# A Lightweight Approach of Human-Like Playtest for Android Apps

Yan Zhao

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of science
in
Computer Science

Na Meng, Chair
Haipeng Cai
Eli Tilevich

October 23, 2021
Blacksburg, Virginia

Keywords: automated game testing, playtest, tactic generalization, tactic concretization.

# A Lightweight Approach of Human-Like Playtest for Android Apps

Yan Zhao

(ABSTRACT)

Testing is recognized as a key and challenging factor that can either boost or halt the game development in the mobile game industry. On one hand, manual testing is expensive and time-consuming, especially the wide spectrum of device hardware and software, so called fragmentation, significantly increases the cost to test applications on devices manually. On the other hand, automated testing is also very difficult due to more inherent technical issues to test games as compared to other mobile applications, such as non-native widgets, non determinism , complex game strategies and so on. Current testing frameworks (e.g., Android Monkey, Record & Replay) are limited because they adopt no domain knowledge to test games. Learning-based tools (e.g., Wuji) require tremendous resources and manual efforts to train a model before testing any game. The high cost of manual testing and lack of efficient testing tools for mobile games motivated the work presented in this thesis which aims to explore easy and efficient approaches to test mobile games efficiently and effectively. A new Android mobile game testing tool, called Lit , has been developed.

Lit is a lightweight approach to generalize playtest tactics from manual testing, and to adopt the tactics for automatic game testing. Lit has two phases: tactic generalization and tactic concretization. In Phase I, when a human tester plays an Android game $G$ for awhile (e.g., eight minutes), Lit records the tester's inputs and related scenes. Based on the collected data, Lit infers a set of *context-aware, abstract playtest tactics* that describe under what circumstances, what actions can be taken. In Phase II, Lit tests $G$ based on the generalized

tactics. Namely, given a randomly generated game scene, Lit tentatively matches that scene with the abstract context of any inferred tactic; if the match succeeds, Lit customizes the tactic to generate an action for playtest.

Our evaluation with nine games shows Lit to outperform two state-of-the-art tools and a reinforcement learning (RL)-based tool, by covering more code and triggering more errors. This implies that Lit complements existing tools and helps developers better test certain games (e.g., match3).

# A Lightweight Approach of Human-Like Playtest for Android Apps

Yan Zhao

(GENERAL AUDIENCE ABSTRACT)

Testing is recognized as a key and challenging factor that can either boost or halt the game development in mobile game industry. On the one hand, manual testing is expensive and time-consuming, especially the wide spectrum of device hardware and software significantly increase cost to test applications on devices manually. On the other hand, automated testing is also very difficult due to more inherent technical issues to test games as compared to other mobile applications. The two factors motivated the work presented in this thesis.

A new Android mobile game testing tool, called Lit , has been developed. Lit is a lightweight approach to generalize playtest tactics from manual testing, and to adopt the tactics for automatic game testing. A playtest is the process in which testers play video games for software quality assurance. When a human tester plays an Android game $G$ for awhile (e.g., eight minutes), Lit records the tester's inputs and related scenes. Based on the collected data, Lit infers a set of *context-aware, abstract playtest tactics* that describe under what circumstances, what actions can be taken. In Phase II, Lit tests $G$ based on the generalized tactics. Namely, given a randomly generated game scene, Lit tentatively matches that scene with the abstract context of any inferred tactic; if the match succeeds, Lit customizes the tactic to generate an action for playtest. Our evaluation with nine games shows Lit to outperform two state-of-the-art tools and a reinforcement learning (RL)-based tool, by covering more code and triggering more errors. This implies that Lit complements existing tools and helps developers better test certain games (e.g., match3).

# Acknowledgments

First and most important, I would like to thank my advisor, Professor Meng, for providing me the opportunity to pursue my master work at Virginia Tech. This research is completely inspired and supported by . This work is a direct outcome of her hands-on supervision and guidance.

I would like to thank my committee members, Dr. Eli Tilevich, Dr. Cai for their suggestions and support.

I would also like to express my grateful thanks to all the professors and staff members in Computer Science Department.

I would like to thank all my former and present colleagues in Computer Science department, for their help and support. They make my research life here in Virginia Tech more enjoyable.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Introduction

Mobile application and game business grows incredibly over the past few years. Until December 2019, the number of available mobile applications in the Google Play store is over 2.9 million[7] and 25% of them are mobile games. The mobile game is also a high user engagement app category, which accounts for 43% of all Smartphone use and 68% of all App revenue[17]. The growth speed of mobile industry has lead to decreased quality in the apps. Testing tools has not kept up with the growth and the quality. A study shows that popular games receive a large number of reviews each day. Bugs in a mobile game bring bad user experiences, and also cost the game company excessively for bug fixing and user losing[31]. Hence, early-stage testing is a critical stage for detecting bugs and ensuring the reliability of mobile games.

Testing such mobile games is a challenge, because covering sufficiently exercising a game often requires heavy user interactions and complicated input behaviors. Some games even contain execution paths that can be exercised only after the user has passed certain difficult intermediate tasks. Owing to these reasons, most game companies heavily rely on manual testing without systematic solutions[3]. However, these manual testing solutions are often time-consuming and inefficient, which require intensive manpower.

Automatic testing techniques have been widely studied, such as search-based testing [4], [16], coverage-guided fuzzing [1], symbolic execution [8], random-based testing (Monkey [24]) and model-based testing (Stoat [35]) . However, such techniques are challenging in testing games as the game playing is a continuous interaction process with rich graphical user interfaces (GUIs) between the gamer and the game. These techniques are not effective in game testing since they are not "smart" to accomplish the complicated goal of the game.

In recent years, scientists applied machine learning, specifically reinforcement learning, to play games and made a great progress, like IBM's Deep Blue[ 6 ] and DeepMind's AlphaGo[ 24 ].But the existing DRLs mostly focus on winning the game rather than testing game. So researcher are also investigating how to test mobile applications automatically with the intelligence of various machine learning modules and algorithms, like Wuji[]. we know that reinforcement learning requires millions of attempts before learning to solve a task such as playing an game.It has a high requirement of hardware and computing resource which might not work well with frequent iterative development of Mobile games. In this paper, we try to present an light weight solution for game playtest.

In the video game industry, **playtest** refers to the process of exposing a game to its intended audience, so as to reveal potential software flaws during the game prototyping, development, soft launch, or after release. Game vendors sometimes recruit human testers from playtest platforms [1, 2, 44] and pay testers money for game playing.

In our research, there are three challenges:

1. Different games define distinct rules and require users to play games by taking specialized actions (e.g., "long tap" or "swipe"). Our approach needs to mimic game-specific user actions to test games like a human.

2. A **game scene** is an image to display different information related to one program state (see Fig. 2.1). Scenes can be non-deterministic, so our approach should flexibly react to the changing program states.

3. Games usually define various customized UI items or game icons (i.e., pictures) which are not recognizable by most automatic testing frameworks. To effectively play games, our approach should identify those icons.

To overcome the above-mentioned challenges, we developed LIT to have two phases: tactic generalization and tactic concretization. Here, a **tactic** describes in what context (i.e., program states), what playtest action(s) can be taken and how to take those actions.

Phase I requires users to (1) provide snapshots of game icons and (2) play the game $G$ for awhile. Based on the provided snapshots, LIT uses image recognition [15] to identify relevant icons in a given scene. When users play $G$, LIT recognizes each user action with respect to game icon(s) and further records a sequence of $\langle context, action \rangle$ pairs. Here, **context** removes scenery background but keeps all recognized game icons. From the recorded pairs, LIT generalizes tactics by (1) identifying abstract contexts $AC = \{ac_1, ac_2, \ldots\}$ and major action types $AT = \{at_1, at_2, \ldots\}$ and (2) calculating alternative parameters and/or functions to map each abstract context to an action type. Phase II takes in any generalized tactics and plays $G$ accordingly. Given a scene $s$, LIT extracts the context $c$, and tentatively matches $c$ with any abstract context $ac \in AC$ involved in the tactics. If there is a match, LIT randomly picks a corresponding parameter setting and/or synthesized function to create an action for game testing.

For evaluation, we applied LIT , two state-of-the-art testing tools (i.e., Monkey [24] and Sapienz [34]), and a reinforcement learning (RL)-based tool to a set of game apps. Our evaluation shows that with an eight-minute user demo for each open-source game, LIT out-

performed all tools by achieving higher test coverage and triggering more runtime errors.

Specifically, we tested three famous commercial mobile games, including *Angry Birds*, *Ketchapp Basketball*, and *Pop Star*. With about 8-minute user-playing demos that achieve level 2 and get 179294 points in *Angry Birds*, Lit infers reacting tactics and covers various scenarios that achieve level 7, and get 1184084 points. Choudhary et al.[14] present that Monkey achieves higher test coverage and reports more failures than other testing tools, but it can only get 52667 points and achieve level 0.

We further evaluate Lit on popular open-source games, such as *Android 2048*, *Archery*, *CasseBonbons* [52] (a game similar to *Candy Crush Saga* etc. The results show that Lit significantly improves the testing coverage and trigger two runtime errors. In average, our framework improved line coverage by 58.14% and branch coverage by 100% over Monkey. The average coefficient of variation is reduced by 97.27%.

Our experiments show that Lit is capable of testing games of three popular categories: match3, shooting, and basic board games. As there are hundreds of games belonging to these categories [35, 37], we believe that Lit can tremendously help many game developers to efficiently test games and improve software quality.

To sum up, we made the following contributions:

- We designed and implemented a novel algorithm to generalize tactics from user-provided icons and short game-playing demos. The algorithm identifies user actions, records $\langle context, action \rangle$ pairs, and derives functions or parameters to map game contexts to feasible actions.

- We designed and implemented a novel algorithm to test games based on the generalized tactics. Lit reacts to any randomly generated game scene by matching the scene with

context in tactics, and taking actions accordingly.

- We compared LIT with two state-of-the-art tools and one RL-based tool using a dataset of nine games. Our evaluation comprehensively compared the test coverage of different tools in terms of source lines, branches, Java classes, and Java methods. We also compared different tools in terms of earned game scores, passed difficulty levels, and triggered errors. LIT outperformed all tools.

At https://figshare.com/s/c7ac3cfa300e3ede1202, we open-sourced our program and data.

# Chapter 2

# Background

## 2.1 Smartphone and Mobile Operating System

From around 2010, the touchscreen smartphone revolution had a major impact on sales of basic feature phones, as the sales of smartphones increased from 139 million units in 2008 to 1.54 billion units in 2019. In 2020, smartphone sales decreased to 1.38 billion units due to the coronavirus (COVID-19) pandemic. Apple, Samsung, and lately also Xiaomi, were the big winners(Table 2.1) in this shift towards smartphones; BlackBerry and Nokia were among the losers. Nokia's focus on hardware rather than software specifications is one reason their net sales fell by around 30 billion euros in just three years.

