A Test Oracle for Reinforcement Learning Software based on Lyapunov Stability Control Theory

Shiyu Zhang[1], Haoyang Song[1], Qixin Wang[2], Henghua Shen, Yu Pei
Department of Computing, The Hong Kong Polytechnic University, Hong Kong SAR, China.
Email: {shiyu-comp.zhang, haoyang.song}@connect.polyu.hk
{csqwang, henghua.shen, max.yu.pei}@polyu.edu.hk

***Abstract*—*Reinforcement Learning* (RL) has gained significant attention in recent years. As RL software becomes more complex and infiltrates critical application domains, ensuring its quality and correctness becomes increasingly important. An indispensable aspect of software quality/correctness insurance is testing. However, testing RL software faces unique challenges compared to testing traditional software, due to the difficulty on defining the outputs' correctness. This leads to the RL test oracle problem. Current approaches to testing RL software often rely on human oracles, i.e. convening human experts to judge the correctness of RL software outputs. This heavily depends on the availability and quality (including the experiences, subjective states, etc.) of the human experts, and cannot be fully automated. In this paper, we propose a novel approach to design test oracles for RL software by leveraging the Lyapunov stability control theory. By incorporating Lyapunov stability concepts to guide RL training, we hypothesize that a correctly implemented RL software shall output an agent that respects Lyapunov stability control theories. Based on this heuristics, we propose a Lyapunov stability control theory based oracle, LPEA(ϑ, θ), for testing RL software. We conduct extensive experiments over representative RL algorithms and RL software bugs to evaluate our proposed oracle. The results show that our proposed oracle can outperform the human oracle in most metrics. Particularly, LPEA($\vartheta = 100\%, \theta = 75\%$) outperforms the human oracle by 53.6%, 50%, 18.4%, 34.8%, 18.4%, 127.8%, 60.5%, 38.9%, and 31.7% respectively on accuracy, precision, recall, F1 score, true positive rate, true negative rate, false positive rate, false negative rate, and ROC curve's AUC; and LPEA($\vartheta = 100\%, \theta = 50\%$) outperforms the human oracle by 48.2%, 47.4%, 10.5%, 29.1%, 10.5%, 127.8%, 60.5%, 22.2%, and 26.0% respectively on these metrics.**

***Index Terms*—reinforcement learning, test oracle**

I. Introduction

Reinforcement Learning (RL) has gained significant attention in recent years due to its wide range of applications, such as control/robotics [1] and game playing [2]. RL software outputs an agent, which learns optimal decision-making policies through interacting with an inputted *training environment* (simplified as "*environment*" in the following); the agent's goal in the interaction is to maximize cumulative rewards [3]. As RL software grows more complex and infiltrates critical application domains, ensuring its quality and correctness becomes increasingly important.

One indispensable aspect of software quality/correctness insurance is testing. The purpose of testing is to reveal the existence of bugs[1,2] in software. However, testing RL software faces unique challenges compared to testing traditional software.

To test traditional software, given the input, the correctness of an output is well defined: there is usually a unique and well-defined expected output, and the output is either equal to the unique expected output (i.e. correct), or not equal to it (i.e. incorrect). There is no gray area between the correct and the incorrect outputs. However, for RL software, given the inputted environment, often the expected output (i.e. the trained agent) is neither unique nor well-defined. For example, one major application domain of RL is to train agents (aka "*controllers*" in such contexts) for nonlinear control systems, where the nonlinearity is so complex, that the system is hard (if at all possible) to be mathematically analyzed. For such control systems, it is hard to define which controller is the unique correct controller (i.e. the unique well-defined expected output). All that we can measure is the controllers' performance for a test set; and the performance metrics are usually multi-dimensional, statistical, and dependent on the test set. It is difficult to define the optimum performance, nor to define what performance is correct/incorrect.

For example, in the control task of stabilizing an inverted pendulum [4], given a test set of initial states of the inverted pendulum, controller A on average stabilizes the inverted pendulum in 10 seconds, while controller B on average stabilizes the inverted pendulum in 15 seconds (usually a shorter stabilizing time cost is better). However, in terms of overshoot, controller A causes a worst case overshoot of 15 degrees, while controller B causes a worst case overshoot of 5 degrees (usually a smaller overshoot is better). Due to the multi-dimensionality of the performance metrics, it is difficult to

1. These authors contributed equally to this work.
2. This is the corresponding author.

[1] A more formal term for "*bug*" is "software defect." But for succinctness, we will use the term "bug" in this paper. Correspondingly, the formal terms "software contains defects" and "software contains no defect" will be referred to as "*buggy*" and "*bug-less*" respectively in the following.

[2] There are two types of bugs. Some bugs can cause compilation warnings/failures, or catch-able runtime exceptions/failures (e.g. via the Java/C++ try-catch exception mechanisms, or the C error return values). Others are hidden. The hidden bugs are more threatening. Therefore, in the following, unless otherwise denoted, we shall focus on the hidden bugs.

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00074

define which controller is better, nor to define which controller is correct/incorrect: we cannot say controller A is incorrect because it performs worse than controller B on overshoot; nor controller B is incorrect because it performs worse than controller A on stabilization time cost.

The above hard-to-define-output-correctness challenge for RL software leads to a critical problem in testing: the *test oracle problem* [5].

A *test oracle* is an indispensable tool for software testing [5]. It is a mechanism to judge the correctness of software outputs (given the corresponding input) without analyzing the software source code (in the RL literature, the term "oracle" has other meanings; however, in this paper, the term "oracle" never takes those meanings; therefore, in the following, we simplify the term "test oracle" as "oracle"). According to this definition, *the prerequisite to define an oracle is to define the correctness of software outputs*.

The RL software outputs are the agents (controllers). From the previous inverted pendulum example, we can see that to judge if an agent (controller) is correct, all that we can have is the agent's statistical performances (analytical proof of correctness is infeasible). However, as the inverted pendulum example shows, it is hard to use common statistical metrics to judge which agent is better, not to speak of which agent is correct/incorrect. That is, it is hard to define the correctness of RL software outputs. In other words, the prerequisite to define RL software oracle is yet to be fulfilled.

To deal with this problem, the traditional approach in testing RL software relies on *human oracles*, i.e. a panel of human experts are convened to manually judge the correctness of the RL software outputs (i.e. the agents). However, this approach heavily depends on the availability and quality (including experiences, subjective states, etc.) of the human experts, and cannot be fully automated.

When high quality human experts are unavailable, or when full automation is needed, we need an alternative to the human oracle. This is the focus of this paper.

Specifically, we propose an oracle based on the Lyapunov stability control theory [6] [7]. The heuristics goes as follows.

It is well-known [6] [7] in control theory that given a linear system (including the linear state dynamics, and the linear controller), where the system's physical state at time t is described by an n-dimensional vector $X(t) \in \mathbb{R}^n$, then we can construct a so-called *Lyapunov equation*. If the equation has positive-definite symmetric matrix solution $\mathbf{P} \in \mathbb{R}^{n\times n}$, then the linear system is stable. Meanwhile, given this $\mathbf{P}$, we can define a so-called *Lyapunov potential energy* (aka *Lyapunov function*) $V(X(t)) \overset{\text{def}}{=} X(t)^{\mathsf{T}}\mathbf{P}X(t)$, where $X(t)^{\mathsf{T}}$ is the transpose of $X(t)$. The Lyapunov potential energy $V(X(t))$ can only decrease over time.

We propose our RL software oracle by using the Lyapunov stability control theory inversely. We randomly generate many stable linear systems using the Lyapunov stability control theory. For each such system (including its linear state dynamics, its linear controller, and its Lyapunov potential energy $V(X(t))$), we input the following environment to the RL software under test: the state transition dynamics of the environment is the linear state dynamics. Meanwhile, we use $V(X(t))$ as the core of the environment's reward function, and punish any agent (i.e. the RL learnt controller) that ever increases $V(X(t))$ over time. Expectedly, if the RL software is bug-less, the RL learnt controller should be smooth (for example, linear), aware of the hidden Lyapunov potential energy $V(X(t))$ of the environment, and its control actions shall always decrease $V(X(t))$. Otherwise, the RL software is considered buggy.

Based on the above RL software oracle design heuristics, this paper makes the following contributions.

1) As an alternative to the traditional human oracle for testing RL software, we propose a Lyapunov stability control theory based oracle, LPEA(ϑ, θ).
2) We conduct extensive experiments over representative RL algorithms and RL software bugs to evaluate our proposed oracle. The results show that under various configurations of (ϑ, θ), our LPEA(ϑ, θ) oracles can achieve good performances in most metrics. The average accuracy, precision, recall, F1 score, true positive rate, true negative rate, false positive rate, false negative rate, and ROC curve AUC are respectively 75.2%, 94.6%, 55.7%, 68.6%, 55.7%, 94.8%, 5.2%, 44.3%, and 0.751.
3) We conduct extensive experiments over representative RL algorithms and RL software bugs to compare our proposed oracle with the human oracle. The results show that our proposed oracle can outperform the human oracle in most metrics. Particularly, LPEA$(\vartheta = 100\%, \theta = 75\%)$ outperforms the human oracle by 53.6%, 50%, 18.4%, 34.8%, 18.4%, 127.8%, 60.5%, 38.9%, and 31.7% respectively on accuracy, precision, recall, F1 score, true positive rate, true negative rate, false positive rate, false negative rate, and ROC curve's AUC; and LPEA$(\vartheta = 100\%, \theta = 50\%)$ outperforms the human oracle by 48.2%, 47.4%, 10.5%, 29.1%, 10.5%, 127.8%, 60.5%, 22.2%, and 26.0% respectively on these metrics.

