tool, we recorded the number of target functions covered by the generated events and compared their results.

Q2. Efficiency in testing commit-associated changes: *Is Hawkeye more effective in exercising changed functions associated with commits in open source Apps, when compared to ARES and Fastbot2?* For each commit in an open-source App, we manually identified changed functions associated with it. We then ran Hawkeye, *ARES*, and *Fastbot2* to generate 1000 events each to exercise the AUT. The number of events required by each approach to exercise the changed functions associated with the commit were recorded. The approach that required the lowest number of events to exercise all the changed functions was deemed the most effective for that particular commit. When a tool failed to exercise all the changed functions within 1000 events, the result was recorded as 1000.

Q3. Effectiveness on a commercial App: *Is Hawkeye better than Fastbot2 in exercising changed functions within the commercial App, Feishu?* For this question, we use developer identified changed functions for updates in Feishu. As the commercial App is large and complex, we set 200 GUI events as the maximum number of attempts for each changed function and report whether these functions are covered by Hawkeye. We then execute Hawkeye and *Fastbot2* to assess number of events required to execute all changed functions.

Q4. Analysis of Hawkeye performance: *What factors impact the effectiveness of Hawkeye in exercising changed functions?* We conduct a study using the commercial App, Feishu, to better understand the reasons for Hawkeye being effective at exercising some changed functions but not others. We randomly select 100 listener functions (callback functions that are directly bound to GUI events, such as `onClickListener`) and check whether Hawkeye is able to exercise them within 200 GUI events for each function. In contrast to the random selection, we also sample functions according to their heat – triggered frequency during training. We report the success rate of exercising heat-based sampled functions within the same 200-events budget as used with the random selection.

Selected Tools. We select test input generation tool, *ARES* for comparison in research question 2 since it is the SOTA reinforcement learning-based GUI testing tool. We attempted to use *ARES* for comparison over the commercial App in research question (Q) 3 but found the time needed to generate events over the commercial App was immeasurably long and the tool timed out after generating few events. *Fastbot2* is a reusable automated model-based Android GUI testing tool that leverages probabilistic model and reinforcement learning to utilise prior exploration data and prioritise new GUI states that was not covered by previous testing runs. *Fastbot2* is not designed to target changes. We compare with *Fastbot2* as it is the in-house testing tool used by the Feishu developer team.

As mentioned in Section 3.4, it is not possible to make a comparison with CAT and ATUA with equivalent testing capabilities for change-focused testing, since the static analysis module they depend on is unable to handle large applications, even when utilising a devbox with 128 GB of RAM.

Experiment Platform. We conducted our experiment on multiple Samsumg Galaxy S10 devices running Android 11 systems. Hawkeye and *Fastbot2* were run on the mobile devices, while ARES was run on a MacBook Pro equipped with a 2.6GHz Intel Core i7 Processor.

6 RESULTS

We present results from our experiment in the context of the research questions in Section 5.

6.1 Q1: HAWKEYE versus Fastbot2 Exploration

App Name	Fastbot2	Hawkeye	Intersection
Amaze File Manager	**37**	33	31
Diary	40	**43**	39
OpenTasks	**50**	39	39
Simple Draw	**28**	27	22
Simple File Manager	39	**49**	39
WiFi Analyzer	**50**	**50**	50
Hibi	17	**39**	16
Suntimes	25	**37**	22
Nani	49	**50**	49
Currency	32	**45**	32
Avg.	36.7	**41.2**	N/A

Table 2: Target Functions Coverage

For the first research question, we assess if randomly selected 50 target functions in each of the open source Apps can be exercised better by Hawkeye than *Fastbot2*. Function coverage over the 50 target functions is reported for both tools over each App in Table 2. The final column in Table 2 shows the number of target functions covered by both Hawkeye and *Fastbot2*. On average, across the Apps, we find Hawkeye outperformed *Fastbot2* in terms of target function coverage. Specifically, Hawkeye covered an average of 45 out of 50 target functions, whereas *Fastbot2* covered only 39. Furthermore, Hawkeye achieved a higher function coverage for six out of the ten open source Apps than *Fastbot2*. Hawkeye and *Fastbot2* both achieved a 100% coverage on the WiFi Analyzer App. This is because the target functions for this App were easily accessible through a few actions on the MainActivity.

