IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received October 5, 2020, accepted October 22, 2020, date of publication October 26, 2020, date of current version November 6, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3033888

Using Deep Reinforcement Learning for Exploratory Performance Testing of Software Systems With Multi-Dimensional Input Spaces

TANWIR AHMAD, ADNAN ASHRAF, DRAGOS TRUSCAN, ANDI DOMI, AND IVAN PORRES
Faculty of Science and Engineering, Åbo Akademi University, 20500 Turku, Finland
Corresponding author: Tanwir Ahmad (tanwir.ahmad@abo.fi)

This work was supported by the Electronic Component Systems for European Leadership Joint Undertaking under Grant 737494.

ABSTRACT During exploratory performance testing, software testers evaluate the performance of a software system with different input combinations in order to identify combinations that cause performance problems in the system under test. Performance problems such as low throughput, high response times, hangs, or crashes in software applications have an adverse effect on the customer's satisfaction. Since many of today's large-scale, complex software systems (e.g., eCommerce applications, databases, web servers) exhibit very large multi-dimensional input spaces with many input parameters and large ranges, it has become costly and inefficient to explore all possible combinations of inputs in order to detect performance problems. In order to address this issue, we introduce a method for identifying input combinations that trigger performance problems in the software system under test. Our method, under the name of iPerfXRL, employs deep reinforcement learning in order to explore a given large multi-dimensional input space efficiently. The main benefit of the approach is that, during the exploration process, it learns and recognizes the problematic regions of the input space that have a higher chance of triggering performance problems. It concentrates the search in those problematic regions to find as many input combinations as possible that can trigger performance problems while executing a limited number of input combinations against the system. In addition, our approach does not require prior domain knowledge or access to the source code of the system. Therefore, it can be applied to any software system where we can interactively execute different input combinations while monitoring their performance impact on the system. We implement iPerfXRL on top of the Soft Actor-Critic algorithm. We evaluate empirically the efficiency and effectiveness of our approach against alternative state-of-the-art approaches. Our results show that iPerfXRL accurately identifies the problematic regions of the input space and finds up to 9 times more input combinations that trigger performance problems on the system under test than the alternative approaches.

INDEX TERMS Exploratory performance testing, deep reinforcement learning, test data generation.

I. INTRODUCTION

One of the most critical and challenging tasks for developers is to identify and fix performance problems of software systems [1]. Performance problems such as low throughput, high response times, hangs, or crashes in software applications have an adverse effect on the customer's satisfaction. According to [2], there are higher chances of a software system crashing due to performance problems rather than functional failures.

Recent reports [1] show that certain input combinations can trigger more than half of the performance bottlenecks identified in non-trivial software systems. The reason is that certain input combinations can invoke inefficient code sequences or resource-intensive operations, which result in overall system performance degradation commonly referred to as performance bottlenecks. Molyneaux [3] defines a *performance bottleneck* as a software defect which degrades the performance of the *System Under Test* (SUT) unexpectedly.

The associate editor coordinating the review of this manuscript and approving it for publication was Porfirio Tramontana.

Performance testing is an approach to identify performance bottlenecks [2], [4]. A performance test typically comprises a sequence of *actions* and the *test data* for those actions. During the performance testing process, different *Key Performance Indicators* (KPIs) of the SUT are monitored (as test outputs) to uncover performance bottlenecks (i.e., deviations from the expected KPIs threshold). Such KPIs can be, for instance, latency, elapsed execution time, bandwidth, resource utilization, and throughput.

There are several ways in which the test input values used in performance testing are generated in current practices [5]; however, each has some disadvantages:

1) manually crafted by the test engineers based on their experiences and intuition - requires that the test engineers have rigorous domain knowledge about the SUT;
2) generated by performance profilers - requires access to the source code of the SUT, which is often unavailable. In addition, it requires test inputs to instrument the program under test for performance profiling;
3) collected during the regular usage of the SUT - they represent the most common input combinations executed by users. However, they miss the rare and infrequent, yet problematic, combinations;
4) extracted from crash reports sent by customers - collecting the test input values post-production is not recommended since the failure has already affected the end-users of the system. Thus, the damage in reputation of the company has been produced already. In addition, the testing process should be proactive.

Since many of today's systems exhibit very large multi-dimensional input spaces with many input parameters and large ranges, it has become costly and inefficient to explore all possible input combinations in order to detect performance bottlenecks [1], [6]. The reason behind this is that not only the number of input combinations is very large, but also the time for executing a single combination against the SUT can be relatively long. These two factors make the exhaustive testing of all input combinations virtually infeasible. This is also accentuated in the case where the SUT is regarded as a *black-box* and one does not have access to the internal dynamics of the SUT.

Exploratory Testing (ET) is a software testing technique which does not rely on a pre-defined set of test cases; instead, the tester continuously learns, creates, and executes test cases [7]. The tester extracts new information and insights from the results of the previously executed test cases and creates new, better test cases. The goal of ET is to find software defects by learning the system behavior and being less dependent on the test documentation. ET is usually performed manually and requires rigorous domain knowledge and substantial efforts and time.

In this paper, we propose an automated exploratory performance testing method that explores the input space of the SUT and identifies, in an effective manner, a limited number of input combinations that trigger performance bottlenecks. Our method, under the name of *iPerfXRL*, employs *Deep Reinforcement Learning* (DRL) [8] in order to explore non-exhaustively a given large multi-dimensional input space. It does not require prior knowledge about the system or domain. We can apply iPerfXRL to any software system where we can interactively execute different input combinations while monitoring their performance impact on the SUT. During the exploration process, the method learns to focus on those regions of the input space that have a higher chance of triggering performance bottlenecks.

The work presented in this article is an extension of our work published in [9], where we initially formulated the input space exploration problem for performance testing as a DRL problem. However, in the current article, we provide several improvements:

1) We reformulate the input space exploration problem in the context of DRL by redefining the action space and the reward function. This resulted in improved bottleneck detection rate.
2) We provide the tool support for our approach using Python's *Stable Baselines* [10] library to automate the exploration process.
3) We empirically evaluate the efficiency and effectiveness of our method against our previous approach (i.e., PerfXRL [9]), random testing, deterministic grid search, and combinatorial interaction testing.
4) We experimentally show that the improved PerfXRL (iPerfXRL) is able to detect more performance bottlenecks than the PerfXRL [9] and, at the same time, it identifies up to 9 times more bottlenecks than the other approaches.

The rest of the paper is structured as follows: Section II provides an introduction to RL. In Section III, we describe our approach. We empirically evaluate our approach in Section IV. Section V presents an overview of the related work. Section VI specifies threats to the validity of this work. Finally, Section VII discusses conclusions and future work.

II. REINFORCEMENT LEARNING

Reinforcement Learning (RL) is a reward-driven machine learning technique in which an *agent* learns by interacting with an unknown *environment* in order to accomplish a goal. The agent collects feedback (or a *reward*) from the environment by performing an *action* according to the current *state* of the environment. The goal of the agent is to maximize the expected cumulative rewards over time by finding the optimal (or a near-optimal) sequence of actions.

An RL process can be characterized as a *Markov Decision Process* (MDP). The process is represented as a tuple $\langle S, A, P, R, \gamma \rangle$, where

- S is a finite set of states of the environment;
- A represents a finite set of permissible actions;
- $P : S \times A \rightarrow S$ is a transition function;
- $R : S \times A \rightarrow \mathbb{R}$ is a reward function;
- $\gamma \in [0, 1]$ is a discount factor.

At every time step t, the agent observes the current *state* $s_t \in S$ and then it performs an *action* $a_t \in A$. After each action, the agent receives a scalar *reward* $r_{t+1} \sim R(s_t, a_t)$[1] from the environment, as well as the next state $s_{t+1} \sim P(s_t, a_t)$.

Based on the reward, the agent recognizes which actions are *good* or *bad* with respect to a given state. The main property of an MDP is that, given the current state at any particular time step, the future rewards and states are conditionally independent of the previous states. In other words, the current state contains all the required information which is needed by the environment to make state transitions and to return rewards corresponding to the agent actions. Therefore, reward function R and transition function P only require the current state s_t and action a_t as input parameters in order to calculate the reward r_{t+1} and the next state s_{t+1}, respectively.

The agent chooses an action according to a *policy* π, which maps the states of the environment to the actions that can be performed in those states. In other words, $\pi(a|s) = \Pr\{a_t = a|s_t = s\}$ represents the probability of selecting action a in state s while following policy π. The objective of RL is to find an *optimal policy* π^* which maximizes the total excepted discounted return:

$$\pi^* = \arg\max_{\pi} \mathbb{E}[G|\pi] \tag{1}$$

where G_t specifies the *total expected discounted return* at a time step t: $G_t \doteq \sum_{k=t+1}^{T} \gamma^{k-t-1} R(s_k, a_k)$, where T is the maximum number of time steps in a finite MDP. This duration is known as an *episode*.

A. DEEP REINFORCEMENT LEARNING

In this work, we use a DRL algorithm, called SAC [12], to implement our method. DRL [8] is a combination of RL and *deep learning* [13] principles where we utilize *Deep Neural Networks* (DNNs) as function approximators to learn the environment. DNNs [14] are multi-layer neural networks that are designed to approximate complex functions by learning different levels of representations of the given training data. The main benefit of DRL is that the agent can learn various levels of abstractions from the data and generalize the knowledge from the observed input combinations to the unseen input combinations.

The SAC algorithm consists of two components: actor and critic. The actor is responsible for choosing the actions based on the given states. The critic does not directly influence the selection of the actions; instead, the actor utilizes the feedback from the critic to improve the policy. This architecture allows actor-critic methods to perform better than the other RL methods by reducing the variance and accelerating the learning process [11]. The SAC algorithm approximates the value function $V_\psi(s_t)$ and the soft action-value function $Q_\theta(s_t, a_t)$ to implement the *critic*. The *actor* is based on the tractable policy $\pi_\phi(a_t|s_t)$. ψ, θ and ϕ represent the parameters of the DNNs [13].

[1]We follow the conventions by Sutton *et al.* [11] where they use r_{t+1} to denote the reward in response to action a_t in state s_t.

The algorithm is based on the *maximum entropy framework* [12]. Unlike the standard RL algorithms where the goal is to maximize the expected reward (see Equation 1), in the maximum entropy framework, we learn a policy π^* that maximizes the sum of the expected rewards and the entropy of the policy at each visited state:

$$\pi^* = \arg\max_{\pi} \sum_{t=0}^{T} \mathbb{E}_\pi[R(s_t, a_t) + \alpha \mathcal{H}(\pi(.|s_t))] \tag{2}$$

where $\mathcal{H}$ returns the entropy of the policy and the temperature parameter α specifies the relative importance of the entropy. The entropy term determines the randomness in the policy: $\mathcal{H}(\pi(.|s_t)) = \mathbb{E}_\pi[-\log \pi(.|s_t)]$. In other words, as we increase the value of α, the agent performs more random actions which result in a more extensive exploration of the state space.