The rest of the paper is organized as follows. Section II provides background on RL and Lyapunov stability control theory. Section III proposes our oracle for testing RL software. Section IV evaluates our proposal. Section V discusses related work. Section VI concludes the paper.

II. Background

A. Reinforcement Learning

Reinforcement Learning (RL) is a paradigm of machine learning, where the RL software trains an *agent* by letting it interact with an inputted environment. The key concepts related to RL include:

1) **Environment:** The external system with which the agent interacts. It includes the following concepts.
   a) **State:** We denote the state at time t of the environment as $X(t)$. The set of all possible states is called the *state space*, denoted as $\mathcal{X}$.

b) **Action:** We denote the action applied to the environment at time t as $U(t)$. The set of all possible actions is called the *action space*, denoted as $\mathcal{U}$.

c) **State Transition Dynamics:** Given the current state $X(t)$ and action $U(t)$ of the environment, the way that the state $X(t)$ will evolve is called the environment's *state transition dynamics*.

d) **Reward Function:** A function $r : \mathcal{X} \times \mathcal{U} \mapsto \mathbb{R}$ that maps the current state $X(t)$ and action $U(t)$ to a scalar real value (aka the *reward*), which measures the desirability of the current state and action.

2) **Agent:** The learning entity that interacts (issue actions to and receive states/rewards from) with the environment. The RL software takes charge of structuring and modifying the agent, aiming to maximize the cumulative reward. The final version of the agent is the output of the RL software.

B. Lyapunov Stability Control Theory

The most well-studied control theories are about *linear systems* [8], where the state of a system at time t can be modeled by an n-dimensional vector $X(t) \in \mathbb{R}^n$, and it evolves according to the so called *linear state dynamics*:

$$X(t+1) = \mathbf{A}X(t) + \mathbf{B}U(t), \tag{1}$$

where $\mathbf{A} \in \mathbb{R}^{n\times n}$ and $\mathbf{B} \in \mathbb{R}^{n\times m}$ are respectively n-by-n and n-by-m matrices; $U(t) \in \mathbb{R}^m$ is the m-dimensional action (i.e. controller feedback) applied to the system at time t; $X(t{+}1) \in \mathbb{R}^n$ is the state of the system at the next time tick, i.e. $(t+1)$; note as the focus is on testing RL software, which is discrete, we only use the discrete-time control theories in this paper.

Besides the linear state dynamics, the other key component of a linear system is the *linear controller*, which takes the following form:

$$U(t) = -\mathbf{K}X(t), \tag{2}$$

where $\mathbf{K} \in \mathbb{R}^{m\times n}$ is a m-by-n matrix.

The key to designing the linear controller is to find a proper $\mathbf{K}$ for Eq. (2). Given $\mathbf{A}$ and $\mathbf{B}$ in Eq. (1), there are many well-established ways to propose candidate values for $\mathbf{K}$, e.g. LQR [9], LMI [10] etc.

Once a candidate value for $\mathbf{K}$ is proposed, we can construct the so-called *Lyapunov equation*:

$$\tilde{\mathbf{A}}^\mathsf{T}\mathbf{P}\tilde{\mathbf{A}} - \mathbf{P} + \mathbf{Q} = 0, \tag{3}$$

where $\tilde{\mathbf{A}} \overset{\text{def}}{=} \mathbf{A} - \mathbf{B}\mathbf{K}$; $\tilde{\mathbf{A}}^\mathsf{T}$ is the transpose of $\tilde{\mathbf{A}}$; $\mathbf{Q} \in \mathbb{R}^{n\times n}$ is an arbitrarily given positive-definite symmetric matrix (e.g. we can set $\mathbf{Q}$ to be the identity matrix $\mathbf{I}$). The only unknown variable in Eq. (3) is $\mathbf{P} \in \mathbb{R}^{n\times n}$.

The well-known Lyapunov stability control theory [6] [7] says, if Eq. (3) has a positive-definite symmetric matrix solution $\mathbf{P} \in \mathbb{R}^{n\times n}$, then the original linear system of Eq. (1)(2) is *globally asymptotically stable*. That is, starting from any initial state $X(0)$, the trajectory of $X(t)$ will always converge to the origin point (denoted as $\mathbf{0}$) of the state space $\mathbb{R}^n$. In this paper, we simply refer to "globally asymptotically stable" as "*stable*."

There are well-established tools to solve the Lyapunov equation, i.e. to find a positive-definite symmetric matrix $\mathbf{P}$ that upholds Eq. (3) [10] [11]. When a solution cannot be found, there are also well-established tools to propose other candidate values for $\mathbf{K}$ (i.e. propose other linear controllers), so as to change the Lyapunov equation. But still, there are chances that for certain given $\mathbf{A}$ and $\mathbf{B}$, no proper $\mathbf{K}$ exists to construct a solvable Lyapunov equation. Fortunately, such chances are low in practice.

Once we solve the Lyapunov equation, i.e. found a positive-definite symmetric matrix $\mathbf{P} \in \mathbb{R}^{n\times n}$ that upholds Eq. (3), we can define the so-called *Lyapunov potential energy*, aka *Lyapunov function*:

$$V(X(t)) \overset{\text{def}}{=} X(t)^\mathsf{T}\mathbf{P}X(t), \tag{4}$$

where $X(t)^\mathsf{T}$ is the transpose of $X(t)$. The Lyapunov stability control theory also tells us that the Lyapunov potential energy $V(X(t))$ always decreases over time.

III. Solution

Suppose a to-be-tested RL software implementation R takes in as input an environment $\mathcal{E}$, then trains and outputs an agent a. Following the heuristics described in Section I, we propose our RL software oracle as follows.

- **Step 1:** Generate I stable[3] linear systems $\{\mathcal{S}_i\}_{i=1}^{I}$. System $\mathcal{S}_i$ ($i = 1, \ldots, I$) includes linear state dynamics of

  $$X_i(t+1) = \mathbf{A}_i X_i(t) + \mathbf{B}_i U_i(t), \tag{5}$$

  where $X_i(t) \in \mathbb{R}^n$, $U_i(t) \in \mathbb{R}^m$, $\mathbf{A}_i \in \mathbb{R}^{n\times n}$, $\mathbf{B}_i \in \mathbb{R}^{n\times m}$; and a linear controller of

  $$U_i(t) = -\mathbf{K}_i X_i(t). \tag{6}$$

  where $\mathbf{K}_i \in \mathbb{R}^{n\times m}$.
  Meanwhile, as a stable linear system, $\mathcal{S}_i$ shall also have a Lyapunov potential energy (aka Lyapunov function) defined as

  $$V_i(X_i(t)) \overset{\text{def}}{=} X_i(t)^\mathsf{T}\mathbf{P}_i X_i(t), \tag{7}$$

  where $X_i(t)^\mathsf{T}$ is the transpose of vector $X_i(t)$, and $\mathbf{P}_i \in \mathbb{R}^{n\times n}$ is an n-by-n positive-definite symmetric matrix that solves the Lyapunov equation of $\mathcal{S}_i$ (see Section II-B and Eq. (3)(4)).

- **Step 2:** For each linear system $\mathcal{S}_i$, we construct an environment $\mathcal{E}_i$ as follows. First, we adopt $X_i(t)$, $U_i(t)$, and Eq. (5) of $\mathcal{S}_i$ respectively as $\mathcal{E}_i$'s state, action, and

[3] According to Section II-B, given any arbitrarily generated $\mathbf{A}$ and $\mathbf{B}$ for Eq. (1), there are chances that a stable linear system cannot be designed. However, such chance is low in practice. Hence generating I stable linear systems is practically feasible, e.g. via repetitive trial.

state transition dynamics (see Section II-A). Second, we define the reward function r_i of $\mathcal{E}_i$ as follows.

$$\begin{aligned} & r_i(X_i(t), U_i(t)) \\ \overset{\mathsf{def}}{=}\ & \mathsf{exp}(-V_i(X_i(t))) + \alpha\,\mathsf{exp}(-||X_i(t)||_2) \\ =\ & \mathsf{exp}(-X_i(t)^{\mathsf{T}}\mathbf{P}_i X_i(t)) + \alpha\,\mathsf{exp}(-||X_i(t)||_2), \end{aligned} \quad (8)$$

where $\alpha \in [0, 1]$ is a weighting constant, exp is the exponent function, and $||X||_2$ is the 2-norm of vector X (i.e. vector X's Euler length). That is, *we reward the decrease of the Lyapunov potential energy*, and we reward converging $X_i(t)$ to the origin point $\mathbf{0}$ in $\mathbb{R}^n$.

