Wuji: Automatic Online Combat Game Testing Using Evolutionary Deep Reinforcement Learning

Yan Zheng[1,⁎], Xiaofei Xie[2,⁎], Ting Su[2], Lei Ma[3], Jianye Hao[1,🖂], Zhaopeng Meng[1],
Yang Liu[2], Ruimin Shen[4], Yingfeng Chen[4], Changjie Fan[4]

[1] *College of Intelligence and Computing, Tianjin University, China.*
[2] *Nanyang Technological University, Singapore.*
[3] *Kyushu University, Japan.*
[4] *Fuxi AI Lab, Netease, Inc., Hangzhou, China.*

***Abstract*—Game testing has been long recognized as a notoriously challenging task, which mainly relies on manual playing and scripting based testing in game industry. Even until recently, automated game testing still remains to be largely untouched niche. 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. The recent success of deep reinforcement learning (DRL) sheds light on advancing automated game testing, without human competitive intelligent support. However, the existing DRLs mostly focus on winning the game rather than game testing. To bridge the gap, in this paper, we first perform an in-depth analysis of 1349 real bugs from four real-world commercial game products. Based on this, we propose four oracles to support automated game testing, and further propose *Wuji*, an on-the-fly game testing framework, which leverages evolutionary algorithms, DRL and multi-objective optimization to perform automatic game testing. *Wuji* balances between winning the game and exploring the space of the game. Winning the game allows the agent to make progress in the game, while space exploration increases the possibility of discovering bugs. We conduct a large-scale evaluation on a simple game and two popular commercial games. The results demonstrate the effectiveness of *Wuji* in exploring space and detecting bugs. Moreover, *Wuji* found 3 previously unknown bugs[1], which have been confirmed by the developers, in the commercial games.**

***Index Terms*—Game Testing, Artificial Intelligence, Deep Reinforcement Learning, Evolutionary Multi-Objective Optimization.**

I. Introduction

Gaming has evolved into an all-around entertainment phenomenon. The game market is growing rapidly. According to *Global Games Market Report* [31] from NewZoo in 2018, there are more than 2.3 billion active gamers in the world, of which 46% spend money on games. The game market around the world reaches around $138 billion and is expected to more than $180 billion in 2021. With so many users, the quality of games becomes especially important. A game bug may bring bad gaming experience, even cause financial losses for gamers or game companies. Game companies not only put a lot of efforts to detect and fix the bugs but also suffer from losing users. Hence, an early-stage testing is of great importance in discovering bugs and guaranteeing the game quality.

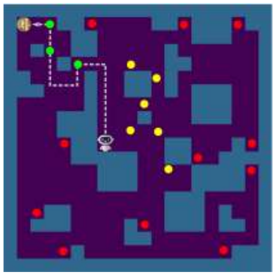


Fig. 1: Block maze with bugs (red, green and yellow dots)

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]. Semi-automatic script-based testing can improve efficiency. However, it still requires substantial manual efforts to develop the scripts. In addition, the ad-hoc scripting is also costly and usually focuses on main scenarios of the game following a pre-defined strategy. From the internal report of the *NetEase, Inc.*, there are 30 testers for testing even one game (which was used as the benchmark in our evaluation). The direct loss caused by the bugs in this game is about $2 million each year. A systematic and automated testing technique towards better quality assurance is urgent in the game industry [25].

Automatic testing techniques have been widely studied, such as search-based testing [4], [16], coverage-guided fuzzing [1] and symbolic execution [8]. 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. Some techniques, such as random-based testing (Monkey [17]) and

⁎ Equal contribution. 🖂 Corresponding author.

[1] The video reproduction of the bugs can be found in our website [41].

model-based testing (Stoat [35]), have been proposed for testing GUI applications. These techniques improve the code coverage by covering the sequences of different events during UI change. However, in a game, events in each UI are usually fixed (i.e., the skills), and it is very easy to cover all events inside game UIs. These techniques are not effective in game testing since they are not "smart" to accomplish the complicated goal of the game. For example, Fig. 1 shows the Block Maze game in which a robot hunts for gold. Bugs will be triggered if the robot reaches one of the dots. The yellow dots are easily reachable by Monkey and Stoat since they are near the initial place. However, the red and green dots are more difficult to reach as the robot needs to find a better path.

Recently, deep reinforcement learning (DRL) has achieved great success in automatic game playing (e.g., AlphaGo [13] on the board game, OpenAI on Atari games [29], and OpenAI-five on the large-scale real-time strategy games Dota2 [14]). This brings an opportunity for automated game testing with the intelligent support of DRL. However, the challenge is that the current DLR focuses on finding a better solution to accomplish the mission (e.g., obtaining higher rewards) rather than testing games [44]–[47]. For example, in Fig. 1, DRL will learn a better strategy (the dotted arrow) that will earn the gold quickly. Hence, the green dots can be reached but other dots (e.g., the red dots) are hard to cover. In addition, to capture various types of bugs (e.g., glitches, gaming balance) during playing the game, the test oracle is needed.

This paper performs a two-stage study towards addressing the game testing challenges. In the first stage, we conduct an empirical study towards understanding the characteristics of bugs in real-world games. We analyzed 1,349 real bugs from four commercial online games and classify the bugs into five categories based on their manifestation, i.e., *crash bugs*, *stuck bugs*, *logical bugs*, *game balance bugs* and *user experience bugs*. Based on the manifestation of different types of bugs, we proposed four oracles, that facilitate the detection of such bugs. At the second stage, we develop an automated game testing framework, named *Wuji*, to test the games and discover different types of bugs. The objective of *Wuji* is to detect game bugs by exploring states as much as possible. To achieve the goal, *Wuji* not only considers accomplishing the mission, but also considers searching different directions. These two strategies complement each other. Consider Fig. 1, the strategy of accomplishing mission helps reach the green dots that are difficult to cover by the general random strategy. The strategy of exploration will help reach the red dots that may not be located in the mainline of the game.

To test a game, *Wuji* first trains a set of agents (i.e., deep neuron networks) that can play the game automatically. The agent policies will always be updating towards exploring the states and accomplishing the mission. Based on proposed oracles, *Wuji* performs an on-the-fly testing while constantly training the agent policies (rather than the usual AI solutions that can be used only after training). We propose a technique by a combination of evolutionary algorithms (EA), multi-objective optimization (MOO) and deep reinforcement learning (DRL). EA and MOO mainly contribute to ***exploring game space*** while DRL contributes to ***accomplishing the mission***. Specifically, in each evolution iteration, the population first performs crossover and mutation, considering the two objectives (i.e., exploring states and accomplishing mission). Then, each policy in the population will be separately trained by the DRL for enhancing the capability of accomplishing the mission. Next, the MOO together with a *non-dominated sorting* is used to select a better set of policies as the seed population of the next evolution. By optimizing the population iteratively, more states of the game could be explored and tested.

We implemented *Wuji* and evaluated the usefulness by applying it to one simple game and two commercial massively multiplayer online games (MMOGs) including a mobile game and a PC game. MMOG is one of the most popular games nowadays, where thousands of players join together. The two commercial games have more than 4 million and 880 thousand peak daily players, respectively. The results demonstrated that *Wuji* significantly outperforms random testing, EA alone and DRL alone in terms of bug detection and state coverage. Moreover, *Wuji* also successfully discovered previously unknown bugs in the two active commercial games.

To the best of our knowledge, this is the first automated game testing work on real-world online combat games. The main contributions of this paper are summarized as follows:

1) We perform an empirical study to characterize game bugs by analyzing 1,349 real bugs from four industrial games. We propose four test oracles for four types of bugs.
2) We propose an on-the-fly automated testing framework based on evolutionary deep reinforcement learning.
3) We implement *Wuji* and evaluate its effectiveness in two real-world games. Moreover, we find 3 previously unknown bugs, which are confirmed by the developers.