While *Fastbot2* demonstrated similar performance to Hawkeye across many applications, notable performance disparities were observed in `Currency`, `Hibi`, and `Suntimes`, where Hawkeye effectively exercised 12–22 functions more in each of these Apps. This discrepancy can be attributed to the reachability of the target functions within these Apps, which, upon manual inspection, were found to require a greater number of steps on average to be exercised. Additionally, it is important to highlight the intersection column in Table 2, that shows a significant overlap between

the covered functions of both Hawkeye and *Fastbot2*. This finding strongly suggests that while there is a common subset of functions exercised by both tools, Hawkeye has the distinct capability to go beyond and exercise additional functions within the AUTs. The results indicate that Hawkeye is more effective in generating events that comprehensively exercise a broader range of target functions within the AUT.

6.2 Q2: Efficiency in Testing Commit-Associated Changes

Table 3 displays the commits for each open-source App to be tested using Hawkeye, *Fastbot2*, and *ARES*. We selected the five most recent commits with code changes for evaluation. The second column represents the count of changed functions associated with each commit, averaging at 6.8 functions updated per commit. We recorded the number of GUI events needed by each approach to exercise the changed functions linked to the respective commit. A stopping criterion of 1000 events was set for the tools. In Table 3, cells containing the value 1000 indicate that the corresponding tool failed to exercise the changed functions for that commit. On the other hand, numbers less than 1000 are considered a success for the tool, with lower number of events indicating better efficiency in reaching and exercising the changes. For each commit, the winning tool is the one with the lowest number of events (shown in bold).

Table 3 makes it evident that all three tools struggle to exercise changes in a significant portion of the examined commits. Our manual analysis revealed that this is largely due to a substantial 18% of the updated methods being callable only under specific Android or database versions, rendering them inaccessible when the version of Android or the database in the AUT does not match. Additionally, in the case of four commits in `Suntimes`'s (App# 9), exercising the target functions necessitates adding an App widget to the home screen. However, this action is unlikely to occur because adding home screen widgets requires very specific system setting operations. Even if the widget is added by default, during testing, when an event leads to the home screen, all the tools tend to relaunch the App to avoid operations not related to the App, making it unlikely to perform this specific action. `Currency` is the only App where all five commits were successfully exercised by all three tools. This can be attributed to the fact that the number of updated functions associated with the commits for `Currency` is small and not contingent on specific Android or database versions. Additionally, the updated functions require a relatively small number of steps to be exercised.

Success rates for the tools (last row of Table 3), measured as the fraction of commits that needed less than 1000 events, are comparable. In terms of the average number of events needed for each tool to successfully exercise the commits (second to last row of Table 3), Hawkeye required the fewest number of events on average – 191 events – to exercise the updated methods of a commit, while *ARES* required 202 events and *Fastbot2* required 261 events.

Understanding performance based on results from the open-source App commits proves challenging due to the intrinsic fragility and lack of portability of these Apps across various Android and Database versions. It is crucial to acknowledge and consider these limitations when interpreting the performance presented in Table 3