Note we exclude $U_i(t)$ from the right hand side of Eq. (8) to feedback less info to the RL, so as to increase the learning difficulty, hoping to expose more problems of the RL software implementation.

Also note an agent (aka "controller" in this context) that matches such reward function exists, which is the linear controller described by Eq. (6). According to the Lyapunov stability control theory described in Section II-B, as the linear system $\mathcal{S}_i$ is stable, the linear controller of Eq. (6) guarantees the Lyapunov potential energy always decreases over time, and $X(t)$ converges to $\mathbf{0}$. An ideal RL software shall figure out this controller based on the reward function of Eq. (8).

- **Step 3:** Input each environment $\mathcal{E}_i$ ($i = 1, \ldots, I$) to the to-be-tested RL software implementation R, which respectively outputs the trained agent (aka "controller" in this context) a_i. Given any state $X_i(t) \in \mathbb{R}^n$, controller a_i maps $X_i(t)$ to an action of $a_i(X_i(t)) \in \mathbb{R}^m$. In other words, a_i is a function of $\mathbb{R}^n \mapsto \mathbb{R}^m$. We then define a *controller goodness* function to quantify the quality of a_i with a scalar value, and denote this function as $g_i(a_i)$. Note $g_i(a_i) \in \mathbb{R}$. Upon $\{g_i(a_i)\}_{i=1}^{I}$, we can further define an *implementation goodness* function to quantify the quality of R. Denote the implementation goodness function as $q(\{g_i(a_i)\}_{i=1}^{I})$. Note $q(\{g_i(a_i)\}_{i=1}^{I}) \in \mathbb{R}$. We can then set a threshold constant $\theta \in \mathbb{R}$, and claim R to be "buggy" if $q(\{g_i(a_i)\}_{i=1}^{I}) < \theta$, and "bug-less" otherwise.

Step 1 $\sim$ **3** is not yet an oracle, unless the controller/implementation goodness functions and threshold θ are defined. We propose to define them as follows, and name the resulted oracle the *Lyapunov Potential Energy Abnormality* (LPEA) oracle.

For each controller a_i ($i = 1, \ldots, I$) outputted in **Step 3**, we uniformly random sample J initial states in the state space $\mathcal{X}_i = \mathbb{R}^n$, and denote these initial states as $\{X_{i,j}(0)\}_{j=1}^{J}$. Correspondingly, a_i generates the respective actions $\{a_i(X_{i,j}(0))\}_{j=1}^{J}$ to be fed to the environment $\mathcal{E}_i$. Then $\mathcal{E}_i$ will respectively evolve the J initial states by 1 time tick, to generate the next states $\{X_{i,j}(1)\}_{j=1}^{J}$. The corresponding change of Lyapunov potential energy is

$$\Delta V_{i,j}(0) \overset{\mathsf{def}}{=} V_i(X_{i,j}(1)) - V_i(X_{i,j}(0)). \quad (9)$$

Correspondingly, define

$$\mathcal{V}_i^- \overset{\mathsf{def}}{=} \{j | j \in \{1, \ldots, J\} \text{ and } \Delta V_{i,j}(0) < 0\}.$$

Apparently, $0 \leqslant |\mathcal{V}_i^-| \leqslant J$, hence $\frac{|\mathcal{V}_i^-|}{J} \in [0, 1]$.

We have the following conjecture.

> *Conjecture 1:* If the RL software implementation R is bug-less, then it can figure out the underlying meaning of the reward function Eq. (8). Particularly, almost all of the outputted agent (controller) a_i ($i = 1, \ldots, I$) shall largely comply with the Lyapunov stability control theory: the Lyapunov potential energy shall decrease over time. That is, almost all $\Delta V_{i,j}(0)$s ($i = 1, \ldots, I$; $j = 1, \ldots, J$) should be negative; in other words, almost all $\frac{|\mathcal{V}_i^-|}{J}$s ($i = 1, \ldots, I$) should be close to 100%. Otherwise, R is buggy.

Based on Conjecture 1, we define the following *controller goodness function*

$$g_i(a_i) \overset{\mathsf{def}}{=} \begin{cases} 1 & (\text{if } \frac{|\mathcal{V}_i^-|}{J} \geqslant \vartheta), \\ 0 & (\text{otherwise}), \end{cases} \quad (10)$$

where $\vartheta \in [0, 1]$ is a configurable threshold constant.

Furthermore, we should set ϑ close to 100%. In this way, according to Conjecture 1, if the RL software implementation R is bug-less, then almost all $g_i(a_i)$s ($i = 1, \ldots, I$) will take the high value of "1"; while if R is buggy, then many $g_i(a_i)$s will take the low value of "0". $(*)$

Based on the controller goodness function, we define the *implementation goodness function*

$$q(\{g_i(a_i)\}_{i=1}^{I}) \overset{\mathsf{def}}{=} \frac{\sum_{i=1}^{I} g_i(a_i)}{I}. \quad (11)$$

Apparently, $q(\{g_i(a_i)\}_{i=1}^{I}) \in [0, 1]$. Hence we can configure the threshold constant θ to a value in $[0, 1]$, and claim the RL software implementation R "bug-less" if $q \geqslant \theta$, and "buggy" if $q < \theta$.

Furthermore, due to $(*)$, if R is bug-less, then we can conjecture almost all $g_i(a_i)$s ($i = 1, \ldots, I$) will take the value of "1", raising the q value close to "1"; while if R is buggy, then we can conjecture many $g_i(a_i)$s ($i = 1, \ldots, I$) will take the value of "0", dropping the q value toward "0." $(**)$

We can see different configurations of the ϑ and θ values result in different LPEA oracles, denoted respectively as LPEA(ϑ, θ). Though $(*)$ and $(**)$ give guidelines on the configuration of the (ϑ, θ) value, the optimal configuration of the (ϑ, θ) value is still an open problem. A brute force solution is to exhaustively try a reasonable set of candidate values. For example, guided by $(*)$ and $(**)$, we can try $\vartheta = 100\%$, 97.5%, 95%, 92.5%, 90%, and $\theta = 75\%$, 50%, 25%, resulting in $5 \times 3 = 15$ different combinations: LPEA$(100\%, 75\%)$, LPEA$(100\%, 50\%)$, ..., LPEA$(90\%, 25\%)$.

IV. Evaluation

A. Overall Evaluation Process, Metrics, and Research Questions

We are interested in comparing our proposed oracles with the traditional human oracle in testing RL software.

Specifically, we create benchmarks containing multiple implementations of RL software $\{R_{k,l}\}_{k=1,\ldots,K;l=0,1,\ldots,L_k}$, where $R_{k,0}$ ($k = 1, \ldots, K$) is the bug-less implementation of the kth RL software, and $R_{k,l}$ ($l = 1, \ldots, L_k$, where $L_k \in \mathbb{N}^+$) is the lth buggy implementation of the kth RL software. Define

$$\mathcal{R}' \overset{\text{def}}{=} \{R_{k,0} | k = 1, \ldots, K\}, \quad (12)$$

$$\bar{\mathcal{R}} \overset{\text{def}}{=} \{R_{k,l} | k = 1, \ldots, K \text{ and } l = 1, \ldots, L_k\}. \quad (13)$$

Note for each piece of RL software, if we adopt only one bug-less implementation but multiple buggy implementations, then $|\mathcal{R}'| << |\bar{\mathcal{R}}|$. This makes the benchmark imbalanced. To balance the benchmark, for each bug-less RL software implementation $R_{k,0}$ ($k = 1, \ldots, K$), we create $(L_k - 1)$ additional bug-less implementations: $\{R_{k,-l}\}_{l=1,\ldots,(L_k-1)}$, where $R_{k,-l}$ is a varied bug-less implementation of $R_{k,0}$ by adding random irrelevant instruction(s) into $R_{k,0}$, e.g., by adding no-op instructions randomly into $R_{k,0}$. Correspondingly, we define

$$\mathcal{R} \overset{\text{def}}{=} \mathcal{R}' \cup \{R_{k,-l} | k = 1, \ldots, K \text{ and } l = 1, \ldots, (L_k - 1)\}. \quad (14)$$

We call $\mathcal{R}'$ the *raw bug-less RL software benchmark*, while $\mathcal{R}$ the *expanded bug-less RL software benchmark* (or simply the *bug-less RL software benchmark*). We call $\bar{\mathcal{R}}$ the *buggy RL software benchmark*; and call $(\mathcal{R} \cup \bar{\mathcal{R}})$ the *RL software benchmark*.