II. Background and Problem Definition

A. Game Playing

Game playing is a sequential decision-making process where a player needs to continuously make decisions and take actions based on received observations before the game ends. Such a problem can be modeled as a Markov Decision Process (MDP) that is a classical formalization of sequential decision making [38]. Fig. 2 depicts an MDP which shows a typical overall interaction of playing the game. In general, the MDP consists of 5 elements - *Agent*, *Environment*, *State*, *Action* and *Reward*. An environment is what an agent interacts with (e.g., the game). The agent utilizes a specific policy for interacting with the environment. The state is the observation of the environment which is usually different at different timestamps. The action is a set of possible decision (e.g., moves, fire) the agent can make. The reward (e.g., scalar value) is the feedback by which we measure the success or failure of an agents actions with regard to achieving some goals (e.g., winning the fight or achieving points).

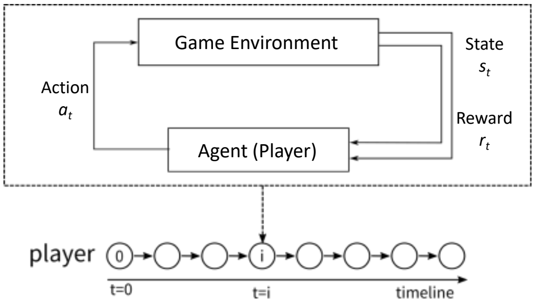


Fig. 2: The interaction of an agent and game environment.

For example, to be specific, given the state s_t at time t, the agent selects an action a_t to interact with the game environment and receives a reward r_t from the environment. The environment turns into a new state s_{t+1}, affecting the agent selection of the next action. The environment and agent interact continually until the game ends, and gives rise to a *trajectory* as follows:

$$(s_0, a_0, r_0, \cdots, s_i, a_i, r_i, \cdots) \tag{1}$$

B. Problem Definition

A large online combat game (OCM) usually contains many instance dungeons or scenarios. It is challenging to test the whole game one time, the common way is to test each scenario separately. Another overwhelming challenge is that the OCM is usually played by multiple players. In this initial game testing work, we mainly focus on game testing with one agent.

Based on the Equation 1, we define a game as follows:

Definition 1 (Game)**.** A game G is a function $(S, R) = G(A)$, where A is a sequence of actions from an agent, S is a sequence of states and R is a sequence of rewards. For simplicity, we also use $\|G(A)\|_S$ and $\|G(A)\|_R$ to represent the outputs of the game G (i.e., S and R), respectively.

A test case of the game G is a sequence of actions A, which explore and drive the game into different states. Note that G is often a non-deterministic function because of the randomness in the game (e.g., the randomness in the environment, the randomness effect of an action).

Suppose $\|G(A)\|_R := (r_1, \ldots, r_n)$, the rewards of A on the game G (denoted as $RW(A, G)$) is computed as follows:

$$RW(A, G) = \sum_{t \in 1 \ldots n} \gamma^t r_t \tag{2}$$

where $\gamma \in [0, 1]$ is a discount factor.

Definition 2 (Policy)**.** A policy π is a function, $a = \pi(s)$, that maps a state to an action, where s and a represent a state and an action, respectively.

The policy is the strategy that an agent adopts to decide the next action based on the current state. For example, a policy can be the strategy from the human or a trained deep neural network that can make the decision.

Definition 3 (Game State)**.** A state s of the game G is a fixed-length vector $\langle v_0, v_1, \ldots, v_n \rangle$, where each value of the vector represents a certain aspect of the state of the game.

The vector of a state represents the current status of the game, and it differs for different games. For example, the position of the player character, the health points, etc.

Definition 4 (Game Testing)**.** For a game G, a game tester T can be defined as a function $S, B = T(G, \Pi)$, where Π is a set of policies to interact with G, S is a set of states explored by T and B is a set of bugs by T.

A game tester employs different policies and tries to explore different states of the game, where a bug might hide. The goal of the game testing is to generate test cases (i.e., valid actions) that can reach the diverse states, on which bugs will be triggered. To check whether a specific state triggers a bug, the test oracle is usually needed. However, due to the uniqueness of game software, the test oracle also quite differs from other kinds of software, which is yet not well-studied.

Thus, our first goal is to perform an empirical study and investigate the potential useful test oracles for checking game bugs. Then, we propose effective testing policies (i.e., Π) for effective state exploration to cover those bugs might hide, which is then captured by our proposed test oracle.

III. Empirical Study

Massively multi-player online game (MMOG) is one of the most popular games nowadays, where thousands of players join together. To achieve better user experience, game companies strive to detect and fix as many bugs as possible before each version release. Online combat is among the core and most complicated elements of MMOGs, which is the key challenge for MMOG testing. In this paper, we mainly focus on such scenarios but our approach is general to other types of games.

To better understand the characteristics of existing game bugs, we first perform an empirical study that analyzes 1,349 bugs from the bug lists of four popular commercial OCMs that are actively used by over 10 million peak daily players in total. All these games are developed and maintained by *Netease, Inc.*[2]. Informed by the knowledge of game developers, we categorize them into 5 different types according to their consequences. We order them by the fixing priority from highest to least as follows:

- **Crash bugs**, which leads the entire game to crash and exit. For example, the zero-dividing problem, memory leak issue or recursive function calls will result in game crash. Crash bugs are severe defects and should be fixed first.
- **Stuck bugs**, which freezes the game. Game players are unable to continue the interactions. Such bugs may be caused when the player enters into an abnormal state, getting stuck the whole game. Although there is no crash, the game cannot response to any player's actions.

[2] https://www.neteasegames.com/

TABLE I: Bug distribution in four commercial games.

Games	L10	NSH	L12	LH01	Total
Crash	89	16	10	30	145(**10.75%**)
Stuck	22	7	9	27	65(**4.82%**)
Logical	379	25	22	62	488(**36.17%**)
Balance	7	2	2	1	12(**0.89%**)
Experience	476	88	20	55	639(**47.37%**)
Total	973	138	63	175	1349 (**100.00%**)

- **Logical bugs**, which often do not break the game like the previous two types of bugs, but lead to unexpected results (e.g., errors on score computation). This type of bugs are usually caused by incorrect implementation of game logic.
- **Gaming balance bugs**, which disturb the gaming balance between human and computer-aided players (e.g., too strong or too weak), and thus violating the designers' original expectation. This type of bug is specific but critical to combat games.
- **User experience bugs**, which downgrade user experience, e.g., missing video (e.g., ray tracing) and audio effects, failing to give proper hints, text typos, etc.

Among these types of bugs, game developers give higher priority for the first four types of bugs, since they directly affect the business logic, stability, and correctness of games. These bugs are often strived to be fixed before releasing games. Table I shows the detailed classification results of 1,349 bugs. We can observe that these four critical types of bugs account for half of all bugs in 4 studied games. Based on these, in this paper, we define four types of oracles, with the intention to detect these four typical bugs respectively, which are urgent to be fixed with a higher priority:

- **Oracle 1 (Crash)**, the crash bugs can be automatically captured by monitoring whether the game process is terminated.
- **Oracle 2 (Stuck)**, the stuck bugs can be automatically captured by: 1) monitoring the change of the game screen, or 2) monitoring the states of the game, i.e., comparing the difference between the current state and the previous n consecutive states. In this paper, we used the state-level check since monitoring the game screen is too heavy and not scalable. Specifically, during game playing, the moving averaged state $\bar{s} = \sum_{i=t-n+1,\ldots,t} s_i/n$ is calculated by averaging n previous consecutive states. If the difference between current observed state s and $\bar{s}$ is less than a threshold $\mathcal{T}_s$, a stuck potential bug is discovered.
- **Oracle 3 (Logical)**, the logical bugs are difficult to be discovered automatically. It requires specific assertions to reveal logical bugs.
- **Oracle 4 (Gaming Balance)**, the balance bugs can be captured by checking whether one player achieves unexpected performance (e.g., too strong or weak) beyond a predefined threshold. Gaming balance bugs are subjective and need the confirmation of game designers. We will not consider this type of bugs in this paper.

