Empirical Software Engineering (2022) 27: 140
https://doi.org/10.1007/s10664-022-10158-x

GBGallery : A benchmark and framework for game testing

Zhuo Li[1] · Yuechen Wu[2] · Lei Ma[1,3,4] · Xiaofei Xie[5] · Yingfeng Chen[2] · Changjie Fan[2]

Accepted: 28 February 2022 / Published online: 27 July 2022

Abstract

Software bug database and benchmark are the wheels of advancing automated software testing. In practice, real bugs often occur sparsely relative to the amount of software code, the extraction and curation of which are quite labor-intensive but can be essential to facilitate the innovation of testing techniques. Over the past decade, several milestones have been made to construct bug databases, pushing the progress of automated software testing research. However, up to the present, it still lacks a real bug database and benchmark for game software, making current game testing research mostly stagnant. The missing of bug database and framework greatly limits the development of automated game testing techniques. To bridge this gap, we first perform large-scale real bug collection and manual analysis from 5 large commercial games, with a total of more than 250,000 lines of code. Based on this, we propose *GBGallery*, a game bug database and an extensible framework, to enable automated game testing research. In its initial version, *GBGallery* contains 76 real bugs from 5 games and incorporates 5 state-of-the-art testing techniques for comparative study as a baseline for further research. With *GBGallery*, we perform large-scale empirical studies and find that the current automated game testing is still at an early stage, where new testing techniques for game software should be extensively investigated. We make *GBGallery* publicly available, hoping to facilitate the game testing research.

Keywords Game testing · Bug database · Automated testing · Deep reinforcement learning

1 Introduction

With the rapid technology evolution (e.g., software, hardware) over the past decades, game software has already become an essential part of our daily life. According to the Newzoo *Global Games Market Report* (Newzoo 2020), the game industry will generate $159.3

Communicated by: Shaowei Wang, Tse-Hsun (Peter) Chen, Sebastian Baltes, Ivano Malavolta, Christoph Treude and Alexander Serebrenik

This article belongs to the Topical Collection: *Collective Knowledge in Software Engineering*

✉ Xiaofei Xie
xfxie@smu.edu.sg

Extended author information available on the last page of the article.

billion annual revenues with over 3 billion global game players by the end of 2020. As a core problem and pain point of the game industry, the quality issues in the game software can post risks, causing negative user experience, losing user confidence, leading to financial losses and even severe security accidents. Therefore, ensuring the quality of game software is of great importance, for which testing is still a *de facto* standard in the game industry.

However, the current industrial game software testing exhibits many challenges. Game software is often highly interactive, dynamic, and non-deterministic. A good tester often requires having a certain level of intelligent strategy to perform sequential actions cleverly to reach the state, where a bug potentially lies, among the huge game state space. What makes it even harder is that industrial games (especially the recent games) are often quite large and complex. The modern game software also evolves at a rather rapid pace, often ranging from daily to weekly. Although game companies do hire quite a lot of professional human players as testers to perform testing activities before releasing, it is labor-intensive and still far from being enough under the time pressure, leaving many bugs being discovered by the users after releasing. Therefore, automated game testing techniques and tool support are highly desirable, and oftentimes can be essential, especially with the increase of game complexity and frequent evolution.

We have witnessed the leap of automated testing in the past 30 years with many state-of-the-art automated testing techniques proposed (e.g., random testing, search-based testing, symbolic execution) and tools available (e.g., Randoop (Pacheco and Ernst 2007), EvoSuite (Fraser and Arcuri 2011), KLEE (Cadar et al. 2008)). Multiple bug databases were also proposed (e.g., iBug (Dallmeier and Zimmermann 2007), Defects4J (Just et al. 2014)), which play an important role in accelerating the development of automated testing techniques. However, even until this moment, little progress has been made on automated game testing, and to the best of our knowledge, no game bug database is publicly available.

To bridge this gap and further ignite game testing research towards addressing the long-standing industrial pain point and demand, in this paper, we made substantial efforts to initiate the construction of a real game bug database (containing 76 real game bugs) and automated framework (with 5 current state-of-the-art automated techniques for game testing), named *GBGallery*, to enable automated game testing research and support further in-depth studies. In particular, the real game bugs are manually collected from 5 industrial games of different types developed by *NetEase*. Together with professional game developers at *NetEase*, we made careful collection, analysis, classification of the real bugs, and integrate them into a bug database. Each bug is accompanied by at least one manual test case, allowing the bugs to be reproduced. *GBGallery* also enables the reproducible study of automated game testing. We also integrate 5 current state-of-the-art automated game testing techniques into *GBGallery* framework for comparative study and as a baseline for further research. Both bug database and automated framework are designed to be extensible to support further automated game testing research.

With *GBGallery*, we also perform a large-scale empirical study to investigate and benchmark the current automated techniques for game testing. In particular, we mainly investigate the following research questions:

- **RQ1 (Statement and Branch Coverage):** How much statement and branch coverage can be achieved by current automated game testing techniques?
- **RQ2 (State Coverage):** How many game states can be covered by different testing techniques?
- **RQ3 (Bug Detection):** How effective are the current automated testing techniques in detecting real game bugs?

- **RQ4 (Correlation):** Is there a positive correlation between the achieved coverage (i.e., statement, branch, state coverage) and game bug detection?
- **RQ5 (Missed Bugs):** What can be the common reasons that game bugs are not detected by existing automated testing techniques?

Through answering these questions, we aim to understand the current status of automated game testing, and provide useful findings for further game testing research. For example, our study reveals that ❶ the statement and branch coverage are rather weak indicators of game testing sufficiency, ❷ the number of covered game states can better reflect the exploration capability of the game testing techniques, ❸ existing automated game testing techniques can detect a certain number of bugs but are still limited and struggling to covering corner-case bugs. Our study also reveals that ❹ the state coverage is more strongly co-related to bug detection capabilities than statement and branch coverage. Furthermore, our in-depth manual analysis on bugs, which are not covered by existing automated testing techniques, reveals that although many portions of game code fragments are relatively easy to be covered, the game state exploration capability of existing automated testing techniques is still limited. The evaluation results confirm that testing strategy intelligence can be particularly important in games software, calling the attention of researchers and practitioners for further investigation. Our automated framework *GBGallery* is expected to directly enable several follow-up research, e.g., automated game test case generation, game testing criteria design, fault localization, repair, *etc.*

In summary, this paper makes the following contributions:

- We spent a large amount of effort on manual analysis of 5 commercial games at *NetEase*, and collected 76 real game bugs under 5 categories, based on which we construct a game bug database.
- We propose and implement an extensible framework, incorporating both the bug database and 5 state-of-the-art automated game testing techniques.
- We also perform a large-scale empirical study and benchmark the current game testing techniques, in terms of code coverage, state coverage, bug detection capability, and investigate their correlations.
- We further perform in-depth analysis on those bugs missed by existing automated game testing techniques, and pinpoint the challenges and opportunities for further game testing research.

  GBGallery and the study results are made publicly available at the accompanying website of this paper, https://sites.google.com/view/gbgallery.

2 Related work

Although automated software testing achieved quite a lot of progress over the past 30 years, little progress has been made on automated game testing in public from the research community. This section summarizes the related work to this paper.

2.1 Software bug database

Software bug database plays an important role in pushing testing techniques forward. The early Siemens benchmark suite provides bugs from 7 small C programs, ranging from 141 to 512 lines of code. These bugs are manually injected and are similar to simple mutations (Hutchins et al. 1994). Software-artifact infrastructure repository(SIR) (Do et al.

2005), which is a milestone work maintained for more than 10 years, contains 85 projects written in C, C++, Java, PHP, C#, *etc.* , where most of the bugs are manually seeded or by mutations.

Another important work iBugs (Dallmeier and Zimmermann 2007) provides a real bugs database of Java programs, with a benchmark specially focused for fault localization, containing 233 bugs with exposing test cases. A more recent milestone work Defects4J (Just et al. 2014), a real bug database and framework, leads the trend of software quality assurance studies on real bugs. It has been updating for more than 5 years since its initial released version containing 5 programs with 357 bugs. The current version Defects4J 2.0 contains 17 projects with 835 bugs, broadly impacting many recent works (e.g., automated testing studies (Shamshiri et al. 2015; Papadakis et al. 2018), fault localization (Pearson et al. 2017), static application programming interfaces (API) testing such as MUBench (Amann et al. 2016) and MUC (Amann et al. 2018), repair (Liu et al. 2019; Madeiral et al. 2019a)) in the software engineering community. Bugs.jar (Saha et al. 2018) and Bears (Madeiral et al. 2019b) also proposed advanced bug benchmarks that can be used for automated Java software testing. Different from these works, we propose a database that focuses on game bugs, which not only contains implementation bugs, but also game logic and game balance bugs.