Given oracle ω, we denote

$$\mathcal{R}^{\ominus}_{\omega} \overset{\text{def}}{=} \{R | R \in \mathcal{R} \text{ and is labeled “bug-less” by } \omega\}, (15)$$

$$\mathcal{R}^{\oplus}_{\omega} \overset{\text{def}}{=} \{R | R \in \mathcal{R} \text{ and is labeled “buggy” by } \omega\}, \quad (16)$$

$$\bar{\mathcal{R}}^{\ominus}_{\omega} \overset{\text{def}}{=} \{R | R \in \bar{\mathcal{R}} \text{ and is labeled “bug-less” by } \omega\}, (17)$$

$$\bar{\mathcal{R}}^{\oplus}_{\omega} \overset{\text{def}}{=} \{R | R \in \bar{\mathcal{R}} \text{ and is labeled “buggy” by } \omega\}. \quad (18)$$

We define the following key metrics for ω:

$$\text{True Positive Count TP}(\omega) \overset{\text{def}}{=} |\bar{\mathcal{R}}^{\oplus}_{\omega}|, \quad (19)$$

$$\text{False Positive Count FP}(\omega) \overset{\text{def}}{=} |\mathcal{R}^{\oplus}_{\omega}|, \quad (20)$$

$$\text{True Negative Count TN}(\omega) \overset{\text{def}}{=} |\mathcal{R}^{\ominus}_{\omega}|, \quad (21)$$

$$\text{False Negative Count FN}(\omega) \overset{\text{def}}{=} |\bar{\mathcal{R}}^{\ominus}_{\omega}|, \quad (22)$$

$$\text{Accuracy } \gamma_{\mathsf{AC}}(\omega) \overset{\text{def}}{=} \frac{|\mathcal{R}^{\ominus}_{\omega}| + |\bar{\mathcal{R}}^{\oplus}_{\omega}|}{|\mathcal{R}| + |\bar{\mathcal{R}}|}. \quad (23)$$

Additional metrics like precision, recall, and F1 score are standard and well-defined in the literature, so they are not explicitly redefined here to save space.

We want to study the following research question:

RQ: How good is the LPEA(ϑ, θ) oracle compared to the human oracle?

B. Bug-less RL Software Benchmark

To implement the overall evaluation process described in Section IV-A, we still need to address several problems.

The first implementation problem is which pieces of RL software shall be included in the bug-less RL software benchmark $\mathcal{R}$ (see Eq. (14)). Or more fundamentally, which pieces of RL software shall be included in the *raw* bug-less RL software benchmark $\mathcal{R}'$ (see Eq. (12)).

One choice is the *Stable Baselines3* (SB3) [12] [13] library, a reasonably sized (over 100 source code files, and over 18000 lines of source code (excluding comments and blanks)), well-known and well-maintained open-source implementation of existing RL algorithms. There are two main families of existing RL algorithms: *on-policy* and *off-policy* (see [3] for their rigorous definitions). One of the most representative on-policy RL algorithms is *Advantage Actor-Critic* (A2C) [14]; while one of the most representative off-policy RL algorithms is *Twin Delayed Deep Deterministic policy gradient* (TD3) [15], a descendant of the well-known Q-Learning algorithm [16]. We therefore include the state-of-the-art SB3 stable implementations of A2C and TD3 into our raw bug-less RL software benchmark $\mathcal{R}'$. Note this assumes the state-of-the-art SB3 stable implementations are trustworthy enough to be regarded as the *ground-truth* bug-less implementations. We have to adopt this best-effort practice because perfectly bug-less implementations are empirically unavailable.

In addition to A2C and TD3, OpenAI also strongly recommends the *Proximal Policy Optimization* (PPO) RL algorithm [17] [18]. Therefore, we also include the state-of-the-art SB3 stable implementation of PPO into $\mathcal{R}'$.

In summary, the bug-less RL software included in $\mathcal{R}'$ are the SB3 stable implementation (specifically, the GitHub commit hash is `4efee92`, see [12] [13]) of A2C, PPO, and TD3. Note both A2C and PPO are on-policy RL algorithms, and TD3 is an off-policy algorithm. Also note A2C, PPO, and TD3 are general purpose RL algorithms for both continuous and discrete state/action environments.

We then carry out harmless variations (see the paragraph before Eq. (14) in Section IV-A) upon the state-of-the-art SB3 stable implementations of A2C, PPO, and TD3, to expand $\mathcal{R}'$ to $\mathcal{R}$, i.e. creating the bug-less RL software benchmark $\mathcal{R}$. Specifically, $\mathcal{R}$ includes respectively 19, 22, and 15 bug-less implementations of A2C, PPO, and TD3. These numbers respectively match the numbers of buggy implementations of A2C, PPO, and TD3 included in $\bar{\mathcal{R}}$, the buggy RL software benchmark (see Section IV-C paragraph $(***)$). In this way, the bug-less and buggy benchmarks, $\mathcal{R}$ and $\bar{\mathcal{R}}$, are balanced.

C. Buggy RL Software Benchmark

The second implementation problem is how to create the *buggy* RL software benchmark $\bar{\mathcal{R}}$ (see Eq. (13)).

We create $\bar{\mathcal{R}}$ by injecting bugs into the bug-less RL software implementations included in $\mathcal{R}'$. Specifically, we inject two kinds of bugs: *artificial* and *real-world*.

We create the artificial bugs based on state-of-the-art surveys of RL software bugs. Specifically, as per the survey of

Nikanjam et al. [19], Table I summarizes the types of RL software bugs[4].

TABLE I: RL Software Bug Types from Survey [19]

Level-1 Type	Level-2 Type
1. RL Specific	1.1. Exploring the environment bugs
	1.2. Updating network bugs
2. NN Specific	2.1. Model bugs
	2.2. Training bugs
	2.3. Tensor and inputs bugs

Based on Table I, we injected artificial bugs into the bug-less A2C, PPO, and TD3 implementations of $\mathcal{R}'$, to create various buggy implementations, collectively referred to as $\bar{\mathcal{R}}_{\mathsf{art}}$. Respectively, there are 17, 21, and 15 buggy implementations of A2C, PPO, and TD3 in $\bar{\mathcal{R}}_{\mathsf{art}}$. Table II summarizes the distributions of various types of bugs in these implementations. Note the distribution is not even. This is because whether we can inject a specific type of bug into a bug-less implementation depends on the semantics of the bug and the implementation. Some semantics are easy for injection, some are difficult, even impossible.

TABLE II: Number of Buggy Implementations in $\bar{\mathcal{R}}_{\mathsf{art}}$

Artificial Bug's Type*	A2C	PPO	TD3
1.1	0	0	1
1.2	13	16	8
2.1	1	1	1
2.2	2	2	4
2.3	1	2	1

* See Table I column 2 for the meanings of the bug type IDs.

TABLE III: Real-World Bugs

Bug ID	Description
RW1	SB3 GitHub commit hash: `d5986da` File: `stable_baselines3/common/on_policy_algorithm.py` Line 167: "`dones`" should be "`_last_dones`"; Line 178: should insert one more line "`self._last_dones = dones`" before this line.
RW2	SB GitHub commit hash: `689afd1` File: `stable_baselines/a2c/a2c.py` Line 325: "`np.int32`" should be "`self.env.action_space.dtype`".

Meanwhile, we collect real-world bugs by manually analyzing the GitHub commit history of SB3 [12] [13], including SB3's ancestor repository: *Stable Baselines* (SB) [20]. Many real-world bugs recorded in the commit history are *non-hidden bugs* (i.e. they show themselves via compilation warnings/failures and/or runtime exceptions/failures), which can be detected without oracle, hence are not the focus of this paper. Our manual analysis finally collected 2 real-world hidden bugs, listed in Table III. Bug RW1 resides in a parent class used by both the A2C and PPO implementations. By re-injecting bug RW1 into the bug-less A2C and PPO implementations respectively, we create two buggy implementations, respectively for A2C and PPO. Bug RW2 resides in a class used by the A2C implementation. By re-injecting bug RW2 into the bug-less A2C implementation, we create another buggy implementation of A2C. We refer to the three created buggy implementations collectively as $\bar{\mathcal{R}}_{\mathsf{rw}}$, and thus $\bar{\mathcal{R}} = \bar{\mathcal{R}}_{\mathsf{art}} \cup \bar{\mathcal{R}}_{\mathsf{rw}}$.

Based on the above narrations, we can see $\bar{\mathcal{R}}_{\mathsf{art}}$ includes respectively 17, 21, and 15 buggy (using artificial bugs) implementations of A2C, PPO, and TD3; while $\bar{\mathcal{R}}_{\mathsf{rw}}$ includes respectively 2, 1, and 0 buggy (using real-world bugs) implementations of A2C, PPO, and TD3. Therefore, $\bar{\mathcal{R}}$ includes respectively $(17+2) = 19$, $(21+1) = 22$, and $(15+0) = 15$ buggy implementations of A2C, PPO, and TD3. $(***)$

D. Human Oracle and Comparison Configurations

The third implementation problem is the design of human oracle, the traditional oracle for testing RL software.

However, so far, there is no standard specification on the design of human oracle for testing RL software. To our best knowledge, the most relevant specification is given by Zolfagharian et al. through their paper on STARLA [21]. The focus of STARLA is on how to use genetic algorithms to create test cases for RL software. Nevertheless, it also mentions how to judge if an agent (the output of RL software) is abnormal: by checking if the agent drives the test environment into a "*faulty state*;" and the definitions of "faulty states" are given by human experts.