Android and iOS are the two mostly used operating systems (OS) for smartphones. Android maintained its position as the leading mobile operating system worldwide in June 2021, controlling the mobile OS market with a close to 73 percent share. Google's Android and Apple's iOS jointly possess over 99 percent of the global market share. This thesis will concentrate on Android, because at the moment it is dominating the mobile OS niche, see

Table 2.1: Smartphone shipment market shares

| **Brands** | **2019Q1** | **2019Q2** | **2019Q3** | **2019Q4** | **2020Q1** | **2020Q2** | **2020Q3** | **2020Q4** | **2021Q1** | **2021Q2** |
|---|---|---|---|---|---|---|---|---|---|---|
| Samsung | 21% | 21% | 21% | 18% | 20% | 20% | 22% | 16% | 22% | 18% |
| Xiaomi | 8% | 9% | 8% | 8% | 10% | 10% | 13% | 11% | 14% | 16% |
| Apple | 12% | 10% | 12% | 18% | 14% | 14% | 11% | 21% | 17% | 15% |
| Other | 59% | 60% | 59% | 56% | 56% | 56% | 54% | 52% | 47% | 52% |

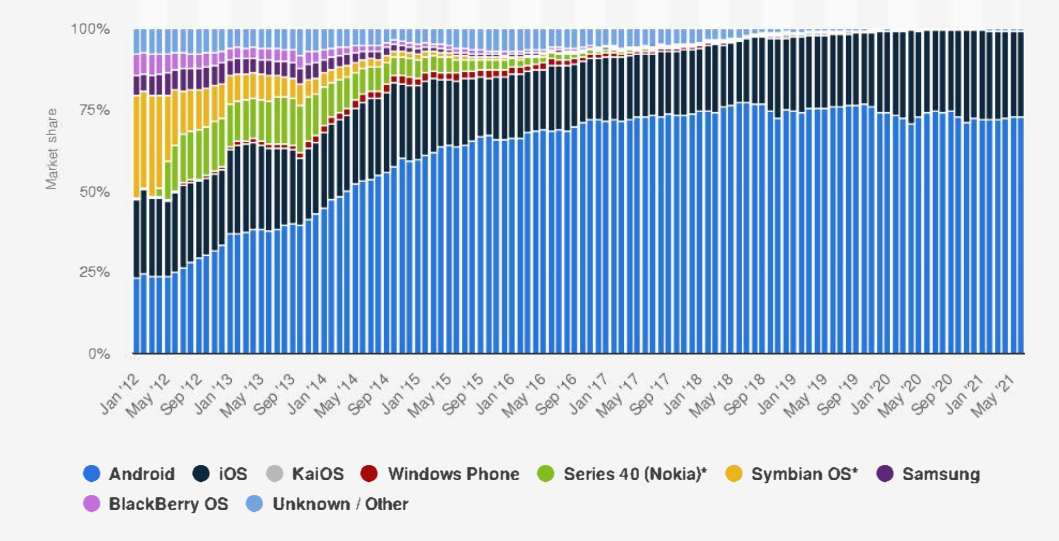


Figure 2.1: Mobile Operating Systems' Market Share Worldwide

Figure 2.1.

### 2.1.1 Fragmentation

Fragmentation refers to a concern over the number of distinct devices with different combinations of software and hardware. The number of distinct Android devices is increasing and there were 24093 distinct devices in August 2015. Figure 2.2 shows state of Android Fragmentation. Each cell represent a device and its market share.

A major challenge in mobile application development is the inability to 'write once and run anywhere'. Developers often have to customize a mobile application to suit a multitude of diverse mobile devices. This increases the effort required in all aspects of application development, such as testing. Android fragmentation truly emphasizes the need for cross-browser testing. Because the operating system versions, hardware specifications, screen sizes, and system UI all differ from device to device and brand to brand, test results will vary in the same way.

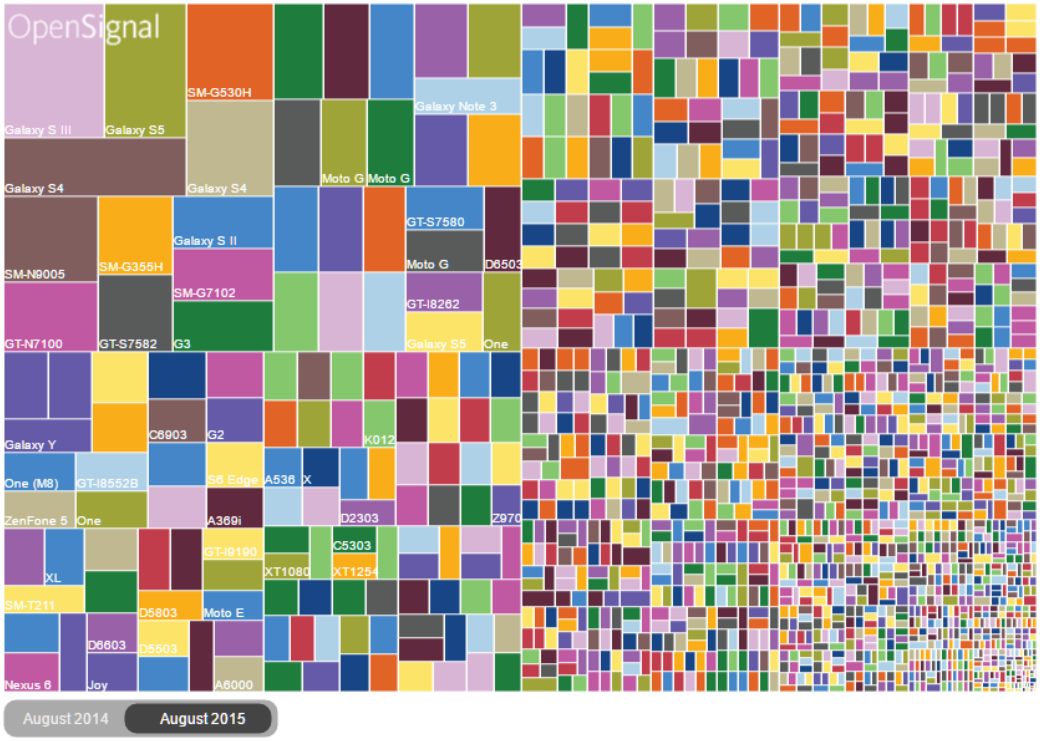


Figure 2.2: The Sorry State of Android Fragmentation

Manual testing is extremely expensive and time-consuming. Having a test automation tool, is truly a good way to optimize usability for various Android devices in order to be inclusive of their current market as well as their potential market of mobile users.

## 2.2 Mobile Games

Mobile application and game business grows incredibly over the past few years. Until December 2019, the number of available mobile applications in the Google Play store is over 2.9 million[7] and 25% of them are mobile games. When gaming becomes an all-around entertainment, the mobile game is also a high user engagement app category, which accounts for 43% of all Smartphone use[17].

There are around 2.7 billion mobile gamers worldwide. The Earth's population is 7.8 bil-

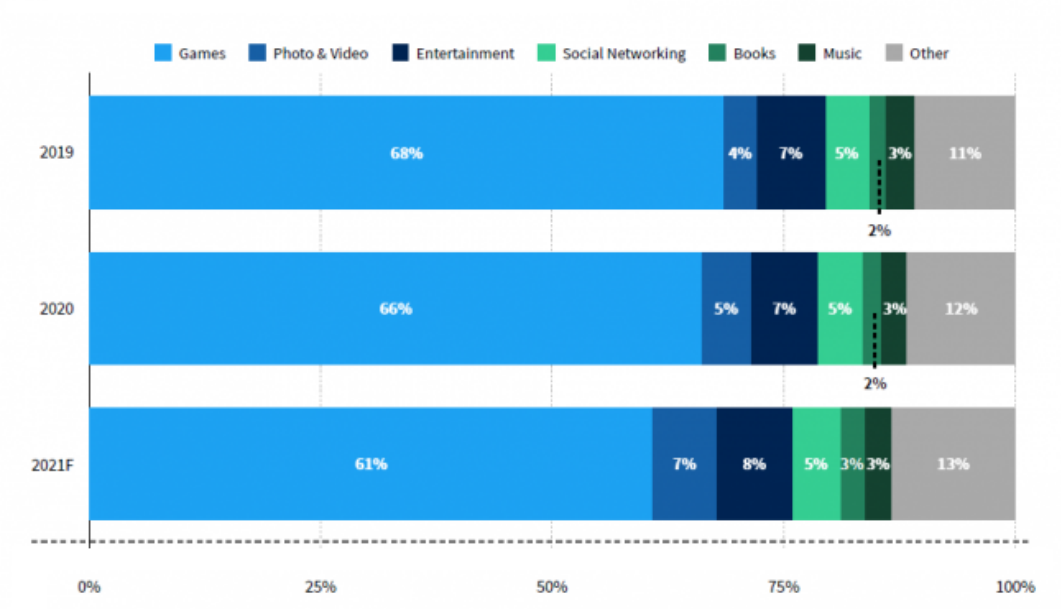


Figure 2.3: App Store Revenue by Categories

lion. Mobile games market size statistics show that of the world citizens, nearly 3 billion are avid phone game players. By the way, the modern-day gamer is clinically healthy, civically engaged, educated, and socially active. What's more, the increasing number of smartphone users ensures that the mobile games industry continues to receive massive amounts of attention from game developers worldwide. And mobile games player penetration will be 53.3% by 2025.

It's no secret that the mobile game market is booming. As shown in Figure 2.3, games account for more than half revenue in recent year. In 2020 the mobile games market generated close to $160 billion. The mobile gaming industry brought in $159.3 billion in revenue in 2020. That is a $39.2 billion increase from the previous year and up to $43.8 billion compared to 2018's income. The global mobile game revenue increased by 50% in 2020. According to mobile game statistics, in 2019, the mobile game industry had total revenue of $64.4 billion, representing a 10% YoY increase from 2018. In 2020, the coronavirus kept many at home. This subsequently led to increased activity of mobile gamers. As such, the industry

experienced a 13.3% YoY growth from 2019.

In 2021, the casual game genre is by far the most popular genre downloaded with 78% of the games downloaded falling into this category. Casual games include simple games like match-3 games, bubble shooters, hidden object games, word games, and puzzle games[21]. Considering that the game play of casual games is easy to understand and very addictive, it's easy to understand why this genre is being downloaded the most. As App Annie explains, this genre plays a significant part in turning "non-gamers" into gamers.

## 2.3 Mobile Game Testing

Application quality is a real concern for companies around the world these days. If the application has low quality, the users can easily uninstall it. 79 percent of the users will remove the application if it does not work within the first two or three attempts. For 84 percent of the users the rating of the application in app store is important. So if a game application has a bad rating in the app store, it will affect the downloads. However, due to the complexity and heavy user interactions of games, currently, game testing is mainly dependent on human testers. Most game companies adopt some ad-hoc manual testing without using systematic and automated testing solutions [3]. The ad-hoc manual testing is costly and is inefficient in discovering bugs for large games. As a result, many bugs are still discovered long after the official release. Unfortunately, most of these bugs are discovered by gamers. A study shows that popular games receive a large number of reviews each day, making it very time-consuming for developers to handle them [24] Meanwhile, Mobile games require frequent updates especially in the early-stage of their development process for adding features and fixing bugs. Even after release, companies still want to have a continuous deployment to attract users, such as releasing a deployment for every week

On the other hand, Game testing has been long recognized as a notoriously challenging task. The following list shows difficulties of the mobile game test automation.

- Domain Knowledge

  A key challenge is that game testing often requires to play the game as a sequential decision process. A bug may only be triggered until completing certain difficult intermediate tasks, which requires a certain level of intelligence. And sufficiently testing a game often requires agilely generating complicated input behaviors that can complete certain difficult intermediate game tasks. This challenge is aggravated when it is required to adapt to changes in the game during its development process.

- Non-deterministic Process The game environments are often non-deterministic and time-sensitive, making the testing process unstable and hard to reproduce. Take Candy Crush as an example, the layout of candies are always random. Some testing tools, like record and replay, are not able to handle the situation.

- Game Widgets

  Another challenge is that various customized game widgets in these apps are very difficult to recognize with most existing test frameworks. We take Google Chrome browser and Angry Birds as examples to explain the difference.

  Figure 2.4 shows test tools can recognize Android built-in widgets of Chrome. With this information, test tools can test their functions with corresponding inputs like typing, clicking.