IV. Automated Game Testing

In this section, we first describe the overview of our approach. Then, we present our detailed techniques and algorithms for automated game testing.

A. Overview

The general idea of *Wuji* is to generate effective policies that explore more states of the game where the bugs potentially hide. In company with the proposed oracle, such bugs are more likely to be detected. Fig. 3(a) shows the simplified workflow of *Wuji*. Given a game, *Wuji* randomly initializes a population of policies P (i.e., a specified number of DNNs), after which the main testing iteration starts on the basis of an evolutionary process. Notably, all individuals of the first generation are randomly initialized (DNNs with random weights).

From the high level, in each iteration, *Wuji* adopts the evolutionary multi-objective optimization (EMOO) and deep reinforcement learning (DRL) to achieve efficient game testing. EMOO is used for improving the diverse state exploration while DRL facilitates improving the capability of accomplishing the mission of the game. The combination of EMOO and DRL altogether makes *Wuji* potentially be able to complete more missions and explore more states of the game.

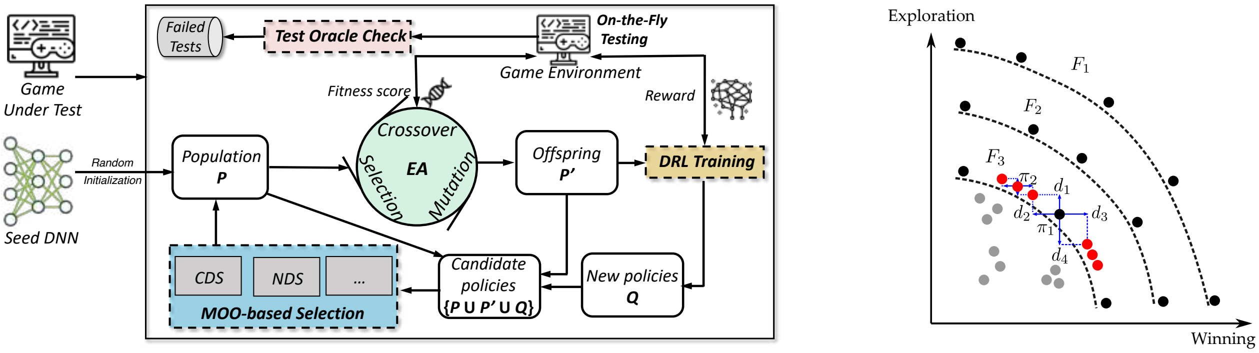
The policies (i.e., DNNs) P are randomly selected by the binary tournament selection [27] to perform crossover and mutation to evolve the population to P', towards generating policies that are capable of exploring more states (Section IV-C). The fitness score of each policy is computed by playing and interacting with the game. A challenge is that, if the policy is not smart and is always unable to complete the mission of the game, then the policy would not explore the states after the mission. To mitigate the problem, *Wuji* leverages a DRL-based technique which aims at improving the capability of accomplishing the mission of the game (Section IV-D). Each obtained DNN policy in P' will be further trained by DRL, and an advanced population Q that is good at completing missions is obtained.

Afterwards, *Wuji* performs a multi-objective based selection form the population P, P', and Q that could survive in the next iteration (Section IV-E). In particular, the non-dominated sorting (Algorithm 2) and crowding distance sorting (Algorithm 4) are used to sort the population $P \cup P' \cup Q$.

Wuji continually evolves its testing policy by EMOO and DRL iteration by iteration. At the same time, the policies obtained from both procedures are leveraged to perform on-the-fly game testing, until the given testing time or resource budge exhausts. *Wuji* eventually outputs a set of failed traces that facilitate reproducing the bugs and a set of obtained competitive testing policies.

B. Multiple Objectives

To be specific, *Wuji* iteratively evolves the population towards achieving two objectives: exploring more states and winning the game (i.e., complete the mission). We capture these two objectives by exploration score and the winning score, respectively, which are measured as follows. For the game G, we use the policy π to play the game for m times, after which m sequence of actions (i.e., $A_0, A_1, \ldots, A_m$) will be generated. Each of these sequences represents the actions



(a) The workflow of *Wuji*

(b) Visualization of MOO-based selection

Fig. 3: The overview and workflow of Wuji and MOO-based selection.

taken by an agent and the corresponding game environment state changes step by step. Then, the exploration score of π on the game G is computed as follows:

$$ES_G^{\pi} := \frac{1}{m}\sum_{i=1}^{m}(size(\|G(A_i)\|_S)) \tag{3}$$

The wining score of π on the game G is calculated as:

$$RS_G^{\pi} := \frac{1}{m}\sum_{i=1}^{m}(RW(G, A_i)) \tag{4}$$

Intuitively, the exploration score is computed as the average of the total number of states in m runs of the game. The winning score is computed as the average of the total rewards in m runs of the game. The higher the exploration score, the more states the policy will potentially explore. The higher winning score, the policy has a higher chance to accomplish more intermediate missions.

To better capture the capability relation of different policies, we define the policy domination relation as follows:

Definition 5 (Pareto domination)**.** A policy π_0 dominates another policy π_1 (denoted as $\pi_0 \succ \pi_1$) on game G if $(ES_G^{\pi_0} \geq ES_G^{\pi_1} \wedge RS_G^{\pi_0} > RS_G^{\pi_1})$ or $(ES_G^{\pi_0} > ES_G^{\pi_1} \wedge RS_G^{\pi_0} \geq RS_G^{\pi_1})$.

C. Crossover and Mutation

Algorithm 1 shows the crossover and mutation on a population P. $|P|$ represents the total number of the policies in P. *Wuji* adopts a binary tournament selection [27] to select two policies (i.e., DNNs) π_f and π_m for the crossover (Line 3-10). Specifically, two candidates π_1 and π_2 are picked randomly. If π_1 dominates π_2, π_1 will be selected (Line 5). If π_2 dominates π_2, π_2 will be selected (Line 7). If they do not dominate each other, we randomly select one of them (Line 9). For the two selected DNNs, single-point crossover (SPX) [11] is performed on the weights of DNN (Line 11). After that, random mutation is further used by applying additive Gaussian noise (GN) [37] to each weights of DNNs in P. By perturbation, diverse new policies with different strategies are generated. Consequently, more states may be explored by utilizing such policies.

Algorithm 1: Cross_Mutate

```
input  : $P$: the population, $G$: the game
output : $P'$: a new population
1  $P' \leftarrow \emptyset$;
2  for $i \in \{1, \ldots, |P|\}$ do
3      Randomly select two policies $\pi_1$, $\pi_2$ from $P$;
4      if $\pi_1 \succ \pi_2$ then
5          $\pi_f \leftarrow \pi_1$;
6      else if $\pi_2 \succ \pi_1$ then
7          $\pi_f \leftarrow \pi_2$;
8      else
9          $\pi_f \leftarrow random(\pi_1, \pi_2)$;
10     Similarly, select another candidate $\pi_m$;
11     $\pi_c \leftarrow weights\_cross(\pi_f, \pi_m)$; ▷ (see SPX in [11])
12     $\pi \leftarrow weights\_mutate(\pi_c)$;      ▷ (see GN in [37])
13     $P' \leftarrow P' \cup \{\pi\}$;
14 return $P'$;
```

D. DRL Training

Deep reinforcement learning (DRL) is an effective approach that interacts with the real environment and learns the optimal policy from the reward signal, enabling the agent to play really complex games automatically and achieving a certain level of human-competitive performance [33]. *Wuji* adopts a standard DRL algorithm named advantage actor-critic (A2C) [28], which tries to find a policy to maximize the rewards in terms of accomplishing the mission.