There are four main reasons why we chose SAC over the other RL algorithms: (1) SAC is an off-policy algorithm, which means that the algorithm can reuse the previous experiences for multiple learning updates to improve the learning efficiency of the agent. The algorithm stores the previous experiences (i.e., recently executed input combinations and their performance impact on the SUT) in a cyclic buffer, called *replay memory*. It uniformly samples the reply memory to train the agent. This process is known as *experience replay* [15]. (2) The performance of the algorithm is less sensitive to the different hyperparameter values than the other RL algorithms [16]. (3) As we have mentioned at the beginning of this section that SAC is a DRL algorithm and has the ability to generalize the knowledge from the previously executed input combinations to the unseen input combinations. Therefore, we can apply our approach to the systems with large continuous input spaces and find performance bottlenecks. (4) The maximum entropy objective encourages the agent to explore the state space more extensively while avoiding unrewarding regions of the space. Consequently, iPerfXRL should be able to find more potential performance bottlenecks than other approaches.

III. PERFORMANCE EXPLORATION APPROACH

iPerfXRL employs RL to identify performance bottlenecks in a SUT without any prior domain knowledge about the SUT or of its internal implementation (see Figure 1). The identification of performance bottlenecks is made by executing different input combinations against the SUT and monitoring the deviations of the KPI values from certain pre-configured *acceptable performance thresholds*. Such thresholds can be derived from different sources such as requirement specifications or *Service Level Agreements* (SLAs), and vary from system to system. The main goal of iPerfXRL is to identify as many input combinations as possible that can trigger performance bottlenecks. From hereon, we refer to those combinations as *relevant combinations*. We achieve the identification by recognizing those regions of the input space that have a higher potential to trigger performance bottlenecks.

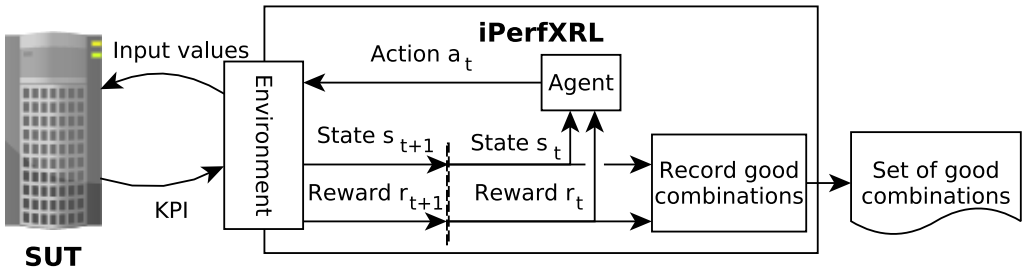

FIGURE 1. General view of the iPerfXRL approach (Adapted from [11]).

To illustrate our approach, we start with a hypothetical example and show how we place it in the context of DRL.

A. CONCEPTUAL EXAMPLE

Let us consider a conceptual example of a system which accepts three input parameters. For convenience, we will refer to it as *Conceptual Black-box System* (CBS). The parameters of CBS can take integer values from 1 to 50 for the first two parameters, and from 1 to 400 for the third one.

CBS is just another software system with a large multi-dimensional input space where different combinations can exhibit different performance behavior of the system. The input parameters can be actual input values, or they can be configuration parameters of the SUT that will influence the performance of the system. For instance, database systems (e.g., MySQL, Postgres) and web servers (e.g., Apache HTTP Server) are the real-world examples of CBS. These systems have hundreds of configurable parameters [17]. Finding a set of values for these parameters that greatly affects the performance of the system is a challenging task. In this section, we will discuss how our approach can be used to identify such values of the input parameters without executing all possible combinations.

B. DEEP REINFORCEMENT LEARNING PROBLEM FORMULATION

We represent the input space exploration problem for performance testing as a search problem over a large multi-dimensional input space S representing all possible input combinations. The goal is to identify a near-complete subset of input combinations (i.e., relevant combinations) $\mathcal{B} \subset S$ that yield resource-intensive computations on the SUT. These can be observed as degradations of the performance, for instance, lower throughput and higher response time in relation to specified thresholds. We denote the complement of relevant combinations, $S \setminus \mathcal{B}$, as the *set of irrelevant combinations* with respect to our approach.

The DRL agent interactively tries out different input combinations while observing their performance impact on the SUT formulated as a *reward* function. This is a continuous process. At each step, the agent suggests a new action to be executed, which results in a new input combination. The environment executes the new combination against the SUT and returns a reward to the agent, which uses the reward to improve the selection of future actions.

During this process, the approach maintains a list of relevant combinations. These combinations can be later on filtered or ranked with respect to their performance impact on the system, and can be further reviewed or clustered to assist debugging performance bottlenecks in the system.

We define different components of our approach as follows:

1) STATE SPACE

We define the state space S of the iPerfXRL as the set of all possible input combinations $s_t \in S$ through which the agent can transition during the exploration. Each state represents a single input combination expressed as a vector $\{v_t^{(1)}, v_t^{(2)}, \ldots, v_t^{(N)}\}$ at a given time step t, where N is the number of input parameters (or dimensions of the input space) of the SUT. Thus, the size of the state space is equal to the size of the input space.

In our example, the three parameters $v^{(1)}$, $v^{(2)}$, and $v^{(3)}$ of CBS create a multi-dimensional input space S_{CBS} with a size of 50*50*400 states, that is 10^6 states.

2) ACTION SPACE

The agent observes the current state s_t (i.e., the current input combination) and provides an action a_t to the environment. Based on the action a_t, the environment modifies the s_t in order to produce the next state s_{t+1} (i.e., a new input combination). Thus, we can specify the action space A as a set of potential modifications that can be made to the current state in order to produce the next state.

In our previous work [9], the selection of actions was restricted to a finite discrete action space, which means that at every time step t, the agent could either increase or decrease the value of a single input in the current state by a fixed amount in order to get the next state. Therefore, the agent had to go through numerous unrewarding states in order to get to the rewarding regions of the input space, which resulted in decreasing the overall bottleneck detection rate of the

TABLE 1. Examples of state updates according to the actions.

No.	Action	if $s_t = \{15, 40, 186\}$ then $s_{t+1} =$
1	$\{0.10, 0.15, -0.35\}$	$\{20, 47, 46\}$
2	$\{-0.40, 0.27, 0.17\}$	$\{0, 53, 254\}$
3	$\{0.30, 0.40, -0.22\}$	$\{30, 50, 98\}$

approach. In order to address this issue, iPerfXRL extends the discretized action space to continuous action space. At every time step t, the agent can increase or decrease the values of all the inputs at once in order to get the next state. Furthermore, the input values are not updated by a fixed amount; instead, the agent decides how much value of each input should be increased or decreased:

$$\begin{aligned} A &= \{a^{(i)} | -1.0 \leq a_{low} \leq a \leq a_{high} \leq 1.0\} \\ vrange^{(i)} &= vmax^{(i)} - vmin^{(i)} + 1 \\ v_{t+1}^{(i)} &\leftarrow clip(v_t^{(i)} + a_t^{(i)} * vrange^{(i)}, i) \\ &\forall i \in [1, 2, \ldots, N] \end{aligned} \tag{3}$$

where function $clip(x, i)$ returns x in the range: $[vmin^{(i)}, vmax^{(i)}]$ of the input variable $v^{(i)}$. The function *clip* helps to avoid invalid updates to the inputs. a_{low} and a_{high} are hyperparameters, which control the magnitude of the updates to the inputs.

For example, we set a_{low} and a_{high} to -0.4 and 0.4, respectively, for the action space A_{CBS} of the CBS. Thus, the maximum size of the update to the inputs that the agent can perform is restricted to the 40% of the $vrange_{CBS}^{(i)}$, where $vrange_{CBS} = \{50, 50, 400\}$. Table 1 lists several examples of the possible actions that the agent can select and how they modify the current state in order to produce the next state. In our previous work [9], we addressed only integer input parameters, but iPerfXRL can be applied to float inputs without any modification. Furthermore, iPerfXRL can easily be extended to support other types of inputs (e.g., string, categorical) by modifying the action and the input space accordingly.

3) TRANSITION FUNCTION

A transition function P accepts a *state* and an *action* as inputs and returns a *new state*. In Markov chains, the function is stochastic for a non-deterministic environment. However, in our case, the environment is *deterministic*, which means that given the current state and the selected action, we can calculate the exact outcome of the transition function at any time step.

4) POLICY

The goal of our agent is to maximize the maximum entropy objective as defined in Equation (2). The *temperature parameter* α is used to balance between exploitation and exploration. A higher value of α corresponds to more exploration, whereas a lower α correlates to more exploitation. In order words, increasing the value of α would inspire to explore the input space by performing more random actions. This ensures that the agent does not precipitately converge to a poor strategy. However, it is difficult to manually set the optimal value for α because the entropy term in the objective function can fluctuate erratically during the optimization of the policy. Therefore, Haarnoja *et al.* [16] have updated the SAC algorithm to automatically learn and adjust the value of α during the training of the agent. This allows the agent to explore more extensively the uninvestigated regions of the input space and exploit the acquired knowledge when traversing familiar regions. Thus, in this work, we do not set the value of α manually; instead, we let the SAC algorithm regulate it.

5) REWARD

The reward value is the primary basis for updating the policy; if the chosen action by the policy is followed by a low reward, then the policy may be updated to choose some other action on that state in the future.

In our case, the agent gets high rewards when it finds relevant combinations. We compare the measured KPI value (e.g., CPU load, disk usage, or elapsed execution time of the SUT) against the given *acceptable performance threshold* $\mathcal{L}$ to identify relevant combinations. The conjecture is that the measured KPI value would be different from the given threshold $\mathcal{L}$ for relevant combinations. As we have defined in Section III, *relevant combinations* refer to those input combinations that are most likely to cause performance bottlenecks.

The agent can obtain reward only once per relevant combination. The reason is to encourage agent to explore input space and find more unique relevant combinations. The reward $r_{t+1}^{\mathcal{L}}$ is defined as

$$r_{t+1}^{\mathcal{L}} = \begin{cases} x \in \mathbb{Z}_{>0} & \text{if } \mathrm{E}(\mathrm{s_{t+1}}) \lesseqgtr \mathcal{L} \textbf{ and } isNew(s_{\mathrm{t+1}}) \\ x \in \mathbb{Z}_{<0} & \text{otherwise.} \end{cases} \tag{4}$$

where E is an executor function which runs the performance test cases using the provided input combination s_{t+1} against the SUT and returns the measured KPI value, and function $isNew(s)$ checks whether the given input combination s has been executed before or not. In summary, Equation (4) specifies that if the measured KPI value for the combination s_{t+1} is less than or greater than the given acceptable performance threshold $\mathcal{L}$ and the combination s_{t+1} has not been executed so far, the agent receives a positive reward (i.e., a positive integer) because it has found a new relevant combination.

C. FINDING BOTTLENECKS

Algorithm 1 summarizes our approach. Our agent explores the input space in an episodic manner. In every episode, the agent starts from a random state (line *5*) and performs a certain number of steps. On each step, the agent suggests an action to the environment that computes a new input combinations based on the proposed action and the current input combination (lines *9* to *10*). Next, the environment executes the newly created input combination against the SUT and calculates the reward (line *11*). We keep track of the reward that the agent has collected so far in this episode (line *12*).