Although in the past there are public game bug log platform (Buglog 2015) that provides to host and track the game bugs, due to the lack of game environments and test cases, these bugs are rather difficult to be analyzed and automatically reproduced. In *GBGallery*, we intend not only to provide curation of game bugs, but also game entities, oracles, test cases, as well as current state-of-the-art automated game testing techniques, to enable research on automated testing and analysis of game software.

2.2 Game testing

Khalid et al. (2014) conduct the user review study from 99 mobile games, which found that most negative reviews come from a small subset of devices due to the lack of testing. Iftikhar et al. (2015) proposes a UML-based model-based method to support system-level game testing. They manually construct and use the model to generate and execute test cases. However, manual modeling is labor-intensive and difficult to scale up to the current large size industrial games under a fast evolution pace. Aleem et al. (2016) conduct a quantitative survey to identify key developer's factors for a successful game development process. Lin et al. (2017) and Lovreto et al. (2018) point out that most popular games on the market in fact are in the lack of sufficient testing. Borrelli et al. (2020) demonstrated that game bugs can be caused by the game design and bad code smells. Wu et al. (2020) apply a reinforcement learning-based regression testing to explore differential behaviors between multiple versions of multiplayer online role-playing games (MMORPG) for detecting potential regression bugs. Most of these works indicate that manual testing and ad-hoc exploratory testing still play the predominant role for current game testing but are rather costly. It might be tempted to think that game testing is very similar to simple GUI testing (Banerjee et al. 2013), which also performs extensive interactions. However, different from simple GUI testing and other bug trackers, game software is often rendered by using game engines with many consecutive tightly connected scenes, instead of traditional GUI widgets. In other words, for effective game testing, a tester needs to reach a certain level of intelligence and expertise (i.e. policy) in playing these games, e.g., reaching more deep game states and scenes. Otherwise, the diverse game states containing hidden bugs will not be covered.

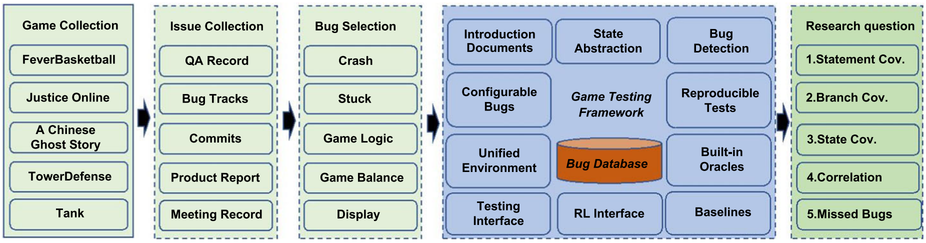
Recently, leveraging artificial intelligence (AI) to test games is being paid more attention and explored (Nordin et al. 2018). With the recent progress in deep reinforcement learning (DRL) (Mnih et al. 2015), automatically training an agent (policy) to play the game, which exhibits a certain level of intelligence, becomes possible. Some open platforms (e.g., OpenAI Gym (Brockman et al. 2016)) were proposed and become the foundation to train advanced agent policy to better complete the game. However, training an agent to complete a game scenario and to detect game software bugs is different. An agent that is capable to complete the game is not necessarily able to detect the bugs, since the bugs might not be contained in the policy action (frequently visited states) scope of the agent.

In this work, we aim to construct a game bug database and automated framework to enable automated game testing research. Besides the collection and analysis of real game bugs from *NetEase*, we also incorporate 5 diverse game testing strategies, and perform comparative studies to benchmark the current automated game testing techniques to identify the challenges and opportunities in automated game testing.

3 Design of *GBGallery*

Figure 1 summarizes the overview of this paper that includes 3 main components: 1) the game collection including the issue collection and bug selection, 2) the game bug database including *GBGallery* framework construction and 3) a large-scale empirical study on exiting game testing techniques. In particular, at its early stage, the initial version of *GBGallery* contains 5 industrial games from *NetEase*. Overall, we spend a lot of manual effort on the bug analysis and collection based on the 5 game commit histories, issue lists, product quality management reports and other records of game development. Eventually, 76 real bugs were selected and we manually injected them into the 5 corresponding games, forming the game bug database. In this process, lots of engineering work was required for building *GBGallery* such as history bug analysis, bug injection and reproducing, framework construction, which takes more than 6 man-months efforts.

Based on the bug database, we further propose an automated framework to enable the reproducible study of game testing research. The framework is designed by taking usability, flexibility, and extensibility into consideration. It provides multiple interfaces (i.e. , command-line interface (CLI), graphic user interface (GUI), reinforcement learning interface), which allows flexible configuration on bugs, oracles, *etc.* It also supports essential analytics for game testing, e.g., statement coverage, branch coverage, state coverage, bug



(a) Game Bug Database and Framework Construction.

(b) Empirical Study.

Fig. 1 The overview and summary of *GBGallery*, including game bug database construction, game testing framework construction, and empirical studies for benchmarking game testing techniques

detection, and incorporates 5 current automated game testing techniques for comparative studies. In addition, the *GBGallery* framework is extensible, where new game bugs or game testing techniques can be easily integrated into *GBGallery*. We further perform large-scale empirical studies to evaluate different game testing techniques to identify the main challenges and potential opportunities, providing the basics and guidance for further game testing research.

3.1 Subject game collection

To create a representative game bug database, the subject games and bugs are the most important factors. Although we initially intended to also include publicly available open-source games, our early attempts to collect open-source games found that most open-source games are not mature and ready for game bug database inclusion, which has few bug records and relevant information. Thus, we selected 5 commercial games from *NetEase* (see Table 1) as the subjects, which contain well-maintained game development information, e.g., source code, quality records and issues from the bug tracking systems. These games were selected based on the following rules: 1) a candidate game should have a bug tracking repository that is well maintained such that we can easily analyze the previous issues. This enables to increase the selection opportunity for subject bugs in *GBGallery*; 2) the selected games can be granted open access by *NetEase* considering the commercial regulation and policy in the company; 3) we consider the size of the game and the complexity of bug analysis. Game products with too high complexity often require too much manual effort for analysis, which can be difficult to complete within the reasonable time. On the other hand, games that are too small or simple often do not have many bugs for injection. Moreover, since there are often a number of internal game versions for a commercial game, we only select one internal version that is representative, stable and easy to integrate the game bugs.

Specifically, *Fever Basketball* (*FB*), *Tower Defense* (*TD*), and *Tank* are currently active in the commercial operation status, which forces us only be able to release the binary code based on the company's rules. For *A Chinese Ghost Story* (*CGS*) and *Justice Online* (*JO*), although they are currently popular and there are quite lots of users, we eventually obtained the permission to release the source code of a part of the game to be included in *GBGallery*, i.e. , typical scenarios of battles and mission scenes in the game.

Table 1 summarizes subject game information. Column #Tot_Issues and Column #Sel_Issues show the number of history issues and selected issues over all the analyzed documents (i.e. , commit histories, quality assurance records, product reports, bug tracking system and meeting records), respectively. We eventually found more than 11,000 history bugs of *JO* and 4,400 bugs of *CGS*, but due to the size of the granted game versions[1], our study considered 120 and 113 issues from the issue histories of *JO* and *CGS*, respectively. These games were developed with a mixture of Python, Lua, C++, Visual Basic and C# code. In particular, the information processing and network data flow control of all games are developed in Python. Other game modules such as major logic, graphic rendering and game resource loading are mainly built with Python, Lua, C++, Visual Basic and C#. Most of the game bugs are distributed in those modules developed by C# and Python, which are mainly used to construct the game logic and control system.

A brief description of each game is summarized as follows.

[1] The full game version is not granted due to the permission restriction.

Table 1 Subject game information of *GBGallery*

Game	Programming language	#LOC	#Tot_Issues	#Sel_Issues
Fever basketball	C#, Python, Lua	229,103	4,903	67
Justice online	C#, Python	1,697	120	19
A Chinese ghost story	C#, Python	1,271	113	15
Tower defense	C#, Python, VB	16,861	1,604	28
Tank	C#, Python, C++	8,053	691	14

- **Fever Basketball (FB):** An online basketball competition game that allows two teams to play against each other in a half court, where each team has 3 players. A user controls one of the players to perform certain actions (e.g., ball passing, shooting, defense, rebound, *etc.*). *FB* also involves builtin-AI to collaborate with players. Each team needs to score as much as possible to defeat its opponents.
- **Justice Online (JO):** A large-scale and popular MMORPG currently with over 1 million peak daily active users. This game has rather diverse elements, including a complex task dispatch system and a variety of props. For *JO*, bugs usually occur in the middle to complete a game mission, e.g., prop trading, mission acquisition, mission update, *etc.* Together with the *JO* game developers, we select 2 representative missions in this game and develop a mirroring open-source version.
- **A Chinese Ghost Story (CGS):** *CGS* is an online combat scene role-playing game (RPG). We follow a similar process as *JO*, select 3 different professions and develop the mirroring open-source version (a representative combat instance from the full game) for its inclusion into *GBGallery*.
- **Tower Defense (TD):** A tower defense game with independent client and server, developed in C#, Python and Visual Basic. *TD* requires players to fight against builtin-AI or other online players, and to defend against monsters attacks by building more towers. The game logic is rather complex and requires players to balance between the number and quality of defense towers properly.
- **Tank:** This is a tank battle game, in which players can choose a variety of types of tanks and control a tank to attack other tanks that are built-in non-player characters (NPC). Each tank has 3 different attacking skills that can be used by the player. In addition, the moving direction of the tank is not limited to only up, down, left, and right, but can be in a smooth 360-degree direction.