However, the STARLA specification still lacks details (after all, STARLA's focus is not on oracle). In the following, we expand the STARLA specification into a more detailed human oracle specification, assuming the to-be-tested RL software implementation is R.

- **Step 1:** We choose I classic environments $\{\mathcal{E}_i\}_{i=1}^{I}$.
- **Step 2:** Repeat **Step 2.1** H ($H \in \mathbb{N}^{+}$) times.

  Step 2.1: In the hth ($h \in \{1, \ldots, H\}$) iteration, input each environment $\mathcal{E}_i$ ($i = 1, \ldots, I$) to R, which outputs the corresponding trained agent $a_{i,h}$ ($i = 1, \ldots, I$). Let $a_{i,h}$ interact with $\mathcal{E}_i$, and record the *interaction trace*, which is a time series of the states of $\mathcal{E}_i$, denoted as $\{\mathsf{state}_{i,h,t}\}_t$.
- **Step 3:** Distribute the recorded IH interaction traces to human experts.
- **Step 4:** For each human expert, and for each interaction trace she/he got, she/he *subjectively*[5] decides a set of "*faulty states*." The interaction trace is labeled "*abnormal*" if the trace ever reaches a "faulty state;" and is labeled "*normal*" otherwise.
- **Step 5:** R is labeled "*buggy*" if there exists an "abnormal" interaction trace; and is labeled "*bug-less*" otherwise.

In practice, we choose the classic *Mountain Car Continuous* (MCC) and the *Pendulum* environments from the famous

[4] As mentioned in footnote 2, this paper focuses on hidden bugs, hence the non-hidden bugs (those show themselves via compilation warnings/failures and/or runtime exceptions/failures) listed in [19] are not listed in Table I.

[5] The human experts have the freedom to check all the common statistical metrics on the provided traces. In this sense, the common statistical metrics are already included in the human oracle design.

Gymnasium repository [22] [23], respectively as our $\mathcal{E}_1$ and $\mathcal{E}_2$ (hence $I = 2$). We choose $H = 3$.

We use the same bug-less RL software benchmark $\mathcal{R}$ and buggy RL software benchmark $\bar{\mathcal{R}}$ respectively described in Section IV-B and IV-C. As mentioned in Section IV-B, $\mathcal{R}$ includes respectively 19, 22, and 15 bug-less implementations of A2C, PPO, and TD3 RL software. As mentioned in Section IV-C, $\bar{\mathcal{R}}$ includes respectively 19, 22, and 15 buggy implementations of A2C, PPO, and TD3 RL software. Hence in total, we have $|\mathcal{R}|+|\bar{\mathcal{R}}| = (19+22+15)+(19+22+15) = 112$ different RL software implementations. Together these implementations create $(|\mathcal{R}| + |\bar{\mathcal{R}}|)IH = 112 \times 2 \times 3 = 672$ interaction traces.

We randomly distribute the 672 interaction traces to a panel of three human experts. Two of the human experts have bachelor's degrees in computer science, and the other has PhD degree in computer science and robotics related engineering. The three panel members respectively reported spending at least 1.5, 2, and 2.5 hours on labeling these traces. This totals at least 6 human-hour of human resources allocated to run the human oracle.

In contrast, the human resources needed to run our proposed LPEA(ϑ, θ) oracle is always nearly 0 human-hour (no matter what ϑ and θ values we choose): the human operator only needs to enter a command line to start the automated workflow. Our evaluations hence are biased in favour of the human oracle in the sense that way more human resources are allocated to it than to the LPEA(ϑ, θ) oracle.

In terms of computing resources. The human oracle costs 24 hours of computing, mainly to generate the $(|\mathcal{R}| + |\bar{\mathcal{R}}|)IH = 672$ interaction traces. The computing platform is a laptop with AMD Ryzen 7 5800H CPU and NVIDIA GeForce RTX 3060 Laptop GPU.

For an LPEA(ϑ, θ) oracle (see Section III), we choose the following learning hyper-parameters: $I = 20$; $J = 800$; $n = 2$; $m = 2$; learning rate for A2C, PPO, and TD3 are respectively 0.0004, 0.0012, and 0.001; learning steps for A2C, PPO, and TD3 are respectively 90000, 120000, and 90000. The resulted computing cost is always within 129 hours (no matter what ϑ and θ values we choose), using the same computing platform as that for the human oracle.

As the computing platform costs USD1600 to purchase, and can be used for at least 3 years. This corresponds to an hourly price of 0.061USD/hour. In contrast, the human resource cost is about 10USD/hour. Therefore, the total monetary cost of the human oracle is $10 \times 6 + 0.061 \times 24 = 61.464$(USD); while the total monetary cost of an LPEA oracle is $10 \times 0 + 0.061 \times 129 = 7.869$(USD). Therefore, in terms of total monetary cost, our evaluations are still biased in favour of the human oracle (i.e. more money is allocated to the human oracle).

E. Evaluation Results

Table IV lists all the raw data on the true positive count (TP), false positive count (FP), true negative count (TN), and false negative count (FN) (see Eq. (19)-(22)) over all oracles, including 15 LPEA oracles of different (ϑ, θ) configurations and the human oracle; while Fig. 1 and Fig. 2 plot the derived performance metrics.

Fig 1 (a) visualizes the accuracy (see Eq. (23)), precision, recall, and F1 score metrics using the overall data from Table IV. In the figure, the X-axis lists the names of the oracle, while the Y-axis lists the metrics' values.

In terms of accuracy, the LPEA oracles show strong performance across different (ϑ, θ) configurations. The highest accuracy of 0.804 is achieved by LPEA$(\vartheta = 97.5\%, \theta = 75\%)$ and LPEA$(\vartheta = 95\%, \theta = 75\%)$. Even the lowest accuracy (0.679 by LPEA$(\vartheta = 90\%, \theta = 25\%)$) is higher than the human oracle's accuracy of 0.5.

In terms of precision, many LPEA oracles achieve the perfect score of 1.0. These include LPEA$(\vartheta = 97.5\%, \theta = 50\%)$, LPEA$(\vartheta = 97.5\%, \theta = 25\%)$, and all other LPEA oracles with a ϑ of 95%, 92.5%, and 90%. Even the lowest precision (0.737 by LPEA$(\vartheta = 100\%, \theta = 50\%)$) is higher than the human oracle's precision of 0.5.

In terms of recall, the human oracle, with a recall of 0.679, beats many of the LPEA oracles. But still, LPEA$(\vartheta = 100\%, \theta = 75\%)$ and LPEA($(\vartheta = 100\%, \theta = 50\%)$ beat the human oracle respectively with a recall of 0.804 and 0.750. The high recall achieved by the human oracle is mainly due to the overly pessimistic (on whether an implementation is buggy) subjective views of the human experts. From Table IV-Overall-TP and FP columns, we can see that the human experts labeled $(38 + 38) = 76$ implementations out of the 112 implementations as "positive" (i.e. "buggy"), though only half of the implementations (i.e. $112/2 = 56$) are actually buggy. These overly pessimistic subjective views lead to picking out most buggy implementations at the cost of mistakenly labeling many bug-less implementations as "buggy." That is, at the cost of other metrics, such as accuracy, precision, and F1 score.

In terms of F1 score, a metric that comprehensively combines the accuracy, precision, and recall metrics, nearly all the LPEA oracles (with F1 scores ranging from 0.600 to 0.776) outperform the human oracle (with an F1 score of 0.576), except LPEA$(\vartheta = 92.5\%, \theta = 25\%)$ and LPEA$(\vartheta = 90\%, \theta = 25\%)$ (respectively with a F1 score of 0.545 and 0.526). Particularly, LPEA$(\vartheta = 100\%, \theta = 75\%)$ achieves the highest F1 score, 0.776. Due to the comprehensiveness of F1 score, this evidences that LPEA$(\vartheta = 100\%, \theta = 75\%)$ has the overall best performance compared to other LPEA configurations and the human oracle.

Fig 1 (b) visualizes the false positive rate and false negative rate metrics using the overall data from Table IV. In the figure, the X-axis lists the names of the oracle, while the Y-axis lists the metrics' values. For false positive rates and false negative rates, the lower the values, the better.

In terms of false positive rates, many LPEA oracles achieve the perfect score of 0.0. These include LPEA$(\vartheta = 97.5\%, \theta = 50\%)$, LPEA$(\vartheta = 97.5\%, \theta = 25\%)$, and all other LPEA oracles with a ϑ of 95%, 92.5%, and 90%. Even the highest false positive rate (0.268 by LPEA$(\vartheta = 100\%, \theta = 75\%)$ and LPEA$(\vartheta = 100\%, \theta = 50\%)$) is much lower than the human oracle's false positive rate of 0.679.

