A Lightweight Approach of Human-Like Playtest for Android Apps

Yan Zhao* Enyi Tang† Haipeng Cai‡ Xi Guo§ Xiaoyin Wang¶ Na Meng*
*Virginia Tech** *Nanjing University*† *Washington State University*‡
University of Science and Technology Beijing§ *The University of Texas at San Antonio*¶
yanzhao@vt.edu eytang@nju.edu.cn haipeng.cai@wsu.edu xiguo@ustb.edu.cn xiaoyin.wang@utsa.edu nm8247@vt.edu

***Abstract*—A playtest is the process in which testers play video games for software quality assurance. Manual testing is expensive and time-consuming, especially when there are many mobile games to test and every game version requires extensive testing. Current testing frameworks (e.g., Android Monkey) are limited as they adopt no domain knowledge to play games. Learning-based tools (e.g., Wuji) require tremendous manual effort and ML expertise of developers.**

This paper presents LIT—a lightweight approach to generalize playtest tactics from manual testing, and to adopt the tactics for automatic testing. LIT has two phases: tactic generalization and tactic concretization. In Phase I, when a human tester plays an Android game G for a while (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. LIT complements existing tools and helps developers test various casual games (e.g., match3, shooting, and puzzles).

***Index Terms*—automated game testing, playtest, tactic generalization, tactic concretization**

I. INTRODUCTION

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 often hire human testers for game playing. Meanwhile, the mobile gaming industry has been growing incredibly fast. According to Sensor Tower, the worldwide spending in games grew 12.8% across the App Store and Google Play in 2019 [1]. By the end of 2019, 45% of the global gaming revenue came directly from mobile games [2]. The booming mobile game industry has led to a rapid growth in game testing demand, although manual testing is always costly and time-consuming.

Researchers and developers proposed approaches to automatically test Android apps and video games, but the tool support is insufficient. For instance, random testing [4] and model-based testing [5] execute apps by generating various input events (e.g., button clicks). These approaches only recognize the standard UI controls defined by Android, such as buttons and checkboxes [6]. They cannot identify any customized UI items (e.g., the birds and pigs shown in Fig. 1), neither do they use any domain knowledge to effectively play games. Some approaches use machine learning (ML) to test games by training models [7], [8]. However, these approaches are heavyweight: they require developers to learn to design, program, and tune deep neural networks (DNN), which process is challenging and time-consuming. More importantly, such heavy workload must be repetitively fulfilled for each game, as there is no universal DNN available to test multiple games.

Fig. 1: A screenshot of the game *Angry Birds* [3]

To help general developers efficiently test Android games without using ML, we created a lightweight game testing approach—LIT. 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) 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.
3) A **game scene** is an image to display different information related to one program state (see Fig. 1). Scenes can be generated non-deterministically, so our approach should flexibly react to the changing program states.

To overcome all challenges, we developed LIT to have two

This work was supported by NSF Grants of US (CCF-1845446, CCF-2006278, and CCF-2007718), Fundamental Research Funds for the Central Universities of China (FRF-IDRY-19-023).

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 [9] 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 $AC = \{ac_1, ac_2, \ldots\}$ as well as 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 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 [4] and Sapienz [10]), 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 outperformed all tools by achieving higher test coverage and triggering more runtime errors. Specifically for *CasseBonbons* [11] (a game similar to *Candy Crush Saga* [12]), LIT achieved 79% branch coverage. On the other hand, Monkey, Sapienz, and RL-based tool separately got 1%, 33%, and 58% branch coverage. LIT triggered two runtime errors in tested games, while the other tools triggered none. Our experiments show that LIT can effectively test three categories of casual games: match3, shooting, and puzzles. As there are hundreds of games belonging to these categories [13], [14], we believe that LIT can tremendously help developers 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 contexts in tactics, and taking actions accordingly.
- We conducted a comprehensive evaluation, to empirically compare LIT with two state-of-the-art tools and one RL-based tool. LIT outperformed all tools.

At https://github.com/NiSE-Virginia-Tech/yzhao-Lit, we open-sourced our program and data.