3.2 Bug collection and bug database construction

We select the bugs based on the following steps: 1) we try our best to collect issues as more as possible, which are from the internal quality records, bug trackers, commit history, product development documents and meeting minutes; 2) we manually analyze and select issues which were relevant to the game implementation. Specifically, the selected bugs should have been successfully fixed, since we intend to incorporate both the buggy versions and corresponding fixed versions; 3) we also remove the bugs that are relevant to sensitive functions with the commercial restrictions of the company (e.g., the payment process); 4) we only select bugs that were compatible with the game versions that can be accessible; 5) we address disagreements by voting and seeking advice from professional game developers and teams.

After these steps, the selected issues have been reduced a lot and some categories of bugs cannot be injected into the games. Specifically, in *FB*, we were granted a game version that can only support several basic roles, and bugs that related to other roles (e.g., customized roles with different clothing and skills) were excluded, thus most of the game bugs based on the customized roles have to be removed as well. In *JO* and *CGS*, we were granted the mirroring open-source versions that contain 2 representative missions and 3 different professions, respectively. Therefore the selection scopes of game issues were limited to those bugs related to the granted game versions, which reduces the number of injected game bugs. For example, we cannot inject the *Display* bugs into these 2 games since the selected game versions do not include the GUI modules. In *TD* and *Tank*, although we obtained a complete latest game version, these 2 games were developed 5 years ago, we can access the issue information but cannot find some related programs and resources (e.g., old database) due to the updates of game versions. It is worth mentioning that the updates between the buggy version and the fixed version may be quite large, e.g., the framework update, game engine update, new data types and structure, new elements and objects. Additionally, in all games, many of bugs were security-sensitive and related to the payment system, commercial events (e.g., game character discount sales in Christmas), which post concerns for opening and therefore were excluded.

We performed an in-depth analysis of the selected issues and evaluated the difficulty of injecting corresponding bugs into the automated framework. We select one version that was more suitable to inject these bugs. Although all the selected issues are selected from the 5 games, some of them are incompatible with the selected version. Specifically, the update between the buggy version and the selected version may be quite large, e.g., the framework update, game engine update, new data structure and objects design. Thus, these bugs are difficult to be injected into the version and we remove them. Finally, we include 76 bugs in *GBGallery*, which can be inserted and reproduced in the selected version. It is worth mentioning that we used collective voting when there are some disagreements in the bug injection.

We analyzed the triggering condition of each bug and reproduced them in the target version. Specifically, we inserted the bugs in 3 different ways: 1) some bugs can be triggered only if the corresponding code snippet is covered and executed such as game display bugs, crash and stuck bugs. For these bugs, we manually extracted the buggy code snippets from the commit history, inserted it into the code file (mainly C# and Python files) besides the fixed code snippet, and use a switch (i.e. , a Boolean variable) to control the program logic and determine if the bug can take effect; 2) Some bugs break the game logic and balance because the wrong data of a critical attribute has been loaded (e.g., attack value, number of reward coin, cool down time of a skill, *etc.*). These data are not stored in program files but in data files such as game design tables, so we modified the corresponding data to trigger the bugs; 3) Some bugs have triggered conditions conflict with others (e.g., 2 bugs are dependent and affect the same module), we manually revised the game code and make sure these bugs can be triggered in the same game version independently. All the injected bugs are tested and manually checked to ensure that the insertion operation is correct, effective, and reproducible.

Overall, there are a total of 6 people including 3 professional game developers from *NetEase* in our work. We spent more than 6 man-months effort to build the game testing framework including bug collection, analysis and framework development. Specifically, we spent 1 man-month effort to select games, analyze the issues and collect bugs. Then, we spent 2 man-months effort to analyze the bugs including investigating the bug video and report, and selecting the suitable game versions for injecting these bugs. Finally, we took another 3 man-months to inject and reproduce these bugs, develop the functionality of the framework (e.g., test case replay, RL interfaces, testing baselines). This whole process is rather time-consuming and requires a lot of engineering effort.

Table 2 summarizes the distribution of bugs in these games. To better understand these bugs, we summarized these bugs into 5 categories. The reason of introducing the categories of bugs into *GBGallery* are: 1) these categories are defined by the game experts. They cover the common game bugs in *NetEase* and have high priority in the company. 2) Such categories correspond to the oracles, where different types of bugs can be captured by different oracles.

- **Crash:** The game exits abnormally when these bugs are triggered. The crash bugs may be caused by division by zero, memory management, object recycling, *etc.* Crash is one of the most critical bugs that can introduce bad experience and security issues.
- **Stuck:** The graphical user interface (GUI) or background service is stuck, preventing the user from continuing. Our collected stuck bugs are largely caused by the delayed response, the infinite loop, game design, and *etc.*
- **Game logic:** Except for crash bugs and stuck bugs that are relatively easier to observe, the collected real bugs contain many logic bugs that are difficult to discover by testers or perceived by players. For example, some skills do not take effect under the corner case; Some game states could not be reached; Some actions might violate the game's setting or some abnormalities caused by improper decimal truncation.
- **Game Balance:** Game balance issues are non-functional bugs, which may affect the experience of players. The balance bugs are usually caused by poor design or improper difficulty setting. Specifically, if some tasks are too hard (e.g., the monsters are too strong to defeat), players may lose confidence and give up the game. On the contrary, if a task is too easy, the user can quickly complete the game and lose interest. The *Game Balance* bugs in *GBGallery* are usually caused by the unsuitable game configuration (e.g., the configurable parameters for the monster, attack speed and attack effects).
- **Display:** There are some rendering issues including GUI or audio playing, which may or may not affect the normal playing of games. For example, some displayed issues may be caused by the inconsistency between the game client side and server side, the resource files loading failure and *etc.*

3.3 *GBGallery* framework design and implementation

Based on the constructed game bug database, we build an automated and extensible framework, named *GBGallery*, to enable reproducible game testing. In general, *GBGallery*

Table 2 Inserted bugs from selected issues on each game

Types	FB	JO	CGS	TD	Tank	Total
Crash	2	–	1	5	2	10
Stuck	3	–	–	2	2	7
Logic	11	8	4	11	5	39
Balance	2	3	4	4	2	15
Display	3	–	–	2	–	5
Total	21	11	9	24	11	76

framework is designed with the following properties: 1) detailed introductions and documents to make sure users can access and use the framework with less effort, 2) flexible bug configuration, i.e. , a bug can be easily configured to be turned on or off, 3) the packaged execution environment for all 5 games which makes it easier to use, 4) bugs are reproducible with accompanying tests, 5) the flexible interfaces (e.g., RL, search-based) for users to apply the existing DRL methods and integrate newly proposed testing techniques, 6) the state-of-the-art game testing methods can be easily integrated and compared, 7) different kinds of oracles which are used to test if the bugs can be detected and 8) a detailed public repository contains the issue list and other information to ease the usage.

Introduction Documents. We developed the documents for *GBGallery* including the tutorial for game environment setup, game program introduction, and the usage of the game testing framework.

Configurable Bugs. In our bug database, for each game, we integrate all the selected bugs into a single *clean* game version. We adopt a fined-grained flag-based approach, widely used in some software product lines, to control whether the faulty relevant code fragments are enabled or not. This makes our framework flexible in controlling each bug activation status in a fine-grained way. In particular, for each game, we provide a bug configuration file and a Boolean flag parameter to configure whether the bug is turned on or off for analysis.

Unified Environment. We have packaged the game in an execution environment. To be specific, *FB*, *TD* and *Tank* have the compiled client which must be executed on MS-Windows system, the other games and modules of *GBGallery* can be executed both on MS-Windows and UNIX-style systems, such as the game servers and testing frameworks.

Reproducible Tests. As a shred of important evidence to confirm the existence and enable bug investigation, in *GBGallery*, we provide at least one test for each bug, which enables bug reproducing and further comparative testing analysis. To better support the analysis of tests generated by automated tools, we provide the functionality to replay a given test case (action sequence) on the game, where the replay supports both GUI and CLI to enable the observation of concrete execution and in-depth investigation by users.