App Index	#Changed Functions	Commit	Fastbot2	Hawkeye	ARES
1	1	6d6a192	1000	1000	1000
	1	92256a7	1000	54	**6**
	6	27d1e2b	1000	1000	1000
	3	67d6712	1000	1000	1000
	33	538fd8a	1000	1000	1000
2	3	c4aabf2	309	399	**115**
	1	54c1335	**354**	487	873
	1	7c51891	55	**4**	21
	1	2070eb0	305	**5**	6
	1	f66a966	**71**	81	197
3	1	f53cddd	1000	1000	**633**
	1	1a1669f	1000	1000	**109**
	1	f175a71	1000	**773**	1000
	2	700a773	1000	701	**20**
	4	47d6676	1000	1000	1000
4	2	fd7bca5	630	**9**	754
	1	621d932	1000	1000	1000
	6	c660f5d	1000	1000	1000
	6	4e2e9f7	1000	1000	1000
	3	c1c9be2	1000	1000	1000
5	2	ea56927	1000	1000	1000
	4	440df4e	1000	1000	1000
	3	0488984	40	27	**3**
	2	ce18931	1000	1000	1000
	1	0b5d6ae	1000	1000	**183**
6	22	b8c4544	1000	1000	1000
	1	e750aca	**3**	10	5
	51	8ddf52d	1000	1000	1000
	1	d7f08ff	1000	1000	1000
	5	238fae0	1000	1000	1000
7	53	2c4dccd	1000	1000	1000
	2	294609c	1000	1000	1000
	1	944848a	1000	1000	1000
	2	b7f113b	240	**4**	25
	4	1456648	**850**	1000	1000
8	1	69ccd65	**462**	524	1000
	4	4dfa28c	1000	1000	1000
	11	69db31a	1000	1000	1000
	2	2f0e4a3	165	133	**6**
	2	ab47bca	1000	1000	1000
9	3	f833032	1000	1000	1000
	1	95d6d30	**2**	3	563
	5	3b27600	1000	1000	1000
	27	ad10290	1000	1000	1000
	12	cbdf999	1000	1000	1000
10	30	3ba6c7c	1000	1000	1000
	5	7156da1	488	**20**	1000
	1	966b094	52	196	**40**
	1	81e79d8	382	1000	**277**
	3	dc3bf39	22	3	**2**
Avg.	6.8	-	261	**191**	202
Success Rate	-	-	34%	36%	**38%**

Table 3: Changed Function Coverage on Open-source Apps.

Commit ID	# Changed Functions	Fastbot2		HawkEye	
		# Covered	Coverage (%)	# Covered	Coverage (%)
1	11	0	0.0%	**10**	**90.9%**
2	8	**4**	**50.0%**	2	25.0%
3	68	0	0.0%	**34**	**50.0%**
4	14	**1**	**7.1%**	0	0.0%
5	393	0	0.0%	**5**	**1.3%**
6	138	0	0.0%	**17**	**12.3%**
7	209	0	0.0%	**15**	**7.2%**
8	216	0	0.0%	0	0.0%
9	36	0	0.0%	0	0.0%
10	59	0	0.0%	**10**	**16.9%**
11	40	0	0.0%	**14**	**35.0%**
12	133	2	1.5%	**7**	**5.3%**

Table 4: Changed Function Coverage on Feishu

and drawing conclusions about the effectiveness of the testing tools. To address these limitations and provide a more comprehensive evaluation, we include a well-maintained commercial App in our analysis in the next section.

6.3 Q3: Effectiveness on a Commercial App

Table 4 shows twelve commits for the Feishu App, the number of changed functions identified by the developers for each commit, and the function coverage achieved by Hawkeye and the baseline *Fastbot2*. The commits were chosen from merge requests by Feishu developers that transpired within a one-week time period. Across the 12 commits, *Fastbot2* performed poorly, achieving less than 10% coverage on 11 out of the 12 commits. Hawkeye performs considerably better than *Fastbot2* on 8 of the 12 commits with improvements in the range of 1.3 – 90.9%. *Fastbot2* performs better than Hawkeye on 2 out of the 12 commits, when the random exploration in *Fastbot2* fortuitously identified the correct sequence of events, resulting in more effective testing for those particular commits in this evaluation instance.

Function coverage with Hawkeye is 5% or less for half the commits. This is primarily because of the stopping condition we imposed of 200 events per changed function. This condition was set to ensure that results could be obtained within a reasonable time frame but *Fastbot2* does not have this constraint. For a commercial App of Feishu's scale, limiting the number of events to 200 per changed function may indeed result in relatively low coverage. Despite this limitation, Hawkeye emerges as the most promising tool, showcasing the potential to handle the complexity of a commercial App and effectively test updates. We anticipate that lifting this event limit will lead to improved coverage. It's important to recognize that this is an initial step in testing commit-associated changed functions for commercial Apps, and further advancements are necessary, particularly in refining the reinforcement learning technique to better exercise the changes. Nonetheless, Hawkeye's performance surpasses that of the in-house company tool, *Fastbot2*, and has been positively received by developers who find it efficient and time-saving for exercising changes.

6.4 Analysis of Hawkeye Performance

6.4.1 Listener Function Coverage. We randomly selected 100 listener functions from Feishu as target functions and set the maximum number of GUI events for each function as 200. We repeated this experiment 13 times.