II. MOTIVATING EXAMPLE

This section uses an example to intuitively explain our research. *Archery* [15] is an open-source Android game (see Fig. 2). The game rule is to shoot a target board with a bow and

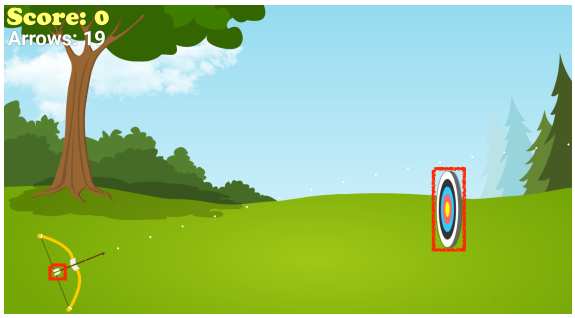

Fig. 2: A snapshot of the game *Archery*

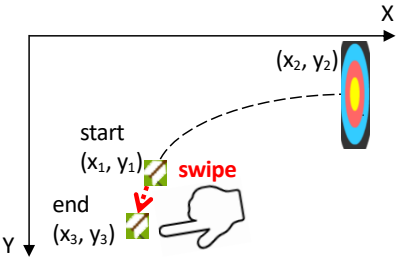

Fig. 3: Visualizing any $\langle context, action \rangle$ pair for *Archery*

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 record-and-replay approach does not quite help as the game scenes are generated nondeterministically. Neither random testing nor model-based testing works well for two reasons. First, arrow and board are game-specific icons instead of standard UI controls; existing tools cannot recognize the 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 rule.

Our insight is that *when a user plays a game, user actions reflect the gameplay 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 mimics 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. 2); 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. 3. Namely, every pixel of a 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

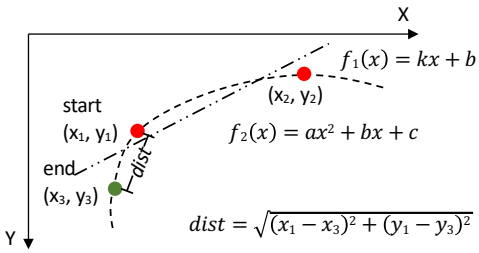

Fig. 4: Tactic inference from any $\langle context, action \rangle$ pair

TABLE I: The inferred tactic for *Archery*

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

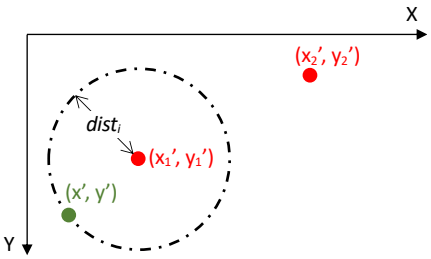

Fig. 5: Tactic application given a random scene of *Archery*

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-oriented.

LIT then characterizes three property parameters for each target-oriented swipe: (i) distance $dist$, (ii) direction dir, and (iii) duration dur. For simplicity, here we only explain the calculation of parameters (i) and (ii) for any pair p_i. As shown in Fig. 4, 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 parameter 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 [16]. To sum up, LIT generates a tactic from Alex's inputs (see Table I).

Tactic Application: When testing *Archery*, given a randomly generated scene, LIT first identifies all game icons. To swipe an arrow towards the board, LIT needs to decide the end point (x', y') for the swipe operation (see Fig. 5). 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 equation group (1) shown below to get (x', y'); otherwise, if $p = a_l$, LIT solves the equation group (2). Due to the random combination between inferred parameters and uncontrollable equation-solving procedure, 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.

$$\begin{cases} (y' - y_1')^2 + (x' - x_1')^2 = dist_i^2 \\ y_1' - y' = k_j \times (x_1' - x_1') \end{cases} \tag{1}$$

$$\begin{cases} (y' - y_1')^2 + (x' - x_1')^2 = dist_i^2 \\ y_1' - y' = a_l \times (x_1'^2 - x'^2) + b \times (x_1' - x') \\ y_2' - y_1' = a_l \times (x_2'^2 - x_1'^2) + b \times (x_2' - x_1') \end{cases} \tag{2}$$

III. Approach

As shown in Fig. 6, LIT consists of seven steps to implement two phases. In this section, we will explain each step in detail.

A. Recording

To record the screenshots and traces while a user plays game G, we used a command-line tool Android Debug Bridge (adb) [18]. The tool provides access to a Unix shell that we can use to run a variety of commands on an Android device. Specifically through the adb shell, we issued the `getevent` command [19], to collect human-computer interaction data from an Android phone and to save the data to our computer. Prior work also uses this command to collect traces [20], [21].

The length of demo time can influence 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* [22]), 1-minute user demo can lead to comparable testing coverage with a 10-minute demo. For games with complex contexts (e.g., *Angry Birds* [3]), a longer user demo (i.e., 16-minute long) is usually better. Due to the time limit and with the consideration of differences among games, we set the default length of demo time 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 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 of (1) users' response time and (2) the cost of automatic screenshot-taking.