Game State Abstraction and Reinforcement Learning Interfaces. A typical RL environment includes state acquisition, action input, reward acquisition, game reset, and *etc.* . Among these, the design of game state representation can be a key issue for efficient game testing and RL strategy learning. Existing RL frameworks (e.g., OpenAI Gym) mostly adopt 2 ways for game states representation: image-based states (Mnih et al. 2013) and vector-based states (Brockman et al. 2016). In *GBGallery*, we used vector-based states representation, that is, interval status of the game environments such as health points (HP), magic points (MP), position of the player/NPC and the time used, were sequentially summarized as a vector. The information in the state vectors can help to tune the deep neural network and finally generate a useful policy for the game agent. We referenced the methods and details of designing states in Brockman et al. (2016).

Usually, there is not a best standard way to design the game state representations for RL training. We could use different state representations (e.g., the screenshot, the internal values of games), which may have different granularity (i.e. , different game information). The states with different granularity may lead to different performance for training the DRL model. In particular, we developed game states by 3 steps: (1) we analyzed the characteristics and mechanism of the games, and extracted all variables related to the progress of the game, such as HP, MP and scores; (2) we summarized the values of these key variables as a state vector (all the state vectors keep the same formality and length but differ from values at each moment); (3) we integrated the internal reinforcement learning policies in *NetEase* and public algorithms (Hill et al. 2018) on 5 games, and use the state vectors as input data to train the polices of the agents. We verify whether these agents can learn useful knowledge from the state vectors, and gradually improve the game performance.

Additionally, we defined possible actions of the game based on one-hot encoding. The user only needs to input the unique action ID during testing and the action can be automatically interpreted and executed in the game environment. We also developed other interfaces such as resetting (i.e. , restart the game), initialization (i.e. , prepare the game environment) and obtaining observation (i.e. , load the real-time states from the game).

Automated Game Testing Baselines. To enable comparative studies, we also tried our best to collect classic and state-of-the-art game testing techniques as the baselines. The baselines are representative automated game testing techniques with different strategies (e.g., random-based strategy, reinforcement learning (RL) based strategy DQN, A2C, A2C+C, and combined RL and genetic algorithm-based strategy *Wuji* (Zheng et al. 2019)). Users can propose a new approach and conduct a comparison with them.

Built-in Oracles. The oracles we inserted for each type of bugs are summarized as follows. It is worth mentioning that the built-in oracles for all 76 bugs cannot be directly accessed and modified by users, since most of them were implemented into the compiled game client.

- *Oracle 1 (Crash):* The crash oracle directly checks whether the game process is terminated.
- *Oracle 2 (Stuck):* In general, game stuck status can be difficult to detect. One way is to check whether a particular action is completed within a certain time limit. In addition, since we know the triggering condition for each stuck bug, a specific oracle that checks whether the triggering condition is satisfied is also introduced. For example, once a special game state is reached, the game will get stuck.
- *Oracle 3 (Game logic):* We directly insert assertions to check whether the desired logic constraints are violated.
- *Oracle 4 (Game Balance):* With the understanding of the balancing constraint of each case, we design the oracle to check whether the expected balance properties are violated, e.g., the feasible time to complete a target task cannot go under a minimum threshold value, the health point of a character (e.g., monster) should not go beyond a maximum threshold that should not occur in practice.
- *Oracle 5 (Display):* The display issue is difficult to detect automatically (e.g., the graph rendering and the audio effects). The display issues in *GBGallery* can be captured by checking the inconsistency of data flow and status of the display modules between the client and server, e.g., whether the displayed text can match the back-end computation.

3.4 The major components and usage of *GBGallery*

In this subsection, we introduce the major components of the bug database and framework, usage of *GBGallery*, e.g., how the *GBGallery* can be used to support the testing of video games. More detailed discussion can be found at the accompanied website of this paper (GBgallery 2021).

The Major Components of *GBGallery*. As an opening source game bug database and game testing framework, *GBGallery* contains 5 commercial games and 76 real industrial game bugs in total, and *GBGallery* contains 5 independent software packages. Each game contains the following elements:

- *Game Software:* The game entities (including the server programs and client programs) and unified environment setup packages. Some programs have been compiled into binary files and others are Python scripts, they can be directly executed.
- *Game Bugs:* The game bugs were planted in the game software. The details of the bugs can be found at our website (GBgallery 2021).
- *Configuration File:* The configuration allows users to make any number of bugs take effect. Users can configure the specific bugs by modifying the *JSON*-style file, i.e. , *bug_config.JSON*.
- *Bug Oracles:* The built-in bug oracles are in the compiled programs.
- *DRL Interfaces:* A set of reinforcement learning interfaces in *Python* files.
- *Baselines:* 5 existing automated game testing strategies.
- *Testing Frameworks:* We prepared a Python script named *start_testing.py* in each game package, in which the testing framework can be executed.

The Usage of *GBGallery*. The users can use *GBGallery* as follows: (1) run the game client and server; (2) configure the game bugs required for testing; (3) configure the integrated testing baselines and starting to run the game testing framework; (4) design their own testing strategies. For (1)-(3), the users only need to modify the configuration files (in *JSON*-style) and run the specific programs. For (4), the users can develop the customized algorithm based on the provided sample solution. In summary, with the game bugs in *GBGallery*, users can understand the challenges of automated game testing, implement their own algorithms and compare the algorithms with integrated baselines.

4 Empirical study

Based on the proposed framework, we perform a large-scale empirical study to investigate existing game testing methods (i.e. , the integrated baselines in *GBGallery*). In particular, with *GBGallery*, we first investigate the code coverage and state coverage that can be achieved by the 5 testing strategies (RQ1 and RQ2). Then, we evaluate the bug detection capability of these techniques (RQ3). We further analyze the correlation between different coverage and the capability of bug detection (RQ4). At last, we perform an in-depth analysis on those bugs that are not detected by existing techniques and summarize the main challenges that need to be addressed in future game testing research (RQ5).

4.1 Evaluation setup

We test each of the 5 games with the 5 testing strategies in *GBGallery*. For the RL based strategies, including *DQN* (Hester et al. 2018; Mnih et al. 2013) and advantage actor-critic *A2C* (Konda and Tsitsiklis 2000). *DQN* is one of the most famous DRL algorithms, which uses DNN to represent the *q-table*, it drives the agent to interact with the environment and learn a decision policy based on the feedback of each action, thus the agent can select those actions which can bring more rewards from the environment in each state. *A2C* is a more complex RL algorithm which uses a DNN model *Actor* to take the state from the environment as input and output an action decision for the agent, and a *Critic* to criticize the actions made by the *Actor*, thus the policy can be updated and improved. We referenced the curiosity-based random network distillation (Burda et al. 2018) on *A2C* to formulate the testing baseline *A2C+C*, which is more exploratory than *A2C* because of the extra reward if the agent can meet more novel states in the environment. *Wuji* is a combination of evolution algorithm (EA) and reinforcement learning, which is designed to learn a policy by RL, to avoid the local optimum and further explore more states by EA. In *Wuji*, we also followed the basic settings from the original publication (Zheng et al. 2019).

Each run of the testing strategy is allocated a time budget of 12 hours. We have tried to set a longer time budget and it shows that after 12 hours for each single testing task, the testing results become stable and the learning-based policies rarely change. To counteract the randomness during the testing process, we repeat the execution of each configuration 5 times and calculate the average results. Overall, the game testing takes a total of 1,500 (=5*5*12*5) CPU hours.

In particular, for RQ1, we analyze the statement coverage and branch coverage achieved by different testing strategies. For RQ2, we collect how many states (via state discretization) are explored on a target game for measuring the exploration baselines. Note that the state of each game is defined based on the domain knowledge of developers. For RQ3, we analyze the number of bugs detected by each method. Based on the results of RQ1, RQ2 and RQ3, we measure the correlation between the code (state) coverage and the number of bugs detected in RQ4. For RQ5, we perform a manual analysis of un-detected bugs and study possible reasons. All the experiments are run on 4 servers with the same configuration, i.e. , CPU (Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHZ) with 64 cores and 128G RAM.

4.2 RQ1: Statement and branch coverage

Tables 3 and 4 summarize the averaged statement coverage and branch coverage obtained by each method, respectively. The first column lists the subject games and the first row shows different testing methods. Overall, we can observe that the code coverage of all testing methods is not very high. However, different strategies obtain similar code coverage results with only slight deviations, i.e. , ≤12.35% and ≤8.98% difference for statement coverage and branch coverage, respectively, across all cases.