Algorithm 1 Pseudocode of iPerfXRL

Require: *TtlSteps* - total number of steps, *MaxNegR* - maximum negative reward per episode, *Env* - RL environment
Ensure: $\mathcal{B}$ - list of relevant combinations

1: *Agent* ← *SACAgent*(*Env*){create a SAC agent}
2: $\mathcal{B}$ ← {} {initialize a set for storing relevant combinations}
3: *step* ← 0
4: **while** *step* < *TtlSteps* **do**
5: *s* ← *Env.reset*() {reset the environment for the new episode and return a random state}
6: *done* ← **false**
7: *current_ep_r* ← 0 {store the reward for the current episode}
8: **while** *step* < *TtlSteps* **or not** *done* **do**
9: *action* ← *Agent.act*(*s*) {*agent* observes the current state *s* and returns an *action*}
10: *next_s* ← *Env.P*(*s*, *action*) {get the next state using the transition function *P* defined in the environment}
11: *r* ← *Env.R*(*next_s*) {calculate the reward using Equation (4)}
12: *current_ep_r* ← *current_ep_r* + *r*
13: **if** *current_ep_r* ≤ *MaxNegR* **then**
14: *done* ← **true**
15: **end if**
16: *Agent.learn*(*s*, *action*, *next_s*, *r*, *done*) {*agent* uses the provided arguments to train the deep neural network to select better actions in the future}
17: **if** *r* > 0 **then**
18: $\mathcal{B} \leftarrow \mathcal{B} \cup$ *next_s* {add a relevant combination to the set $\mathcal{B}$ }
19: **end if**
20: *s* ← *next_s*
21: *step* ← *step* + 1
22: **end while**
23: **end while**

An episode ends when the reward for the current episode becomes less than or equal to *MaxNegR* (e.g., −100) (lines *13* to *15*). The conjecture is that the agent can be stuck in some region of the input space where it could not find relevant input combinations anymore. Thereupon, the environment begins the new episode by generating a new random input combination. Starting every episode from a new random input combination allows the agent to explore different regions of the state space in each episode.

Furthermore, the agent stores the results of the recently executed input combinations (i.e., current input combination s_t, action a_t, next input combination s_{t+1}, and reward r_{t+1}) (line *16*), at each time step t, in the *replay memory*. These results are reused for multiple learning updates to improve the learning efficiency of the agent.

D. PARAMETERS TUNING

Both parameters, number of steps (*TtlSteps* $\in \mathbb{Z}_{>0}$) and the maximum negative reward per episode (*MaxNegR* $\in \mathbb{Z}_{<0}$), mainly depend on the complexity of the SUT and the duration of test runs against the SUT and vary from case to case. Nevertheless, a higher value of *TtlSteps* would allow the agent to explore the input space more extensively and yield better results.

The value of *MaxNegR* is used to control the length of an episode. A large negative value of *MaxNegR* indicates that the agent can execute a large number of irrelevant combinations which do not trigger bottlenecks before the environment resets and starts a new episode. As a result, the duration of episodes would be long, and if the *TtlSteps* is small, the agent would only explore a few regions of the input space. Thus, due to the lack of broad exploration of the input space, the agent might not find an adequate number of relevant combinations. On the other hand, when the value of *MaxNegR* gets close to zero, the episodes become shorter, and the environment resets frequently. This also leads to reduced bottleneck detection rate because the agent cannot adequately explore the regions of the input space. With the right value of *MaxNegR*, the agent balances diversification and exploration of the input space regions, leading to higher bottleneck detection rate.

IV. EVALUATION

In this section, we experimentally evaluate our RL approach by answering the following research questions:

- RQ1: Can iPerfXRL adequately identify the relevant regions of the input space which contain relevant combinations by exploring only a subset of the input space?
- RQ2: How effective is iPerfXRL compared to other deterministic and non-deterministic approaches?
- RQ3: How is the performance of iPerfXRL affected by the values chosen for the hyperparameters of the algorithm?

RQ1 investigates how well iPerxXRL learns and determines those regions of the input space which contain relevant combinations (i.e., input combinations that can triggers performance bottlenecks on the SUT, as stated in Section III). To answer RQ2, we compare the number of relevant combinations identified by iPerfXRL against the alternative approaches. RQ3 focuses on the effects of different parameters on the bottleneck detection rate of iPerfXRL.

We have conducted several computational experiments to answer the research questions. We follow the guidelines suggested by Barr *et al.* [18] to perform the experiments. The goal of an experiment is to examine the effect of changing one or more variables while keeping the rest of the variables unchanged. Within an experiment, all the variables that we control and modify are called *independent variables*, whereas the variables that are measured to study the effects of variations in the independent variables are known as *dependent variables*.

The following subsection describes the subject application used for evaluation. Then, we present the methodology and the setup of our experiments. Lastly, we discuss the results with respect to each research question.

A. SUBJECT APPLICATION

We use the reference web application RUBiS [19] as a subject application in our experiments. RUBiS is a web-based application that implements the core functionality of an auction site. It has been widely used in academia for performance evaluation, with over 300 citations on Google Scholar [20]. We use an Apache HTTP web server [21] 2.4.29 with PHP [22] 7.2.10 to host the front-end of the application. The backend database is MySQL Server [23] 5.7.

The inputs to RUBiS are provided as HTTP requests. Table 2 defines a performance test case for RUBiS as a scenario (i.e., a sequence of HTTP requests to three URLs).

TABLE 2. RUBiS test case.

Request ID	Request URL
R1	/SearchItemsByRegion.php?category=CID& categoryName=CN&region=RID
R2	/ViewItem.php?itemId=IID
R3	/ViewUserInfo.php?userId=UID

RUBiS test case requires four integer input parameters listed in Table 3. These input parameters constitute the RUBiS input space S_{RUBiS} with a size of *3 100 000* (i.e., the total number of input combinations). The input parameter *Category Name* (CN) is not considered as an input parameter because its value depends on the value of input parameter *Category ID* (CID). The size of the action space A_{RUBiS} for RUBiS is four because we have four input parameters. An input combination $s_t \in S_{RUBiS}$ can be represented as: $s_t = \{CID_t, RID_t, IID_t, UID_t\}$ at time step t. Even though the input parameters are used in different URLs and are provided sequentially, we consider them as an atomic definition of the state.

TABLE 3. RUBiS input space S_{RUBiS}.

Input parameter	Range
Category ID (CID)	[1, 20]
Region ID (RID)	[1, 62]
Item ID (IID)	[1, 50]
User ID (UID)	[1, 50]

In order to calculate the reward r_{t+1} for the agent, we need to define the executor function E_{RUBiS} for RUBiS. In our evaluation, we use the *Elapsed Execution Time* (EET) of the SUT for a given input combination as a KPI. Our approach will attempt to identify those input combinations that will take the EET over a given *acceptable performance threshold* $\mathcal{L}$. The executor function $E_{RUBiS}(s_{t+1})$ runs the RUBiS performance test case (listed in Table 2) using the input values s_{t+1} and returns the total EET of the test case (i.e., the sum of EETs of the URL requests). For instance, if $s_{t+1} = \{5, 30, 23, 35\}$, the function $E_{RUBiS}(s_{t+1})$ runs the following URL requests:

R1: /SearchItemsByRegion.php?category=5 &categoryName=Coins&region=30
R2: /ViewItem.php?itemId=23
R3: /ViewUserInfo.php?userId=35

In this case, the EET of the RUBiS performance test case for s_{t+1} would be equal to the sum of EET of R1, R2, and R3:

$$E_{RUBiS}(s_{t+1}) = \overbrace{0.59}^{R1} + \overbrace{0.10}^{R2} + \overbrace{0.04}^{R3} = 0.73 \; seconds \quad (5)$$

B. METHODOLOGY

In order to evaluate iPerfXRL in a controlled setting, we benchmark the performance of the RUBiS by exploring the complete input space and measuring EET for the given scenario by exercising all possible input combinations. The results showed that different input combinations have no considerable impact on the performance of RUBiS.

For the evaluation purposes, we injected 20 clusters of synthetic performance bottlenecks into the source code, as delays that are triggered on certain input combinations. Since our approach is intended to perform exploratory performance testing of systems in a black-box manner (i.e., without access to the internal implementation), iPerXRL is not aware of these bottlenecks beforehand and tries to identify them during the exploration.

A cluster of injected bottlenecks would emulate computationally expensive operations and increase the elapsed execution time of the RUBiS performance test case by 5 seconds. For instance, s_{t+1} input combination would trigger one of our injected clusters of bottlenecks because the input combination $s_{t+1} = \{5, 30, 23, 35\}$ satisfies the following condition:

$$7 \leq CID \leq 10 \wedge 23 \leq RID \leq 39 \wedge 12 \leq IID \leq 28 \wedge 31 \leq UID \leq 47 \quad (6)$$

After injecting the bottlenecks, the EET of the RUBiS performance test case for s_{t+1} would be 5.73 seconds, instead of 0.73 seconds (as calculated in Equation 5).

The reason for injecting clusters of bottlenecks instead of individual bottlenecks is that the performance problems usually tend to affect a group of adjacent combinations contrary to a single combination. Since each injected performance bottleneck adds a delay of 5 seconds to the elapsed execution time of the performance test, we set the value of $\mathcal{L}$ to 5 and use the following equation for reward calculation:

$$r^{\mathcal{L}}_{t+1} = \begin{cases} 10 & \text{if } E_{RUBiS}(s_{t+1}) > 5 \textbf{ and } isNew(s_{t+1}) \\ -1 & \text{otherwise.} \end{cases} \quad (7)$$

As we have discussed in Section III, in real-world situations, the value of $\mathcal{L}$ is usually extracted from requirement specifications or SLAs. In other words, the $\mathcal{L}$ threshold specifies a satisfactory level of performance with respect to a given KPI. The threshold value does not affect the performance of our approach. In our preliminary tests, we got good results by setting the positive and negative reward to 10 and -1, respectively, instead of typical values of 1 and -1.