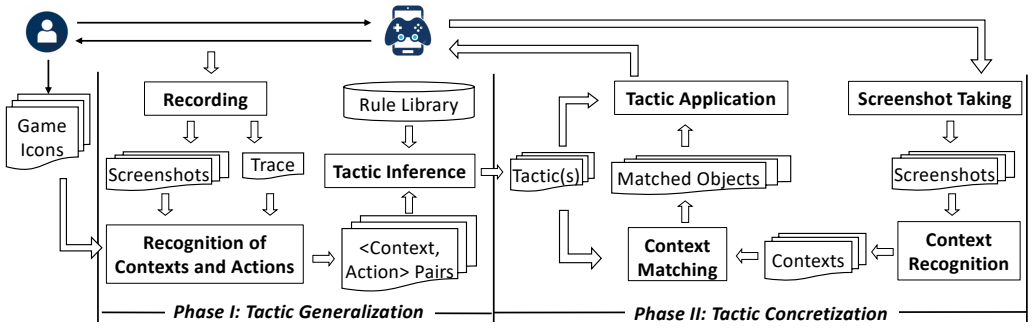

Fig. 6: LIT consists of two phases: tactic generalization and tactic concretization

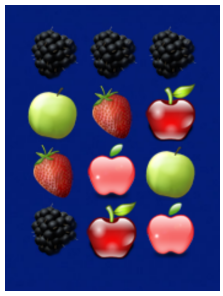

Fig. 7: A screenshot of *AndroidLinkup* [17]

```
add device 9: /dev/input/event2
  name:     "synaptics_dsxv26"
[ 377065.779086] /dev/input/event2: EV_ABS       ABS_MT_TRACKING_ID   000009ae
[ 377065.779086] /dev/input/event2: EV_ABS       ABS_MT_POSITION_X    000002ba
[ 377065.779086] /dev/input/event2: EV_ABS       ABS_MT_POSITION_Y    0000022a
[ 377065.779086] /dev/input/event2: EV_ABS       ABS_MT_TOUCH_MAJOR   00000003
[ 377065.779086] /dev/input/event2: EV_ABS       ABS_MT_TOUCH_MINOR   00000002
[ 377065.779086] /dev/input/event2: EV_ABS       ABS_MT_PRESSURE      0000002e
```

Fig. 8: An excerpt of a trace file

Fig. 9: Exemplar function icons in *Angry Birds*

Fig. 8 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.

B. 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
- *Function*—the icons that a user manipulates to switch major game phases, such as moving on to the next difficulty level or retrying the current level. Fig. 9 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. 9. 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. We expect the manual effort of specifying game icons to be little, because game developers need to define icons anyway. In many scenarios, they can simply reuse or tailor the icons in their projects' `assets` folder for inputs.

To recognize specified icons in given screenshots, we used OpenCV (i.e., Open Source Computer Vision Library) [9] for image recognition. Specifically, OpenCV offers a function `cv.matchTemplate(...)` [23] to search for the location of a template image in a larger image; we configured the function to use `TM_CCOEFF_NORMED` as the comparison method. The function 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** includes one or more touch gestures made for a valid move in games (e.g., shooting an arrow towards the board in *Archery*). Our research focus on two types of gestures: taps (i.e., clicks) and swipes. To recognize user actions in trace files, we took an intuitive approach. Namely, we observed that the recorded event sequence for each gesture always (i) starts with `ABS_MT_TRACKING_ID 0000xxxx`, (ii) ends with `ABS_MT_TRACKING_ID 0000`, and (iii) 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, a swipe was made.
H2: If $dist \leq 20$ || $dur \leq 0.2$ second, a tap gesture was made.

We defined the two heuristics by experimenting with different gestures in several games, observing the recorded traces, and summarizing gesture-trace mappings. Our heuristics are similar to those of prior work [24]. This step outputs $\langle context, action \rangle$ pairs, with each pair for one timestamp t.

C. Tactic Inference

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

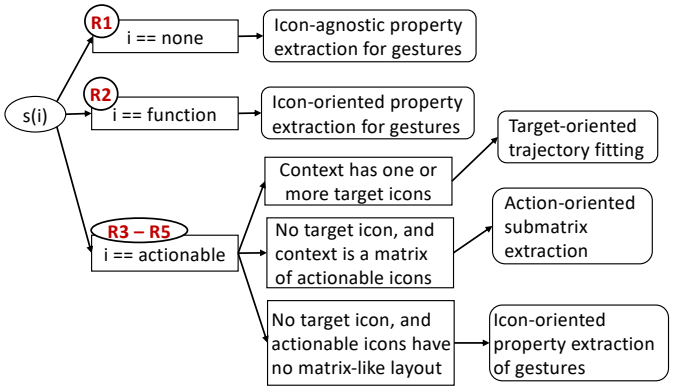

Fig. 10: Rules defined to infer parameters/functions for context-action mappings

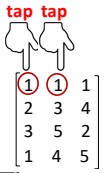

Fig. 11: The numeric representation of Fig. 7

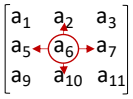

Fig. 13: Neighbors of a matrix element

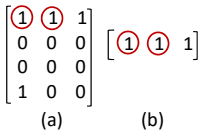

Fig. 12: Normalized context and extracted submatrix

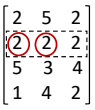

Fig. 14: LIT creates an action for a new context

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. 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 I.