Through analysis of code coverage, and discussion with the game development team in *NetEase*, we found that the core functions are often very easy to be covered with only a few steps while some code fragments that are relevant to the game resources (e.g., display, data reading, model loading) are difficult to be covered. For example, in *FB*, the corresponding code fragments of basic actions such as shoot, pass, defense, steal and congratulation, are

Table 3 Average statement coverage (%)

Items	Random	DQN	A2C	A2C+C	Wuji
FB	44.51	42.39	**47.62**	46.58	46.30
JO	**59.40**	47.05	58.53	56.89	56.03
CGS	47.13	46.18	46.52	45.13	**47.31**
TD	51.75	52.60	**57.13**	56.82	50.06
Tank	72.12	**78.35**	75.06	75.00	77.48
Average	**59.98**	53.31	56.97	56.08	55.44

Table 4 Average branch coverage (%)

Items	Random	DQN	A2C	A2C+C	Wuji
FB	32.45	33.33	31.00	34.55	**37.40**
JO	58.63	53.90	57.80	**58.76**	52.40
CGS	**43.03**	42.21	42.35	41.26	41.67
TD	50.69	51.15	50.30	50.94	**59.17**
Tank	62.80	64.16	62.67	62.69	**67.77**
Average	49.52	48.95	48.82	49.64	**51.68**

easy to be covered, but those advanced skills (e.g., *HookShot*) are hardly triggered, neither the functions of loading relative animation model.

We also found that almost all testing strategies can achieve a certain level of code coverage within a very short time. In particular, statement coverage and branch coverage are often saturated in less than 30 minutes. We executed the 5 different techniques repeatedly and observed that most of the statements/branches covered by 5 strategies are the same, only a small part of code they covered are different from each other. The reason is that most code logic is highly piled in particular regions, most of which are basic modules and functions of the game. And the statement or branch can be covered only if the game enters the related scenes, but reaching such game scenes requires the testing strategies to be smart enough to complete the up-streaming missions. For example, in *Tank*, if the player shoots its enemy by a bullet, the specific program logic related to the bullet management and distance calculating will be mostly covered and the coverage data will not change much by further actions or tests. But if the player cannot win the game, then those programs related to round reset, reward settlement, difficulty upgrade would never be covered.

There are some code fragments related to the core functionalities that the agent cannot cover because some corner game states are hard to be reached. For example, some advanced skills or special actions are required to reach the game states. In *FB*, the skill *NicePass* can only be triggered if the player successfully passes the ball under the very tight defense condition, which is often hard to trigger.

Answer to RQ1: *Different testing strategies achieve similar code coverage results on each of the studied games, indicating that code coverage is coarse for testing sufficiency measurement of the game software. Code related to basic functions can often be easily covered during game playing, while corner-cases need to trigger specific conditions.*

4.3 RQ2: State coverage

As shown in RQ1, code coverage can be coarse and insufficient to reflect the game testing sufficiency. We evaluate the coverage of game states in this section. Different from the code coverage, states of games require to be defined based on the domain knowledge of the games. In particular, the full state representation of each game is defined by the game developers based on their domain knowledge. Then, we discretize and abstract the concrete game states, where each variable in the game state vector will be normalized to a reasonable range and divided into finite intervals, based on which our state coverage is measured.

Table 5 summarizes the state coverage achieved by different testing strategies. Compared with code coverage, the state coverage achieved by different strategies is diverse. Overall, for state coverage, we found that no single method outperforms the others in all cases. *DQN*, *A2C+C*, and *Wuji* maintain a relatively high state coverage on all 5 games. Based on the reward definition, we found that RL-based strategies often perform well in the complex game environment.

We can also see that *Random* can also achieve certain state coverage in some cases. In particular, it achieved the best result in *TD* because 1) the current *TD* version in *GBGallery* is not large, i.e. , there are not many complex operations and logic, 2) the pre-defined reward of RL may limit the exploration (e.g., tends to accomplish missions) and 3) there can be some randomness in this game, e.g., the location of new tower and type of casted skills are all randomly assigned, which greatly affects the policy learning process.

The state coverage achieved by testing methods could be different with different initial states. Therefore, in *GBGallery*, we defined the initial state of testing as the starting point of the game, which may make that some scenes and states of the game never be covered. This is also one of the reasons for such a relatively low state coverage. In addition, although the covered state spaces overlapped in *GBGallery*, different test methods still covered some different states, this is why they detected different bugs.

Answer to RQ2: *Compared with code coverage, state coverage is shown to be a more fine-grained indicator for game testing. Our results show that no single method in the 5 studied methods outperforms all others in terms of state coverage. Reinforcement learning-based methods can achieve higher state coverage in many cases.*

4.4 RQ3: Bug detection

Table 6 summarizes the average percentage of bugs detected by different strategies and the corresponding boxplot results are shown in Fig. 2. In general, all strategies can only detect a small number of bugs. For example, *Wuji* detected the largest number of bugs, which is only 23.16% of the bugs in *GBGallery*.

DQN, *A2C+C* and *Wuji* outperform *Random* and *A2C* in all cases, i.e. , more bugs are detected in each game. Compared *A2C* with *A2C+C*, we found that the curiosity mechanism can improve the capability of state exploration and reach more diverse game states. For example, *A2C+C* achieved higher state coverage (see Table 5) and detected more bugs.

In Fig. 2, we can see that although the average number of bugs detected by *Random* is generally less than other methods, it occasionally detected more bugs on *JO* and *CGS*,

Table 5 Average game state coverage (%) of each method

Items	Random	DQN	A2C	A2C+C	Wuji
FB	36.48	46.90	43.15	46.87	**59.43**
JO	56.47	54.00	55.87	**66.25**	54.56
CGS	57.59	**72.90**	51.54	54.50	59.11
TD	**38.96**	38.02	36.77	38.24	30.94
Tank	24.97	10.92	21.00	**52.10**	43.98
Average	42.89	44.55	41.67	**51.59**	49.60

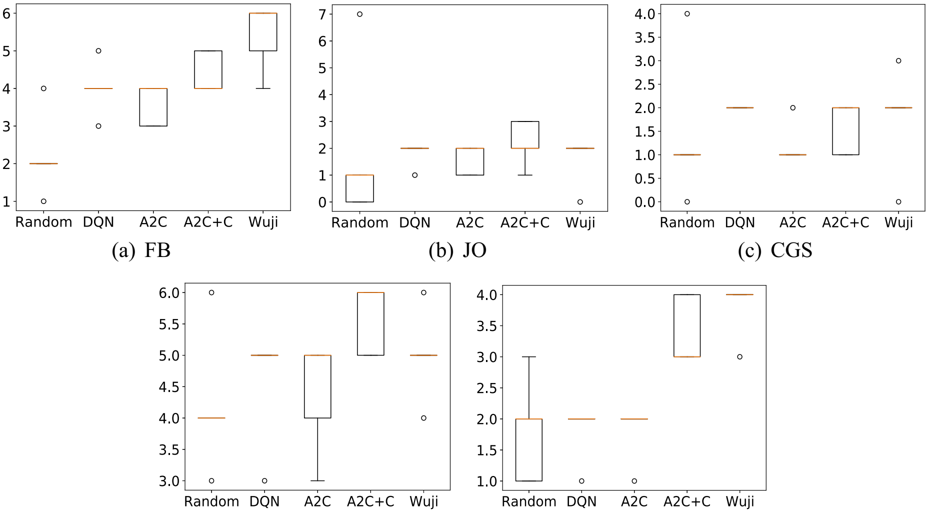
Table 6 Average percentage of detected bugs on each game. (%)

Items	Random	DQN	A2C	A2C+C	Wuji
FB	10.48	19.05	17.14	20.95	**25.71**
JO	16.36	16.36	14.55	**20.00**	14.55
CGS	15.56	**22.22**	13.33	17.78	20.00
TD	17.50	19.17	18.33	**23.33**	20.83
Tank	16.36	16.36	16.36	30.91	**34.55**
Average	15.00	18.68	16.58	22.63	**23.16**

indicating that the reward mechanism may miss some game states that can be reached by *Random*.

Our further analysis shows that bug detection depends not only on the testing strategy but also on the characteristics of games. Some games (e.g., *TD*, *FB*) are strategic games that need better strategies with some level of intelligence to complete the mission. RL-based approaches can perform much better on such games. Some games rely more on operations instead of the intelligent strategy. For example, *CGS* requires real-time actions and a high response speed of the player and the strategy is not so dominant. Thus we found that all methods achieved similar results in terms of detected bugs on *CGS*.

Table 7 shows the average percentage of detected bugs by 5 testing tools on each category. Table 8 shows all the failures caused by the game bugs in *TD*, including the bug descriptions, categories, and detected by which method. *Wuji* detected the most *Game Logic*,



(a) FB (b) JO (c) CGS

(d) TD (e) Tank

Fig. 2 The box-plots of detected bug number by each technique on the corresponding games

Table 7 Average percentage of detected bugs on each categories. (%)

Items	Random	DQN	A2C	A2C+C	Wuji
Crash	18.00	20.00	16.00	**36.00**	22.00
Stuck	11.43	**25.71**	22.86	20.00	20.00
Game Logic	17.95	19.49	17.95	22.05	**25.13**
Game Balance	6.67	6.67	5.33	**14.67**	**14.67**
Display	12.00	36.00	28.00	28.00	**40.00**

Game Balance and *Display* bugs on all 5 games, and *A2C+C* detected the most bugs on *Crash* and *Game Balance*. In addition, *DQN* detected the most *Stuck* bugs. From the results, we observe that *Wuji* and *A2C+C* can detect more game bugs on different categories and RL-based methods are better than *Random* policy due to the numbers. This trend is consistent with the result of the average number of bugs in all categories. In addition, the *Game Logic* bugs are the most detected category.