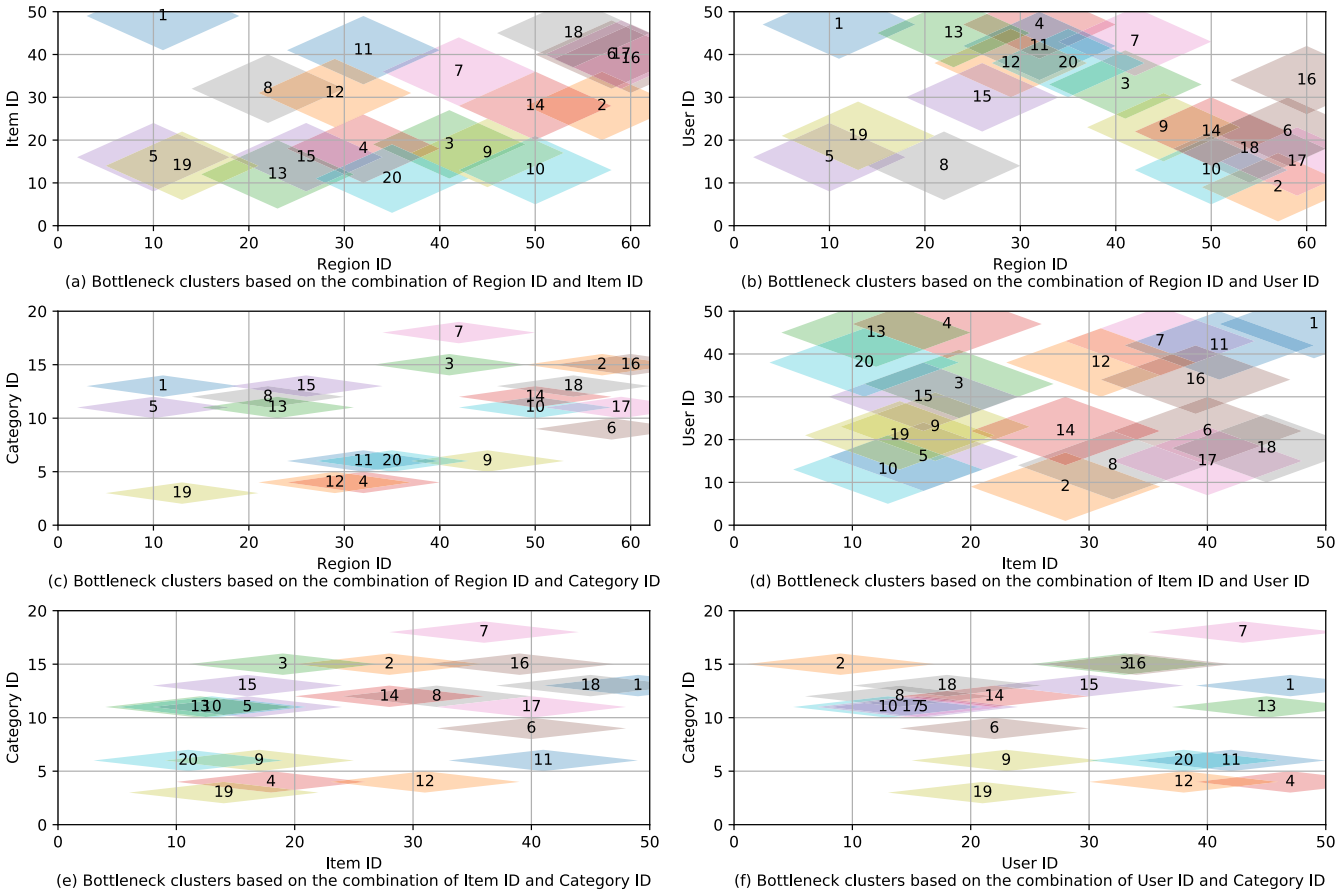

FIGURE 2. Distribution of clusters of injected bottlenecks for each pair of the input variables (listed in Table 3) in RUBiS$_{UNI}$. Each diamond shape in the subfigures (a) to (f) corresponds to a bottleneck cluster. The values on the diamonds indicate the identifier of the clusters.

Injecting artificial bottlenecks raises the question of *how the distribution of injected clusters of performance bottlenecks affects the performance of iPerfXRL in finding the bottlenecks*. To address this question, we created two variants of the RUBiS web application. In RUBiS$_{UNI}$, the clusters of bottlenecks are *uniformly* [24] distributed, which is not an ideal situation for iPerfXRL (or any other tool using heuristics) because the clusters are widely spread over the entire input space. In the second variant, RUBiS$_{POI}$, we used *Poisson distribution* [24] to distribute the clusters of bottlenecks. In the case of RUBiS$_{POI}$, as opposed to RUBiS$_{UNI}$, the clusters are packed together in the input space. Figures 2 and 3 show the distributions of clusters of bottlenecks in RUBiS$_{UNI}$ and RUBiS$_{POI}$, respectively, where each diamond shape represents one cluster of bottlenecks, and the values on the diamonds indicate the identifier of the clusters. These figures are two-dimensional representations of the four-dimensional input space of RUBiS$_{UNI}$ and RUBiS$_{POI}$. In total, we have injected bottlenecks on 250 184 and 283 391 unique input combinations in RUBiS$_{POI}$ and RUBiS$_{UNI}$, respectively. These bottlenecks roughly cover 9% of the total input space. We have made the input spaces of RUBiS$_{UNI}$ and RUBiS$_{POI}$ with injected bottlenecks publicly available in [25].

We repeated every experiment described in this study 30 times to establish the statistical significance of the results. For each experiment, the approach under evaluation and the SUT ran on different machines. Each machine featured an Intel Core i7-3770K CPU, 16 GB of memory, 7200 rpm hard drive, and Ubuntu 18.04 Operating System. To reduce the network latency, the machines were connected via a 1Gb Ethernet connection in an isolated environment.

C. RQ1: IDENTIFYING RELEVANT REGIONS OF THE INPUT SPACE

The objective of this research question was to investigate whether iPerfXRL can learn and recognize the relevant regions of the input space, which contain relevant combinations. To answer this question, we conduct an experiment where we ran iPerfXRL on both RUBiS$_{UNI}$ and RUBiS$_{POI}$. The values of the different parameters (or independent variables) in iPerfXRL are as follows:

- *Number of steps* (*TtlSteps*): 775 000
- *Maximum negative reward per episode (MaxNegR)*: -100
- *Action space bounds*: we set a_{low} to -0.05 and a_{high} to 0.05. In our preliminary experiments, we got better

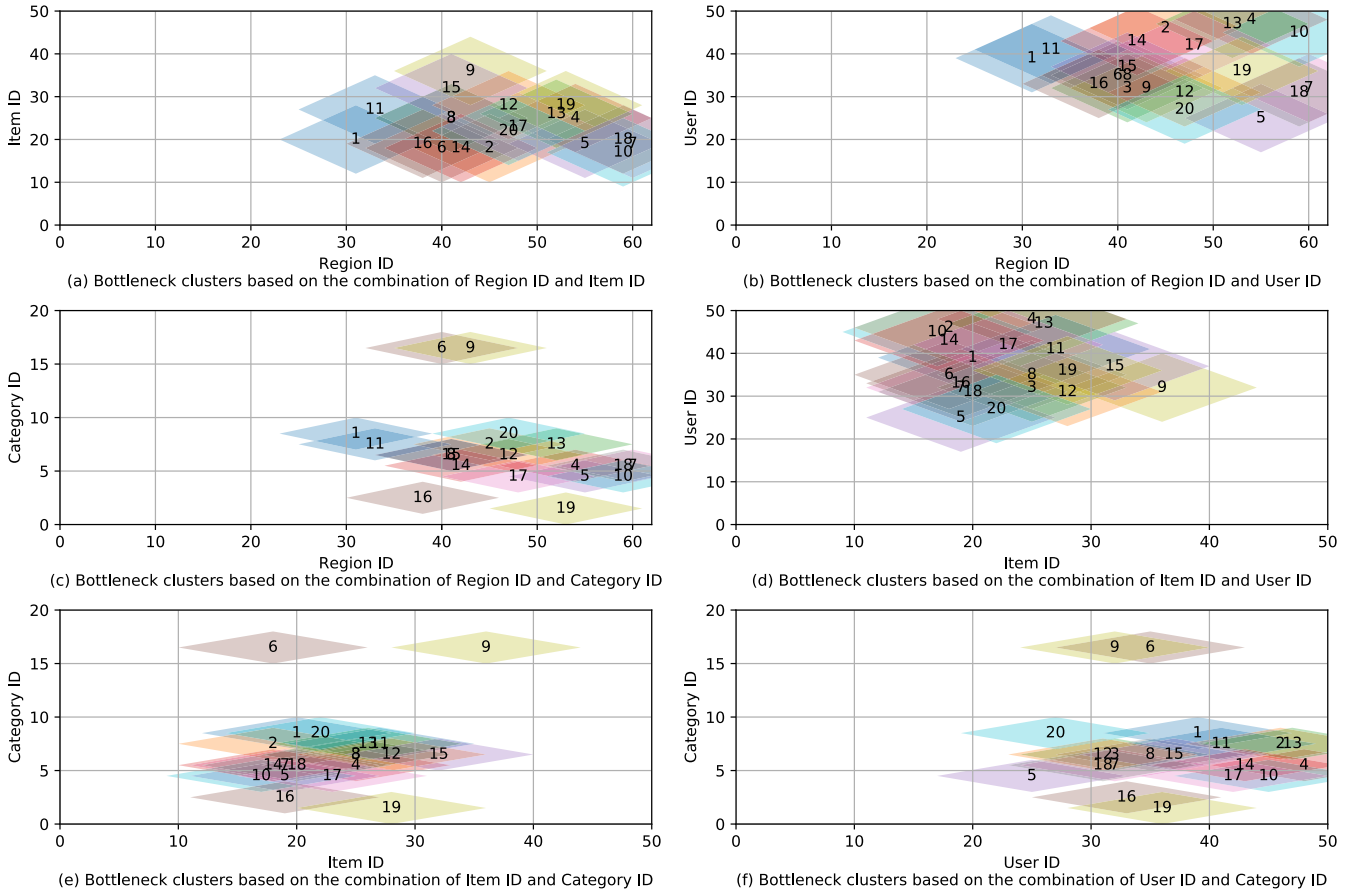

FIGURE 3. Distribution of clusters of injected bottlenecks for each pair of the input variables (listed in Table 3) in RUBiS$_{POI}$. Each diamond shape in the subfigures (a) to (f) corresponds to a bottleneck cluster. The values on the diamonds indicate the identifier of the clusters.

results when the a_{low} and a_{high} values were between -0.01 to -0.1 and 0.01 to 0.1, respectively.

- *Discount factor* (γ): 0.99
- *Size of the replay memory*: 300 000
- *Deep neural networks*: we used three fully-connected hidden layers with 32 nodes. The layers used the *hyperbolic tangent tanh*(x) [13] activation function. We used the *Adam* [26] optimization algorithm for DNNs with the *learning rate* (lr) set to 0.0003. Furthermore, we have noticed, in our preliminary experiments, that providing the *last five* executed input combinations as input to the SAC agent improves the learning capability of the agent significantly. This modification allowed the agent to learn the patterns and dependencies behind the input combinations and performance bottlenecks.

The dependent variable is a list of all the input combinations executed by the agent. iPerfXRL took around 110 minutes to execute 775 000 input combinations (i.e., 25% of the RUBiS input space S_{RUBiS} defined in Section IV-A) against the SUT. We performed a frequency analysis of the input combinations executed by the agent in order to build a heatmap for each pair of input variables, as shown in Figures 4 and 5 for RUBiS$_{UNI}$ and RUBiS$_{POI}$, respectively. Each entry in the heatmap corresponds to the relative execution frequency of the input combination. For instance, the more executed input combinations are indicated by a redder shade while the least executed input combinations are a bluer shade in the heatmaps.