TABLE IV: Raw Data

Oracle	A2C				PPO				TD3				Overall (A2C, PPO, and TD3)			
	TP	FP	TN	FN	TP	FP	TN	FN	TP	FP	TN	FN	TP	FP	TN	FN
LPEA($\vartheta = 100\%, \theta = 75\%$)	15	0	19	4	15	0	22	7	15	15	0	0	45	15	41	11
LPEA($\vartheta = 100\%, \theta = 50\%$)	15	0	19	4	12	0	22	10	15	15	0	0	42	15	41	14
LPEA($\vartheta = 100\%, \theta = 25\%$)	13	0	19	6	9	0	22	13	14	12	3	1	36	12	44	20
LPEA($\vartheta = 97.5\%, \theta = 75\%$)	15	0	19	4	9	0	22	13	12	2	13	3	36	2	54	20
LPEA($\vartheta = 97.5\%, \theta = 50\%$)	15	0	19	4	8	0	22	14	10	0	15	5	33	0	56	23
LPEA($\vartheta = 97.5\%, \theta = 25\%$)	13	0	19	6	6	0	22	16	10	0	15	5	29	0	56	27
LPEA($\vartheta = 95\%, \theta = 75\%$)	15	0	19	4	8	0	22	14	11	0	15	4	34	0	56	22
LPEA($\vartheta = 95\%, \theta = 50\%$)	13	0	19	6	7	0	22	15	10	0	15	5	30	0	56	26
LPEA($\vartheta = 95\%, \theta = 25\%$)	11	0	19	8	5	0	22	17	8	0	15	7	24	0	56	32
LPEA($\vartheta = 92.5\%, \theta = 75\%$)	13	0	19	6	8	0	22	14	11	0	15	4	32	0	56	24
LPEA($\vartheta = 92.5\%, \theta = 50\%$)	13	0	19	6	6	0	22	16	10	0	15	5	29	0	56	27
LPEA($\vartheta = 92.5\%, \theta = 25\%$)	9	0	19	10	5	0	22	17	7	0	15	8	21	0	56	35
LPEA($\vartheta = 90\%, \theta = 75\%$)	13	0	19	6	8	0	22	14	10	0	15	5	31	0	56	25
LPEA($\vartheta = 90\%, \theta = 50\%$)	13	0	19	6	5	0	22	17	8	0	15	7	26	0	56	30
LPEA($\vartheta = 90\%, \theta = 25\%$)	8	0	19	11	5	0	22	17	7	0	15	8	20	0	56	36
Human Oracle	12	15	4	7	15	15	7	7	11	8	7	4	38	38	18	18

In terms of false negative rates, the human oracle, with a false negative rate of 0.321 beats many of the LPEA oracles. But still, LPEA($\vartheta = 100\%, \theta = 75\%$) and LPEA($\vartheta = 100\%, \theta = 50\%$) beat the human oracle respectively with a false negative rate of 0.196 and 0.25. The reason for human oracle's good performance on false negative rate is similar to that on recall.

Fig 1 (c) visualizes the true positive rate and true negative rate metrics using the overall data from Table IV. In the figure, the X-axis lists the names of the oracle, while the Y-axis lists the metrics' values. For true positive rates and true negative rates, the higher the values, the better.

In terms of true positive rates, the human oracle, with a true positive rate of 0.679, beats many of the LPEA oracles. But still, LPEA($\vartheta = 100\%, \theta = 75\%$) and LPEA($\vartheta = 100\%, \theta = 50\%$) beat the human oracle respectively with a true positive rate of 0.804 and 0.750. The reason for human oracle's good performance on true positive rate is similar to that on false negative rate (in fact, true positive rate and false negative rate always sum up to 1).

In terms of true negative rates, many LPEA oracles achieve the perfect score of 1.0. These include LPEA($\vartheta = 97.5\%, \theta = 50\%$), LPEA($\vartheta = 97.5\%, \theta = 25\%$), and all other LPEA oracles with a ϑ of 95%, 92.5%, and 90%. Even the lowest true negative rate (0.732 by LPEA($\vartheta = 100\%, \theta = 75\%$) and LPEA($\vartheta = 100\%, \theta = 50\%$)) is much higher than the human oracle's true negative rate of 0.321.

In addition to the metrics visualized by Fig. 1, Fig. 2 visualizes the ROC curves [24] using the raw data from Table IV. In the figure, the X-axis is the false positive rate and the Y-axis is the true positive rate. Similar to the F1 score, ROC curve provides another way to comprehensively reflect a classification tool's performance. An ideal ROC curve should be bent toward the top-left corner of the plot as much as possible, reaching high true positive rates at the cost of low false positive rates. Conversely, an ROC curve close to the diagonal line (aka the *random baseline*) represents a performance similar to random guessing, which should be avoided. Based on this intuition, a quantitative metric on ROC curve's quality is the *Area Under Curve* (AUC), and the bigger the AUC the better.

As per Fig. 2, in terms of the AUC, nearly all LPEA oracles are better than the human oracle, no matter based on the A2C, PPO, TD3, or overall data.

With all the data and metrics presented above, in summary and to answer the research question of **RQ** in Section IV-A, our LPEA(ϑ, θ) oracles (where ϑ = 100%, 97.5%, 95%, 92.5%, 90%, and θ = 75%, 50%, 25%) outperform the human oracle in majority of the metrics. The average accuracy, precision, recall, F1 score, true positive rate, true negative rate, false positive rate, false negative rate, and ROC curve AUC are respectively 75.2%, 94.6%, 55.7%, 68.6%, 55.7%, 94.8%, 5.2%, 44.3%, 0.751. Particularly, LPEA($\vartheta = 100\%, \theta = 75\%$) outperforms the human oracle by 53.6%, 50%, 18.4%, 34.8%, 18.4%, 127.8%, 60.5%, 38.9%, and 31.7% respectively on accuracy, precision, recall, F1 score, true positive rate, true negative rate, false positive rate, false negative rate, and ROC curve AUC; and LPEA($\vartheta = 100\%, \theta = 50\%$) outperforms the human oracle by 48.2%, 47.4%, 10.5%, 29.1%, 10.5%, 127.8%, 60.5%, 22.2%, and 26.0% respectively on these metrics.