  On the contrast, Figure 2.5 shows test tools were not able to recognize any widgets of game Angry Birds. Test tools have no idea these colorful birds are widgets and should

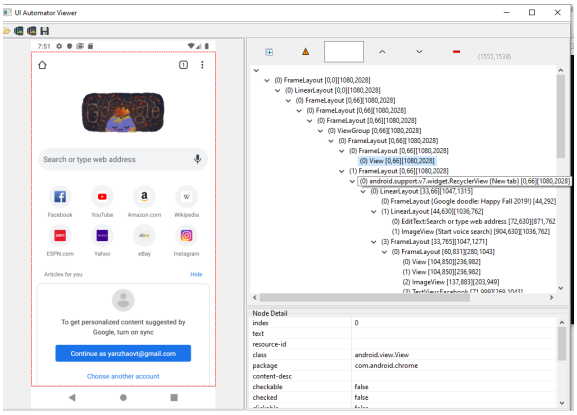


Figure 2.4: The Widgets of Chrome

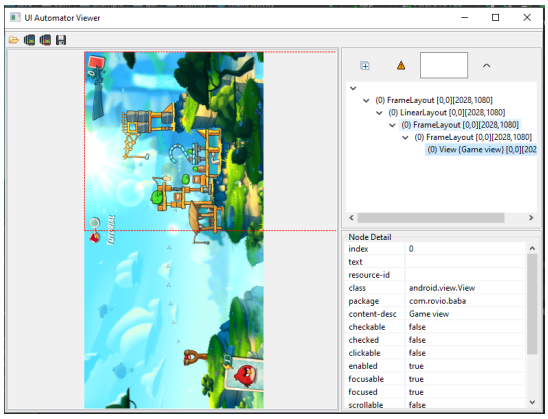


Figure 2.5: The Widgets of Angry Birds

drag and drop them.

## 2.4 tools

OPENCV[15]: OpenCV is an open source computer vision library which includes several hundreds of computer vision algorithms. It was originally developed by Intel[12]. It has C, C++, Java, Python and MATLAB interfaces and it supports Linux, Android, Mac OS and Windows.

Android Debug Bridge (ADB)[23] : is a toolkit for Android development. ADB can control emulator instances or real device from command line. It is a client-server program which consists of three components, client, server and daemon. Client runs on the development machine and it is connected to the server. Server runs also in development machine and is connected to the daemon which runs in the target instance.

GYM[40]: is a toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents everything from walking to playing games like Pong or Pinball.

# Chapter 3

# Review of Literature

The related work of our research includes automated testing for Android apps, empirical studies on automated testing for Android apps, and automated game testing.

## 3.1 Automated Testing for Android Apps

Various tools were proposed to automate testing for Android apps [4, 5, 11, 22, 24, 28, 30, 32, 33, 34, 45, 48].

Random-based tools [24, 32] test apps by generating random UI events and system events. Given an app to test, model-based tools [4, 28, 48] use static or dynamic program analysis to build a model for the app as a finite state machine (FSM). An FSM represents activities as states and models events as transitions. The built model is then used to generate events and explore program behaviors. Since random-based and model-based tools cannot trigger certain program behaviors that require for specific inputs, systematic exploration tools [5, 33] were proposed to reveal such hard-to-trigger behaviors in order to increase test coverage. In particular, ACTEve [5] is a concolic-testing tool that symbolically tracks events from the point where they originate to the point where they are handled, infers path constraints accordingly, and generates test inputs based on the inferred constraints. However, these approaches do not recognize customized UI items, neither do they observe domain-specific rules to test games.

Record-and-replay tools [22, 45] record inputs and program execution when users manually test apps, and then replay the recorded data to automatically repeat the testing scripts. The record-and-replay methodology assumes that GUIs are always organized in a deterministic way and UI items are always put at fixed locations. However, when game scenes are randomly generated and game icons move, the above-mentioned assumptions do not hold. Humanoid [30] is closely relevant to Lit . It uses deep learning to train a model with the recorded human-computer interaction traces from lots of existing apps. To test a new app $A$ based on the model, Humanoid generates input events depending on (1) $A$'s similarity with existing apps and (2) the frequent actions users take given similar GUIs. However, Humanoid cannot test games when there is no Android widget (e.g., buttons); it is insensitive to any app-specific interaction modes because the trained model focuses on the commonality between apps.

## 3.2 Empirical Studies on Android App Testing

Researchers conducted studies on automated testing for Android apps [14, 38, 42, 56, 57]. Specifically, Choudhary et al. [14] studied test-input generation tools for Android. Among the seven tools explored, Monkey [24] was found to execute or test most code. Based on the study, Zeng et al. [57] applied Monkey to WeChat—a popularly used Android app, and revealed two limitations of Monkey. First, Monkey generated many redundant events. Second, Monkey is oblivious to the locations of widgets (e.g., buttons) and GUI states. Mohammed et al. [38] recruited eight users to test five Android apps, and also applied Monkey to the same apps. They revealed that Monkey could mimic human behaviors, when apps have UIs full of clickable widgets to trigger logically independent events. However, Monkey was insufficient to test apps that require information comprehension and problem-solving skills like games. Our research was inspired by prior work. Some of our observations

and experience corroborate prior findings.

## 3.3 Automated Game Testing

Several approaches were introduced to automate game testing [9, 27, 50, 53, 54, 59]. Specifically, online testing (e.g., TorX [50] and Spec Explorer [53]) is a form of model-based testing. With online testing, testers use a specification (or model) $M$ of the system's behavior to guide testing and to detect the discrepancies between the implementation under test (IUT) and $M$. Both IUT and $M$ are viewed as interface automata to establish formal conformance relations between them. However, these testing methods require users to use domain-specific languages to prescribe models. Sikuli [47] is an open-source GUI based test automation tool. It uses techniques like "Image Recognition" and "Control GUI" to interact with elements of web pages and windows popups. Sikuli requires users to script the testing procedure for automation. In comparison, Lit does not require users to prescribe any model or script; it infers playtest tactics from user demos and uses the tactics to automate testing.

Deep learning-based approaches train models with lots of playtest data and use those models to predict the most "human-like" action in a given game scene [9, 27, 59]. For instance, Wuji [59] is the state-of-the-art tool that uses evolutionary algorithms, deep reinforcement learning, and multi-objective optimizations to perform automatic game testing. When testing a game, Wuji intends to balance between winning the game and exploring the space. Since Wuji is not available even though we contacted the authors, we could not compare Lit with it empirically. These learning-based approaches usually (1) consume lots of computing time and resources for game-specific training, and (2) require users to build DNN architectures and tune hyperparameters. When developers cannot afford the time, resource, and effort required by the usage of learning-based tools, Lit can serve as a lightweight alternative

that generates human-like inputs to test games efficiently and effectively.

# Chapter 4

# Motivating Example

## 4.1 Motivating Example

A report from Influencer Marketing Hub, a media company, pointed out that, "In 2021, the casual game genre is by far the most popular genre downloaded with 78% of the games downloaded falling into this category. Casual games include simple games like match-3 games, target shooters, hidden object games, word games, and puzzle games"[21]. Our research will focus on the dominating category and implement test automation for majority.

This section uses a target shooter game, a subcategory of casual games, as an example to intuitively explain our research. *Archery* [51] is an open-source Android game. As shown in Fig. 4.1, the game rule is to shoot a target board with a bow and arrows in order to make a great score. The game is challenging because the target board is placed randomly after each target hit. Suppose that a developer Alex wants to automatically test this game. A naïve record-and-replay approach does not help because the game scenes are generated nondeterministically. Neither random testing nor model-based testing works for two reasons. First, arrow and board are game-specific graphic objects instead of standard GUI items; existing tools cannot recognize game-specific icons. Second, a user scores only if s/he pulls the arrow, shoots the arrow towards the board, and has the arrow hit the board; existing tools blindly test games without following any scoring rule.

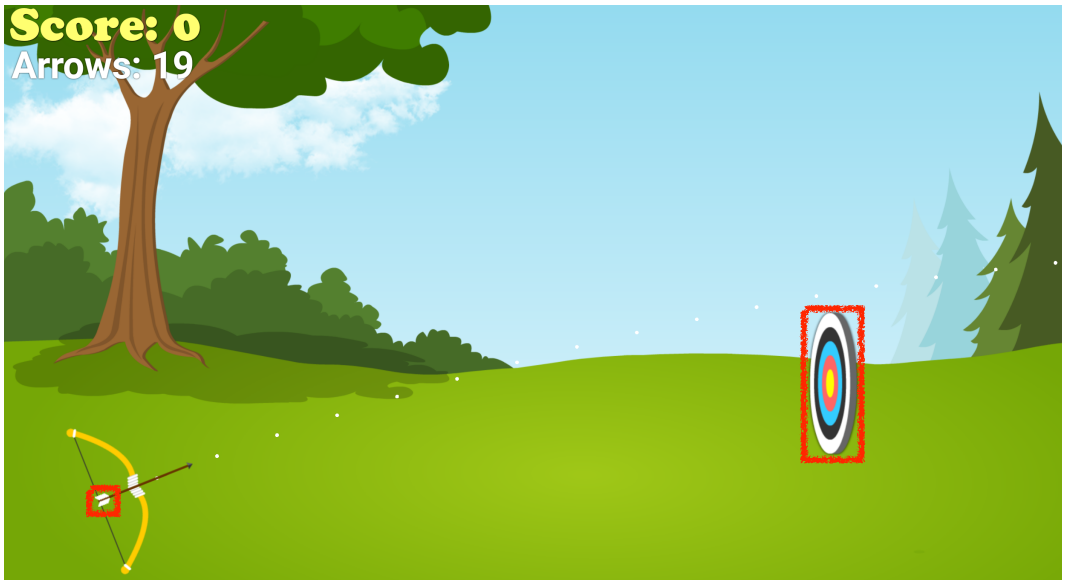


Figure 4.1: A snapshot of the game *Archery*

Our insight is that *when a user plays a game, user actions reflect the game-play tactics that are usable for automatic game testing.* Thus, we designed Lit to work in two modes: **demo mode** and **test mode**. Lit monitors users' playtest in the demo mode and mimic game play in the test mode. To use Lit , Alex should provide two inputs: (i) snapshots of all game icons and (ii) a demo for a limited timespan. For the first input, Alex can take a snapshot of the game and cut out images of board and arrow (see regions marked with **red rectangles** in Fig. 4.1); Alex may also specify the arrow to be **actionable** (i.e., manipulable) and the board to be **target** (i.e., unmanipulable). For the second input, Alex can play the game in the demo mode such that Lit records game scenes and traces Alex's finger gestures. This process continues until timeout.

**Recognition of Contexts and Actions** Based on the inputs and recorded data, Lit analyzes traces to identify Alex's action sequence and analyzes each scene snapshot to identify the context. By indexing actions and contexts based on their timestamps, Lit creates a sequence of $\langle context, action \rangle$ pairs $P = \{p_1, p_2, \ldots, p_n\}$. The information captured by a $\langle context, action \rangle$ pair $p_i$ $(i \in [1, n])$ is illustrated in Fig. 4.2. Namely, every pixel of a

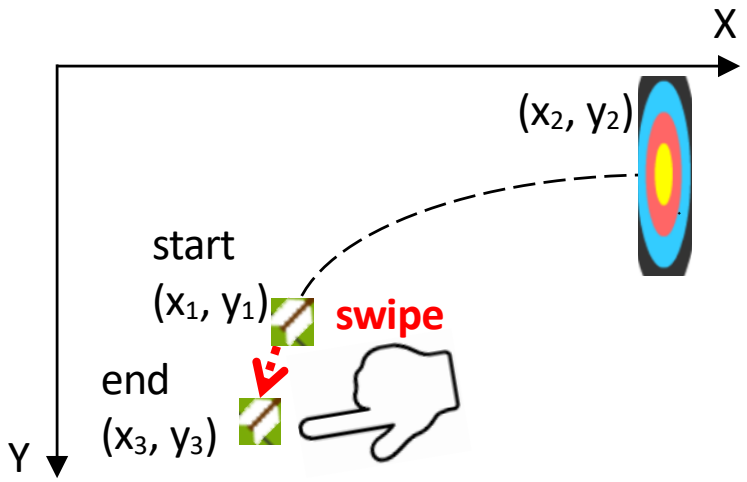


Figure 4.2: Visualizing any $\langle context, action \rangle$ pair for *Archery*

display is represented with an xy-coordinate. The location of each game icon $o$ is represented with the coordinate of $o$'s centroid, such as $(x_1, y_1)$ for arrow and $(x_2, y_2)$ for board. The swipe operation is represented with a starting point $(x_1, y_1)$ and an ending point $(x_3, y_3)$, as indicated by the **red dotted directed edge**. Our goal of tactic inference is to generalize mappings from contexts to actions.