In order to visually inspect how effectively iPerfXRL manages to recognize the relevant regions of the input space, we generate Figures 6 and 7 by overlaying the heatmaps (shown in Figures 4 and 5) on top of the bottleneck distributions (illustrated in Figures 2 and 3) for RUBiS$_{UNI}$ and RUBiS$_{POI}$, respectively. Figures 6 and 7 positively answer *RQ1* that iPerfXRL can identify the relevant regions of the input space. Furthermore, one can notice that the agent identified the relevant regions for RUBiS$_{POI}$ (where the clusters of bottlenecks are packed together) more precisely as compared to RUBiS$_{UNI}$. This is because there are numerous regions of the input space in RUBiS$_{UNI}$, which contain relevant combinations, but the agent can only execute a limited number of input combinations. Therefore, it converges to those regions, which have higher densities of relevant combinations. However, we can modify the convergence properties of the agent by tuning the temperature parameter α (discussed in Section III-B4). We will investigate it in our future work.

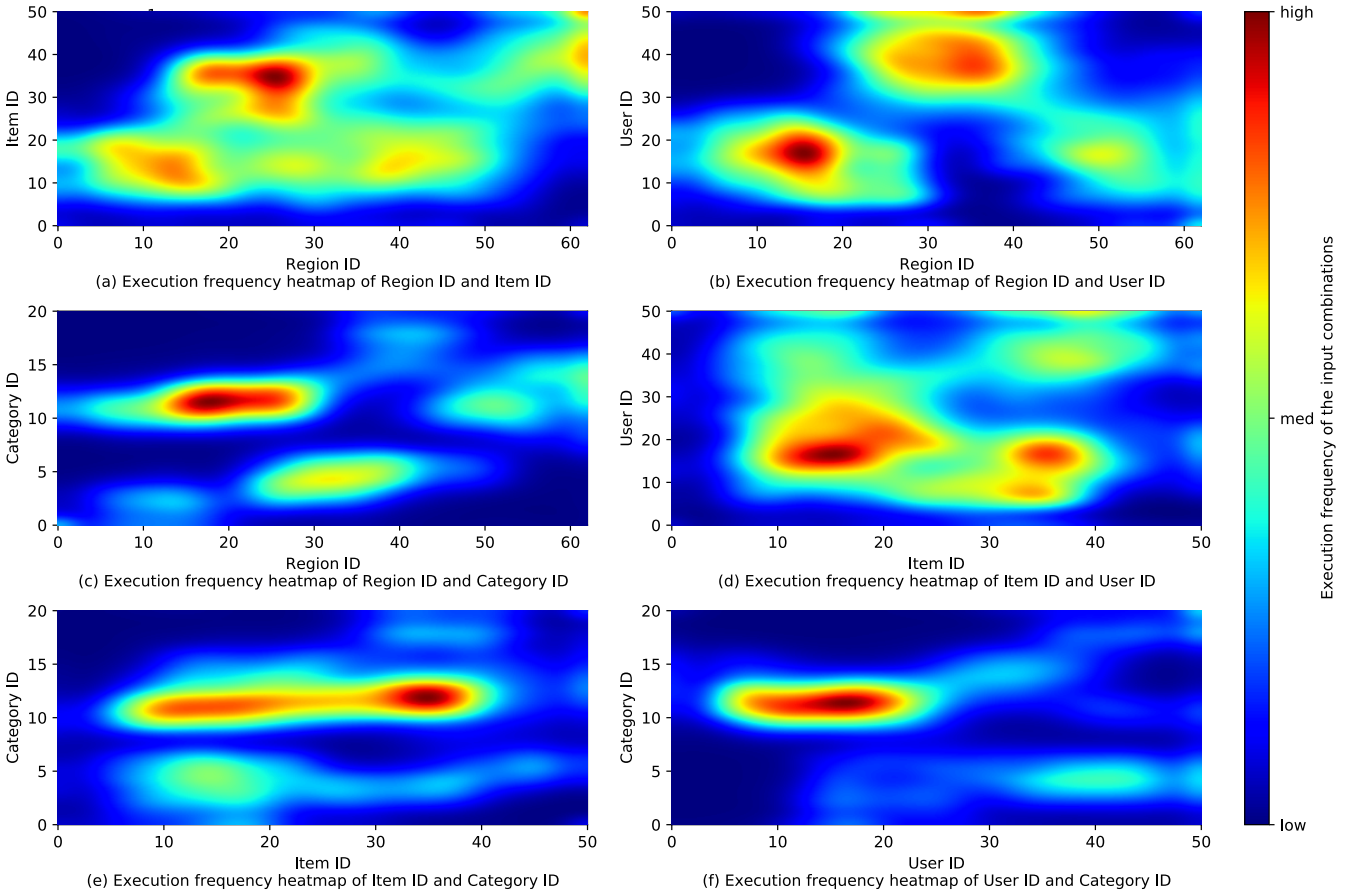

FIGURE 4. **Heatmaps built for RUBiS$_{UNI}$ with respect to each pair of the input variables (listed in Table 3) by performing a frequency analysis on the input combinations executed by the agent. A redder shade in the subfigures (a) to (f) indicates more frequently executed input combinations, whereas the least executed input combinations are depicted in a bluer shade.**

D. RQ2: EFFECTIVENESS OF iPerfXRL

The purpose of RQ2 is to measure the effectiveness of iPerfXRL by comparing the number of relevant combinations identified by it against the following alternative approaches: PerfXRL [9], *random testing*, *Deterministic Grid Search* (DGS), and *Combinatorial Interaction Testing* (CIT).

In random testing, we uniformly sample unique input combinations from the input space of the SUT for a given number of times. We choose *random testing* because it is a robust approach [27], [28] as compared to many other systematic testing approaches. Hamlet [29] recommended using random testing for large and irregular input spaces.

DGS is a deterministic approach. It uniformly samples the input space to obtain the input combinations using a *Low Discrepancy Sequence* (LDS) generator [30]. An LDS generator can sample the space more uniformly than pure random uniform draws [31]. The conjecture is that a higher degree of uniformity in the distribution of the sampled input combinations would increase the chances of finding the relevant combinations. There are several LDS generators, for example, Halton [32], Sobol [33], Haselgrove [34], and Galanti and Jung [35]. However, in our preliminary experiments, Sobol provided the best results as compared to the other generators. We use Chaospy [36] library to implement Sobol's DGS approach.

CIT [37] is a software testing technique that systematically explores and tests the subset of the input space by limiting the degree of interaction between the values of its input parameters. A CIT approach generates a test suite that contains every possible combination of input values for every combination of t input parameters at least once [38]. For instance, in pairwise testing (i.e., $t = 2$) of a system with ten binary input parameters, we would need to execute six input combinations to test every valid pair of input values of any two input parameters. However, the size of the test suite increases as we increase the value of t [39]. For instance, the size of the test suite to cover all 3-way interactions (i.e., $t = 3$) of the previously mentioned system would be 13. In this paper, we compare iPerfXRL against the four deterministic (or greedy) CIT approaches (i.e., IPOG [40], IPOG-D [40], IPOG-F [41], IPOG-F2 [41]) as implemented in ACTS [42] and one non-deterministic (or meta-heuristic) CIT approach called CASA [43]. These are commonly used CIT approaches with proper tool support [44].

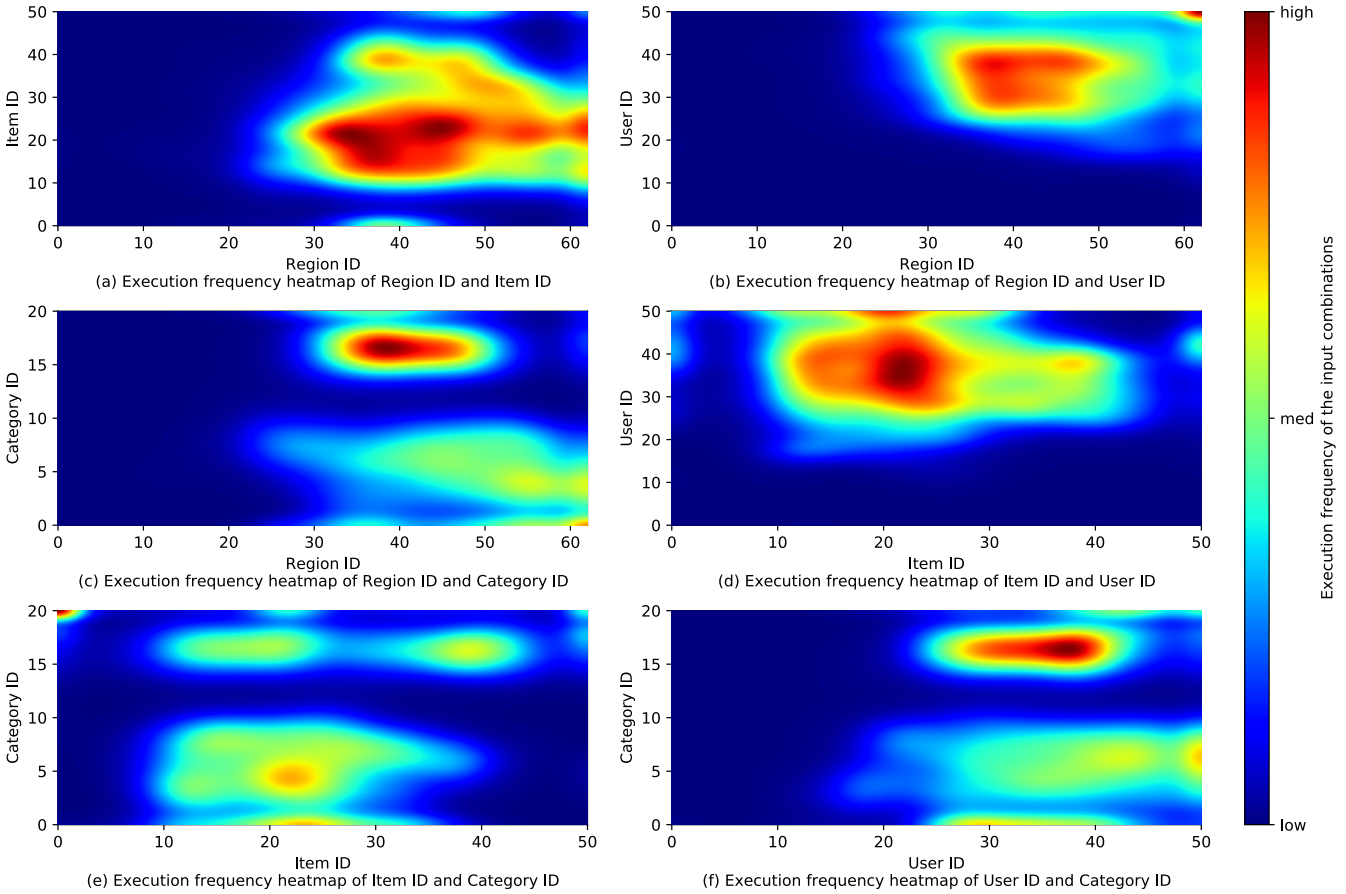

FIGURE 5. Heatmaps built for RUBiS$_{POI}$ with respect to each pair of the input variables (listed in Table 3) by performing a frequency analysis on the input combinations executed by the agent. A redder shade in the subfigures (a) to (f) indicates the more frequently executed input combinations, whereas the least executed input combinations are depicted in a bluer shade.