On average, Hawkeye achieved coverage for 52 out of the 100 functions, utilizing an average of 42 GUI events per function. Cumulatively, across all 13 testing runs, Hawkeye covered a total of 89 out of the 100 functions. It is worth noting that, Hawkeye is able to cover the first 44 functions efficiently within the initial 5 minutes. However, it then expended the entire 3-hour budget to cover the remaining functions. As with research question 3, we believe that function coverage will improve if Hawkeye uses more than 200 GUI events for each function given the complexity and size of Feishu.

6.4.2 Function Coverage based on Heat. As shown in Table 5, following 10 hours of model training based on random exploration, 196 functions were exercised more than 300 times, 192 functions were exercised between 100 and 300 times, 159 functions were exercised between 50 and 100 times, 260 functions were exercised between 20 and 50 times, 208 functions were exercised between 10 and 20 times and 558 functions were exercised less than 10 times.

As anticipated, we observed a clear correlation between function coverage achieved by Hawkeye and the exploration frequency during model training. Functions that were more frequently explored (exceeding 300 times) during training were also more readily covered by Hawkeye's tests. Conversely, functions that had a lower exploration frequency during training (less than 50 times) posed a challenge for Hawkeye in achieving coverage. This correlation aligns with expectations and underscores the importance of training the model on a broad set of functions. Our current training was limited by resource availability. However, we are optimistic that this limitation can be overcome by scaling up our training infrastructure, increasing the number of devices used for training, and thereby covering a more extensive range of GUI states and functions.

Developer Feedback and Scale. Hawkeye has gained widespread adoption within multiple product teams at ByteDance, including Douyin, Toutiao, and Feishu, which collectively boast billions of user installations. In a comprehensive one-year review, Hawkeye demonstrated an improved change-impacted statement coverage by 11%. Additionally, it considerably improved the detection of update-related crashes, achieving a threefold increase in identification.

The product teams using Hawkeye have consistently praised its effectiveness in facilitating thorough testing of App changes and preemptively exposing potential App crashes before they impact end-users. This positive feedback underscores Hawkeye's value in maintaining App quality and user experience.

Furthermore, Hawkeye is actively invoked hundreds of times each week. Whenever a new merge request is submitted to the codebase, the continuous integration (CI) pipeline automatically initiates the creation of a corresponding APK file (App installation package). Subsequently, Hawkeye is triggered to conduct a smoky test, ensuring the quality of the introduced code changes before they

Function Heat Range	>=300	100-300	50-100	20-50	10-20	<10
Number of Functions	196	192	159	260	208	558
Success Rate >30%	76.5% (150/196)	29.2% (56/192)	11.9% (19/159)	-	-	-

Table 5: Case Study on Feishu

are deployed. This seamless integration emphasizes Hawkeye's role in upholding code quality and robustness in the development process.

6.5 Lessons Learned

Both *Fastbot2* and Hawkeye have been deployed at ByteDance for Android App testing for several years. Based on this deployment experience, we provide valuable lessons for practitioners in the field of automated Android App testing:

Lesson 1: While static analysis-based tools are precise, they struggle to scale to commercial Apps. Existing methods use static analysis to construct App state transition graphs and infer changes by analyzing and mapping widget definitions and listener functions [22, 25, 35]. However, Android App development offers various ways to define widgets and bind their callbacks. Dynamic addition of widgets in the source code is common practice in complex commercial Apps but the support for this feature is cumbersome, accompanied by high overhead in static analysis. In our experience, we found static analysis-based tools failed to analyze our Apps due to out-of-memory crashes or incomplete results when reducing precision levels. This limitation drove us to adopt a fully dynamic approach, building the transition graph during runtime.

Lesson 2: Leveraging historical exploration data. In an industrial setting, the continuous daily testing of Apps utilizing hundreds of devices generates a wealth of historical exploration data, including transition graphs and screenshots. *Fastbot2* and Hawkeye represent our initial efforts to harness this valuable data to prioritize untested activities and change-impacted functions. It is worth noting that the form of exploration data and the methods used for capturing and storing them are vital considerations. Existing techniques commonly store exploration data as transition graphs. However, we found this method proves impractical for large, complex Apps, consuming excessive memory. To overcome this challenge, we adopted a more efficient approach to store exploration data by capturing and storing only the mapping between events and functions, not the full graph, ensuring both memory and processing efficiency. This adjustment proved to be effective and scalable in storing exploration data for ByteDance Apps.