**Tactic Inference** Based on recognized pairs, Lit analyzes three things for automated testing:

- What is the commonality between contexts?
- What kind of actions are frequently applied?
- How is each context mapped to the corresponding action?

Lit infers any common context by comparing collected contexts, and finds the board and arrow to always exist while Alex plays the game. Similarly, Lit compares all identified actions and recognizes arrow-swiping as the major action type. In our research, we differentiate between two types of swipe operations: target-oriented swipes and swipes without target. Because board is specified as target, Lit infers all arrow-swiping operations to be target-

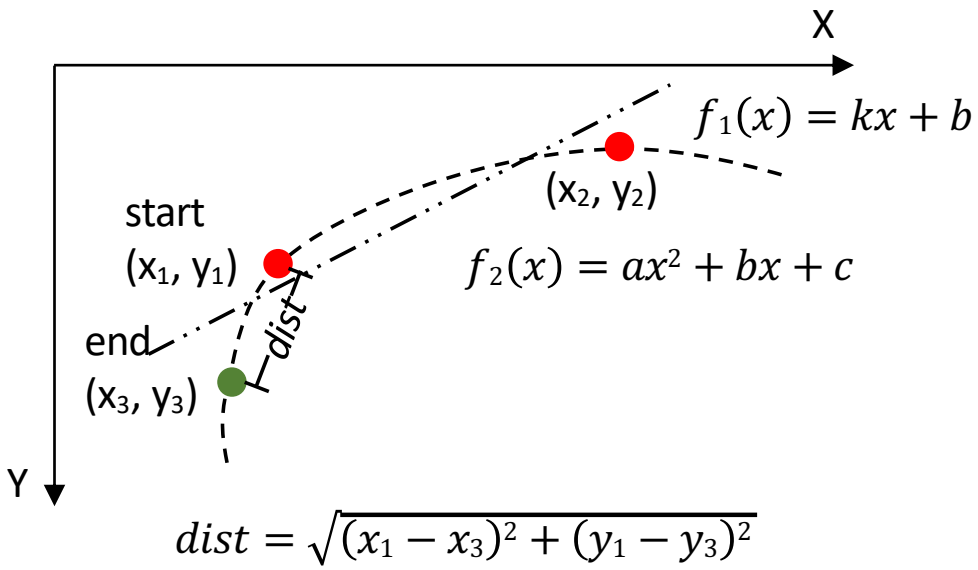


Figure 4.3: Tactic inference from any $\langle context, action \rangle$ pair

oriented.

Lit then characterizes three properties for each target-oriented swipe: (i) distance *dist*, (ii) direction *dir*, and (iii) duration *dur*. For simplicity, here we only explain the calculation of properties (i) and (ii) for any pair $p_i$. As shown in Fig. 4.3, Lit computes *dist* based on the coordinates of the start and end points. Lit calculates *dir* by fitting functions to the coordinates of all three points, because such functions reflect Alex's potential angles to shoot the arrow. Intuitively, Lit fits a linear function $f_1(x) = kx + b$ to the coordinates; it also fits a quadratic function $f_2(x) = ax^2 + bx + c$. For each linear function, Lit records $\boldsymbol{k}$ as the inferred direction because $\boldsymbol{k}$ decides the slope of $f_1$'s line. For each quadratic function, Lit records $\boldsymbol{a}$ because $\boldsymbol{a}$ decides the width and direction (up or down) of a parabola's opening [36]. To sum up, Lit generates a tactic from Alex's inputs (see Table 4.1).

Table 4.1: The tactic inferred from Alex's inputs

| | |
|---|---|
| "Abstract Context": | Actionable (arrow)<br>Target (board) |
| "Action Type": | Swipe (actionable) |
| "Swipe Distance": | $dist_1, dist_2, \ldots, dist_n$ |
| "Swipe Direction": | Linear $(k_1, k_2, \ldots, k_n)$<br>Quadratic $(a_1, a_2, \ldots, a_n)$ |
| "Swipe Duration": | 0.26 (second), 1.26, ..., 0.33) |

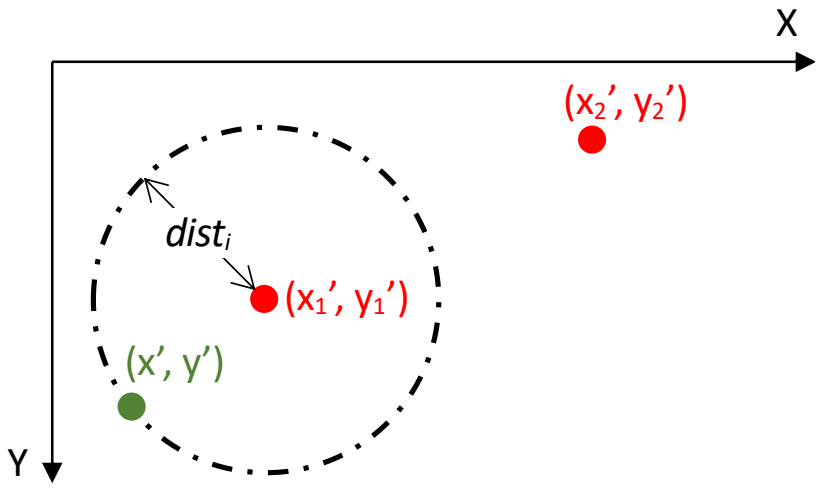

Figure 4.4: Tactic application given a random scene of *Archery*

**Tactic Application** When testing *Archery*, given a randomly generated scene, LIT first identifies all game icons (i.e., arrow and board). To swipe an arrow towards the board, LIT needs to decide the end point $(x', y')$ for the swipe operation (see Fig. 4.4). To do that, LIT randomly picks a distance $dist_i$, a direction parameter $p$, and a duration $dur_j$ from the inferred tactic. If $p = k_j$, LIT solves the equations below to get $(x', y')$:

$$\begin{cases} (y' - y'_1)^2 + (x' - x'_1)^2 = dist_i^2 \\ y'_1 - y' = k_j \times (x'_1 - x'_1) \end{cases}$$

Otherwise, if $p = a_l$, LIT solves the following equations:

$$\begin{cases} (y' - y'_1)^2 + (x' - x'_1)^2 = dist_i^2 \\ y'_1 - y' = a_l \times (x'^2_1 - x'^2) + b \times (x'_1 - x') \\ y'_2 - y'_1 = a_l \times (x'^2_2 - x'^2_1) + b \times (x'_2 - x'_1) \end{cases}$$

Due to the random combination between inferred parameters, LIT does not guarantee all arrows to hit the board. However, all generated actions are valid arrow-shootings and some

actions are highly likely to score. By diversifying the generated actions, Lit can test the game like humans, and save Alex significant amount of time and effort for manual testing.

# Chapter 5

# Approach

## 5.1 Approach

The purpose of Lit is to implement test automation for mobile games and this section describes how Lit is designed and implemented. Section 5.1 shows the high level system design and the rest sections, digs deeper into the system.

### 5.1.1 System overview

As shown in Fig. 5.1, Lit consists of two phases and seven steps to implement two phases. In this section,

**Phase I: Tactic Generalization (Demo Mode)**

- Lit records information while a user plays game $G$.

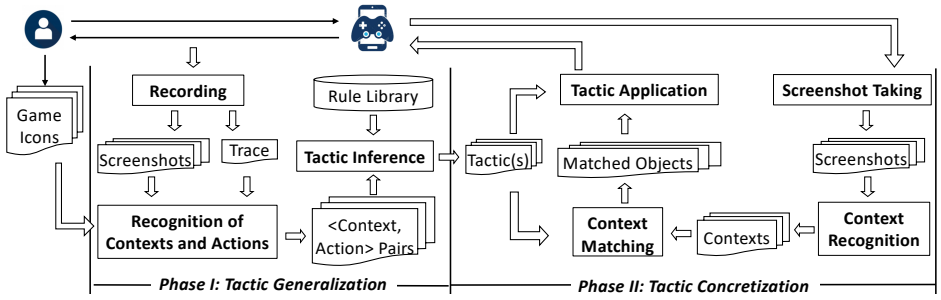


Figure 5.1: Lit consists of two phases: tactic generalization and tactic concretization

- Based on the recorded data and user-specified game icons, LIT recognizes game contexts $C$ and related actions $A$.

- LIT infers tactics by extracting abstract contexts $AC$ and action types $AT$, and revealing mappings from each context $c \in C$ to the related action $a \in A$.

**Phase II: Tactic Concretization (Test Mode)**

- When playing $G$, LIT periodically takes snapshots for game scenes; it repeats the steps below for each snapshot.

- LIT recognizes concrete context $c'$ in the snapshot.

- LIT tentatively matches $c'$ with any $ac \in AC$.

- If a match is found successfully, LIT concretizes the related tactic for action generation to play the game.

The following sections explain each step in detail. (Section 5.1.2–Section 5.1.7).

### 5.1.2 Recording

To record the screenshots and traces while a user plays game $G$, we used Android Debug Bridge (adb) [23]—a command-line tool to collect human-computer interaction data from an Android device, and to save the data to our computer. The length of demo time can affect both manual workload and automated testing effectiveness. Due to time limit, we set the length to eight minutes. During the demo, in every nine seconds, LIT reads the system time $t$, takes a screenshot, and saves it as "png.$t$". Depending on how complex a game scene is, LIT may spend 1–2 seconds creating an image file. Afterwards, LIT creates a trace file "txt.$t$" to record finger movements. At a terminal, LIT then prompts the user to taken an

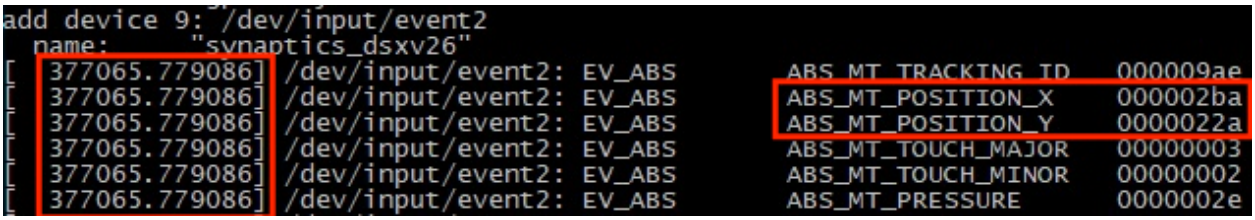


Figure 5.2: An excerpt of a trace file

action and records all corresponding input events in the trace file. In this way, screenshots and trace files can be aligned based on their common timestamps. We set the time interval to nine seconds based on our observations on (1) users' response time and (2) the cost of automatic screenshot-taking.

Fig. 5.2 shows an excerpt of a trace file. In the file, the first column lists the timestamps of events, although these timestamps cannot be mapped to the system-level timestamp $t$ mentioned above. All `ABS_MT` events report details on how an object (e.g., a finger) touches the screen and makes movements. Particularly, `ABS_MT_POSITION_X` and `ABS_MT_POSITION_Y` events show the xy-coordinates of contact points in a temporal order. When a finger moves on the screen, multiple xy-coordinates are recorded for the trajectory.

### 5.1.3 Recognition of Contexts and Actions

Lit recognizes contexts based on user-specified game icons. Currently, users are supported to specify three types of icons:

- *Actionable*—the icons that a user controls or manipulates to score (e.g., arrow in *Archery*),
- *Target*—the icons that a user does not operate but are helpful for the user to decide how to operate actionable icons (e.g., board in *Archery*), and


Figure 5.3: Exemplar function icons in *Angry Birds*

Figure 5.4: A screenshot of *AndroidLinkup* [16]

- *Function*—the icons that a user manipulates to switch between major game phases, such as moving on to the next difficulty level or retrying the current level. Fig. 5.3 lists some function icons used in *Angry Birds*.