Specifically, all perturbed individuals in offspring P' are further evolved by leveraging the gradient-based optimization (A2C). The optimization continuous until either of the following criteria is satisfied: 1) reaching maximum training time, or 2) none performance improvement after certain consecutive training rounds. These criteria are referred to as the "early stop", which achieves a good trade-off between improving performance and saving computational resources. It is worth mentioning that, to be effective, *Wuji* adopts a parallel approach for training all DNNs in the offspring, asynchronously and simultaneously. The choices of hyper

Algorithm 2: ND_Sort

```
input  : $P$: the population
output : $P'$: a sorted set of Pareto frontiers.
1  $P' \leftarrow \emptyset$;
2  while $|P| > 0$ do
3      $F \leftarrow \{\pi \in P \mid \nexists \pi' \in P, \pi' \succ \pi\}$;
4      $P' \leftarrow P' \cup \{F\}$;
5      $P \leftarrow P \setminus F$;
6  return $P'$;
```

Algorithm 3: Pop_Select

```
input  : $P$: the population, $n < |P|$: the number of
         agents to be selected
output : $P'$: a selected population
1  $P' \leftarrow \emptyset$;
2  $\{F_1, \ldots, F_m\} \leftarrow$ ND_Sort($P$);
3  foreach $F_i$ in $\{F_1, \ldots, F_m\}$ do
4      if $|P'| + |F_i| \leq n$ then
5          $P' \leftarrow P' \cup F_i$;
6      else
7          $F' \leftarrow$ CD_Select($F_i, n - |P'|$);
8          $P' \leftarrow P' \cup F'$ ;
9          break
10 return $P'$
```

parameters of DRL training and early stop will be detailed in the later Section V-A4, respectively.

E. Population Selection

After the evolution, we obtain two new strategy populations P' and Q. The policies in P' are evolved via crossover and mutation, which pay more attention to the exploration. On this basis, population Q are obtained via DRL training, which focuses on completing the mission. Given all candidate policies (i.e., $P \cup P' \cup Q$), *Wuji* utilizes the non-dominated sorting and crowding distance sorting to dynamically select the better policies in terms of the two objectives (see Section IV-B).

Algorithm 2 shows the non-dominated sorting algorithm. Given a population P, it identifies a number of prioritized Pareto frontiers [12] (a subset of the policies) by a repeating process. In each iteration, a policy, which cannot be dominated by other policies, is added into the current selected Pareto frontier (Line 3). Then, the Pareto frontier will be added to the new population (Line 4) and be removed from the original population (Line 5). In the next iteration, it continues to select next Pareto frontier from the remaining agent policies.

The intuition of our selection strategy is that if a policy is not dominated by others, it is at least no worse than any other one in terms of the multi-objectives. Fig. 3(b) shows an example of the non-dominated sorting that outputs a sequence of Pareto frontiers $\{F_1, F_2, F_3, \ldots\}$. Each point is a policy within a Pareto frontier. We could see that F_1 generally exceeds F_2 because $\forall \pi \in F_1.(\nexists \pi' \in F_2.\pi' \succ \pi)$.

Algorithm 4: CD_Select

```
input  : $F = \{\pi_1, \pi_2, \cdots, \pi_l\}$: a Pareto frontier, $n < |F|$:
         the number of polices to be selected
output : $P'$: a set of selected policies
1  $\forall 0 < i \leq l, dist(\pi_i) \leftarrow 0$;
2  $E \leftarrow Sort^{e}_{\downarrow}(F)$;
3  $W \leftarrow Sort^{w}_{\downarrow}(F)$;
4  $dist(E[0]) \leftarrow \infty$;
5  $dist(W[0]) \leftarrow \infty$;
6  foreach $\pi \in (F \setminus \{E[0], W[0]\}$ do
7      $i \leftarrow IndexOf(E, \pi)$;
8      $j \leftarrow IndexOf(W, \pi)$;
9      $dist(\pi) \leftarrow$
         $(ES^{E[i-1]} - ES^{E[i+1]}) + (RS^{W[j-1]} - RS^{W[j+1]})$;
10 $Z \leftarrow Sort^{dist}_{\downarrow}(F)$;
11 return $\{Z[1], \ldots, Z[n]\}$;
```

Algorithm 3 presents the procedure that takes policies P and a number n as the input, and outputs a better population P'. We first perform a non-dominated sorting to prioritize the policies. Based on the prioritization, the Pareto frontiers will be added to the new population in order (Line 5). If the total number of the selected population P' and the current Pareto frontier F_i exceeds n (Line 4), we have to select part of the polices (whose size is $n - |P'|$) from F_i such that we can exactly select n policies. Crowding distance sorting strategy is then adopted to select the remaining policies (Line 7).

Algorithm 4 shows the crowding distance sorting algorithm. Crowding distance (CD) measures the density of the policies and *Wuji* tends to select the sparse points for constructing a more diverse population. The CD of each policy is initialized as zero (Line 1). Then we sort the policies F in descending order based on the exploration score and winning score, respectively (Line 2-3). The CD of the policy, which has the largest winning score or the exploration score, is set as the max distance value (Line 4-5). The two policies are the best ones in terms of exploring/winning and we will always select them. For each of the other policies (Line 6), we compute the CD by the distances from the surrounding polices. We get the indexes in the ordered sequences E and W (Line 7-8). The CD is computed based on the distance between two surrounding policies in terms of exploration score and winning score (i.e., $ES^{E[i-1]} - ES^{E[i+1]}$ and $RS^{W[j-1]} - RS^{W[j+1]}$). Afterward, the polices are sorted in descending order based on the CD and the first n policies are returned (Line 10-11).

Consider the two policies π_1 and π_2 in Fig 3(b), the CD of π_1 is computed as $d_1 + d_2 + d_3 + d_4$, which is larger than the CD of π_2. Intuitively, the polices around π_2 are very dense while the polices around π_1 are sparse.

V. Evaluation

We have implemented *Wuji* (7,000+ lines of code) based on PyTorch [32]. To demonstrate the effectiveness of *Wuji*, we investigate the following research questions.

Fig. 4: Examples of combat scenarios for L10 (left) and NSH.

RQ1 (Bug Detection): How effective is *Wuji* in detecting game bugs? How do the key components of *Wuji*, i.e., *EA*, *DRL* and *MOO*, affect the performance when functioning separately?
RQ2 (Coverage): How is the exploration capability of *Wuji* in terms of code coverage and state coverage?
RQ3 (Bug Analysis): What are the root cause and characteristics of those bugs found by *Wuji*?

A. Experimental Design and Settings

1) Datasets: A simple game (Block Maze) and two commercial online combat games (L10[3] and NSH[4]) are selected in the evaluation. Since L10 and NSH are quite complex real-world commercial games. To make our study feasible, for each game, we only selected one instance (i.e., a combat scenario) for testing. Fig. 4 depicts the online combat scenarios for the two commercial games. Table II summarizes the detailed information about the games. Column *#Code* shows the number of lines of source code for each game. In particular, for L10 and NSH, the left number shows the code size of the selected scenario while the right number is the code size of the whole game. Column *INST.Size* represents the installation size and Column *#Players* shows daily peak player count (4,000,000+ for L10 and 880,000+ for NSH), from which we could see the selected games are indeed large-scale and practical.
Block Maze: the agent controls the robot to reach the goal coin. The player has 4 actions to choose: {north, south, east, west}. Every movement leads the agent to move into a neighbor grid in the corresponding direction, except that a collision on the block (dark green) results in no movement.
L10** and **NSH: the two games are online combat games, where a game character fights against an opponent in the game. To be specific, L10 is 2-dimensional scenario running on the mobile platform. NSH is a 3-dimensional game running on the PC platform. In L10, the game character is equipped with 5 skills while it has 8 skills in NSH. Also, the game character can move in four directions. The goals of the games are to win the fighting by discovering a combination of skills that damage the opponent to the largest extent.