We ran iPerfXRL, PerfXRL [9], random testing, DGS, and CIT approaches on both RUBiS$_{UNI}$ and RUBiS$_{POI}$. For every experiment, the dependent variable is the number of relevant combinations identified by an approach. We used the same values of the independent variables as defined in Section IV-C for iPerfXRL and PerfXRL [9]. To allow for a comprehensive comparison of iPerfXRL and the alternative approaches, we present the comparison between iPerfXRL and the CIT approaches separately from the rest of the alternative approaches. The reason is that, unlike the other approaches, we cannot generate an arbitrary number of input combinations using the CIT approaches. In addition, different CIT approaches generate different numbers of input combinations, even with the same inputs.

Figures 8 and 9 show the cumulative number of relevant combinations identified by iPerfXRL, PerfXRL [9], DGS, and random testing after executing the 775 000 input combinations (i.e., *TtlSteps*) for RUBiS$_{UNI}$ and RUBiS$_{POI}$, respectively. The solid lines in the figures show the average values, while the shaded regions around the lines represent the standard deviation. The standard deviation of the results using random testing was very low; thus, it is not visible in the figure. For the DGS approach, the standard deviation is zero because the approach is deterministic. It is evident from the figures that iPerfXRL performed better than the rest of the approaches. We have published the results related to iPerfXRL in [25] to allow for future comparisons.

TABLE 4. Number of relevant combinations identified by all four approaches in RUBiS$_{UNI}$ and RUBiS$_{POI}$ after executing 775 000 input combinations (Bold values are the highest.).

Approach	Measurement	RUBiS$_{UNI}$	RUBiS$_{POI}$
iPerfXRL	max	**216 744**	**204 367**
	avg	**204 458.38**	**201 587.38**
	min	**195 910**	**196 868**
PerfXRL	max	112 801	109 907
	avg	100 799.75	105 419.62
	min	89 409	100 397
Random testing	max	67 324	58 855
	avg	67 020.4	58 526.6
	min	66 674	58 231
Deterministic grid search	deterministic	63 352	54 104

Table 4 lists the maximum, average and the minimum number of relevant combinations identified by each of the four approaches after executing 775 000 input combinations. iPerfXRL found on average 2, 3, and 3.2 times more relevant

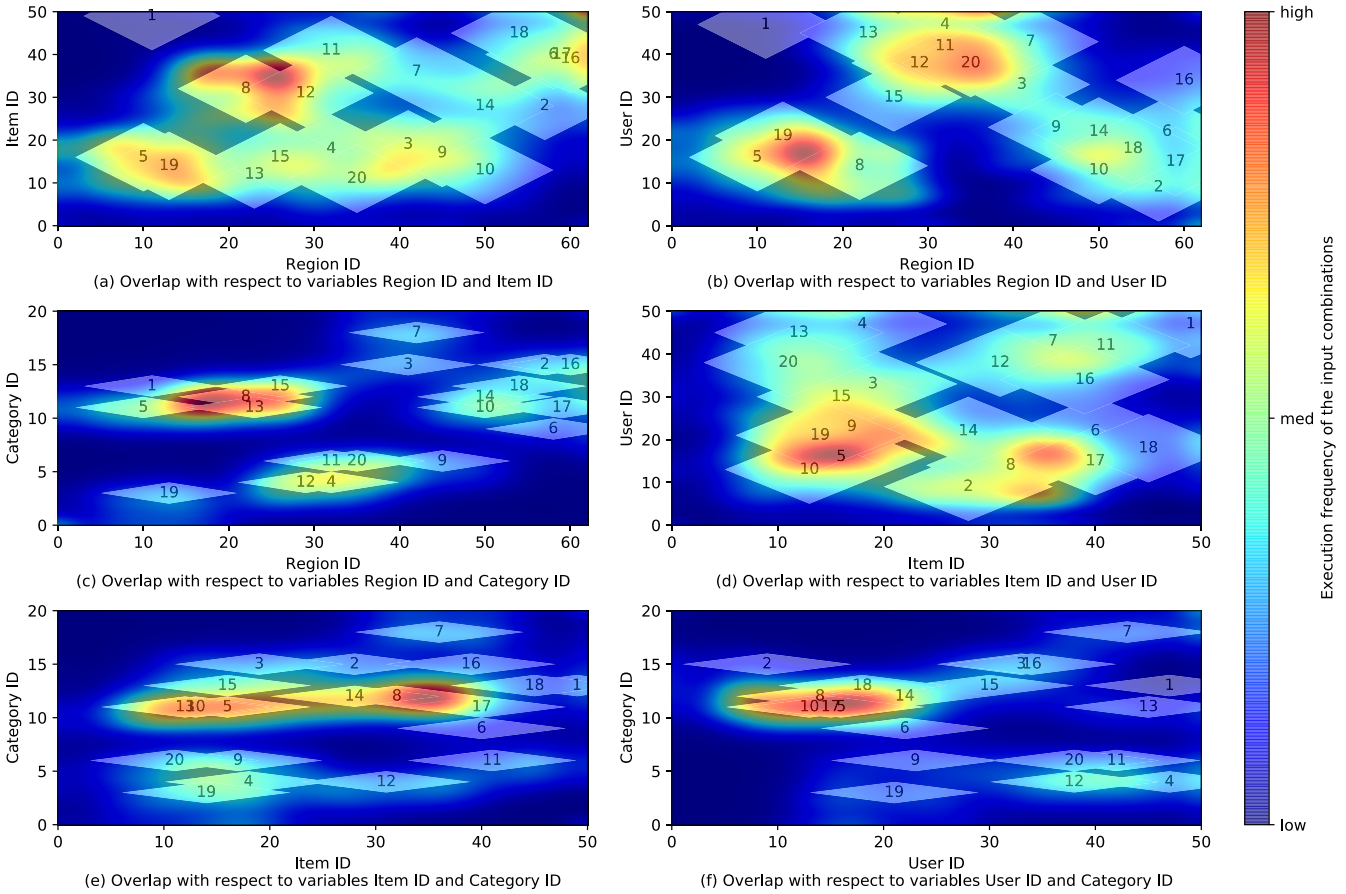

FIGURE 6. Overlap of the injected bottleneck clusters (in Figure 2) and the execution frequency heatmap (in Figure 4).

combinations than PerfXRL [9], random testing, and DGS in RUBiS$_{UNI}$, respectively. In the case of RUBiS$_{POI}$, iPerfXRL identified on average 1.9, 3.4, and 3.7 times more relevant combinations than PerfXRL [9], random testing, and DGS, respectively. In summary, iPerfXRL is better at finding relevant combinations compared to PerfXRL [9], random testing, and DGS.

For the CIT approaches, we set $t = 3$ that is one less than the number of RUBiS input parameters (i.e., four). This is the maximum value of t which we can use in the case of RUBiS because if t is set to the number of input parameters, a CIT approach will test all possible input combinations, i.e., exhaustive testing. To perform a matched comparison of iPerfXRL and the CIT approaches, we ran iPerfXRL on both RUBiS$_{UNI}$ and RUBiS$_{POI}$ for 158 100 input combinations which is the highest number of input combinations generated by one of the CIT approaches (i.e., IPOG-D).

Figures 10 and 11 show the cumulative number of relevant combinations identified by iPerfXRL, IPOG, IPOG-D, IPOG-F, IPOG-F2, and CASA for RUBiS$_{UNI}$ and RUBiS$_{POI}$, respectively. Table 5 lists the maximum, average and minimum number of relevant combinations identified and the number of input combinations executed by each approach. iPerfXRL found approximately 5 and 9 times more relevant combinations on average than the CIT approaches in RUBiS$_{UNI}$ and RUBiS$_{POI}$, respectively. In conclusion, the overall results provide an empirical answer to *RQ2* that iPerfXRL is better at finding relevant combinations compared to the other approaches.

TABLE 5. Number of relevant combinations identified and the number of input combinations executed by iPerfXRL and the CIT approaches in RUBiS$_{UNI}$ and RUBiS$_{POI}$ (Bold values are the highest.).

Approach	Measurement	RUBiS$_{UNI}$	RUBiS$_{POI}$	Input combinations executed
iPerfXRL	max	**73 609**	**110 043**	158 100
	avg	**62 484.88**	**97 945.72**	
	min	**47 340**	**78 847**	
CASA	max	12 661	11 009	156 347
	avg	12 511.25	10 896.71	155 577
	min	12 390	10 873	155 140
IPOG	deterministic	12 419	10 787	155 500
IPOG-D	deterministic	12 801	10 982	158 100
IPOG-F	deterministic	12 403	10 947	155 000
IPOG-F2	deterministic	12 476	10 911	155 000

E. RQ3: SENSITIVITY ANALYSIS OF iPerfXRL

The purpose of the research question *RQ3* was to investigate how different parameters (or independent variables) impact the performance of iPerfXRL in finding relevant

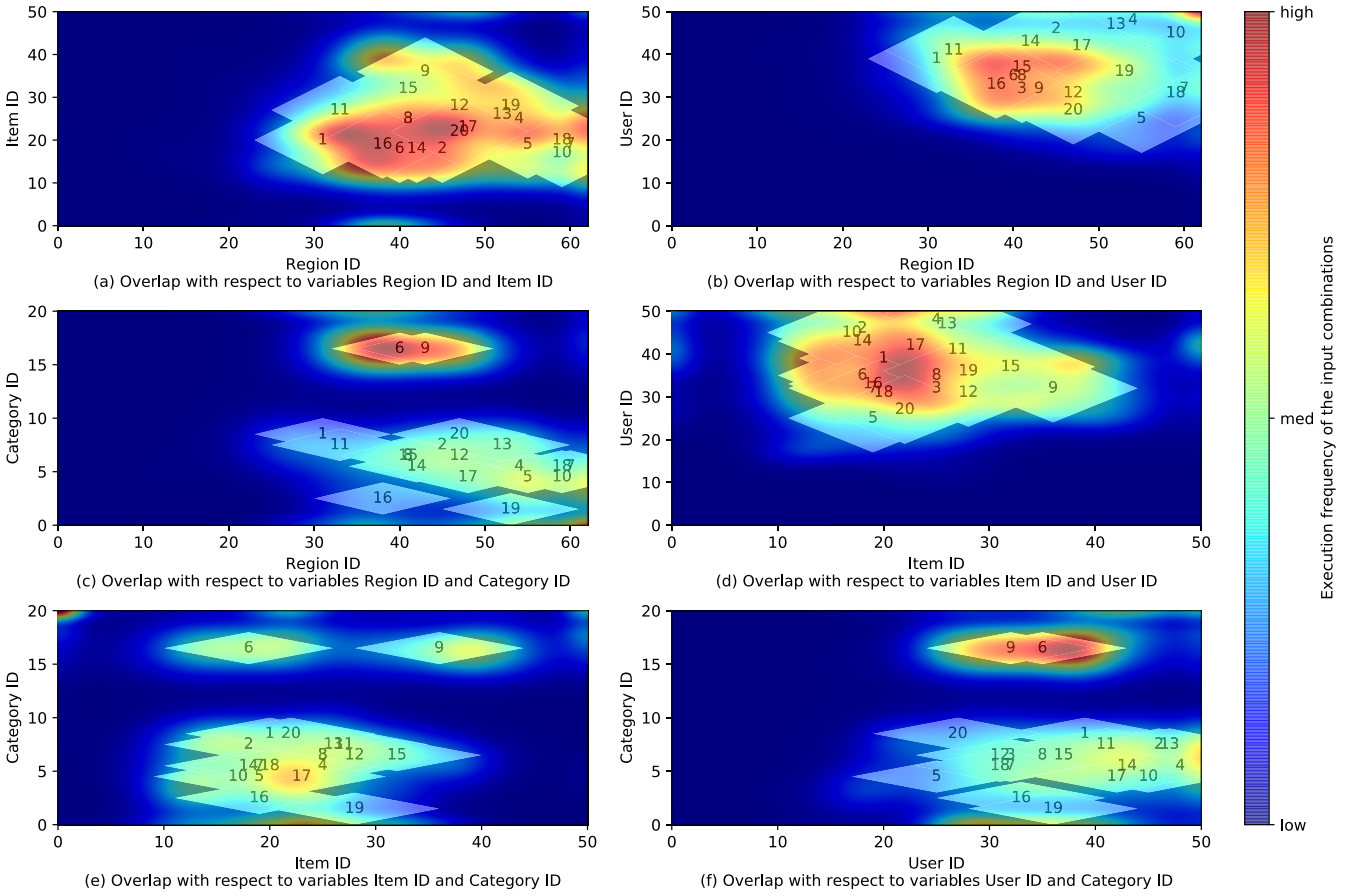

FIGURE 7. Overlap of the injected bottleneck clusters (in Figure 3) and the execution frequency heatmap (in Figure 5).