The user-specified categorized icons serve two purposes. First, they enable Lit to generalize *context-aware* tactics. If no icon is specified, Lit infers tactics solely based on traces. Second, if the user demo presents only a subset of specified icons, the category information allows Lit to generalize inferred tactics from seen icons to unseen ones. For instance, suppose that a demo only uses two of the four function icons shown in Fig. 5.3. Lit generalizes any tactic inferred from these two icons to other same-typed icons. This approach design enables Lit to effectively infer tactics without requiring a long demo. Theoretically, the task of icon specification can be automated, but the inaccuracy of automatic icon recognition can substantially compromise Lit 's performance later on. To avoid data noise, we required users to specify icons. We expect such manual effort to be little, because game developers need to define icons anyway. In many scenarios, they can simply reuse the icons in their projects' `assets` folder as inputs.

To recognize specified icons in given screenshots, we used OpenCV (i.e., Open Source Computer Vision Library) [15] for image recognition. OpenCV can flexibly match similar but different images. Such flexibility is important for LIT to locate game icons in screenshots because the specified icons are sometimes rotated, shadowed, or darkened in game scenarios. For each recognized image, OpenCV outputs coordinates of the matched area.

A **user action** consists of one or multiple touch gestures made for a valid move in games (e.g., shooting an arrow towards the board in *Archery*). These gestures may be taps (e.g., clicks) or swipes. To recognize user actions in trace files, we built an intuitive method. Namely, we observed that the recorded event sequence for each gesture always (1) starts with `ABS_MT_TRACKING_ID 0000xxxx`, (2) ends with `ABS_MT_TRACKING_ID 0000`, and (3) has multiple `ABS_MT_POSTION_X` and `ABS_MT_POSITION_Y` events in between to show xy-coordinates of contact points. Based on this observation, LIT processes any given trace file to identify all segments. Inside each segment, suppose that the first xy-coordinate is $(x_f, y_f)$, the last xy-coordinate is $(x_l, y_l)$, and their related timestamps are separately $ts_f$ and $ts_l$. LIT then calculates two properties: distance $dist = \sqrt{(x_l - x_f)^2 + (y_l - y_f)^2}$ and duration $dur = ts_l - ts_f$'; it derives a gesture using the following heuristics:

*H1:* If $dist > 20 \&\& dur > 0.2$ second, then a swipe gesture was made.

*H2:* If $dist < 20 \| dur < 0.2$ second, a tap gesture was made.

We defined the above-mentioned heuristics by exploring different gestures, observing the recorded traces, and summarizing gesture-trace mappings. At the end of this step, LIT derives a sequence of $\langle context, action \rangle$ pairs, with each pair related to one timestamp $t$.

### 5.1.4 Tactic Inference

Given $\langle context, action \rangle$ pairs, Lit infers tactics by (1) identifying abstract contexts $AC = \{ac_1, ac_2, ...\}$ and action types $AT = \{at_1, at_2, ...\}$ and (2) calculating alternative parameters and/or functions to map each context to the related action. Namely, *each tactic consists of one abstract context, one action type, and a set of parameters and/or functions.*

To identify abstract contexts, Lit clusters collected contexts based on the number of icon types each context contains. For the *Angry Birds* game shown in Fig. 2.1, some contexts include two icon types: actionable (i.e., birds) and target (i.e., pigs), and some contexts include only one icon type: function (i.e., "Next"). Lit considers each cluster to correspond to one abstract context $ac_i$, and represents $ac_i$ with the related icon types, as shown in Table 4.1.

To identify major action types, Lit compares the actions related to each context cluster. If all or most of the actions are composed of the same gesture sequence $s$ (e.g., swipe), the inferred action type is also represented with $s$. Furthermore, in each of these $\langle context, action \rangle$ pairs, Lit tentatively maps the starting coordinate of action to game icons in the context; if the actions are always mapped to the same icon type $i$, the inferred action type is refined to $s(i)$, as shown in Table 4.1.

The major challenge for this step is: *How do we calculate concrete parameters and/or functions to map each abstract context to an action type?* To overcome this challenge, given observed user actions and related contexts for each cluster, Lit follows the rules in our predefined library (see Fig. 5.5) to infer parameters and/or functions from each $\langle context, action \rangle$ pair. The inferred data describes given certain contexts, what concrete actions were taken by users. In later steps (Sections 5.1.6 and 5.1.7), Lit reuses such data to generate actions given a new context. Namely, the inferred data establishes concrete mappings from each

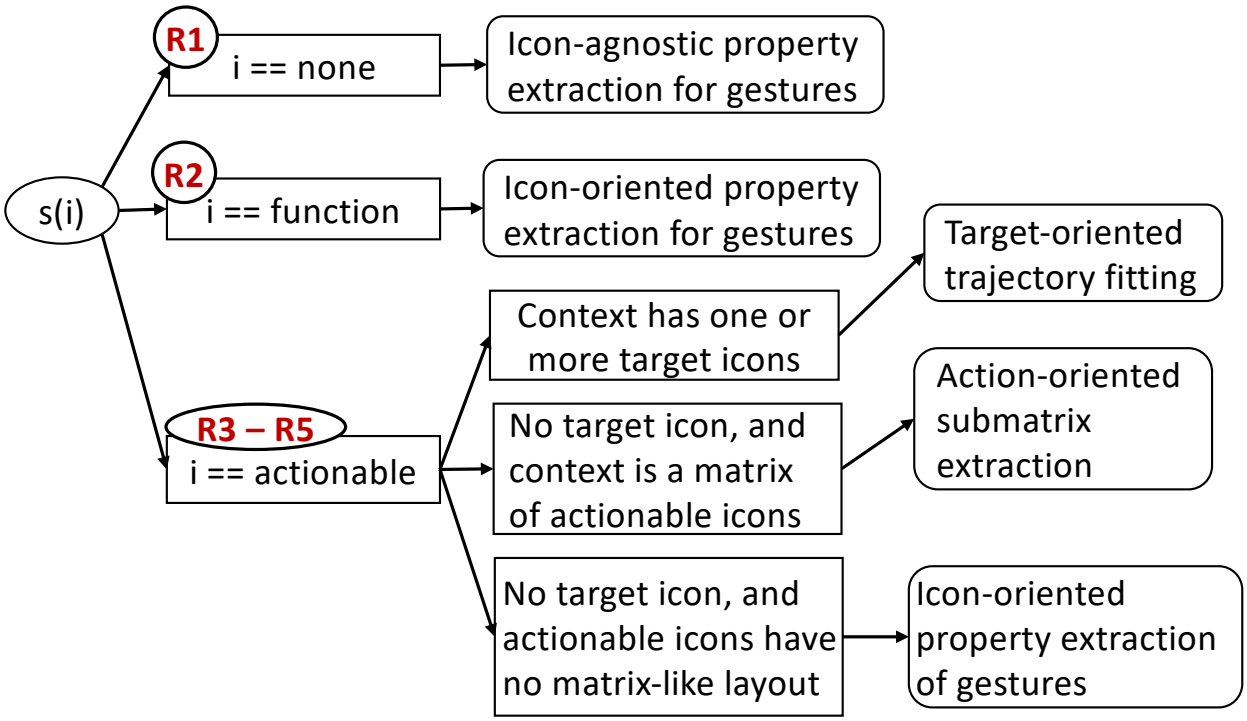


Figure 5.5: Rules defined to infer parameters/functions for context-action mappings

abstract context to the related action type.

As shown in Fig. 5.5, the library currently has five rules. Given $\langle context, action \rangle$, **R1** means that if an action was not applied to any specified icon (i.e., `i == none`), Lit extracts properties of individual gestures contained by the action. Particularly, for any tap, Lit extracts two properties: the starting coordinate $(x_f, y_f)$ and duration $dur$. For any swipe, Lit extracts three properties: distance $dist$, duration $dur$, and angle $\phi = arcsin((y_l - y_f)/dist))$. Similarly, **R2** describes that if an action was applied to a function icon (i.e., `i == function`), Lit extracts gesture properties with respect to that icon. Namely, for any tap, Lit extracts one property—$dur$; for any swipe, Lit extracts three properties: $dist$, $dur$, and $\phi$.

**R3** describes the scenarios where an action was applied to an actionable icon (i.e., `i == actionable`) and the context has one or more target icons (see Fig. 4.1). Many shooting games correspond to such scenarios [37], where users make swipe gestures. Thus, Lit extracts

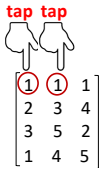


Figure 5.6: The numeric representation of Fig. 5.4

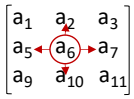


Figure 5.8: Neighbors of a matrix element

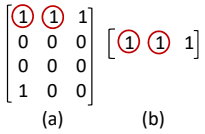


Figure 5.7: Normalized context and extracted submatrix

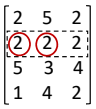


Figure 5.9: LIT creates an action for a new context

three swipe-related properties for each gesture (*dist*, *dur*, and $\phi$), and synthesizes linear and quadratic functions to fit any potential curves between the manipulated icon and a target. In the scenarios where multiple target icons coexist (see Fig. 2.1), it is hard to guess at which target a user aims when manipulating an icon; thus, LIT randomly picks a target to synthesize functions. In our implementation, LIT adopts SciPy [55] to fit both linear and quadratic functions to given coordinates. Although SciPy can synthesize arbitrarily complex functions, based on our experience, the generated linear and quadratic functions are very effective for LIT to test games. Finally, one coefficient of each synthesized function is saved for later use.

**R4** describes the scenarios where an action was applied to an actionable icon, and the context has no target icon but organizes all actionable icons in a matrix. Match3 games [35] adopt such matrix layouts to place actionable icons. As shown in Fig. 5.4, the *AndroidLinkup* game lists different types of fruits in a matrix, and a user needs to tap two fruits of the same type to eliminate them both and earn points. If we use different numbers to refer to different fruit icons, a $\langle context, action \rangle$ pair can be visualized as Fig. 5.6. We decided not to use such context as is in the inferred tactic for two reasons. First, randomly generated scenes can place fruits in arbitrary ways and the reusability of such context is quite limited

in later steps. Second, not all elements in the matrix help explain the user action. Thus, we developed an action-oriented submatrix extraction algorithm to facilitate tactic inference and application.

Based on our experience, icons in matrices are manipulated usually because they are identical to some surrounding icons. Thus, we designed an algorithm to extract an action-relevant submatrix (i.e., pattern) that reflects the commonality. In this algorithm, Lit first initializes a rectangle $sc$ based on the layout of $c$ to cover all elements in $E$. Secondly, Lit normalizes $c$ to another matrix $c_1$ as follows: if an element is identical to any member $e \in E$, the element is converted to "1"; if the element is different from all members in $E$, it is converted to "0"; otherwise, if a grid in $c$ has no element, "-1" is used. For instance, Fig. 5.7 (a) shows the normalized representation for the matrix in Fig. 5.6. Thirdly, Lit enqueues all elements in $E$. For each dequeued element $e$, Lit examines the neighbors (see Fig. 5.8). If an unprocessed neighbor $n$ corresponds to "1" in $c_1$, Lit enqueues $n$. Lit also checks whether $sc$ is large enough to cover $n$; if not, $sc$ is enlarged. This process continues until the queue is empty and $sc$ becomes stabilized.

Our algorithm returns $m$—the submatrix in $c_1$ covered by $sc$. Fig. 5.7 (b) shows the submatrix derived from Fig. 5.7 (a). Lit then infers a function $map(m) = E$ from each $\langle context, action \rangle$ pair. As what Lit does for **R2**, Lit also conducts icon-oriented property extraction for gestures. Therefore, the derived tactic includes *map* functions and icon-related gesture properties.