Note that all three games are tested on the server even if L10 is released for mobile devices. GUIs of the games are removed as we mainly focus on functional testing. In industry game development and testing of Netease, Inc., this is a typical method for maximizing the testing efficiency.

TABLE II: Three games used in the evaluation

Name	#Code (KLOC)	INST.Size	#Players
Block Maze	0.768	3 MB	-
L10	45.99/1,472	1.54 GB	4 million+
NSH	292.07/23,457	61.38 GB	880 thousand+

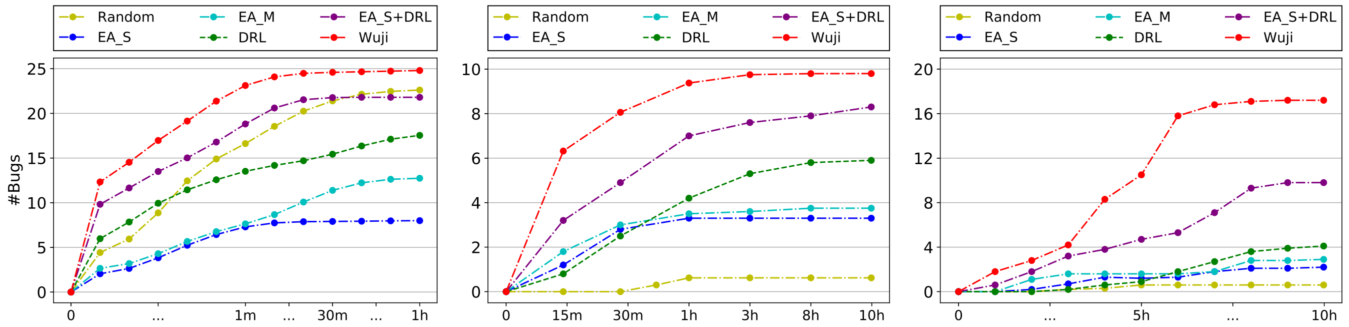
2) Game Bugs: To the best of our knowledge, it still lacks standard benchmarks for game testing at this moment. In our study, to evaluate the usefulness of *Wuji*, the three games are injected with history bugs for controlled study as follows. For *Block Maze*, we injected 25 bugs that are randomly distributed within the environment. The bugs will be triggered if the robot has reached the specific location of the map (see Fig. 1). For *L10* and *NSH*, the game developers of our collaborated company help to inject 10 (2 crash bugs, 2 stuck bugs, and 6 logical bugs) and 18 (4 crash bugs, 3 stuck bugs and 11 logical bugs) previously known bugs in the selected scenarios, based on their experience. These bugs actually appeared in previous different versions of the game, and are now injected in the same game version for our evaluation purpose. Furthermore, developers also insert the assertions for detecting logical bugs.

3) Baseline Approaches: To the best of our knowledge, there still does not exist any automated testing technique on large OCMs at this moment. As mentioned in Section V-A1, our testing environment is deployed on the server and does not contain the GUI part. Hence, the existing GUI testing tools, such as Monkey [17], Sapienz [26], Stoat [35], are not applicable. We tried our best to compare *Wuji* with the five strategies (including one baseline and four variants of *Wuji*) to demonstrate the effectiveness of *Wuji* and evaluate the effectiveness of each strategy of *Wuji* in terms of bug detection and coverage:

- *Random:* during playing the games, it randomly selects one of the available actions. We followed the idea of Monkey [17] and implemented this random strategy as the baseline.
- *EA_S:* the evolutionary algorithm based strategy whose fitness score only considers the winning score.
- *EA_M:*the evolutionary algorithm based strategy that uses a MOO-based fitness score (i.e., Algorithm 1).
- *DRL:* test the games by only considering the winning score with DRL.
- *EA_S+DRL:* the combination of EA and DRL but they only consider the winning score. Compared with *EA_S+DRL*, *Wuji* (*EA_M+DRL*) adopts a multi-objective optimization.

4) Configurations and Implementations: We select the DNN model in the previous work *A2C* [28] as the seeding model for all of the three games, with the input and output dimensions adjusted for different games. Mean square error (MSE) is adopted as the loss function in DRL training, which is minimized by leveraging the Adam [21] optimizer. To be specific, due to the stripping of GUIs, the inputs of the DNNs models are scalar-based vectors. Consequently, we use three fully-connected linear layers with 256,128,128 units as the hidden layers, connected with the output layer. The output

[3] A Chinese Ghost Story (L10) https://xqn.163.com/
[4] Treacherous Water Online (NSH) http://n.163.com/

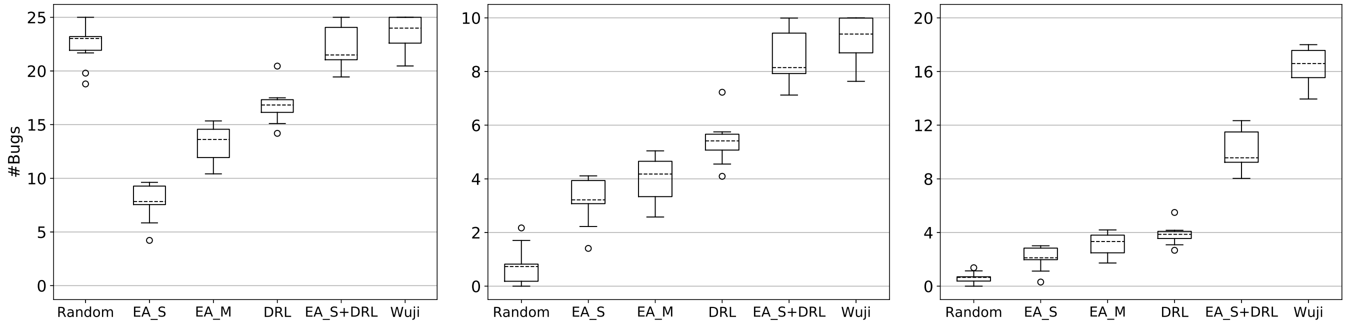


(a) Block Maze

(b) L10

(c) NSH

Fig. 5: Comparison of different strategies regarding the average number of bugs found in three games, respectively



(a) Averaged bugs found in Block Maze

(b) Averaged bugs found in L10

(c) Averaged bugs found in NSH

Fig. 6: The number of bugs discovered using different strategies after 1 hour for Block Maze and 10 hours for L10 and NSH.

layer consists of each valid actions for three games with the corresponding size of 4, 7, 17. The output of each hidden layer is normalized using batch normalization [19], and the activation function ReLU [30] is adopted between two consecutive layers. On the other hand, to ensure sufficient exploration, we set the weight of the entropy item in the loss function of the actor to 10^{-4}, and ensure at least 1% possibility of each valid actions being selected for executing. The epoch for DRL training is set to 20 (BlockMaze), 1000 (L10) and 1000 (NSH), respectively. The learning rate is set to $0.25 * 10^{-3}$ in the Adam optimizer. In addition, the maximum training time and threshold of the consecutive training period in "early stop" are set to 10 minutes and 7 rounds, respectively. For the EA algorithm, the size of the population and offspring are set to 30. The crossover rate and mutation rate are set to 100% and 1%, respectively. Notably, a random noise sampled from a Gaussian distribution $\mathcal{N}(0, 0.005)$ is adopted for mutating the DNNs. All the experiment were run on three powerful servers with CPU(Intel(R) Xeon(R) Gold 6138 CPU @ 2.00GHz) with 100 cores and 256GB RAM.