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. Here, "most" means that a major action type corresponds to (i) at least 90% of all actions, or (ii) at least 10 actions if the majority takes up less than 90%. Furthermore, in each $\langle context, action \rangle$ pair, 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 I.

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. 10) to infer parameters and/or functions from $\langle context, action \rangle$ pairs. The inferred data describes given certain contexts, what concrete actions were taken by users. In later steps (Sections III-E and III-F), LIT reuses such data to generate actions given a new context. Namely, the inferred data establishes concrete mappings from each abstract context to the related action type.

According to Influencer Marking Hub, in 2021, the casual game genre is the most popular genre, with 78% of downloaded games falling into this category [25]. Casual games often involve simple tactics and shorter sessions, requiring less learned skills [26]. Typical casual games include match3 (e.g., *Candy Crush*), shooting (e.g., *Angry Birds*), word games, and puzzles (e.g., *2048*). We focus on casual games because of their popularity and simplicity. Based on our experience with casual games, we defined a library to include five rules (see Fig. 10), which are used to infer frequently applied tactics.

R1 targets the puzzle games that require users to make tap or swipe gestures; users can score even if they blindly take actions without recognizing any icon. LIT extracts properties of gestures for each action. Particularly, for any tap, LIT extracts two parameters: the starting coordinate (x_f, y_f) and duration dur. For any swipe, LIT extracts three parameters: distance $dist$, duration dur, and angle $\phi = arcsin((y_l - y_f)/dist))$.

R2 infers the tactics that (i) start a game or (ii) switch game phases (i.e., difficulty levels). It 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 ϕ.

R3 is defined for swipe-based shooting games [14], in which users swipe an actionable icon (i.e., `i == actionable`) and the context has one or more targets (see Fig. 2). In such scenarios, LIT extracts three swipe-related properties for each gesture ($dist$, dur, and ϕ), and synthesizes linear and quadratic functions to fit any potential curves between the swiped icon and a target. In the scenarios where multiple target icons coexist (see Fig. 1), it is hard to guess at which target a user aims; thus, LIT randomly picks a target to synthesize functions. In our implementation, LIT adopts SciPy [27] 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 focuses on match3 games [13], which lay actionable icons in matrix-like structures and match identical icons in certain places. As shown in Fig. 7, the *AndroidLinkup* game puts 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 fruits, a $\langle context, action \rangle$ pair can be visualized as Fig. 11. We decided not to use such context as is in the inferred tactic for two reasons. First, randomly generated scenes can put fruits in arbitrary ways and the reusability of such context is limited. 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.

Algorithm 1: R4–action-oriented submatrix extraction

Input: context matrix c (identified by LIT), matrix elements (i.e., actionable icons) involved in the action $E = \{e_1, e_2, \ldots\}$
Output: The extracted submatrix m

Initialize $sc = Rectangle(minX, minY, maxX, maxY)$ to cover all elements in E
Normalize c to another matrix c_1 based on E
$q.enqueueAll(E)$
while $q \neq \emptyset$ **do**
    $e = q.dequeue()$
    for *each unprocessed neighbor* n *of* e **do**
        if n *has the value "1"* **then**
            $q.enque(n)$
            Enlarge the rectangle sc as needed to cover n
return the submatrix of c_1—m—that is covered by sc

Based on our experience, icons in matrices are manipulated usually because they are identical to some surrounding icons. Thus, we designed Algorithm 1 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. 12 (a) shows the normalized representation for the matrix of Fig. 11. Thirdly, LIT enqueues all elements in E. For each dequeued element e, LIT examines the neighbors (see Fig. 13). 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. 12 (b) shows the submatrix derived from Fig. 12 (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 property parameters.

R5 is defined for some puzzle games, where actionable icons are specified but context has no target or matrix-like structures. Similar to what it does for **R2**, LIT simply extracts gesture properties with respect to the manipulated icons.

D. 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, relying 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.

E. 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; these contexts can be efficiently enumerated for matching trials. In the worst case where context matching fails, LIT randomly generates an action to proceed ignoring the context.