Lesson 3: Integration into CI pipeline. The seamless integration of Hawkeye into the CI pipeline, coupled with its automatic triggering for every merge request, played a pivotal role in maximizing its utility for testing App updates. This approach ensured a robust quality assurance process and significantly contributed to the acceptance and enthusiastic use of Hawkeye by developers.

7 THREATS TO VALIDITY

A potential threat to internal validity is bugs in Hawkeye's implementation. To mitigate this threat, we conducted careful code reviews and extensive testing. When used on the commercial App, the developers at the commercial site conducted several rounds of manual inspection to mitigate the risk of manual mistakes or omissions in the tool. It is also worth noting that tools adopted by the product team undergo rigorous inspection and evaluation before they are adopted.

A potential threat to the external validity is related to the fact that the set of Android Apps we have considered in this study may not be an accurate representation of a potential App under test. We attempt to reduce the selection bias by using a dataset of 10 open source Apps from different categories with a variety of Android features, and one large, complex commercial App.

A threat to construct validity is caused by restricting the number of GUI events generated by the tools – Hawkeye, *Fastbot2, ARES* – to 1000 events for open source Apps and 200 for the commercial App. Restriction to 1000 input events is inspired from related work in GUI testing [3, 17] that used this in their default settings.

A final threat to validity is the limited number of tools used in comparison. As discussed in Section 3.4, we cannot compare with CAT and ATUA with similar capabilities to test changes as the static analysis module they rely on fail on large Apps even using a devbox with 128 GB RAM. We used *Fastbot2* and *ARES* for comparison as they are the SOTA RL-based GUI testing tools that we are aware of.

8 CONCLUSION

In this paper, we presented Hawkeye, specifically designed for generating GUI test inputs for exercising modified functions associated with Android App updates. Hawkeye uses deep reinforcement learning to learn the mapping between GUI events and functions, producing GUI event sequences that interact with these modified functions. To assess the performance of Hawkeye, we conducted an empirical evaluation, comparing it against *ARES* and *Fastbot2* across 10 open-source Android Apps and a large commercial App. Our evaluation led to the following key observations:

(1) Hawkeye can execute modified functions more frequently than *Fastbot2* and *ARES*, while using fewer GUI events.
(2) For a complex commercial App, Feishu, Hawkeye exhibits better performance than *Fastbot2* in covering listener functions. Hawkeye covers 85% of Feishu's randomly selected listener functions within the first 180 minutes of a testing run.
(3) Hawkeye is effective at exercising functions that were commonly observed during model training.
(4) Internally, within our organization, Hawkeye is integrated into the continuous integration pipeline and is triggered hundreds of times per week whenever changes are submitted to the codebase. On average, the utilization of Hawkeye leads to a 11% increase in change-impacted statement coverage. Moreover, it uncovers App crashes three times more frequently in an industrial setting, showcasing its efficacy in enhancing test coverage and identifying potential issues in App updates.

REFERENCES

[1] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.

[2] Young-Min Baek and Doo-Hwan Bae. 2016. Automated model-based Android GUI testing using multi-level GUI comparison criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 238–249.

[3] Nataniel P Borges, Jenny Hotzkow, and Andreas Zeller. 2018. DroidMate-2: a platform for Android test generation. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 916–919.

[4] Nataniel P Borges Jr, Maria Gómez, and Andreas Zeller. 2018. Guiding app testing with mined interaction models. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. 133–143.

[5] Wontae Choi, Koushik Sen, George Necul, and Wenyu Wang. 2018. DetReduce: minimizing Android GUI test suites for regression testing. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 445–455.

[6] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated test input generation for android: Are we there yet?. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 429–440.

[7] Eliane Collins, Arilo Neto, Auri Vincenzi, and José Maldonado. 2021. Deep reinforcement learning based Android application GUI testing. In *Proceedings of the XXXV Brazilian Symposium on Software Engineering*. 186–194.

[8] Google Developers. [n. d.]. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey. Accessed: 2020-08-20.

[9] Quan Do, Guowei Yang, Meiru Che, Darren Hui, and Jefferson Ridgeway. 2016. Regression test selection for android applications. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*. 27–28.