From a statistical perspective, in *RQ1* and *RQ2*, each strategy is used to test the games for a certain amount of time based on the complexity. For the simple game Block Maze, we allocate one hour time budget. For L10 and NSH, we allocate 10 hours for each. To counter the randomness effect during testing, we repeat each strategy 10 times to average the results. To mitigate the potential bias of bug injection, during each test run of Block Maze, the 25 bugs and the gold position are randomly injected.

Note that, even though we only selected two scenarios in the two commercial games, it still takes quite a lot of human efforts: (1) the game developers spent about 20 man-days and $5,000+ manpower cost to inject these known bugs carefully, and (2) we ran 6 different strategies on three games with 10 repetitions, and it overall took 1,260 machine-hours.

B. *Evaluation on Bug Detection (RQ1)*

Fig. 5 shows the average number of bugs discovered over time. The statistical results of the number of bugs discovered during the allocated testing time (see Section V-A4) are shown in Fig. 6. Overall, we have the following findings.

First, compared with the baseline, we can see that *Random* achieves better results in the smaller game (i.e., Block Maze). The reason is that small game has a relatively small search space. In the small space, *Random* can explore the states given enough time budge. For example, in Block Maze *Random* can find more bugs than *EA_S* and *DRL* quickly. The reason is that *EA_S* and *DRL* focus on how to accomplish the mission. In a smaller game, they can find a better policy quickly and stop exploring more states. This prevents them from discovering more bugs. In commercial games, the state space becomes quite larger. Due to lacking the intelligence, *Random* cannot

TABLE III: The average number of different types of bugs

	L10						NSH					
	Wuji	E_S+DRL	DRL	E_M	E_S	Ran.	Wuji	E_S+DRL	DRL	E_M	E_S	Ran.
Crash	1.4	1.2	0.2	0	0	0	2.4	2.2	0	0	0	0
Stuck	1.8	1.6	1.4	1.2	0.9	0	2.2	2.1	1.1	0	0	0
Logical	5.6	5.5	4.3	2.5	2.4	0.6	10.8	5	3	2.9	2.2	0.6
New	0.8	0	0	0	0	0	1.8	0.5	0	0	0	0
Total	9.6	8.3	5.9	3.7	3.3	0.6	17.2	9.8	4.1	2.9	2.2	0.6

TABLE IV: The results of line coverage with each strategy

	BlockMaze	L10	NSH
Coverage	98%	73%	66%

TABLE V: The results of state coverage with each strategy

Game	*Wuji*	*EA_S+DRL*	*EA_M*	*EA_S*	*DRL*	*Rand.*
B.M	**366.7**	338.2	115.7	91.3	243.2	349.6
L10	**85,764.8**	59,726.7	50,177.7	46,604.7	51,750.6	12,298.8
NSH	**807,344.6**	651,608.6	305,662.2	265,662.2	395,726.8	84,744

accomplish the mission well. It only explores a small number of states and has the worst performance. The results indicate that random testing performs badly in the large games, which is consistent with our intuition that traditional random testing is less effective in testing large game testing because it cannot effectively accomplish the game intermediate missions at the first hand. The AI-based approach (e.g., DRL) can mitigate such a problem by learning a smart policy.

Second, the results also show the effect of *EA*, *DRL* and *MOO* on training smart policies for bug detection. The combination of *EA_S* and *DRL* can achieve much better results than using one of them separately. The reason is that *EA_S* mainly considers the winning score, and it provides various searching directions by leveraging the advantage of population, while the DRL algorithm improves the searching speed in a particular direction through gradient-based optimization. The results also indicate that both gradient-based (*DRL*) and search-based techniques (*EA_S*) contribute to discovering more bugs by complementing each other.

Further, *Wuji* (i.e., *EA_M+DRL*) achieves better results than *EA_S+DRL*, and *EA_M* achieves better results than *EA_S*. This suggests that the multi-objective optimization is helpful for discovering more bugs efficiently. Even though the population can contribute to the exploration (e.g. *EA_S+DRL* performs well in all three games), the performance can be further improved by involving the exploration score in the fitness score (i.e., *EA_M+DRL* for *Wuji*). In a large game like NSH, the improvement is even more obvious.

We also found that *EA_M* is slightly better than *EA_S* while *EA_S+DRL* exceeds *EA_S* even further. Intuitively, adding *DRL* achieves larger improvement than adding exploration score. We perform an in-depth analysis and found that: even if we have added the winning score in *EA*, the performance of accomplishing the mission is still not satisfying since *EA* is gradient-free and it is not effective to improve the winning score. As a result, if it cannot accomplish the mission, the exploration score cannot be improved. The results further demonstrate why the benefits of including *DRL* (gradient-based) in *Wuji*, which facilities the mission accomplishment.

Table III shows the average number of different types of bugs (including known injected bugs and newly discovered bugs) in L10 and NSH. During the 10 rounds of testing, *Wuji* detects 1 new bug in L10 and 2 new bugs in NSH. As for the other strategies, only *EA_S+DRL* finds one new bug in NSH. These three new bugs have been confirmed by the game developers and will be fixed in the next update. More details about the bugs will be discussed in Section V-D. Considering these unknown bugs are detected from the stable release version of these games, it further confirms the usefulness of *Wuji* for detection unknown bugs.

To summarize, *Wuji* enhances the exploration with the MOO-based *EA_M* and improves the capability of accomplishing the mission with *DRL*, which complement each other and achieve the best performance with combination.

Answer to RQ1: *Wuji* is much more effective in detecting game bugs than random testing, which performs poorly on large games. *EA*, *DRL* and *MOO* complement each other in exploration of diverse states and accomplish the game mission. Their combination (i.e., *Wuji*) achieves better performance.

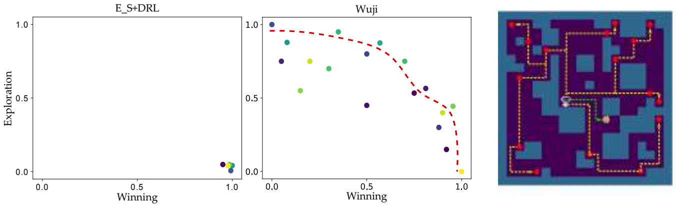
C. Evaluation on Coverage Increase (RQ2)

We continue to analyze the coverage obtained by each strategy on the game. Surprisingly, the line coverage of different strategies is exactly the same for each game, despite the results of bug detection are quite different (see Table IV).

The results indicate that these strategies achieve high line coverage easily. This can be explained by the characteristics of game, i.e., the code related to the function of the game is easily covered by playing games. Hence, it seems to be ineffective to detect game bugs by only improving code coverage.

Furthermore, we also analyze state coverage. The state coverage is computed as follows: each state (see Definition 3) contains a set of features. We adopt an interval approximation to partition each feature into equal buckets. Based on the discretization buckets of the features, we can count covered buckets to estimate the state coverage. For example, if the state has n features and each feature is partitioned into m intervals, the maximum number of states (i.e., buckets) is m^n.

Table V shows the number of states that are covered by each strategy on each game. Compared with the other strategies, *Wuji* cover more states in each game (the bold number). For large games such as *NSH*, the state coverage is relatively low because large games might have more challenging (resp. infeasible) states that are difficult (reps. unable) to reach. For example, if the player is not close to the boss, then the boss cannot be attacked. We further found that the results of state coverage are consistent with the results of bug detection in *RQ1*. In the small game, random testing can achieve competitive results while it is the worst one in the larger games, indicating that it is more useful in detecting bugs



(a) Diversity of Policy (b) Trajectories

Fig. 7: (a) the final distribution of policies in the population, and (b) the trajectories of executing each policy.

by maximizing state coverage than code coverage for game testing.