Figure 3 shows the unique bugs detected by different testing tools as well as their overlap. It shows that, in the single subject game, there is at least one bug that can be uniquely detected by a testing tool. For example, in *CGS*, 2 bugs can only be detected by *Random* and one bug can only be detected by *Wuji*, but in *JO* and *TD*, *DQN* and *A2C* also detected one unique bug, respectively. But overall, *Random* and *Wuji* can detect unique bugs on 4 games which are the most in all methods.

Therefore, even the testing tool that detects the most bugs can not cover all test scenarios (e.g., *A2C+C* and *Wuji*). On contrary, some testing tools (e.g. , *Random*) may not be competitive in terms of overall results, but they can still detect some unique bugs that cannot be detected by others. Based on the testing results from §4.2 and §4.3, a promising direction is to combine the different testing strategies for better testing performance.

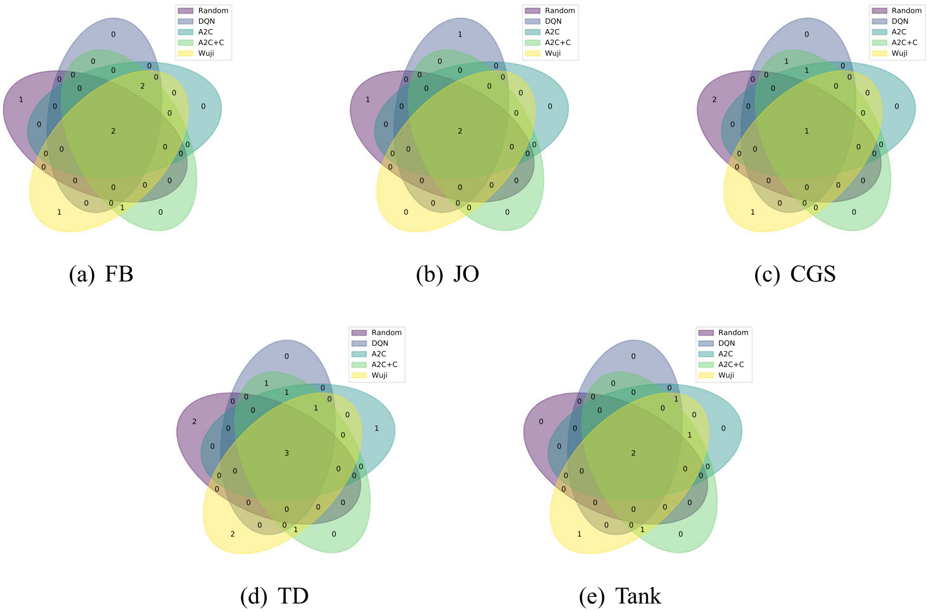
For example, *Bug_9* and *Bug_17* can only be detected by *Random*, and the trigger conditions of these 2 bugs are not related to the game process, but requires some rare actions and strategies. For *Bug_9*, the attack value is the sum of all the attack power of towers on the map. If the player destroys all the towers and leaves the map with no defensive tower, the attack value on the screen should be 0 but it still shows 10, which is the initial attack value. In *Bug_17*, if all the towers are destroyed, the status of the tower is updated to be *destroyed* and the level should be 0. However, the actual tower levels are all set to the initial tower level one. These conditions cannot be covered by other RL-based testing tools, since adding towers always earn reward and destroying tower loses reward. This mechanism tempts the greedy agent to select those actions that can bring more rewards but eventually miss the bugs.

On the contrary, *Bug_16* and *Bug_20* are detected by *Wuji* and *A2C*, respectively. The bug triggering conditions of these 2 bugs depend on the game progress. Specifically, players can use certain skills only if the game proceeds to the last 45 seconds. The bugs can be triggered only if these skills are used. These bugs can be detected by RL-based methods rather than *Random* because only RL-based methods can learn how to play the game better.

In summary, we can leverage the advantages of different testing strategies by considering both the game playing and exploring more rare states, the testing results will be more promising. We believe such a combination (e.g., *Random* and RL-based) can achieve better results.

Table 8 A brief description of the failures due to the corresponding bugs detected by all 5 test tools on *TowerDefense*

Bug_ID	Description of Failures Caused by the Game Bugs	Categories	Detected by Methods
Bug_0	Failed to summon a special defensive tower	Crash	Random, DQN, A2C, A2C+C, Wuji
Bug_1	Correctly calculated but wrongly updated tower buff value	Game Logic	–
Bug_2	Player rarely summoned some types of tower	Game Logic	–
Bug_3	Calculation error when adding buff for player’s towers	Game Logic	Random, DQN, A2C, A2C+C, Wuji
Bug_4	when delete a tower the linked tower should be removed together but failed	Game Logic	–
Bug_5	Enemy’s tower is degraded to level-0 but the lowest level in this game is 1	Game Logic	Random, DQN, A2C, A2C+C, Wuji
Bug_6	The tower has been upgraded but the increased attack power did not change	Game Balance	Wuji
Bug_7	Calculation error of player’s influence range of the skill casting	Game Balance	–
Bug_8	Destroy the tower but did not free its skill object	Game Logic	–
Bug_9	The displayed energy does not match the actual value in the program	Display	Random
Bug_10	The game crashes when open the treasure box (a reward after the game)	Crash	–
Bug_11	There are more than 12 (The maximum of towers) in the map	Game Logic	–
Bug_12	Failed to update the tower attack value on screen	Display	DQN, A2C, A2C+C, Wuji
Bug_13	The HP of last monster is too high that player cannot defeat it	Game Balance	–
Bug_14	Stuck when using skill *speed up gear*	Stuck	DQN, A2C, A2C+C
Bug_15	The game crashes when upgrade a tower to full level (level-5)	Crash	DQN, A2C+C
Bug_16	The skill *reinforce gear* does not work	Game Logic	Wuji
Bug_17	Failed to update the defensive tower’s level	Game Logic	Random
Bug_18	A monster tried to use skills but failed and cause the game crashed	Crash	–
Bug_19	The tower object was freed but a monster cast skill on the tower	Stuck	–
Bug_20	The skill *back stab* causes crash	Crash	A2C, A2C+C
Bug_21.	Died monster’s body should become a bomb, but it does not work	Game Logic	–
Bug_22	The player’s *HP* is too high to win the game all the time	Game Balance	A2C+C, Wuji
Bug_23	Failed to randomly create a defensive tower for player	Game Logic	Random, DQN, A2C, A2C+C, Wuji



(a) FB (b) JO (c) CGS

(d) TD (e) Tank

Fig. 3 The overlapped bug numbers (in 5 runs) detected by different testing tools on 5 games

> **Answer to RQ3**: *All the evaluated testing strategies can detect a number of bugs in the games, but there is still much space for improvement. There is no one single best strategy that can detect more bugs in all games. And a promising strategy is to combine different policies for testing. Overall, A2C+C and Wuji outperformed other methods in many cases.*

4.5 RQ4: Correlation

To further investigate how the coverage metrics indicate the capability of bug detection, we measure the correlation between the achieved coverage (i.e. , statement, branch, and state) and the number of detected bugs. In particular, we adopt the *Kendall* τ coefficient to calculate the correlation, which is also widely used in previous work (Inozemtseva and Holmes 2014). *Kendall* τ is a statistical indicator used to measure the correlation between two variables (e.g., the code coverage and the number of detected bugs). The scale of τ is between -1 and 1, which means that two variables have completely negative correlation and positive correlation, respectively. When τ is 0, two variables are independent.

Table 9 shows the correlation between different coverage obtained and the number of detected bugs. We can observe that statement coverage, branch coverage and state coverage all have a positive correlation with bug detection, indicating that the reason of low code coverage and missed bugs is that only those bugs on the covered elements can be detected. Therefore, the testing strategies need to advance the game progress as much as possible, to cover more game scenes and elements, in order to better improve the game testing performance. On the other hand, the state coverage weakly outperformed the other 2 indicators on the 3 of 5 games in terms of achieving a higher positive correlation score. An exception

Table 9 Correlation between coverage and bug detection

Coverage criterion	FB	JO	CGS	TD	Tank
Statement	0.333	0.215	0.276	0.333	0.447
Branch	0.333	0.645	0.276	0.067	0.149
State	0.733	0.645	0.828	0.200	0.745

is that, *Wuji* detects more bugs on *TD* but achieves lower state coverage, which affects the correlation score.