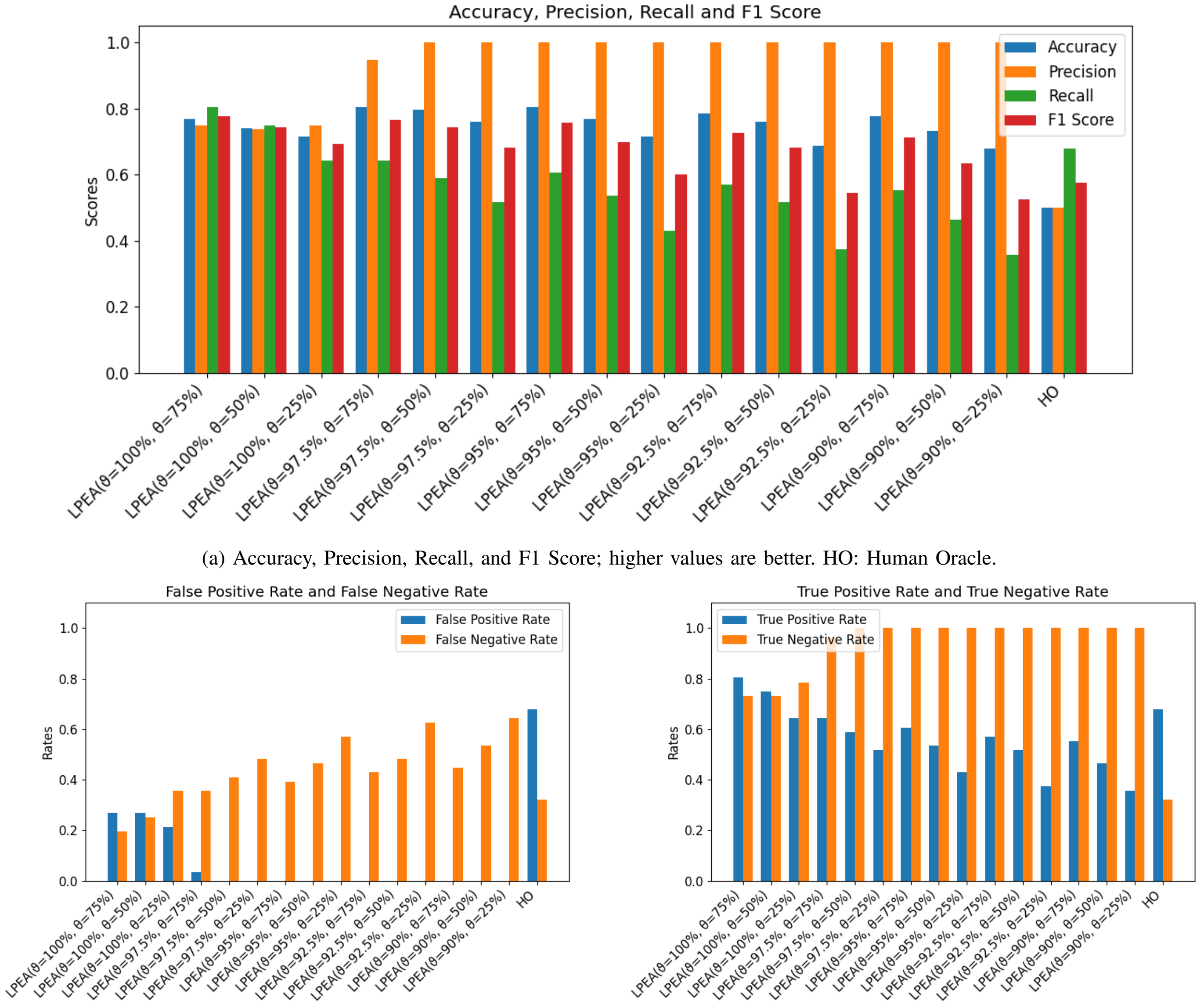
F. Threats to Validity

1. Construct Validity Threats

1.1 Human Oracle Bias

The human experts serving as the human oracles can be biased due to their knowledge and skill differences. To make our evaluations more convincing, two of our three human experts have bachelor's degrees in computer science, and the other has PhD degree in computer science and robotics related engineering. This ensures they have reasonable knowledge and skill on computer science and are reasonably qualified to judge the behaviors of a piece of software. Though the qualities of the human experts can always be better, the human experts we recruit for this paper are reasonable compared to common practices.

The human experts' qualities are also validated by Fig 2. According to the figure, the AUC (Area Under the Curve, the



(a) Accuracy, Precision, Recall, and F1 Score; higher values are better. HO: Human Oracle.

(b) False Positive Rate and False Negative Rate; lower values are better. HO: Human Oracle.

(c) True Positive Rate and True Negative Rate; higher values are better. HO: Human Oracle

Fig. 1: Performance of the LPEA Oracles and the *Human Oracle* (HO)

bigger the better) of the human oracle (HO) ROC curves are significantly greater than that of the "random guessing" ROC curves (i.e. the diagonal dashed lines).

1.2 Ground Truth Validity

It is well-known that for any reasonably sized software (SB3 contains over 100 source code files and over 18000 lines of source code (excluding comments and blanks)), due to the explosion of combinatorial complexity, completely removing all bugs is empirically impossible. Therefore, the "bug-less" RL software implementations used in our evaluations can only be "bug-less" in the sense of best-effort. Specifically, we assume the state-of-the-art stable implementations of the well-known and well-maintained open-source SB3 library as trustworthy enough to be regarded as "bug-less."

1.3 Other Oracle Design Heuristics

There can be other heuristics to construct the oracle for testing RL software. We will explore them in our future work. That said, one reason that the LPEA oracle attracts our attention is that the reward function requests a continuous and differentiable linear controller, hence the resulted changes of environment states form a continuous and differentiable vector field over the vector space of $\mathbb{R}^n$. A hidden bug usually triggers discrete fractures in such continuous and differentiable vector field, hence is easy to spot.

2. Internal Validity Threats:

2.1 Evaluation Platform Implementation Correctness

There can be bugs in the implementation of our oracles. To alleviate this threat, we conducted code reviews on the oracle

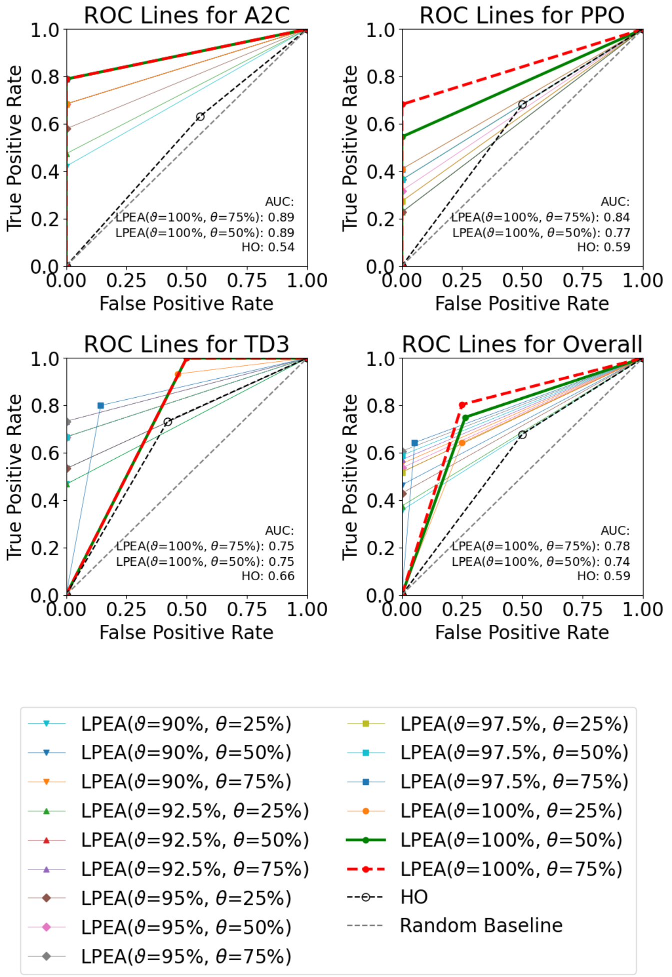


Fig. 2: ROC Curves of the LPEA Oracles and the *Human Oracle* (HO)

implementation before the evaluations.

2.2 Bug Semantics and Injection Locations

The semantics and injection locations of the bugs are constrained by the bug types and the semantics of the bug-less implementations. On the other hand, due to combinatorial explosion of the semantics, we cannot try every possible bug semantics and injection location; but we are trying our best to randomize these bug semantics and injection locations.

3. External Validity Threats:

3.1 Human Oracle Representativeness

To our best knowledge, Zolfagharian et al.'s STARLA [21] paper provides the most rigorous definition in the literature on human oracle for testing RL software. The use of the Gymnasium repository [22] [23] is also a widely-adopted practice to evaluate the RL software implementations.

3.2 RL Software Implementations' Representativeness

We can always expand our benchmark to include more RL software implementations. However, we claim the SB3 implementations of A2C, PPO, and TD3 are reasonably representative, because 1) they are representative algorithms covering the two main categories of RL algorithms: on-policy and off-policy [3]; 2) PPO is strongly recommended by OpenAI [17] [18]; and 3) SB3 is a reasonably sized, well-known and well-maintained open-source library of RL software implementations [12] [13].

V. Related Work

Metamorphic testing [25] has been explored to design oracles for testing machine learning software. Its main idea is to exploit domain-specific knowledge on the relations between the software inputs and outputs, and use such relations as oracles to detect erroneous software outputs. In this sense, the LPEA oracle that we propose in this paper is a metamorphic testing oracle. It exploits the domain-specific knowledge on the decrease of Lyapunov potential energy.

Xie et al. [26] propose a metamorphic testing oracle for testing supervised machine learning software; and the follow up work of [27] proposes a metamorphic testing oracle for testing unsupervised machine learning software for clustering. However, this paper focuses on testing reinforcement learning (RL) software, and RL is neither supervised learning nor unsupervised learning [3].

Besides metamorphic testing oracle, Pang et al. [28] and Tappler et al. [29] study how to test RL software for Markov decision processes. Both papers regard the controlled plant reaching an obvious faulty state (e.g. the plant is crashed) as the indicator (i.e. oracle) of buggy software. This concurs with our human oracle design (see Section IV-D **Step 4**).

Padgham et al. [30] propose a model-based oracle generation method for automated unit testing of agents. This method needs white-box access to the design and implementation of the agents. Nikanjam et al. [19] propose a taxonomy of faults in RL software and propose DRLinter, a model-based fault detection framework for RL software. DRLinter carries out static analysis and graph transformations upon the RL software, hence also needs white-box access to the RL software. Varshosaz et al. [31] propose ways to formally specify temporal-difference based RL algorithms, and check the behavioral properties on the specific steps of the algorithms as oracles. Obviously, this also needs white-box access to the RL software. In contrast, our proposed oracle only needs black-box access to the RL software and the agent. As future work, we can also compare with these white-box solutions. But in case the RL software or the agent source codes are unavailable/proprietary, we can then only use the black-box solutions.

The proposed LEPA oracle can be integrated into the frameworks of property-based testing, such as QuickCheck [32]. A property-based testing framework needs an oracle that exploits certain properties of the software output for testing. Our proposed LPEA oracle fits this need by exploiting the Lyapunov potential energy decreasing properties of the RL learned (outputted) agents.

Jothimurugan et al. [33] propose a formal language to specify RL problems and solutions. This work is also orthogonal to our LPEA oracle, which can be specified using this formal language in the future.

Shen et al. [34] and Hu et al. [35] propose to use mutation analysis [36] to evaluate the quality of a test set for machine learning software. Evaluating the quality of the test set (i.e. inputs to the learned model) is orthogonal to the oracle design, which focuses on labeling the outputs' correctness of the learned model. In addition, mutation analysis requires white-box access to the learned model; while our proposed oracle requires only black-box access of the learned model (agent).