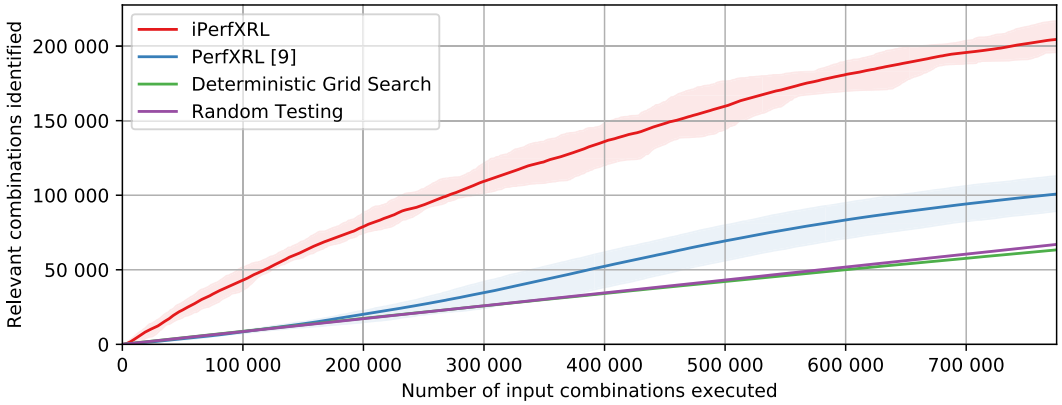

FIGURE 8. Comparison of the cumulative number of relevant combinations found by different approaches for 775 000 data points (input combinations) in RUBiS$_{UNI}$. The solid lines in the figure show the average values, while the shaded regions around the lines represent the standard deviation.

TABLE 6. Parameter values for RQ3.

Parameter	Value
Maximum negative reward per episode (*MaxNegR*)	-10, -100, -5000, -100 000
Neural network designs for the agent (hidden layer configurations)	[32, 32, 32], [64, 32], [64, 64]
Number of steps (*TtlSteps*)	310 000, 465 000, 775 000, 1 085 000

combinations. To answer this question, we conducted an experiment for sensitivity analysis on iPerfXRL. In the sensitivity analysis, we have chosen following independent variables: (1) maximum negative reward per episode (*MaxNegR*) (2) number of steps (*TtlSteps*), and (3) neural network designs for the agent. We conducted several

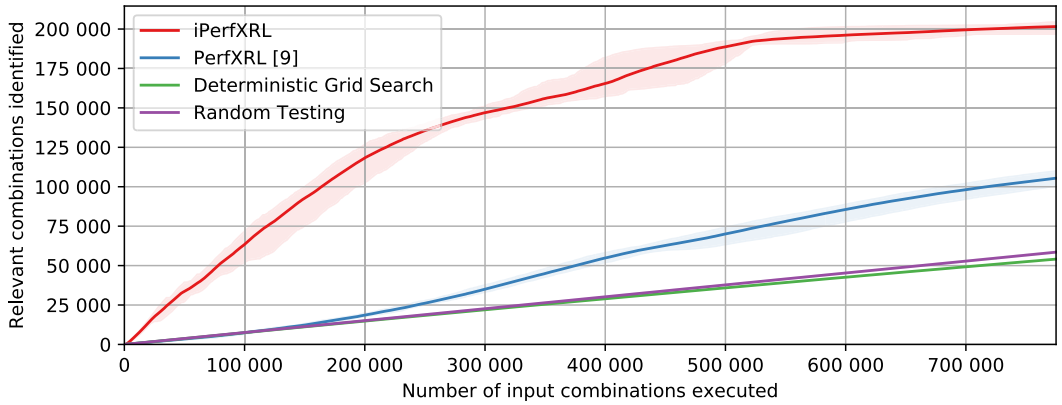

FIGURE 9. Comparison of the cumulative number of relevant combinations found by different approaches for 775 000 data points (input combinations) in RUBiS$_{POI}$. The solid lines in the figure show the average values, while the shaded regions around the lines represent the standard deviation.

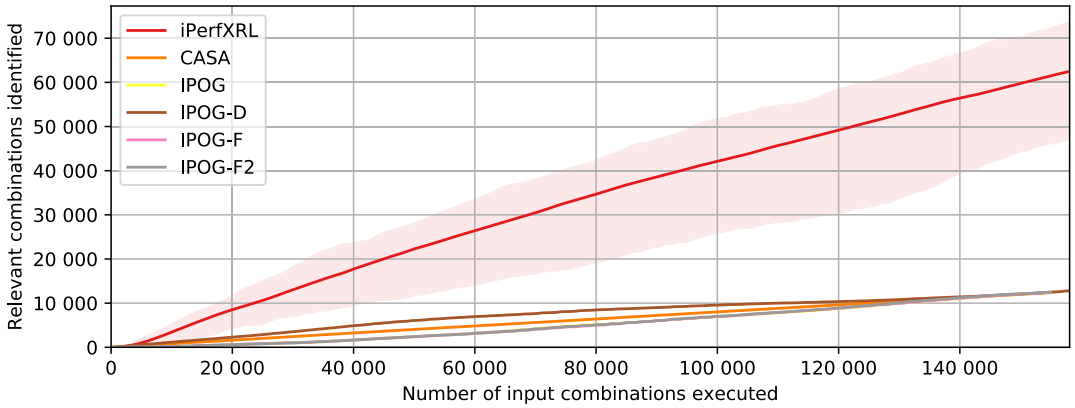

FIGURE 10. Comparison of the cumulative number of relevant combinations found by iPerfXRL and different CIT approaches in RUBiS$_{UNI}$. The solid lines in the figure show the average values, while the shaded regions around the lines represent the standard deviation.

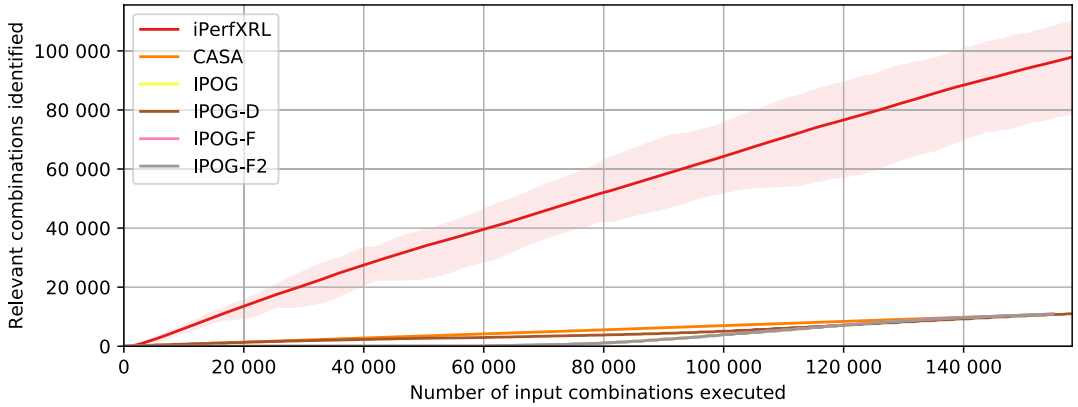

FIGURE 11. Comparison of the cumulative number of relevant combinations found by iPerfXRL and different CIT approaches in RUBiS$_{POI}$. The solid lines in the figure show the average values, while the shaded regions around the lines represent the standard deviation.

experiments where we changed the value of each of the selected parameters, one at a time, while keeping the rest of the independent variables constant as defined in Section IV-C. Table 6 lists the values of selected independent variables. The dependent variable is the number of relevant combinations identified by iPerfXRL. In order to evaluate the statistical significance of the differences in the average number of relevant combinations identified by iPerfXRL with respect

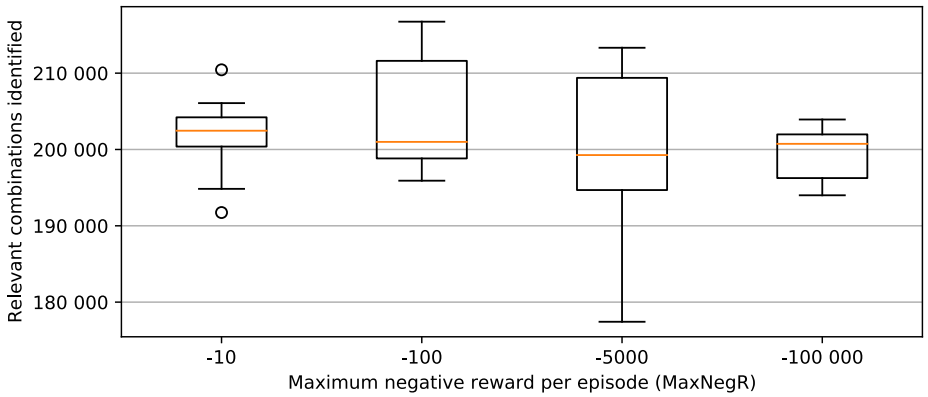

FIGURE 12. Performance of iPerfXRL with respect to the different values of the *MaxNegR* parameter for RUBiS$_{UNI}$.