Answer to RQ4: *The statement coverage, branch coverage and state coverage are all in a positive correlation with the bug detection, which indicates that a game tester should advance the game progress to cover more programs and scenes for detecting more game bugs.*

4.6 RQ5: Study on undetected Bugs

Although the baseline strategies can detect a certain number of game bugs, there is still a large space for further improvement. In this section, we show some case studies on bugs missed by different strategies and analyze the reasons. From Table 10 we observed that there are still many bugs that cannot be detected by all 5 methods, the difficulties of detecting these bugs are not related to the categories, and the missed bugs are distributed among almost all categories in the game. In general, the fundamental challenge is that the game space is usually very large (i.e. , containing too many states) and there is no clear knowledge on where the bugs occurred. We summarize two main reasons: 1) the limitation of testing strategies in game exploration and 2) the difficulty of the bug triggering condition.

4.6.1 Limitation of strategies on game exploration

Due to the large space of games and the difficulty in game playing, the exploration algorithm is especially important for game testing. For example, the state-of-the-art techniques *Wuji* considers both the exploration and the mission accomplishment based on multi-objective optimization. Based on the analysis of the undetected bugs, we found that existing methods are still ineffective in the exploration. Specifically, despite lack of intelligence, *Random* strategy can still obtain certain state coverage as some states (related to the basic functions)

Table 10 A summary of missed bugs in each category on different games

Items	FB	JO	CGS	TD	Tank	Total
Crash	1	1	–	2	–	4
Stuck	1	–	–	1	1	3
Game Logic	8	–	1	7	2	18
Game Balance	2	1	4	1	1	9
Display	2	–	–	–	–	2
Total	14	2	5	11	4	36

are easy to be covered. However, it may miss some critical states that are related to the game task because complex missions require an intelligent strategy to complete. Consequently, *Random* only detects fewer bugs. Based on the reward guidance, RL-based approaches perform better for accomplishing missions and achieving higher state coverage. However, it is still challenging to balance game exploration and accomplishment. We observed that the reinforcement learning converges after some time, which makes the trained model have a stable strategy (i.e. , play games with a fixed strategy), limiting its exploration.

As an example, In *JO*, we found that *Random* detected 4 distinct bugs that are not detected by other strategies. These bugs locate in special states, in which the task could have been completed. Thus, existing RL-based strategies are more inclined to finish this task before reaching such states because the task completion can have a high reward. *Random* could explore these states with a random exploration. Another confirmation is that *A2C+C* (adding curiosity guidance) performs much better than *A2C*, which demonstrates the usefulness of diverse exploration.

As another example, in *TD*, there is a bug that could be triggered if a special tower is upgraded to level 5. It is difficult to obtain a level-5 tower because: 1) too many gold coins are needed and 2) many level-1 towers are required to synthesize the level-5 tower. To win the game, there are usually two different strategies, i.e. , building a large number of low-level towers or a small number of high-level towers. Existing strategies tend to select the former strategy while the latter strategy is not explored. Thus, this bug is not detected.

The results also reveal a potential research direction, i.e. , how to improve testing exploration capability as much as possible while ensuring the completion of game missions.

4.6.2 Difficulty of bug trigger condition

During the testing, we observed that some bugs are quite difficult to be detected and special states are required to be explored. These special states are diverse and strongly related to the game characteristics therefore difficult to briefly summarize. And we will introduce some examples in the following.

In *FB*, there are 14 bugs that cannot be found by all strategies. For example, some bugs are triggered if the offensive player takes some special offensive actions (e.g., stop-jump shot) during a collision with the defender, some bugs are triggered. However, such a condition is difficult to trigger as the defender (builtin-AI) would try to avoid the collision that is likely to cause the foul. In *Tank*, the tank has the skill to recover the HP for friend tanks and itself within a small range. However, there is a logical bug that if the enemy is in the range, its HP can also be recovered. Due to the high attack speed between tanks, tanks are more likely to be destroyed if they are getting closer to the enemy. Differently, the friends can be in this range easily, making the bug hard to detect.

However, existing methods (especially the 5 methods in *GBGallery*) have no such knowledge and are unlikely to reach these states. The results reveal some potential research directions, e.g., how to improve the diversity of game playing to trigger some corner cases states.

Answer to RQ5: *The studied testing strategies are still challenging in bug detection and our study shows that the main reasons include: 1) the exploration of these strategies is limited and many states are not covered and 2) the trigger conditions of some bugs are difficult to be reached.*

4.7 Discussion of testing techniques in *GBGallery*

Based on the above experimental results, we believe that when we propose new automated game testing methods in the future, we can refer to the characteristics and results of the integrated methods. Although *Random* cannot complete game tasks and advance the game progress, it can generate many novel decisions (i.e. , actions) that learning-based methods cannot produce because its decision-making motivation is completely random. It can be seen from the RL-based method that an agent with learning ability can detect a certain number of game bugs in the test. Through the comparison of the experimental results of *A2C+C* and *A2C,DQN*, we found that if the testing methods can improve the exploratory then the agent can strengthen its testing ability. *Wuji* is an instance of improving the exploration of the agent. Its experimental results further proved the importance of the diversity of the game state for game testing.

From the test results achieved by the current testing methods in *GBGallery* still has space for improvement. When users apply the above testing methods, they currently need a lot of efforts to understand the game features and testing methodology, but the various interfaces we provide in *GBGallery* (such as state abstraction) can reduce the needed extra efforts to a certain extent.

5 Threats to validity

5.1 Internal threats to validity

The manual analysis process in our study can be an internal threat, e.g., bug and game selection, bug injection, development of interfaces, and analysis of experimental results. We follow the internal quality control process of *NetEase* to counteract such a threat. Even though, it is still possible that we might introduce some minor issues. To counteract this, we consulted the experience of professional game development teams as much as possible, and discussed those diverged cases among the authors, until reaching a consensus on each issue.

Although we tried our best to develop and test this framework, the implementation may be another threat, which might contain bugs and other reliability issues. For example, we only tested our framework in our described environment, some bugs may not be reproducible in new environments. Moreover, the usability including the GUI interface, the RL interface and other configurations may not be user-friendly.

5.2 External threats to validity

The external threat could from the datasets (i.e. , 5 games and 76 bugs). At its early stage, the scale of the database is relatively small, which could introduce some bias about our conclusion. We plan to integrate more games and bugs in the future. Another threat could be randomness in the testing process, to counteract this, we repeat each testing configuration 5 times and take the average results for the comparative analysis.

6 Conclusion

Automated game testing has been a longstanding challenge in the software engineering community, with intense industrial demands. In this paper, we proposed the first framework

GBGallery to enable reproducible game testing research, which includes 76 representative real game bugs in 5 industrial games. With *GBGallery*, we perform an empirical study to evaluate current game testing techniques. The results demonstrate that the existing automated game testing methods still have rooms for improvement, which is not effective in detecting game bugs and more advanced testing techniques need to be further investigated. In the future, we plan to 1) extend *GBGallery* by introducing more bugs and large-scale games towards facilitating the research on testing different games, 2) develop more methods to improve exploratory of existing testing methods and 3) combine the advantages of multiple automated testing methods.

Acknowledgments This work was supported in part by funding from the Canada First Research Excellence Fund as part of the University of Alberta's Future Energy Systems research initiative, Canada CIFAR AI Chairs Program, Amii RAP program, the Natural Sciences and Engineering Research Council of Canada (NSERC No.RGPIN-2021-02549, No.RGPAS-2021-00034, No.DGECR-2021-00019), the Ministry of Education, Singapore under its Academic Research Fund Tier 1 (21-SIS-SMU-033), as well as JSPS KAKENHI Grant No.JP20H04168, No.JP21H04877, JST-Mirai Program Grant No.JPMJMI20B8, and JST SPRING Grant.

References

Aleem S, Capretz LF, Ahmed F (2016) Critical success factors to improve the game development process from a developer's perspective. J Comput Sci Technol 31(5):925–950

Amann S, Nadi S, Nguyen HA, Nguyen TN, Mezini M (2016) Mubench: A benchmark for api-misuse detectors. In: 2016 IEEE/ACM 13th working conference on mining software repositories (MSR), pp 464–467

Amann S, Nguyen HA, Nadi S, Nguyen TN, Mezini M (2018) A systematic evaluation of static api-misuse detectors. IEEE Trans Softw Eng 45(12):1170–1188

Banerjee I, Nguyen BN, Garousi V, Memon AM (2013) Graphical user interface (GUI) testing: Systematic mapping and repository. Information & Software Technology 55(10):1679–1694

Borrelli A, Nardone V, Di Lucca GA, Canfora G, Di Penta M (2020) Detecting video game-specific bad smells in unity projects. Association for Computing Machinery, New York, NY, USA, pp 198–208. https://doi.org/10.1145/3379597.3387454

Brockman G, Cheung V, Pettersson L, Schneider J, Schulman J, Tang J, Zaremba W (2016) Openai gym. arXiv:1606.01540

Buglog (2015) Video game bug blog. https://airtable.com/universe/expEU1JW4I8ie2zOB/basic-video-game-bug-log