F. 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 rules 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 parameter 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 ϕ from its parameter sets. LIT similarly applies tactics if **R2** or **R5** is in use. When **R3** is used for tactic inference, as illustrated by Section II, 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 (i) the elements matching 1's have the same icon index i and (ii) 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. 14 presents a new context of *AndroidLinkup* that is totally different from the original context in Fig. 11 (a). When matching this context with the s in Fig. 11 (b), LIT can locate two icons and generate two taps accordingly.

IV. EVALUATION

There are two research questions in our evaluation:

- **RQ1**: How effectively can LIT test game apps?
- **RQ2**: How does LIT compare with widely used tools?

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

A. Dataset

We included nine Android games into our evaluation set (see Table II): three closed-source games and six open-source ones. These games were chosen because they present diverse context characteristics and require users to take various actions. With more details, users need to prescribe at least one function icon in each game so that LIT infers how to enter those games. 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 II, column **LOC** shows the number of lines of code for each open-source game.

TABLE II: The nine Android games used in our evaluation

Game	Type (Open or Closed source)	Category	LOC	Player's Actions	Context Characteristics
Angry Birds [3]	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 [28]	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 [29]	C	Match3	-	Tap two or more adjacent identical stars to crush them.	With actionable icons (i.e., stars) organized in a matrix
2048 [30]	O	Puzzle	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 [31]	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 [17]	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 [15]	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 [11]	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 [22]	O	Puzzle	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.

B. Metrics

Similar to prior work [32], [33], 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 adopted two coverage metrics:

$$Line\_Coverage = \frac{\text{\# of lines of code covered}}{\text{Total \# of lines}} \times 100\%$$

$$Branch\_Coverage = \frac{\text{\# of code branches covered}}{\text{Total \# of branches}} \times 100\%$$

In our implementation, we used JaCoCo [34] to collect coverage information. Because JaCoCo uses the ASM library [35] to modify and generate Java byte code for instrumentation purpose, the above-mentioned metrics are only computable for open-source games. **Code coverage is not computable for closed-source software because we have no access to the codebases**. To compare tools based on closed-source software, we adopted two additional metrics: $Game\_Score$ and $Game\_Level$. $Game\_Score$ reflects the points earned by a testing tool after it plays a game for awhile. 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.

C. 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 so that each test run lasted for one hour. In Table III, the **LIT** columns show average results of our tool across five runs. The **Demo** columns show the results 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 (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 III, 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. 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 IV present code coverage measurements for our tool. Among the six open-source games, LIT achieved 50–81% $Line\_Coverage$ and 37–79% $Branch\_Coverage$.

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.*

D. 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 [4] and Sapienz [10]. Monkey implements the most basic random strategy; it treats the app-under-test as a blackbox and randomly generates UI events (e.g., by tapping or swiping a random pixel). 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. [32] 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. [10] 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 [7], [8], [36], but none of the tools is publicly available or executable with Android apps. To ensure the comprehensiveness and representativeness of our empirical

TABLE III: 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	**2,492**	-	-	-	-	-
Apple Flinger	38,290	**83,718**	0	0	14,290	4	**6**	0	0	3
AndroidLinkup	-	-	-	-	-	2	5	0	1	1
Archery	180	**493**	0	0	20	-	-	-	-	-
CasseBonbons	4,050	**21,270**	0	15	2,110	2	**7**	0	1	1
Open Flood	-	-	-	-	-	1	**6**	0	1	3

"-" means that the data is unavailable. For each game, we bolded the highest game score and highest game level.

TABLE IV: Code coverage comparison based on open-source games among user demos, LIT, Monkey, Sapienz, and RLT

Game	$Line\_Coverage$ (%)					$Branch\_Coverage$ (%)				
	Demo	**LIT**	Mon-key	Sapi-enz	RLT	Demo	**LIT**	Mon-key	Sapi-enz	RLT
2048	74	**81**	80	77	**81**	63	**68**	65	62	67
Apple Flinger	50	**53**	19	9	49	50	**52**	17	7	50
AndroidLinkup	70	**77**	63	58	70	63	**72**	41	32	61
Archery	63	**72**	66	20	33	33	**49**	39	6	22
CasseBonbons	60	**77**	4	50	64	52	**79**	1	33	58
Open Flood	36	**50**	32	42	49	33	**37**	20	28	**37**
Average	59	**68**	44	43	58	49	**60**	30	28	49

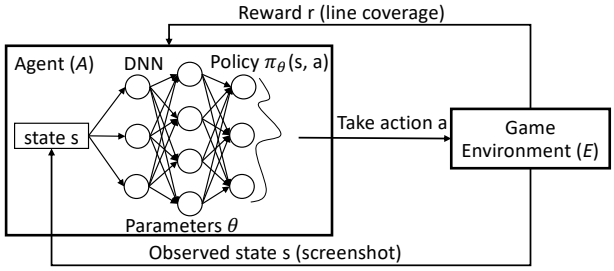

Fig. 15: Overview of RLT—a testing tool based on RL

comparison, we built a vanilla RL-based tool and refer to it as RLT (see Section IV-D1). Among the three baseline tools, Monkey can test all games. Sapienz only tests apps installed on the Android Emulator [37]. 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 IV-D1), 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.