**R5** describes the scenarios when an action was applied to an actionable icon, and the context has no target icon or matrix-like layout. Similar to what it does for **R2**, Lit simply extracts gesture properties with respect to the manipulated icons.

### 5.1.5 Screenshot Taking & Context Recognition

These two steps reuse part of the implementation of Steps 1–2. Specifically, given game $G$, Lit periodically takes snapshots via adb, and relies on OpenCV and user-specified game icons to identify contexts. Because context is represented by the game icons extracted from a screenshot, when developers specify no game icon, Lit recognizes no context.

### 5.1.6 Context Matching

Given an identified context $c'$, Lit tries to match $c'$ with the abstract context $ac$ of any derived tactic based on (1) icon types and/or (2) matrix layouts. According to our experience, such tentative matching often succeeds. This is because Lit extracted at most dozens of abstract contexts from each demo. Those contexts could be efficiently enumerated for matching trials; they were also representative enough to illustrate game rules. In the worst case where context matching fails, Lit randomly generates an action to proceed ignoring the context.

### 5.1.7 Tactic Application

Intuitively, this step is the reverse process of tactic inference. Given a demo, tactic inference characterizes game contexts and derives a set of features to describe user actions. Correspondingly, this step leverages context characterization and derived features to randomly generate actions, and uses adb to issue those actions for playtest. Therefore, depending on the rule adopted for tactic inference, Lit applies tactics differently.

With more details, if **R1** is used for inference, Lit applies tactics by generating actions based on arbitrary property combinations between observed gestures. For instance, if a tap action is needed, Lit randomly picks a recorded coordinate $(x_f, y_f)$ and a duration $dur$ to create a tap. Similarly, if a swipe is needed, Lit creates the gesture by randomly picking $dist$, $dur$, and $\phi$ from its property sets. Lit similarly applies tactics if **R2** or **R5** is in use.

Table 5.1: The nine Android games used in our evaluation

| Game | Type (Open or Closed source) | Category | LOC | Player's Actions | Context Characteristics |
|---|---|---|---|---|---|
| *Angry Birds* [18] | C | Shooting | - | Fling (or swipe) multiple colored birds to defeat green-colored pigs in a structure or tower. | With actionable icons (i.e., birds) and target icons (i.e., pigs) |
| *Ketchapp Basketball* [6] | C | Shooting | - | Swipe the ball towards the basketball hoop. | With actionable icons (i.e., balls) and a target icon (i.e., hoop) |
| *Star Pop Magic* [20] | C | Match3 | - | Tap two or more adjacent identical stars to crush them. | With actionable icons (i.e., stars) organized in a matrix |
| *2048* [19] | O | Board | 1,692 | Swipe any point up/down/left-/right to move the tiles. When two tiles with the same number touch, they merge into one. | Without actionable or target icon |
| *Apple Flinger* [8] | O | Shooting | 14,085 | Shoot (to swipe) apples towards the enemy's base | With actionable icons (i.e., apples), but not organized in a matrix |
| *AndroidLinkup* [16] | O | Match3 | 2,102 | Tap two identical items to connect them with three or fewer line fragments and to crush them. | With actionable icons (i.e., fruits) organized in a matrix. |
| *Archery* [51] | O | Shooting | 2,833 | Shoot (or swipe) arrows towards a board. | With actionable icons (i.e., arrows) and a target icon (i.e., board) |
| *CasseBonbons* [52] | O | Match3 | 2,549 | Swipe colored pieces of candy on a game board to make a match of three or more of the same color. | With actionable icons (i.e., candies) organized in a matrix |
| *Open Flood* [43] | O | Board | 1,659 | Start in the upper left corner of the board. Tap the colored buttons along the bottom of the board to flood all adjacent filled cells with that color. | With actionable icons (i.e., buttons), but not organized in a matrix |

"-" means the data is unavailable.

When **R3** is used for tactic inference, as illustrated by Section 4.1, Lit randomly picks *dist*, direction parameter $p$, and *dur* to decide how to swipe an actionable icon with respect to a target icon.

When **R4** is used for inference, to apply tactics to the given context $c'$, Lit tentatively matches $c'$ with any extracted submatrix $m$. If there is a submatrix $m'$ in $c'$ such that (1) the elements matching 1's have the same icon index $i$ and (2) the elements matching 0's have indexes other than $i$, then Lit identifies elements for operation and creates an action by randomly mixing collected gesture properties. For instance, Fig. 5.9 presents a new context of *AndroidLinkup* that is totally different from the original context in Fig. 5.6 (a). When

matching this context with the $s$ in Fig. 5.6 (b), Lit can locate two icons and generate two taps accordingly.

# Chapter 6

# Evaluation

## 6.1 Evaluation

This section first presents our dataset and evaluation metrics. It then explains the evaluation results for Lit and other tools.

### 6.1.1 Dataset

We included nine Android games into our evaluation set (see Table 5.1); three of the games are closed-source and six games are open-source. We chose these games because they are representative, present diverse context characteristics, and require users to take various actions. With more details, users need to specify at least one function icon in each game so that Lit infers how to enter those games. Additionally, users need to specify actionable icons for some games (e.g., *CasseBonbons*), and specify both actionable and target icons for some other games (e.g., *Archery*). Each game requires for user actions like taps or swipes. In Table 5.1, column **LOC** shows the number of lines of code for each open-source game.

### 6.1.2 Metrics

Similar to prior work [14, 58], we measured code coverage of execution by different testing tools to assess their effectiveness. Theoretically, the more code is executed by a testing tool,

the better. We defined four coverage metrics:

$$Line\_Coverage = \frac{\text{\# of lines of code covered}}{\text{Total \# of lines}} \times 100\%$$
$$Branch\_Coverage = \frac{\text{\# of code branches covered}}{\text{Total \# of branches}} \times 100\%$$
$$Method\_Coverage = \frac{\text{\# of methods covered}}{\text{Total \# of Java methods}} \times 100\%$$
$$Class\_Coverage = \frac{\text{\# of classes covered}}{\text{Total \# of Java classes}} \times 100\%$$

In our implementation, we adopted JaCoCo [39] to collect coverage information. Because Jacoco uses the ASM library [10] to modify and generate Java byte code for instrumentation purpose, the above-mentioned metrics are only computable for open-source games; they are not computable for closed-source software because we have no access to the codebases. To also evaluate tools when they test closed-source software, we defined two additional metrics: *Game_Score* and *Game_Level*. *Game_Score* reflects the points earned by a testing tool after it plays a game for a period of time. We believe that the higher score a tool earns, the more likely that the tool covers more code. Similarly, *Game_Level* shows at which difficulty level a testing tool is when the allocated testing time expires; the higher level, the better.

### 6.1.3 The Effectiveness of Lit

Given a game $G$, the first author manually played $G$ for eight minutes in Lit 's demo mode, and then switched to Lit 's test mode to automatically play $G$ for one hour. Because there is randomness in the test inputs generated by Lit , we ran Lit to play each game five times such that each test run lasted for one hour. In Table 6.1, the Lit columns present average results of our tool across five runs, while the **Demo** columns shows the results achieved

Table 6.1: The comparison of *Game_Score* and *Game_Level* among user demos, LIT , Monkey, Sapienz, and RLT

| Game | *Game_Score* | | | | | *Game_Level* | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Demo | LIT | Monkey | Sapienz | RLT | Demo | LIT | Monkey | Sapienz | RLT |
| *Angry Birds* | 179,394 | **1,147,827** | 35,546 | - | - | 2 | **7** | 0 | - | - |
| *Ketchapp Basketball* | 2 | **37** | 0 | - | - | 1 | **3** | 0 | - | - |
| *Star Pop Magic* | 695 | **2,805** | 225 | - | - | 1 | **2** | 1 | - | - |
| *2048* | 332 | **2,212** | 586 | 600 | 2,142 | **-** | - | - | - | - |
| *Apple Flinger* | 38,290 | **83,718** | 0 | 0 | 81,718 | 4 | **6** | 0 | 0 | 6 |
| *AndroidLinkup* | - | **-** | - | - | - | 2 | **5** | 0 | 1 | 2 |
| *Archery* | 180 | **493** | 0 | 0 | 140 | - | **-** | - | - | - |
| *CasseBonbons* | 4,050 | **21,270** | 0 | 15 | 4,290 | 2 | **7** | 0 | 1 | 3 |
| *Open Flood* | - | **-** | - | - | - | 1 | **6** | 0 | 1 | 6 |

"-" means the data is unavailable

Table 6.2: Code coverage comparison based on open-source games among LIT , Monkey, Sapienz, and RLT

| Game | *Line_Coverage* (%) | | | | *Branch_Coverage* (%) | | | | *Method_Coverage* (%) | | | | *Class_Coverage* (%) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | LIT | Monkey | Sapienz | RLT | LIT | Monkey | Sapienz | RLT | LIT | Monkey | Sapienz | RLT | LIT | Monkey | Sapienz | RLT |
| *2048* | **81** | 80 | 77 | 81 | **68** | 65 | 62 | 68 | **84** | 86 | 84 | 84 | **78** | 78 | 78 | 78 |
| *Apple Flinger* | **53** | 19 | 9 | 53 | **52** | 17 | 7 | 51 | **60** | 27 | 16 | 60 | **59** | 32 | 24 | 59 |
| *AndroidLinkup* | **77** | 63 | 58 | 71 | **72** | 41 | 32 | 63 | **73** | 73 | 68 | 69 | **82** | 79 | 75 | 77 |
| *Archery* | **72** | 66 | 20 | 71 | **49** | 39 | 6 | 47 | **66** | 63 | 20 | 66 | **73** | 66 | 39 | 73 |
| *CasseBonbons* | **77** | 4 | 50 | 71 | **79** | 1 | 33 | 64 | **76** | 6 | 55 | 71 | **70** | 12 | 49 | 70 |
| *Open Flood* | **50** | 32 | 42 | 50 | **37** | 20 | 28 | 37 | **53** | 33 | 51 | 53 | **48** | 35 | 59 | 48 |
| Average | **68** | 44 | 43 | 65 | **60** | 30 | 28 | 54 | **69** | 48 | 49 | 65 | **68** | 50 | 54 | 66 |

by manual testing. In this table, "-" means that the data is not available. Three reasons explain such data vacancy. First, some games do not show game scores (i.e., *AndroidLinkup* and *Open Flood*). Second, some games have a single difficulty level instead of multiple levels (e.g., *Apple Flinger* and *Archery*). Third, some tools do not test the three closed-source games.

By comparing the **Demo** and LIT columns in Table 6.1, we observed LIT to consistently outperform user demos by acquiring higher scores and passing more levels. For instance, in *Angry Birds*, Demo acquired 179,394 points and stopped at the $2^{nd}$ level; LIT obtained 1,147,827 points and stopped at the $7^{th}$ level. This means that LIT did not simply record or repeat what users did. Instead, it effectively inferred tactics from demos, and applied those tactics in reaction to randomly generated scenes. Our observation also indicates that with LIT , users do not need to manually test all games comprehensively. Instead, they can test

the games for only a short period of time, and rely on Lit to spend more time similarly testing those games. The Lit columns in Table 6.2 present code coverage measurements for our tool. Among the six open-source games, Lit achieved 50–81% *Line_Coverage*, 37–79% *Branch_Coverage*, 53–84% *Method_Coverage*, and 48–82% *Class_Coverage*.

**Finding 1:** *Based on eight-minute user demos,* Lit *effectively earned game scores, passed difficulty levels, and executed lots of code within one-hour playtest.*

### 6.1.4 Effectiveness Comparison Among Tools

To assess how well Lit compares with prior work, we also applied two state-of-the-art tools to our dataset: Monkey [24] and Sapienz [34]. Monkey implements the most basic random strategy; it treats the app-under-test as a blackbox and randomly generates UI events (e.g., tap a random point in a game display). Sapienz uses multi-objective search-based testing to automatically explore and minimize test sequences, while maximizing coverage and fault revelation. Three reasons explain why we chose these two tools for experiments. First, Choudhary et al. [14] conducted an empirical study by running multiple automatic testing tools on the same Android apps, and revealed that Monkey outperformed the other tools in terms of code coverage and runtime overhead. Second, Mao et al. [34] conducted a more recent study and showed that Sapienz worked even better than Monkey. Third, similar to Lit , neither tool uses any machine learning technique.