Wan et al. [37] propose to use coverage-guided fuzzing to create better test sets for RL software. Again, the focus is on improving the quality of the test set, i.e. inputs to the learned model. While oracle focuses on the outputs of the learned model (agent). Hence the two problems are orthogonal to each other.

Finally, when we discuss the fairness of the evaluations, we compared the monetary costs of the proposed LPEA oracle and the human oracle (see Section IV-D). Another cost metric of interest is the environmental cost. Recent work by Lacoste et al. [38] quantified the carbon emissions of machine learning systems. Following this work, we can further measure the environmental cost of our proposed LPEA oracle. However, how to measure the environmental cost of human oracle is still an open problem.

VI. Conclusion

In this paper, we address the RL software oracle problem by exploiting the Lyapunov stability control theory, and propose the LPEA(ϑ, θ) oracles. Our evaluations over representative RL algorithms and RL software bugs show that our LPEA(ϑ, θ) oracles (where $\vartheta = 100\%$, 97.5%, 95%, 92.5%, 90%, and $\theta = 75\%$, 50%, 25%) outperform the human oracle in most of the metrics. Particularly, LPEA$(\vartheta = 100\%, \theta = 75\%)$ outperforms the human oracle by 53.6%, 50%, 18.4%, 34.8%, 18.4%, 127.8%, 60.5%, 38.9%, and 31.7% respectively on accuracy, precision, recall, F1 score, true positive rate, true negative rate, false positive rate, false negative rate, and ROC curve AUC; and LPEA$(\vartheta = 100\%, \theta = 50\%)$ outperforms the human oracle by 48.2%, 47.4%, 10.5%, 29.1%, 10.5%, 127.8%, 60.5%, 22.2%, and 26.0% respectively on these metrics.

Acknowledgment

The research project related to this paper is supported in part by the Hong Kong RGC Theme-based Research Scheme T22-505/19-N (P0031331, RBCR, P0031259, RBCP); in part by RGC GRF 15207324 (P0051926, B-QCFM); in part by RGC Germany/HK Joint Research Scheme G-PolyU503/16; in part by HKSAR Government and HKJCCT P0041424 (ZB5A); and in part by the Hong Kong Polytechnic Univ. P0042701 (CE09), P0046487 (CE0F), P0047916 (TACW), P0042699 (CE55), P0045578 (CE1C), P0043884 (CD6R), P0047965 (TAEB), P0047964 (TAEA), P0033695 (ZVRD), P0013879 (BBWH), P0036469 (CDA8), P0043634 (1-TAB2), P0043647 (1-TABF), P0042721 (1-ZVG0), P0051205, P0051074, J2024A005, LTG22-25/IICA/33 (TDG 2022-25), and TDG22-25/SMS-11. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of sponsors.

The authors thank anonymous reviewers for their efforts/advice on improving this paper.

References

[1] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274, 2013.

[2] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the game of go without human knowledge," *nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[3] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[4] N. Muskinja and B. Tovornik, "Swinging up and stabilization of a real inverted pendulum," *IEEE transactions on industrial electronics*, vol. 53, no. 2, pp. 631–639, 2006.

[5] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE transactions on software engineering*, vol. 41, no. 5, pp. 507–525, 2014.

[6] S. Sastry, "Lyapunov stability theory," *Nonlinear systems: analysis, stability, and control*, pp. 182–234, 1999.

[7] Z. Gajic and M. T. J. Qureshi, *Lyapunov matrix equation in system stability and control*. Courier Corporation, 2008.

[8] W. L. Brogan, *Modern control theory*. Pearson education india, 1985.

[9] B. D. Anderson and J. B. Moore, *Optimal control: linear quadratic methods*. Courier Corporation, 2007.

[10] S. Boyd, L. El Ghaoui, E. Feron, and V. Balakrishnan, *Linear matrix inequalities in system and control theory*. SIAM, 1994.

[11] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright *et al.*, "Scipy 1.0: fundamental algorithms for scientific computing in python," *Nature methods*, vol. 17, no. 3, pp. 261–272, 2020.

[12] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021.

[13] "Github stable baselines3 repository," (Date last accessed 2-Aug-2024). [Online]. Available: https://github.com/DLR-RM/stable-baselines3

[14] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*. PMLR, 2016, pp. 1928–1937.

[15] S. Fujimoto, H. Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," in *International conference on machine learning*. PMLR, 2018, pp. 1587–1596.

[16] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, pp. 279–292, 1992.

[17] O. Latypov. (2023) A comprehensive guide to proximal policy optimization (ppo) in ai. [Online]. Available: https://medium.com/@oleglatypov/a-comprehensive-guide-to-proximal-policy-optimization-ppo-in-ai-82edab5db200

[18] OpenAI. (2017) Proximal policy optimization. [Online]. Available: https://openai.com/index/openai-baselines-ppo/

[19] A. Nikanjam, M. M. Morovati, F. Khomh, and H. Ben Braiek, "Faults in deep reinforcement learning programs: a taxonomy and a detection approach," *Automated software engineering*, vol. 29, no. 1, p. 8, 2022.

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

[21] A. Zolfagharian, M. Abdellatif, L. C. Briand, M. Bagherzadeh, and S. Ramesh, "A search-based testing approach for deep reinforcement learning agents," *IEEE Transactions on Software Engineering*, vol. 49, no. 7, pp. 3715–3735, Jul. 2023.

[22] M. Towers, J. K. Terry, A. Kwiatkowski, J. U. Balis, G. d. Cola, T. Deleu, M. Goulão, A. Kallinteris, A. KG, M. Krimmel, R. Perez-Vicente, A. Pierré, S. Schulhoff, J. J. Tai, A. T. J. Shen, and O. G. Younis, "Gymnasium," Mar. 2023. [Online]. Available: https://zenodo.org/record/8127025

[23] F. Foundation, "Gymnasium: A standard api for reinforcement learning and a diverse set of reference environments," https://gymnasium.farama.org/, 2023, accessed: 2024-08-02.

[24] J. N. Mandrekar, "Receiver operating characteristic curve in diagnostic test assessment," *Journal of Thoracic Oncology*, vol. 5, no. 9, pp. 1315–1316, 2010.

[25] T. Y. Chen, S. C. Cheung, and S. M. Yiu, "Metamorphic testing: a new approach for generating next test cases. department of computer science, hong kong university of science and technology," Tech. Rep. HKUST-CS98-01, Tech. Rep., 1998.

[26] X. Xie, J. W. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen, "Testing and validating machine learning classifiers by metamorphic testing," *Journal of Systems and Software*, vol. 84, no. 4, pp. 544–558, 2011.

[27] X. Xie, Z. Zhang, T. Y. Chen, Y. Liu, P.-L. Poon, and B. Xu, "Mettle: A metamorphic testing approach to assessing and validating unsupervised machine learning systems," *IEEE Transactions on Reliability*, vol. 69, no. 4, pp. 1293–1322, 2020.

[28] Q. Pang, Y. Yuan, and S. Wang, "Mdpfuzz: testing models solving markov decision processes," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 378–390.

[29] M. Tappler, F. C. Córdoba, B. K. Aichernig, and B. Könighofer, "Search-based testing of reinforcement learning," *arXiv preprint arXiv:2205.04887*, 2022.

[30] L. Padgham, Z. Zhang, J. Thangarajah, and T. Miller, "Model-based test oracle generation for automated unit testing of agent systems," *IEEE Transactions on Software Engineering*, vol. 39, no. 9, pp. 1230–1244, 2013.

[31] M. Varshosaz, M. Ghaffari, E. B. Johnsen, and A. Wąsowski, "Formal specification and testing for reinforcement learning," *Proceedings of the ACM on Programming Languages*, vol. 7, no. ICFP, pp. 125–158, 2023.

[32] K. Claessen and J. Hughes, "Quickcheck: a lightweight tool for random testing of haskell programs," in *Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming*, 2000, pp. 268–279.

[33] K. Jothimurugan, R. Alur, and O. Bastani, "A composable specification language for reinforcement learning tasks," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[34] W. Shen, J. Wan, and Z. Chen, "Munn: Mutation analysis of neural networks," in *2018 IEEE international conference on software quality, reliability and security companion (QRS-C)*. IEEE, 2018, pp. 108–115.

[35] Q. Hu, L. Ma, X. Xie, B. Yu, Y. Liu, and J. Zhao, "Deepmutation++: A mutation testing framework for deep learning systems," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 1158–1161.

[36] A. Panichella and C. C. Liem, "What are we really testing in mutation testing for machine learning? a critical reflection," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 2021, pp. 66–70.

[37] X. Wan, T. Li, W. Lin, Y. Cai, and Z. Zheng, "Coverage-guided fuzzing for deep reinforcement learning systems," *Journal of Systems and Software*, vol. 210, p. 111963, 2024.

[38] A. Lacoste, A. Luccioni, V. Schmidt, and T. Dandres, "Quantifying the carbon emissions of machine learning," *arXiv preprint arXiv:1910.09700*, 2019.