1) RLT: We built RLT on top of Gym [38]—a toolkit for creating RL algorithms. Because the nine games have distinct icon sets and icon positions, it is infeasible to program a single universal RL agent for all games. Thus, **we programmed an RL agent for each game by hardcoding the icon set, icon positions, and specialized ways to click function icons.** As shown in Fig. 15, a typical RL agent (e.g., intelligent gameplayer) interacts with the environment 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 action set either randomly or based on its deep neural network (DNN), 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 computed by JaCoCo, and an action is a tap or swipe applied to an actionable icon. The goal of A is to learn a **policy** from $\langle state, action \rangle$ pairs that produces actions to maximize the line coverage.

To achieve the goal, we encoded a uniform action set into A for all games. The action set includes two types of actions: tap and swipe. To randomly generate an action, RLT first generates a random number. If the number is odd, it creates a tap; otherwise, it produces a swipe by further randomly generating (1) the end position/coordinate and (2) duration of the swipe gesture. RLT then invokes adb to interact with the tested app accordingly. Additionally, we programmed A to iteratively learn a DNN that outputs actions given game scenes. Intuitively, in the first iteration, A randomly picks actions in the encoded action set, 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 to interact with E. In the third iteration, A refines its policy based on the observed data in the second iteration, and continues generating actions for interactions. Such iterative learning continues until timeout (e.g., after eight minutes). We implemented our DNN by following the architecture design of prior work [39], [40]. 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 3×3 filters; the third convolution layer uses 64 3×3 filters. The pool size in max pooling is 2×2. 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.

2) Comparison Based on $Game\_Score$ and $Game\_Level$: As shown in Table III, LIT outperformed Monkey and Sapienz by always acquiring higher scores and passing more levels. 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. Among the six open-sourced games, LIT

outperformed RLT when testing five games (except for *2048*).

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 the domain knowledge, they cannot properly generate swipe actions for scoring. RLT outperformed Monkey and Sapienz because in each agent, we hardcoded the game-specific icon set, icon positions, and tapping actions for function icons; we also defined a universal action set for all agents. Such coded domain knowledge enables RLT to iteratively try different actions, observe the reward outcomes, and refine its policy.

RLT worked worse than LIT in most scenarios for two reasons. First, RLT derives and refines policies based on actions randomly applied to icons, 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 hardcoded into agents; to learn the unspecified knowledge, RLT has to go through many iterations to well train its policy. Second, the DNN architecture in RLT is very complex; it repetitively processes large images of screenshots and optimizes hundreds of parameters before being stabilized. As we trained RLT for only eight minutes (i.e., the same length with the demo time), it is possible that RLT was not trained sufficiently and it worked less effectively than LIT.

3) Comparison Based on Coverage Metrics: According to Table IV, LIT achieved 68% line coverage and 60% branch coverage on average; Monkey got 44% and 30% on average; Sapienz's values are 43% and 28%; RLT acquired 58% and 49%. LIT achieved higher coverage measurements than other tools; it worked similarly to RLT for *2048* and *Open Flood*.

Two reasons can explain the observation. First, *2048* and *Open Flood* are relatively simple and require for tap gestures; even Monkey and Sapienz could smoothly test those games by randomly clicking pixels on screens. Second, the other four games have more complex contexts (e.g., by including target icons or organizing actionable icons in a matrix), and/or require for carefully planned swipe gestures. LIT managed to infer the tactics, and adopted those tactics to generate high-quality swipe gestures. Nevertheless, with in eight-minute training, RLT was unable to create a policy smart enough to generate as many meaningful actions as LIT does.

4) 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.

Finding 2: *On average,* LIT *outperformed Monkey and Sapienz by playing games more smartly; it outperformed RLT although RLT has a complex DNN design and the agent programming hardcodes a lot more domain knowledge.*

Discussion. LIT outperformed RLT, although we programmed nine specialized agents inside RLT to separately test the nine games. By improving agent programming and optimizing hyperparameter settings, AI experts may be able to create better RL-based tools to outperform LIT. However, LIT can serve as a better tool in the following circumstances. First, developers have little or no expertise in artificial intelligence, and cannot program or optimize agents independently. Second, developers have insufficient computing resources to thoroughly train a deep-learning model for each game app.