> **Answer to RQ2:** The code of games is relatively easy to cover. For state coverage, random testing is limited in increasing state coverage on large games, while *Wuji* is more effective than the other strategies by a combination of *EA*, *DRL*, and *MOO*.

D. Bug Analysis (RQ3)

In this section, we analyze some specific cases towards better understanding the coverage and bugs of games. More analysis and bug reproducing videos can be found on our website [41].

Polices in Block Maze. Fig. 7 shows the final distribution of policies with *EA_S+DRL* and *Wuji* in one run of Block Maze, with the difference between single-objective and multi-objective depicted. With MOO, the policies in the population of *Wuji* are more diverse in terms of exploration score and winning score while the polices in *EA_S+DRL* only favors in winning (i.e., closely distributed at the bottom right). The right figure shows the trajectories of executing each policy. Intuitively, the policies in *Wuji* can explore more directions (yellow arrows) while the policies in *EA_S+DRL* only move towards the gold (green arrow).

Line/State coverage. The following example is a simplified code segment including a bug in L10. The bug will be triggered once the variable *distance* is 0. Intuitively, the code is very easy to cover as long as the skill (i.e., *skill_id*) is used. For state coverage, *distance* is a feature in the state, new states will be covered when *distance* changes. The example shows that line coverage is relatively easy to cover and it cannot capture the state change.

```
if (player.buff_time[skill_id]>0){
    distance = compute_distance(player, boss)
    //will crash if distance == 0
    attack = player.use_skill(skill_id, distance)
    damage = boss.get_damage(attack, ...)...
}
```

Logical bug in L10: The following code shows the calculation of damage from the enemy to the player character. Specifically, the damage from the boss is calculated based on the current damage reduction buffs (e.g. Line 3,5,7) and each buff has a time limit. A logical bug is that the damage (i.e., variable *hurt*) can be negative after some operators, and finally the damage from the boss becomes the treatment for the player (Line 9). We investigate the bug and found that the variable *hurt* becomes negative only when the initial damage (Line 1) is within a certain range and three specific buffs (i.e., 1, 3, 12) exist simultaneously. The state is difficult to reach but *Wuji* discovered it by adopting an effective exploration.

```
1. hurt = boss.atk
2. if (boss.buf[1] == True)
3.   hurt -= boss.buf[1].reduce_val
4. if (boss.buf[3] == True)
5.   hurt -= boss.buf[3].reduce_val
6. if (boss.buf[12] == True)
7.   hurt -= boss.buf[12].reduce_val
   ...
8. // fix: hurt = max(hurt, 0)
9. player.hp -= hurt
```

Crash bug in NSH: A crash bug discovered by *Wuji* and *MA_S+DRL*. Generally, the bug is triggered when some properties of the game are reset with invalid values, which are generated only when the boss is dead with multiple specific buffs. This a very hard corner case to reach by manual testing (e.g., manual playing or scripting). However, it could be triggered during playing by a large number of game players. *Wuji* found the bug by learning diverse policies in favoring to explore more states of the game.

E. Threats to Validity

Randomness is a major factor during testing and the game playing. We counter this issue by repeating 10 times for each testing task and averaging the results. In addition, the oracle we proposed may not be complete, and thus *Wuji* may miss some other unknown bugs like logical bugs and other types of bugs. The selection of the games could be biased. In this paper, we try to counter this issue by using 3 games with different sizes, where two of them are active commercial games. The selection of the DNNs used for training policies could be biased. We counter this problem by selecting models from the state-of-the-art for game playing. The bias of bug injection may be also a threat. We counter this issue by randomly injecting bugs of Block Maze in each testing round, and ask helps from game developers to inject bugs in the commercial games.

On the other hand, false alarms reported by *Wuji* may be a potential threat. Specifically, the thresholds in stuck and balance oracles directly determinate the sensitivity of *Wuji* reporting a bug. The choice of such hyper parameters are not only highly dependent on the domain knowledge, but may also be biased. To address this, for each discovered bugs, *Wuji* will record every necessary information during game playing (including every taken action, the opponent's HP value, attack, and defensive value, etc.). Moreover, an additional replay suit is also developed for replaying the game automatically. On this basis, despite unable to achieve 100% detection accuracy, *Wuji*, however, provides an effective way for testers to discern real bugs efficiently.

VI. Related Work

This section surveys the related work in three aspects.

Game Testing. As games become increasingly popular and complex, testing games is now an important challenge and becomes the key vehicle to improving quality. However, existing testing practice and techniques still need more significant advances. As pointed by Lin *et al.* [24], even the most popular games on the market show the lack of sufficient testing. Alemm *et al.* [3] conduct a quantitative survey to identify key developers factors for a successful game development process. They find that manual testing and ad-hoc, exploratory testing are still predominant than systematic and automated testing. Iftikhar *et al.* [18] proposes a UML-based modeling methodology to support automated system-level game testing of platform games. They manually construct the model and use the model to generate and execute test cases with oracles. However, the manual model construction process still requires a lot of efforts. As for mobile games, existing research is also preliminary. Lovreto *et al.* [25] first reports their experience of using exploratory testing to test 16 mobile games. They manually write test scripts for functional testing and find existing testing techniques still have a lot of limitations. To prioritize testing efforts, Khalid *et al.* [20] mine the user reviews from 99 free mobile game apps and find that most negative reviews come from a small subset of devices, which game developers should emphasize during testing. Game apps are GUI applications, which extensively interact with human players. Although various research work exists for testing of GUI applications [5], [10], these techniques only target at general-purpose GUIs. They do not work for testing game apps because game apps usually use a game engine to render GUIs instead of using traditional GUI widgets. To our knowledge, our technique, *Wuji*, is the first to achieve systematic and automated testing for real-world games and makes great improvements over state-of-the-practice.

Reinforcement Learning for Testing. *Wuji* applies deep reinforcement learning (DRL) to facilitate automatic game testing. Various research work have also applied (deep) reinforcement learning for their own purposes. For example, QBE [22] uses Q-Learning, one type of reinforcement learning technique, to automatically test mobile apps. Specifically, it explores the GUIs of a set of apps to generate the corresponding behavior models, and uses these models to train the transition prioritization matrix with two optimization goals, i.e., activity coverage and the number of crashes, respectively. QBE focuses on improving code coverage and the number of detected crashes for Android apps. Similarly, Adamo *et al.* [2] and Vuong *et al.* [39] also use Q-learning to improve GUI testing of Android apps. However, these work only focus on general apps instead of game apps. Böttinger *et al.* [7] introduces the first program fuzzer that uses reinforcement learning to learn high reward seed mutations for testing traditional software. Specifically, by automatically rewarding runtime characteristics of the target program, this technique obtains new inputs that likely drive program execution towards a predefined goal, *e.g.*, maximizing code coverage. Reinforcement learning is also applied to facilitate test prioritization and selection in regression testing [34]. Different from these work, *Wuji* combines DRL with evolutionary algorithms to improve testing of game apps, which have not been tackled before.

Evolutionary Testing.

Many evolutionary algorithms have been proposed and widely used to solve the test generation problem in software testing. Coverage-guided fuzzing is a popular testing technique which has been proposed for detecting vulnerabilities in traditional software [6], [9], [23], [40]. Moreover, evolutionary testing based techniques have also been applied to test deep neural networks [15], [42], [43]. Compared with them, *Wuji* is designed to explore more states in the game instead of maximizing code/neural coverage.