[10] Quan Chau Dong Do, Guowei Yang, Meiru Che, Darren Hui, and Jefferson Ridgeway. 2016. Redroid: A Regression Test Selection Approach for Android Applications.. In *SEKE*. 486–491.

[11] Yuemeng Gao, Chuanqi Tao, Hongjing Guo, and Jerry Gao. 2023. A Deep Reinforcement Learning-Based Approach for Android GUI Testing. In *Web and Big Data: 6th International Joint Conference, APWeb-WAIM 2022, Nanjing, China, November 25–27, 2022, Proceedings, Part III*. Springer, 262–276.

[12] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 269–280.

[13] Konrad Jamrozik and Andreas Zeller. 2016. DroidMate: a robust and extensible test generator for Android. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*. 293–294.

[14] Bo Jiang, Yu Wu, Yongfei Zhang, Zhenyu Zhang, and Wing-Kwong Chan. 2018. ReTestDroid: towards safer regression test selection for android application. In *2018 IEEE 42nd annual computer software and applications conference (COMPSAC)*, Vol. 1. IEEE, 235–244.

[15] Xiao Li, Nana Chang, Yan Wang, Haohua Huang, Yu Pei, Linzhang Wang, and Xuandong Li. 2017. ATOM: Automatic maintenance of GUI test scripts for evolving mobile applications. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 161–171.

[16] Yuxi Li. 2017. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274* (2017).

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

[18] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2019. Humanoid: A Deep Learning-based Approach to Automated Black-box Android App Testing. *arXiv preprint arXiv:1901.02633* (2019).

[19] Zhengwei Lv, Chao Peng, Zhao Zhang, Ting Su, Kai Liu, and Ping Yang. 2022. Fastbot2: Reusable Automated Model-based GUI Testing for Android Enhanced by Reinforcement Learning. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–5.

[20] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 224–234.

[21] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. 2016. Reducing combinatorics in GUI testing of android applications. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 559–570.

[22] Chanh-Duc Ngo, Fabrizio Pastore, and Lionel C Briand. 2022. ATUA: an update-driven app testing tool. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 765–768.

[23] Paavai Paavai Anand et al. 2021. A brief study of deep reinforcement learning with epsilon-greedy exploration. *International Journal Of Computing and Digital System* (2021).

[24] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement learning based curiosity-driven testing of Android applications. In *ISSTA*. 153–164.

[25] Chao Peng, Ajitha Rajan, and Tianqin Cai. 2021. Cat: Change-focused android gui testing. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 460–470.

[26] Oliviero Riganelli, Simone Paolo Mottadelli, Claudio Rota, Daniela Micucci, and Leonardo Mariani. 2020. Data loss detector: automatically revealing data loss bugs in Android apps. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 141–152.

[27] Andrea Romdhana, Alessio Merlo, Mariano Ceccato, and Paolo Tonella. 2022. Deep reinforcement learning for black-box testing of android apps. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 4 (2022), 1–29.

[28] Aman Sharma and Rupesh Nasre. 2019. QADroid: regression event selection for Android applications. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 66–77.

[29] RM Sharma. 2014. Quantitative analysis of automation and manual testing. *International journal of engineering and innovative technology* 4, 1 (2014).

[30] Wei Song, Xiangxing Qian, and Jeff Huang. 2017. EHBDroid: Beyond GUI testing for Android applications. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 27–37.

[31] Ting Su. 2016. FSMdroid: guided GUI testing of android apps. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 689–691.

[32] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 245–256.

[33] Tommi Takala, Mika Katara, and Julian Harty. 2011. Experiences of system-level model-based GUI testing of an Android application. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 377–386.

[34] Thomas Wetzlmaier and Rudolf Ramler. 2017. Hybrid monkey testing: enhancing automated GUI tests with random test generation. In *Proceedings of the 8th ACM SIGSOFT International Workshop on Automated Software Testing*. 5–10.

[35] Shengqian Yang, Haowei Wu, Hailong Zhang, Yan Wang, Chandrasekar Swaminathan, Dacong Yan, and Atanas Rountev. 2018. Static window transition graphs for Android. *Automated Software Engineering* 25, 4 (2018), 833–873.