Reinforcement learning (RL)-based tools were proposed to test games [9, 27, 59], but none of the tools is publicly available or executable[1]. To ensure the comprehensiveness and representativeness of our empirical comparison, we built a vanilla RL-based tool and refer to it as RLT (see Section 6.1.4). Among the three baseline tools, Monkey can test all games.

[1]Several open-sourced RL-based testing tools need specialized hardware and are inapplicable to mobile apps.

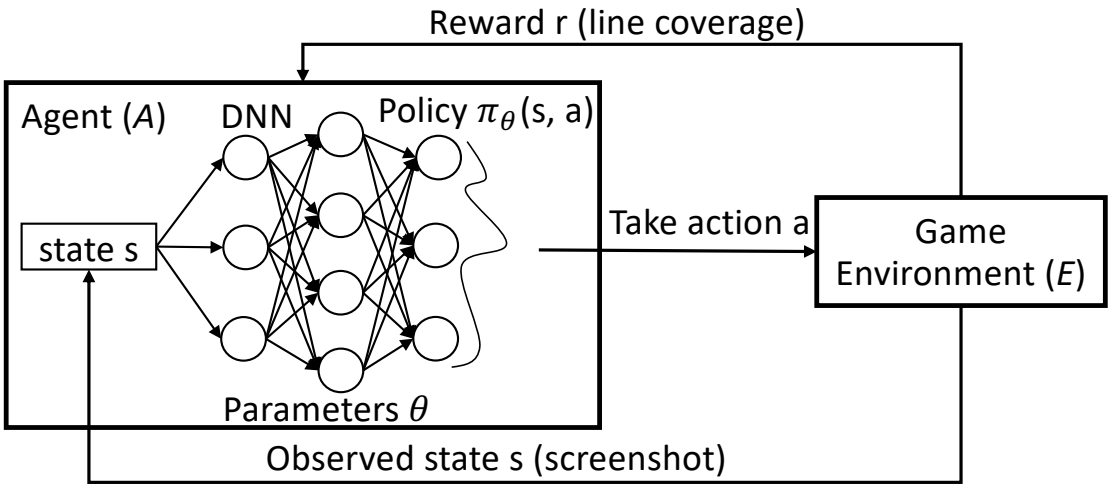


Figure 6.1: Overview of RLT—a testing tool based on RL

Sapienz only tests apps installed on the Android Emulator [26]. As the three closed-source games are not installable on the emulator, Sapienz could not test them. RLT was built to use line coverage values as rewards (see Section 6.1.4), so it is inapplicable to close-source games. Finally, we conducted two experiments with all four tools. In the first experiment, we applied each tool to every game five times, with each test run lasting for one hour; we then compared the average coverage measurements across tools. Second, we used each tool to run every game for five hours, and compared the number of runtime errors triggered.

**RLT**

We implemented RLT on top of Gym [40]—a toolkit for developing RL algorithms. Since different games have distinct rules, we programmed an RL agent for each game. As shown in Fig. 6.1, a typical RL agent (e.g., intelligent gameplayer) interacts with the environment (e.g., game) in discrete time steps. At each time $t$, the agent $A$ receives the current state $s_t$ and reward $r_t$; it then chooses an action $a_t$ from the set of available actions either randomly or based on its deep neural network, and sends $a_t$ to the environment $E$. In our implementation, a state is a game screenshot automatically captured by $A$, a reward is the line coverage output

by JaCoCo at runtime, and an action includes one or more touch gestures conducted to make a valid move in games. The goal of $A$ is to learn a **policy** from $< state, action >$ pairs that produces actions to maximize the line coverage of app execution.

To achieve the goal, we encoded all valid actions (i.e., the action set) into $A$ for individual games. For instance, for *CasseBonbons*, we encoded the swipe operations applicable to a $9 \times 9$ matrix as numbers within $[1, 9 \times 9 \times 4]$, as a swipe has four possible directions (i.e., up, down, left, and right) and is applicable to all elements. Whenever $A$ generates a number, RLT is programmed to invoke adb and manipulate the corresponding matrix element accordingly. Additionally, we also programmed $A$ to iteratively learn a deep neural network (DNN) that outputs actions given game scenes. Intuitively, in the first iteration, $A$ randomly picks actions among the encoded valid ones, and sends actions in sequence to $E$ to observe the corresponding states and rewards.

In the second iteration, $A$ trains a policy based on observed data. It then uses the trained policy together with a random-based strategy to generate actions and interact with $E$. Such iterative learning continues until timeout (e.g., after eight minutes). We implemented our DNN by following the architecture design mentioned in prior work [13, 41]. The architecture has (1) a stack of three convolution layers with a ReLU activation and followed by max-pooling layers, and (2) three fully connected layers followed by a softmax layer. The first two convolution layers separately use 32 $3 \times 3$ filters; the third convolution layer uses 64 $3 \times 3$ filters. The pool size in max pooling is $2 \times 2$. The first two fully connected layers separately have 24 and 48 neurons; the number of neurons in the third fully connected layer is equal to the number of valid actions in a game. The batch size in each iteration is 16.

**Comparison Based on *Game_Score* and *Game_Level***

As shown in Table 6.1, Lit outperformed Monkey and Sapienz by always acquiring higher scores and passing more levels; it worked at least equally well with RLT. For instance, when testing *Apple Flinger*, Lit obtained 83,718 points and arrived at Level 6 with one-hour playtest. Meanwhile, neither Monkey nor Sapienz earned any point or passed any level; RLT got 81,718 points and arrived at Level 6. Two reasons can explain why Monkey and Sapienz worked much worse than Lit . First, both tools do not know how to enter the game, and spent lots of time clicking random pixels on the display before accidentally hitting the "Play" button. Second, *Apple Flinger* requires players to swipe certain icons to hit targets. Because neither tool has such domain knowledge, they cannot properly generate swipe actions for scoring.

RLT worked better than both Monkey and Sapienz because in any agent, we hardcoded the valid action set and programmed the logic to locate actionable icons (i.e., apples) via image recognition. Such coded domain knowledge enables RLT to iteratively try different actions, observe the reward outcomes, and refine its policy. Nevertheless, RLT did not outperform Lit probably due to two reasons. First, RLT generates training data based on random actions, while Lit infers tactics from user demos that indicate not only contexts and actions, but also winning strategies of developers. Namely, there is more domain knowledge manifested by user demos than that coded into agents. Second, the DNN architecture in RLT is very complex; it repetitively processes large images of screenshots and optimizes hundreds of parameters before being stablized; while Lit only has a small library of inference rules to enumerate and explore. Therefore, Lit inferred tactics more efficiently than RLT and tested games more effectively.

### Comparison Based on Coverage Metrics

According to Table 6.2, among the four metrics, Lit achieved 60–69% average coverage, Monkey obtained 30–50% average coverage, Sapienz acquired 28–54%, and RLT got 54–66%. Lit and RLT always achieved higher coverage measurements than Monkey and Sapienz in three games: *Apple Flinger*, *Archery*, and *CasseBonbons*; all tools got similar coverage in the other games: *2048*, *AndroidLinkup*, and *Open Flood*.

Three reasons can explain the observation. First, *2048*, *AndroidLinkup*, and *Open Flood* are relatively simple and require for simple tap gestures; even Monkey and Sapienz could smoothly test those games by randomly clicking pixels on screens. Second, the other three games have more complex contexts (e.g., by including target icons or organizing actionable icons in a matrix), and/or require for carefully planned gestures. Lit and RLT have the domain knowledge to recognize icons and produce valid actions, while the other tools do not. Third, for *2048* and *Open Flood*, Monkey and Sapienz not only tested the game-playing logic, but also tested other UIs (i.e., class `SettingsActivity`). As Lit and RLT focused on game playing, they earned higher scores and passed more levels but did not necessarily cover more code.

### Comparison Based on Triggered Errors

In our experiments, Lit revealed one runtime failure in *Archery* and one program crash in *CasseBonbons*. However, none of the other tools triggered any runtime error. We reported the revealed two issues to developers by filing pull requests, but have not received any response yet.

Table 6.3: Line Coverage% of Lit with different demo time

| Game | 2 min (%) | 4 min(%) | 6 min(%) | 8 min(%) | 10 min(%) |
|---|---|---|---|---|---|
| *CasseBonbons* | 64 | 70 | 72 | 77 | 76 |
| *Open Flood* | 50 | 50 | 50 | 50 | 50 |
| *Apple Flinger* | 50 | 50 | 52 | 53 | 54 |
| *2048* | 80 | 80 | 83 | 81 | 80 |
| *Archery* | 70 | 71 | 72 | 72 | 72 |

**Finding 2:** Lit *outperformed Monkey and Sapienz by playing games more smartly; it worked slightly better than RLT even though RLT has a complex DNN design and built-in domain knowledge.*

### 6.1.5 Manual Cost and Time sensitivity

When testing games via automatic tools, we expect the manual cost of automatic tools is acceptable. The purpose of Lit is to built a light-weight approach for playtesting mobile games. So in this section, we conducted a few experiments with different demo time length from 2 min to 10 min to evaluate the manual cost.

Table 6.3 presents the line coverage that Lit achieves with different demo time. The length of demo time can affect both manual workload and automated testing effectiveness. Based on our experience, the impact of time length on testing effectiveness varies from game to game. For games with simpler contexts (e.g., Open Flood ), 1-minute user demo can lead to comparable testing coverage with a 10-minute demo. For games with complex contexts (e.g., Apple Flinger), a longer user demo (i.e., 10-minute long) is usually better.

**Finding 3:** *All the applications in our evaluation can achieve good coverage in about 10-minute demo time. Hence, the manual cost for* Lit *is acceptable.*

**Summary.** Two reasons explain why Lit worked better than Monkey and Sapienz. First, Lit generated test inputs by inferring and applying game rules based on user demos, while such

domain-specific rules are not observed or exploited by either tool. Second, Lit recognized game icons and extracted contexts, to dynamically decide how to react to random scenes. However, neither Monkey nor Sapienz recognizes manipulable icons, let alone to intelligently control those icons. Additionally, although RL-based techniques were built to test games, in our experiments, we did not see RLT to outperform Lit . Instead, RLT worked slightly worse, even though it is much harder to use. With RLT, developers need to know DNN and RL, program game-specific agents, hardcode actions, and tune various hyperparameters. In comparison, with Lit , developers only need to provide short demos and specify game icons. Lit complements existing tools by providing a different trade-off between manual effort and testing effectiveness.

# Chapter 7

# Discussion

## 7.1 Discussion

### 7.1.1 Generalization

In 2021, the casual game genre is by far the most popular genre downloaded with 78% of the games downloaded falling into this category. Casual games include simple games like match-3 games, target shooters, hidden object games, word games, and puzzle games[21].

Our rule library for tactic inference currently focuses on three major types of games: (1) board games that require no specialized consideration for context (e.g., *2048*), (2) shooting games (e.g., *Archery*), and (3) match3 games (e.g., *CasseBonbons*). we choose the game categories with the most downloads. And we noticed that prior work on automatic game testing evaluates each tool with only 1–3 games [9, 27, 59], so our data set is much larger than the state-of-the-art research.

Beside that, to better understand Lit 's potential application scope in the real world, we examined the most popular 20 games listed on Google Play [25]. 11 games fall into the categories Lit focuses on; the remaining 9 games belong to 4 categories: adventure (e.g., Roblox [12]), race (e.g., Subway Surfers [29]), casual (e.g., Pou [46]), and educational (AB-Cya! [49]).