In addition, there are some techniques using evolutionary testing with multi-objective optimization. In the field of GUI testing, a number of research work exploit multi-objective optimization to improve testing effectiveness. Sapienz [26] uses multi-objective search-based testing to automatically finding bugs in Android apps. It optimizes test sequences by minimizing length, while simultaneously maximizing coverage and fault revelation. Similarly, Stoat [36] use the Gibbs sampling technique to optimize test generation for Android apps by maximizing coverage at both model and code level, while increasing test diversity. In our work, *Wuji* uses a combined evolutionary algorithm and DLR for effective game testing, towards covering diverse states and completing more intermediate missions.

VII. Conclusion

In this paper, we performed a study on the characteristics of bugs in real-world commercial games. We classified the bugs into five categories and proposed four test oracles. Based on the oracles, we developed an automated testing framework for games. By combing the evolutionary multi-object optimization and DRL, *Wuji* can explore more states such that bugs can be more likely to be detected. We demonstrated the effectiveness of *Wuji* on one simple game and two large commercial games. In the future, we plan to 1) extend *Wuji* to support multiplayer games based on multi-agent reinforcement learning, and 2) apply *Wuji* on more types of games.

Acknowledgment

The work is supported by the National key R&D program of China (Grant Nos.: 2018YFB1701700), National Natural Science Foundation of China (Grant Nos.: 61702362, U1836214), the National Research Foundation, Prime Ministers Office Singapore under its National Cybersecurity R&D Program (Award No. NRF2018NCR-NCR005-0001), National Satellite of Excellence in Trustworthy Software System (Award No. NRF2018NCR-NSOE003-0001) administered by the National Cybersecurity R&D Directorate, Singapore. We thank our industrial research partner Netease, Inc., especially the Fuxi AI Lab and Game Testing Department of Leihuo Business Groups for their discussion and support with the experiments.

REFERENCES

[1] American Fuzzy Lop . http://lcamtuf.coredump.cx/afl/, 2018.

[2] David Adamo, Md Khorrom Khan, Sreedevi Koppula, and Renée C. Bryce. Reinforcement learning for android GUI testing. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, A-TEST@ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 05, 2018*, pages 2–8, 2018.

[3] Saiqa Aleem, Luiz Fernando Capretz, and Faheem Ahmed. Critical success factors to improve the game development process from a developer's perspective. *J. Comput. Sci. Technol.*, 31(5):925–950, 2016.

[4] Nadia Alshahwan, Xinbo Gao, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, Taijin Tei, and Ilya Zorin. Deploying search based software engineering with sapienz at facebook. In *International Symposium on Search Based Software Engineering*, pages 3–45. Springer, 2018.

[5] Ishan Banerjee, Bao N. Nguyen, Vahid Garousi, and Atif M. Memon. Graphical user interface (GUI) testing: Systematic mapping and repository. *Information & Software Technology*, 55(10):1679–1694, 2013.

[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2017.

[7] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. Deep reinforcement fuzzing. In *2018 IEEE Security and Privacy Workshops, SP Workshops 2018, San Francisco, CA, USA, May 24, 2018*, pages 116–122, 2018.

[8] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[9] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed greybox fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 2095–2108, New York, NY, USA, 2018. ACM.

[10] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for android: Are we there yet? (E). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 429–440, 2015.

[11] Kalyanmoy Deb and Ram Bhusan Agrawal. Simulated binary crossover for continuous search space. *Complex Systems*, 9(2):115–148, 1994.

[12] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and Tanaka Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii. In *International conference on parallel problem solving from nature*, pages 849–858. Springer, 2000.

[13] DeepMind. Aplhago. https://deepmind.com/research/alphago/, 2019.

[14] DeepMind. Dota2. https://openai.com/five/, 2019.

[15] Xiaoning Du, Xiaofei Xie, Yi Li, Lei Ma, Yang Liu, and Jianjun Zhao. Deepstellar: model-based quantitative analysis of stateful deep learning systems. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 477–487. ACM, 2019.

[16] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419. ACM, 2011.

[17] Google. Ui/application exerciser monkey. https://developer.android.com/studio/test/monkey, 2018.

[18] Sidra Iftikhar, Muhammad Zohaib Iqbal, Muhammad Uzair Khan, and Wardah Mahmood. An automated model based testing approach for platform games. In *18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MoDELS 2015, Ottawa, ON, Canada, September 30 - October 2, 2015*, pages 426–435, 2015.

[19] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv.org*, February 2015.

[20] Hammad Khalid, Meiyappan Nagappan, Emad Shihab, and Ahmed E. Hassan. Prioritizing the devices to test your app on: a case study of android game apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 610–620, 2014.

[21] Diederik P Kingma and Jimmy Ba. Adam - A Method for Stochastic Optimization. *ICLR*, 2015.

[22] Yavuz Köroglu, Alper Sen, Ozlem Muslu, Yunus Mete, Ceyda Ulker, Tolga Tanriverdi, and Yunus Donmez. QBE: qlearning-based exploration of android applications. In *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*, pages 105–115, 2018.

[23] Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. Cerebro: context-aware adaptive fuzzing for effective vulnerability detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 533–544. ACM, 2019.

[24] Dayi Lin, Cor-Paul Bezemer, and Ahmed E. Hassan. Studying the urgent updates of popular games on the steam platform. *Empirical Software Engineering*, 22(4):2095–2126, 2017.

[25] Gabriel Lovreto, André Takeshi Endo, Paulo Nardi, and Vinicius H. S. Durelli. Automated tests for mobile games: An experience report. In *17th Brazilian Symposium on Computer Games and Digital Entertainment, SBGames 2018, Foz do Iguaçu, Brazil, October 29 - November 1, 2018*, pages 48–56, 2018.

[26] 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, Saarbrücken, Germany, July 18-20, 2016*, pages 94–105, 2016.

[27] Brad L Miller, David E Goldberg, et al. Genetic algorithms, tournament selection, and the effects of noise. *Complex systems*, 9(3):193–212, 1995.

[28] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[29] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[30] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[31] Newzoo. Global games market report. https://newzoo.com/solutions/standard/market-forecasts/global-games-market-report, 2018.

[32] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[33] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[34] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, pages 12–22, 2017.

[35] 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. ACM, 2017.

[36] 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, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017, pages 245–256, 2017.

[37] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*, 2017.

[38] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[39] Thi Anh Tuyet Vuong and Shingo Takada. A reinforcement learning based approach to automated testing of android applications. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, A-TEST@ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 05, 2018*, pages 31–37, 2018.

[40] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 579–594. IEEE, 2017.

[41] Wuji. Wuji. https://sites.google.com/view/gametesting, 2019.

[42] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. Deephunter: a coverage-guided fuzz testing framework for deep neural networks. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 146–157. ACM, 2019.

[43] Xiaofei Xie, Lei Ma, Haijun Wang, Yuekang Li, Yang Liu, and Xiaohong Li. Diffchaser: Detecting disagreements for deep neural networks. In *IJCAI*, pages 5772–5778, 2019.

[44] Tianpei Yang, Jianye Hao, Zhaopeng Meng, Chongjie Zhang, Yan Zheng, and Ze Zheng. Towards Efficient Detection and Optimal Response against Sophisticated Opponents. *IJCAI*, 2019.

[45] Tianpei Yang, Jianye Hao, Zhaopeng Meng, Yan Zheng, Chongjie Zhang, and Ze Zheng. Bayes-ToMoP - A Fast Detection and Best Response Algorithm Towards Sophisticated Opponents. *AAMAS*, 2019.

[46] Yan Zheng, Zhaopeng Meng, Jianye Hao, and Zongzhang Zhang. Weighted Double Deep Multiagent Reinforcement Learning in Stochastic Cooperative Environments. *PRICAI*, 2018.

[47] Yan Zheng, Zhaopeng Meng, Jianye Hao, Zongzhang Zhang, Tianpei Yang, and Changjie Fan. A Deep Bayesian Policy Reuse Approach Against Non-Stationary Agents. *NeurIPS*, 2018.