Burda Y, Edwards H, Storkey A, Klimov O (2018) Exploration by random network distillation. arXiv:1810.12894

Cadar C, Dunbar D, Engler DR et al (2008) Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI, vol 8, pp 209–224

Dallmeier V, Zimmermann T (2007) Extraction of bug localization benchmarks from history. In: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, pp 433–436

Do H, Elbaum S, Rothermel G (2005) Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empir Softw Eng 10(4):405–435

Fraser G, Arcuri A (2011) 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, pp 416–419

GBgallery (2021) https://sites.google.com/view/gbgallery

Hester T, Vecerik M, Pietquin O, Lanctot M, Schaul T, Piot B, Horgan D, Quan J, Sendonaris A, Osband I, et al. (2018) Deep q-learning from demonstrations. In: Thirty-second AAAI conference on artificial intelligence

Hill A, Raffin A, Ernestus M, Gleave A, Kanervisto A, Traore R, Dhariwal P, Hesse C, Klimov O, Nichol A, Plappert M, Radford A, Schulman J, Sidor S, Wu Y (2018) Stable baselines. https://github.com/hill-a/stable-baselines

Hutchins M, Foster H, Goradia T, Ostrand T (1994) Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In: Proceedings of 16th international conference on software engineering, pp 191–200

Iftikhar S, Iqbal MZ, Khan MU, Mahmood W (2015) An automated model based testing approach for platform games. In: 2015 ACM/IEEE 18th international conference on model driven engineering languages and systems (MODELS). IEEE, pp 426–435

Inozemtseva L, Holmes R (2014) Coverage is not strongly correlated with test suite effectiveness. In: Proceedings of the 36th international conference on software engineering, pp 435–445

Just R, Jalali D, Ernst MD (2014) Defects4j: A database of existing faults to enable controlled testing studies for java programs. In: Proceedings of the 2014 international symposium on software testing and analysis, pp 437–440

Khalid H, Nagappan M, Shihab E, Hassan AE (2014) Prioritizing the devices to test your app on: A case study of android game apps. In: 22nd ACM SIGSOFT international symposium on foundations of software engineering, pp 610–620

Konda VR, Tsitsiklis JN (2000) Actor-critic algorithms. In: Advances in neural information processing systems, pp 1008–1014

Lin D, Bezemer C-P, Hassan AE (2017) Studying the urgent updates of popular games on the steam platform. Empir Softw Eng 22(4):2095–2126

Liu K, Koyuncu A, Bissyandé TF, Kim D, Klein J, Le Traon Y (2019) You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In: 2019 12th IEEE conference on software testing, validation and verification (ICST), pp 102–113

Lovreto G, Endo AT, Nardi P, Durelli VHS (2018) 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, pp 48–56

Madeiral F, Urli S, Maia M, Monperrus M (2019) Bears: An extensible java bug benchmark for automatic program repair studies. In: 2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER). IEEE, pp 468–478

Madeiral F, Urli S, Maia M, Monperrus M (2019) Bears: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In: Proceedings of the 26th IEEE international conference on software analysis, evolution and reengineering (SANER '19). arXiv:1901.06024

Mnih V, Kavukcuoglu K, Silver D, Graves A, Antonoglou I, Wierstra D, Riedmiller M (2013) Playing atari with deep reinforcement learning. arXiv:1312.5602

Mnih V, Kavukcuoglu K, Silver D, Rusu AA, Veness J, Bellemare MG, Graves A, Riedmiller M, Fidjeland AK, Ostrovski G et al (2015) Human-level control through deep reinforcement learning. nature 518(7540):529–533

Newzoo (2020) Global games market report 2020. https://newzoo.com/insights/trend-reports/newzoo-global-games-market-report-2020-light-version

Nordin M, King D, Posthuma S (2018) But is it fun? software testing in the video game industry. http://www.es.mdh.se/icst2018/live/

Pacheco C, Ernst MD (2007) Randoop: feedback-directed random testing for java. In: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion, pp 815–816

Papadakis M, Shin D, Yoo S, Bae D (2018) Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults. In: IEEE/ACM 40th Intl Conf on Software Engineering (ICSE), pp 537–548

Pearson S, Campos J, Just R, Fraser G, Abreu R, Ernst MD, Pang D, Keller B (2017) Evaluating and improving fault localization. In: 2017 IEEE/ACM 39th international conference on software engineering (ICSE), pp 609–620

Saha RK, Lyu Y, Lam W, Yoshida H, Prasad MR (2018) Bugs. jar: a large-scale, diverse dataset of real-world java bugs. In: Proceedings of the 15th international conference on mining software repositories, pp 10–13

Shamshiri S, Just R, Rojas JM, Fraser G, McMinn P, Arcuri A (2015) Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. In: Proceedings of the 30th IEEE/ACM international conference on automated software engineering. ASE '15, pp 201–211

Wu Y, Chen Y, Xie X, Yu B, Fan C, Ma L (2020) Regression testing of massively multiplayer online role-playing games. In: 2020 IEEE international conference on software maintenance and evolution (ICSME), pp 692–696

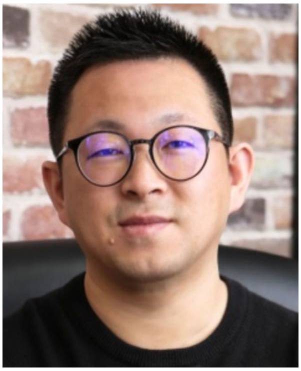
Zheng Y, Xie X, Su T, Ma L, Hao J, Meng Z, Liu Y, Shen R, Chen Y, Fan C (2019) Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning. In: 2019 34th IEEE/ACM international conference on automated software engineering (ASE), pp 772–784

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Zhuo Li received the B.S. degree from Beijing University of Posts and Telecommunications in 2015. He was a software engineer in industry from 2015 to 2019. He received the MA.Eng. degree from Kyushu University in 2022. He is currently pursing Ph.D. degree in Kyushu University. His research interests include software engineering and artificial intelligence.

Yuechen Wu received the B.S. degree in the School of Astronautics from the Harbin Institute of Technology in 2016, and received the M.S. degree in the School of Control Science and Engineering from the Shandong University in 2019. He is currently a Algorithm Engineer with Fuxi AI Lab, NetEase. His research interests include reinforcement learning, computer vision and intelligence testing.

Xiaofei Xie received his Ph.D, M.E. and B.E. from Tianjin University. He is currently an assistant professor in Singapore Management University, Singapore. His research mainly focuses on program analysis, traditional software testing and quality assurance of artificial intelligence systems. He has published some top tier conference/journal papers relevant to software analysis in ICSE, ISSTA, FSE, ASE, TDSC, TIFS, TSE, IJCAI, NeurIPS, ICML and CCS. In particular, he won two ACM SIGSOFT Distinguished Paper Awards in FSE'16 and ASE'19.



Lei Ma received the B.E. degree from Shanghai Jiao Tong University, Shanghai, China, and the M.E. and Ph.D. degrees from The University of Tokyo, Tokyo, Japan. He is currently an Associate Professor with shared appointments between (1) University of Alberta, Canada and (2) Kyushu University, Japan. He is also honorably selected as a fellow and Canada CIFAR AI Chair at Alberta Machine Intelligence Institute (Amii). His recent research centers around the interdisciplinary fields of software engineering (SE) and trustworthy artificial intelligence (AI) with a special focus on the quality and reliability assurance of machine learning and AI Systems. Many of his works were published in top-tier software engineering and AI venues (e.g., TSE, TOSEM, EMSE, ICSE, FSE, ASE, ISSTA, CAV, ICML, NeurIPS, ACM MM, AAAI, IJCAI). He is a recipient of more than 10 prestigious academic awards, including three ACM SIGSOFT Distinguished Paper Awards.

Yingfeng Chen received his Ph.D. from University of Science and Technology of China. He is currently the research director of reinforcement learning in Fuxi AI Lab, NetEase. His research interests include reinforcement learning, robotics and artificial intelligence.

Changjie Fan received his Ph.D. from University of Science and Technology of China. He is currently the director of Fuxi AI Lab, NetEase. His research interests include reinforcement learning, robotics and artificial intelligence.

Affiliations

Zhuo Li[1] · Yuechen Wu[2] · Lei Ma[1,3,4] · Xiaofei Xie[5] · Yingfeng Chen[2] · Changjie Fan[2]

✉ Yingfeng Chen
chenyingfeng1@corp.netease.com

Zhuo Li
li.zhuo.786@s.kyushu-u.ac.jp

Yuechen Wu
wuyuechen@corp.netease.com

Lei Ma
ma.lei@acm.org

Changjie Fan
fanchangjie@corp.netease.com

[1] Kyushu University, Fukuoka, Japan

[2] NetEase Fuxi AI Lab, Hangzhou, China

[3] University of Alberta, Edmonton, Canada

[4] Alberta Machine Intelligence Institute, Edmonton, Canada

[5] Singapore Management University, Singapore, Singapore