The four extra categories mentioned above cannot be tested by Lit for various reasons. First, adventure games have various maps/tracks for players to explore. The scenery and paths along different tracks can be very different from each other, so it is difficult for Lit to infer tactics from the user demo with part of a track and to apply those tactics for playtest on other tracks. Second, race games usually switch scenes so fast that Lit cannot capture screenshots in a timely manner. Third, some casual games (e.g., Pou) provide natural-language hints to players so that they know what to do next. Currently, Lit does not have any natural-language processing capability, so it cannot play such games. Fourth, educational games (e.g., ABCya!) require players to answer questions based on their knowledge background (e.g., to solve word puzzles). Lit needs to be integrated with some databases of knowledge (e.g., dictionary) to test such games.

### 7.1.2 Widget recognition

Users need to specify game icons when testing games with Lit . Then Lit uses OpenCv to locate widgets in the screenshot and manipulates them. For open source games, game icons are available from project source code. For other games, game icons can be captured from game screenshots. The game icons for each game is a quite small set and can be accessed easily. We did not use deep neural network or other automatic object recognition because our solution is already achieve high accuracy and low cost.

# Chapter 8

# Future Work

## 8.1 Future Work

### 8.1.1 Compatibility with Other Mobile Phone Systems

It should be quite easy to add iOS support to Lit as well. The main problem is to find a way to control emulator instances or real device from command line like Android Debug Bridge (ADB) for Android OS.

### 8.1.2 Efficiency

Lit cannot work with fast-paced mobile games, because it is not fast enough. It takes 1 or 2 seconds to transfer the screenshot from mobile device to host with ADB. It is impossible to play fast-paced games with Lit , because the game can end in couple seconds without correct inputs. To overcome the slowness, one solution could be to compress the image in mobile device and then send it to host machine. The Other solution could be to tap into the Android operating system and remove ADB solution completely.

### 8.1.3 Generalization

Games are very diverse and innovative games are put on the market every day. Although we choose the game categories with most download to evaluate Lit , there are still many games

Lit hasn't covered yet. I will add 2 more types and 6 games to our evaluation set in our next experiment.

# Chapter 9

# Conclusions

## 9.1 Conclusion

The thesis focused on mobile game testing. It reviewed the motivation, key milestones and challenges. First, it highlighted why mobile game testing and test automation is harder than testing of traditional mobile applications.One reason is that covering sufficiently exercising a game often requires heavy user interactions and complicated input behaviors. Some games even contain execution paths that can be exercised only after the user has passed certain difficult intermediate tasks. Moreover, typical non-deterministic game environments and rigid time-constraints make it particularly hard to replay tests and achieve high coverage, and various customized game widgets in these apps are very difficult to recognize with most existing test frameworks. Recent research based on deep reinforcement learning techniques yields promising results in game testing. However, these techniques need a lot of computing resources and weeks of running time to train the deep learning models with millions of parameters. As mobile games usually require frequent updates especially in their early development stages, an automated, agile approach to testing these apps is important.

Thus, in this paper, we introduced a novel approach Lit to achieve a better trade-off between the two factors of game testing: the testing effectiveness and the technical complexity. To test a game app, Lit takes in user-specified game icons and a demo; it then infers tactics

from the demo and applies those tactics to automatically test the same game. With a pixel based widget detection and matching algorithm, our framework does not rely on native GUI component recognition, and support various real-world mobile games.

Our work achieves exciting results on famous commercial and popular open source mobile games. As an example for *Angry Birds*, with a 8-minute user-playing demo at difficulty level 2 of the game, the framework automatically infers the reacting tactics that covers various game states, and achieves level 7 with 1184084 game scores, whereas the state-of-the-art testing tool cannot even go through level 0. For popular open-source games, we get the observation that our reacting based framework significantly improves the testing coverage and robustifies game testing with stable testing results. In average, our framework improved line coverage by 58.14% and branch coverage by 100% over Monkey.

There is still significant space for future improvements in game testing, as we discussed in Chapter 7 and Chapter 8. As the future work, we will explore more rules in Lit and apply it to more diverse mobile games. Meanwhile, it is important to improve the efficiency of Lit as well.

# Bibliography


[1] 2k. 2K Playtesting. https://2k.com/en-US/playtest/, 2020.

[2] Activision. Activision | Playtest. https://www.activision.com/company/playtest, 2020.

[3] Saiqa Aleem, Luiz Capretz, and Faheem Ahmed. Critical success factors to improve the game development process from a developer's perspective. *Journal of Computer Science and Technology*, 31:925–950, 09 2016.

[4] 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*, pages 258–261, 2012.

[5] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, New York, NY, USA, 2012. Association for Computing Machinery.

[6] Ketch App. Ketchapp Basketball. https://play.google.com/store/apps/details?id=com.ketchapp.ketchappbasketball&hl=en_US, 2020.

[7] Appbrain. Android and google play statistics, https://www.appbrain.com/stats/stats-index.

[8] ar. apple-flinger. https://github.com/ar-/apple-flinger, 2020.

[9] S. Ariyurek, A. Betin-Can, and E. Surer. Automated video game testing using synthetic and human-like agents. *IEEE Transactions on Games*, pages 1–1, 2019.

[10] ASM. ASM. https://asm.ow2.io, 2020.

[11] Tanzirul Azim and Iulian 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 and Applications*, OOPSLA '13, pages 641–660, New York, NY, USA, 2013. Association for Computing Machinery.

[12] David Baszucki and Erik Cassel. Roblox. https://play.google.com/store/apps/details?id=com.roblox.client&hl=en_US&gl=US, 2021.

[13] Francois Chollet. Building powerful image classification models using very little data. https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html, 2016.

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

[15] Intel Corporation, Willow Garage, and Itseez. OpenCV. https://opencv.org, 2020.

[16] csuyzb. AndroidLinkup. https://github.com/csuyzb/AndroidLinkup, 2020.

[17] Teodora Dobrilova. 14 mobile gaming statistics, 2020 – insights into $2.2b gamers market, https://techjury.net/stats-about/mobile-gaming/#gref.

[18] Rovio Entertainment. Angry Birds. https://www.angrybirds.com, 2020.

[19] Cirulli Gabriele. 2048. https://github.com/gabrielecirulli/2048, 2020.

[20] In Game. Star Pop Magic. https://play.google.com/store/apps/details?id=in.game.starmagic, 2020.

[21] Werner Geyser. Mobile Gaming Industry Stats. https://influencermarketinghub.com/mobile-gaming-statistics/, 2021.

[22] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. Reran: Timing- and touch-sensitive record and replay for android. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 72–81. IEEE Press, 2013.

[23] Google. Adb, https://developer.android.com/studio/command-line/adb.

[24] Google. Monkey. https://developer.android.com/studio/test/monkey, 2020.

[25] Google. games - Android Apps on Google Play. https://play.google.com/store/search?q=games&c=apps&hl=en_US&gl=US, 2021.

[26] Google. Run apps on the Android Emulator. https://developer.android.com/studio/run/emulator, 2021.

[27] S. F. Gudmundsson, P. Eisen, E. Poromaa, A. Nodet, S. Purmonen, B. Kozakowski, R. Meurling, and L. Cao. Human-like playtesting with deep learning. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8, 2018.

[28] Shuai Hao, Bin Liu, Suman Nath, William Halfond, and Ramesh Govindan. Puma: programmable ui-automation for large-scale dynamic analysis of mobile apps. *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, 06 2014.

[29] Kiloo and SYBO Games. Subway Surfers. https://play.google.com/store/apps/details?id=com.kiloo.subwaysurf&hl=en_US&gl=US, 2021.

[30] 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)*, pages 1070–1073, 2019.

[31] Dayi Lin, Cor-Paul Bezemer, and Ahmed E. Hassan. Studying the urgent updates of popular games on the steam platform. *Empirical Software Engineering*, 22:2095–2126, 2016.

[32] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: an input generation system for android apps. In *ESEC/FSE 2013*, 2013.

[33] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. Evodroid: segmented evolutionary testing of android apps. In *FSE 2014*, 2014.

[34] Ke Mao, Mark Harman, and Yue Jia. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 94–105, New York, NY, USA, 2016. Association for Computing Machinery.

[35] match3games. Match 3 games. https://www.match3games.com, 2021.

[36] mathplanet. The graph of $y = ax^2+bx+c$. https://www.mathplanet.com/education/algebra-1/quadratic-equations/the-graph-of-y-ax-2-plus-bx-plus-c, 2020.

[37] MAXFLOW. Shooting Games. https://www.crazygames.com/c/shooting, 2021.

[38] Mostafa Mohammed, Haipeng Cai, and Na Meng. An empirical comparison between monkey testing and human testing (wip paper). *Proceedings of the 20th ACM SIG-*

*PLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, 2019.

[39] Munich Mountainminds GmbH & Co. KG. JaCoCo. https://www.eclemma.org/jacoco/.

[40] OpenAI. Gym. https://gym.openai.com, 2021.

[41] Philip Ossenkopp. Reinforcement learning – Part 2: Getting started with Deep Q-Networks. https://www.novatec-gmbh.de/en/blog/deep-q-networks/, 2018.

[42] Samad Paydar. An empirical study on the effectiveness of monkey testing for android applications. *Iranian Journal of Science and Technology, Transactions of Electrical Engineering*, 44(2):1013–1029, 2020.

[43] Gunship Penguin. open_flood. https://github.com/GunshipPenguin/open_flood/, 2020.

[44] playtestcloud. Remote playtesting for mobile games — PlaytestCloud. https://www.playtestcloud.com, 2020.

[45] Z. Qin, Y. Tang, E. Novak, and Q. Li. Mobiplay: A remote execution based record-and-replay tool for mobile applications. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 571–582, 2016.

[46] Paul Salameh. Pou. https://play.google.com/store/apps/details?id=me.pou.app&hl=en_US&gl=US, 2021.

[47] sikuli. Sikuli. https://www.softwaretestinghelp.com/sikuli-tutorial-part-1/, 2021.

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

[49] Alan Tortolani. ABCya! Games. https://play.google.com/store/apps/details?id=com.abcya.android.games&hl=en_US&gl=US, 2021.

[50] G.J. Tretmans and Hendrik Brinksma. Torx: Automated model-based testing. In A. Hartman and K. Dussa-Ziegler, editors, *First European Conference on Model-Driven Software Engineering*, pages 31–43, 12 2003.

[51] Unknown. Archery. https://github.com/kalina2002/Archery, 2020.

[52] Ismael ussac. casseBonbons. https://github.com/IsmaelCussac/casseBonbons, 2020.

[53] Margus Veanes, Colin Campbell, Wolfram Schulte, and Nikolai Tillmann. Online testing with model programs. *SIGSOFT Softw. Eng. Notes*, 30(5):273?282, September 2005.

[54] Margus Veanes, Pritam Roy, and Colin Campbell. Online testing with reinforcement learning. In Klaus Havelund, Manuel Núñez, Grigore Roşu, and Burkhart Wolff, editors, *Formal Approaches to Software Testing and Runtime Verification*, pages 240–253, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[55] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake Vand erPlas, Denis Laxalde, Josef Perktold,

Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1. 0 Contributors. SciPy 1.0–Fundamental Algorithms for Scientific Computing in Python. *arXiv e-prints*, page arXiv:1907.10121, July 2019.

[56] W. Wang, D. Li, W. Yang, Y. Cao, Z. Zhang, Y. Deng, and T. Xie. An empirical study of android test generation tools in industrial cases. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 738–748, 2018.

[57] Xia Zeng, Dengfeng Li, Wujie Zheng, Fan Xia, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. Automated test input generation for android: Are we really there yet in an industrial case? In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 987–992, New York, NY, USA, 2016. Association for Computing Machinery.

[58] C. Zhang, H. Cheng, E. Tang, X. Chen, L. Bu, and X. Li. Sketch-guided gui test generation for mobile applications. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 38–43, 2017.

[59] Y. Zheng, X. Xie, T. Su, L. Ma, J. Hao, Z. Meng, Y. Liu, R. Shen, Y. Chen, and C. Fan. Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 772–784, 2019.