V. THREATS TO VALIDITY

Threats to External Validity: All inferred tactics and empirical findings mentioned in this paper are limited to our experiment dataset. Our rule library for tactic inference currently focuses on three major types of games: (1) basic puzzle 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 noticed that prior work on automatic game testing evaluates each tool with only 1–3 games [7], [8], [36], so our dataset is much larger than the state-of-the-art research. In the real world, we found hundreds of games belonging to (2) and (3) [13], [14], which fact implies the wide application scope of LIT.

To better understand LIT's potential application scope in the real world, we examined the most popular 20 games listed on Google Play [41]. 11 games fall into the categories LIT focuses on; the remaining 9 games belong to 4 categories: adventure (e.g., Roblox [42]), race (e.g., Subway Surfers [43]), pet (e.g., Pou [44]), and educational (ABCya! [45]).

The four extra categories mentioned above cannot be tested by LIT for various reasons. First, adventure games have 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, pet games (e.g., Pou) provide natural-language hints to players, guiding them to look after pets. Currently LIT does not have any natural-language processing capability. Fourth, educational games (e.g., ABCya!) require players to answer questions based on their knowledge background (e.g., word spelling skills). LIT needs to be integrated with some databases of knowledge (e.g., dictionary) to test such games.

VI. RELATED WORK

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

A. Automated Testing for Android Apps

Various tools were proposed to automate testing for Android apps [4], [5], [10], [20], [46]–[54].

Random-based tools test apps by generating random UI events and system events [4], [48]. Given an app to test, model-based tools use static or dynamic program analysis to

build a model for the app as a finite state machine (FSM) [5], [49], [52]. 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 the program behaviors that require for specific inputs, systematic exploration tools were proposed to reveal such hard-to-trigger behaviors in order to increase test coverage [46], [50], [54]. In particular, ACTEve [46] 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, and creates 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 record inputs and program execution when users manually test apps, and then replay the recorded data to automatically repeat the testing scripts [20], [51]. 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 randomly move, the above-mentioned assumptions do not hold. Humanoid [53] 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.

B. Empirical Studies on Android App Testing

Researchers conducted studies on automated testing for Android apps [32], [55]–[58]. Specifically, Choudhary et al. [32] studied test-input generation tools for Android. Among the seven tools explored, Monkey [4] was found to execute or test most code. Based on the study, Zeng et al. [55] 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. [57] 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 idea was stimulated by prior work. Some of our observations and experience corroborate prior findings.

C. Automated Game Testing

Several approaches were introduced to automate game testing [7], [8], [36], [59]–[61]. Specifically, online testing (e.g., TorX [59] and Spec Explorer [60]) is a form of model-based testing. With online testing, testers use a specification (or model) M of the system's expected behavior to guide testing, and to detect any discrepancy 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 [62] 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 [7], [8], [36]. For instance, Wuji [7] 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 we were unable to execute Wuji even though we contacted the authors for help, 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, or 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.

VII. CONCLUSION

As the mobile game market grows rapidly, there is an increasing demand for advanced testing methods to efficiently test games. Manual testing is expensive and time-consuming, and existing automatic tools are either too simple to test games or too complex for general developers to use. When developers have little domain knowledge of ML and limited resources (i.e., time and computation), we believe lightweight testing methods based on user demos to be more cost-effective. 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. Our evaluation shows exciting results of LIT; it also evidences the strength of rule-based tactic inference. In the future, we will conduct a larger-scale evaluation of LIT, and include more inference rules into LIT to further improve the tool capability.

ACKNOWLEDGMENT

We thank reviewers for their valuable feedback. We also thank Weihao Zhang for his involvement in the project.

REFERENCES

[1] "The mobile games market is getting bigger – and not just for the top ten," https://www.gamesindustry.biz/articles/2020-02-03-the-mobile-games-market-is-getting-bigger-and-not-just\-for-the-top-ten, 2020.

[2] "Mobile gaming is a $68.5 billion global business, and investors are buying in," https://techcrunch.com/2019/08/22/mobile-gaming-mints-money/, 2019.

[3] "Angry Birds," https://www.angrybirds.com, 2020.

[4] "Monkey," https://developer.android.com/studio/test/monkey, 2020.

[5] 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*, 2012, pp. 258–261.

[6] "Android - UI Controls," https://www.tutorialspoint.com/android/android_user_interface_controls.htm, 2020.

[7] 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)*, 2019, pp. 772–784.

[8] 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)*, 2018, pp. 1–8.

[9] "OpenCV," https://opencv.org, 2020.

[10] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, pp. 94–105. [Online]. Available: https://doi.org/10.1145/2931037.2931054

[11] "casseBonbons," https://github.com/IsmaelCussac/casseBonbons, 2020.

[12] "Candy Crush Saga," https://king.com/game/candycrush, 2020.

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

[14] "Shooting Games," https://www.crazygames.com/c/shooting, 2021.

[15] "Archery," https://github.com/kalina2002/Archery, 2020.

[16] "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.

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

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

[19] "Getevent," https://source.android.com/devices/input/getevent, 2021.

[20] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "Reran: Timing- and touch-sensitive record and replay for android," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, pp. 72–81.

[21] Y. Hu, I. Neamtiu, and A. Alavi, "Automatically verifying and reproducing event-based races in android apps," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, pp. 377–388. [Online]. Available: https://doi.org/10.1145/2931037.2931069

[22] "open_flood," https://github.com/GunshipPenguin/open_flood/, 2020.

[23] "Template Matching," https://docs.opencv.org/4.x/d4/dc6/tutorial_py_template_matching.html, 2021.

[24] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Humanoid: A deep learning-based approach to automated black-box android app testing," 2020.

[25] "20 Mobile Gaming Statistics That Will Blow You Away — Mobile Gaming Industry Stats," https://influencermarketinghub.com/mobile-gaming-statistics/, 2021.

[26] "GDC '08: Are casual games the future?" https://www.cnet.com/tech/gaming/gdc-08-are-casual-games-the-future/, 2018.

[27] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. Jarrod Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. Carey, İ. Polat, Y. Feng, E. W. Moore, J. Vand erPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and S. . . Contributors, "SciPy 1.0–Fundamental Algorithms for Scientific Computing in Python," *arXiv e-prints*, p. arXiv:1907.10121, Jul. 2019.

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

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

[30] "2048," https://github.com/gabrielecirulli/2048, 2020.

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

[32] 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)*, Nov 2015, pp. 429–440.

[33] 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)*, 2017, pp. 38–43.

[34] "JaCoCo," https://www.eclemma.org/jacoco/.

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

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

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

[38] "Gym," https://gym.openai.com, 2021.

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

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

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

[42] "Roblox," https://play.google.com/store/apps/details?id=com.roblox.client&hl=en_US&gl=US, 2021.

[43] "Subway Surfers," https://play.google.com/store/apps/details?id=com.kiloo.subwaysurf&hl=en_US&gl=US, 2021.

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

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

[46] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2393596.2393666

[47] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 641–660. [Online]. Available: https://doi.org/10.1145/2509136.2509549

[48] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: an input generation system for android apps," in *ESEC/FSE 2013*, 2013.

[49] S. Hao, B. Liu, S. Nath, W. Halfond, and R. Govindan, "Puma: programmable ui-automation for large-scale dynamic analysis of mobile apps," 06 2014.

[50] R. Mahmood, N. Mirzaei, and S. Malek, "Evodroid: segmented evolutionary testing of android apps," in *FSE 2014*, 2014.

[51] 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)*, 2016, pp. 571–582.

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

[53] 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)*, 2019, pp. 1070–1073.

[54] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury, "Time-travel testing of android apps," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 481–492. [Online]. Available: https://doi.org/10.1145/3377811.3380402

[55] X. Zeng, D. Li, W. Zheng, F. Xia, Y. Deng, W. Lam, W. Yang, and T. 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*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, pp. 987–992. [Online]. Available: https://doi.org/10.1145/2950290.2983958

[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)*, 2018, pp. 738–748.

[57] M. Mohammed, H. Cai, and N. Meng, "An empirical comparison between monkey testing and human testing (wip paper)," *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, 2019.

[58] S. Paydar, "An empirical study on the effectiveness of monkey testing for android applications," *Iranian Journal of Science and Technology, Transactions of Electrical Engineering*, vol. 44, no. 2, pp. 1013–1029, 2020. [Online]. Available: https://doi.org/10.1007/s40998-019-00270-y

[59] G. Tretmans and H. Brinksma, "Torx: Automated model-based testing," in *First European Conference on Model-Driven Software Engineering*, A. Hartman and K. Dussa-Ziegler, Eds., 12 2003, pp. 31–43.

[60] M. Veanes, C. Campbell, W. Schulte, and N. Tillmann, "Online testing with model programs," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, p. 273?282, Sep. 2005. [Online]. Available: https://doi.org/10.1145/1095430.1081751

[61] M. Veanes, P. Roy, and C. Campbell, "Online testing with reinforcement learning," in *Formal Approaches to Software Testing and Runtime Verification*, K. Havelund, M. Núñez, G. Roşu, and B. Wolff, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 240–253.

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