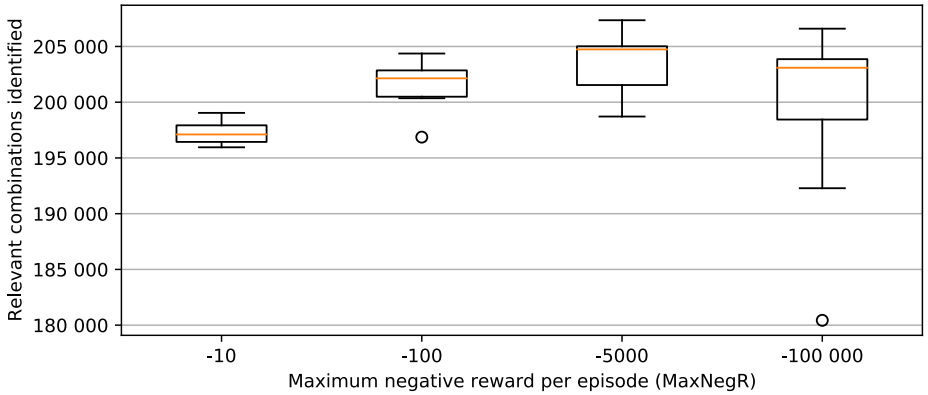

FIGURE 13. Performance of iPerfXRL with respect to the different values of the *MaxNegR* parameter for RUBiS$_{POI}$.

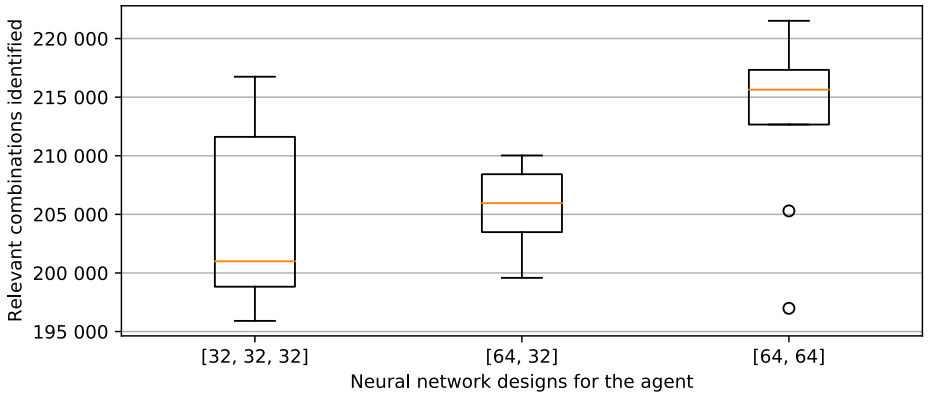

FIGURE 14. Performance of iPerfXRL with respect to the different designs of the neural network for the agent for RUBiS$_{UNI}$.

to different values of a given independent variable, we have introduced the following null hypothesis $H_{0,variable}$ and the alternative hypothesis $H_{A,variable}$ for every selected independent variable:

$H_{0,variable}$**:** There are no statistical differences in the mean numbers of relevant combinations identified by iPerfXRL with respect to different values of a given independent *variable*.

$H_{A,variable}$**:** There are statistical differences in the mean numbers of relevant combinations identified by iPerfXRL with respect to different values of a given independent *variable*.

To test the null hypotheses, we have applied a *t-test* [45] for paired sample means on each pair of values (listed in Table 6) of every selected independent variable. Each t-test examines the differences between the means of both data sets

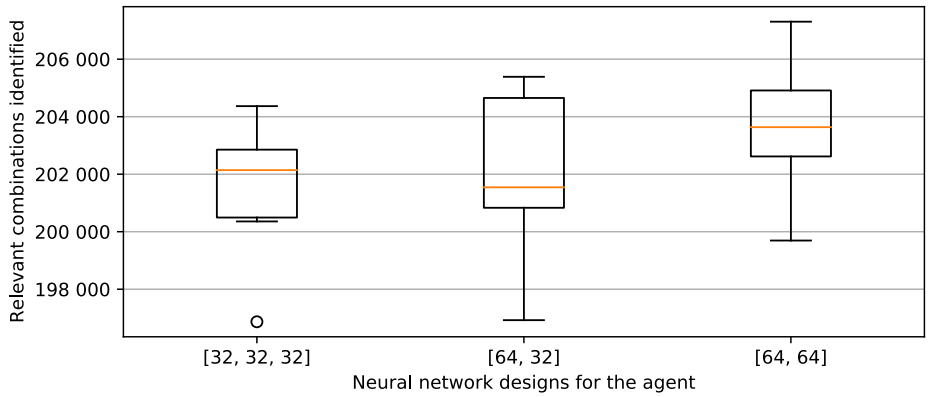

FIGURE 15. Performance of iPerfXRL with respect to the different designs of the neural network for the agent for RUBiS$_{POI}$.

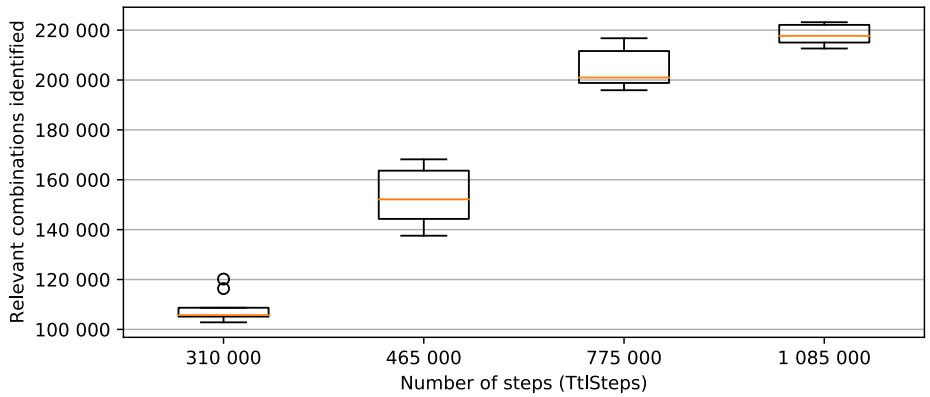

FIGURE 16. Performance of iPerfXRL with respect to the different values of the *TtlSteps* parameter for RUBiS$_{UNI}$.

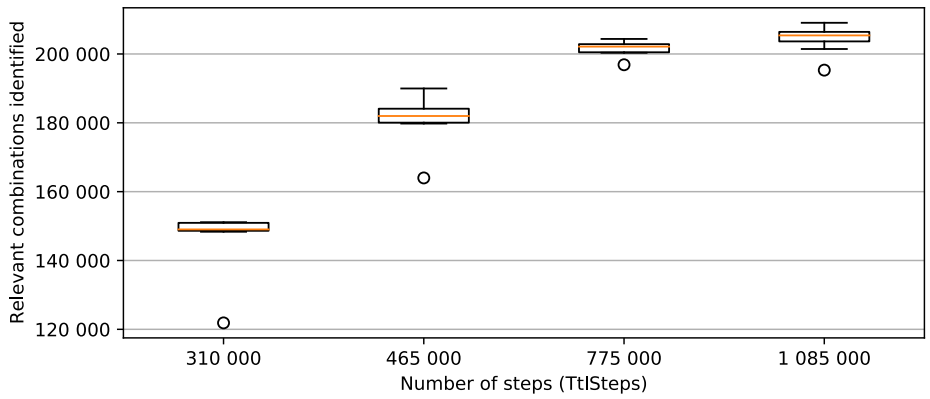

FIGURE 17. Performance of iPerfXRL with respect to the different values of the *TtlSteps* parameter for RUBiS$_{POI}$.

and produces a test statistic that is used to calculate the *p value*. The computed *p* value is compared to the level of significance (α) to decide whether to reject the null hypothesis. The level of significance represents the maximum acceptable probability of rejecting a true null hypothesis [45].

In our first empirical analysis, we only changed the maximum negative reward per episode (*MaxNegR*) parameter to -10, -100, -5000, and $-100\,000$. We selected these values to demonstrate the performance of iPerfXRL when the value of ***MaxNegR*** is either significantly large or small. Figures 12 and 13 show the performance of iPerfXRL when we regulated the value of *MaxNegR* for RUBiS$_{UNI}$ and RUBiS$_{POI}$, respectively. Each boxplot [46] in the figure represents data from 30 experiments. A boxplot expresses four main characteristics of the data: median (line in the box), spread, symmetry and outliers (shown as small circles).

TABLE 7. *p* value of a paired t-test for each pair of selected values of the *MaxNegR* parameter for both RUBiS$_{POI}$ and RUBiS$_{UNI}$. We failed to reject the null hypothesis $H_{0,\text{MaxNegR}}$ for all pairs except for −10, −100 and −10, −5000 for RUBiS$_{POI}$. We have highlighted the cells in the table with the gray color where the null hypothesis $H_{0,\text{MaxNegR}}$ was rejected.

A pair of values for comparison	p value for RUBiS$_{UNI}$	p value for RUBiS$_{POI}$
-10, -100	5.4e-01	6.4e-03
-10, -5000	6.4e-01	1.5e-04
-10, -100 000	4.9e-01	5.0e-01
-100, -5000	2.9e-01	2.7e-01
-100, -100 000	1.9e-01	5.5e-01
-5000, -100 000	9.6e-01	1.4e-01

Table 7 shows the p value of a paired t-test conducted for each pair of the selected values of the *MaxNegR* parameter. In most of the cases, the p values were higher than the level of significance (i.e., $\alpha = 0.05$); thus, we failed to reject the null hypothesis $H_{0,\text{MaxNegR}}$ that different values of the *MaxNegR* parameter do not affect the average number of relevant combinations identified by iPerfXRL. However, the p values were less than 0.05 for the following pairs: -10, −100 and −10, −5000 for RUBiS$_{POI}$. Based on those p values, we can reject the null hypothesis $H_{0,\text{MaxNegR}}$ and accept the alternative hypothesis $H_{A,\text{MaxNegR}}$ that the average numbers of relevant combination identified by iPerfXRL are different with respect to the following values of the *MaxNegR* parameter: −10, −100 and −5000. iPerfXRL found, on average, 4317 and 6353.38 more relevant combinations when the value of *MaxNegR* was set to −100 and −5000 as compared to −10, respectively. As we have discussed in Section III-D when the value of *MaxNegR* gets close to zero, the episodes become shorter, and the environment resets frequently. This leads to reduced bottleneck detection rate because the agent cannot adequately explore the regions of the input space.

In our second empirical analysis, we tried different neural network designs for the agent by altering the number of hidden layers and the number of hidden units in each layer. Table 8 shows the p value of a paired t-test conducted for each pair of the selected neural network designs of the agent. In the majority of the paired t-tests, we were unable to reject the null hypothesis $H_{0,\text{network}}$ that different neural network designs of the agent do not affect the average number of relevant combinations identified by iPerfXRL. The null hypothesis $H_{0,\text{network}}$ was rejected in favor of the alternative hypothesis $H_{A,\text{network}}$ (i.e., different neural network designs of the agent affect the average number of relevant combinations identified by iPerfXRL) only for the following pairs of neural network designs: [32, 32, 32], [64, 64] and [64, 32], [64, 64] for RUBiS$_{UNI}$. The agent with two hidden layers with 64 units (i.e., [64, 64]) found, on average, 6569.76 and 7899.64 more relevant combinations than the agents with the [32, 32, 32] and [64, 32] neural network design for RUBiS$_{UNI}$, respectively. Figures 14 and 15 show the performance distribution of iPerfXRL with respect to different neural network designs of the agent for RUBiS$_{UNI}$ and RUBiS$_{POI}$, respectively.

In our third empirical analysis, we changed the number of steps (*TtlSteps*) to 310 000, 465 000, 775 000, and 1 085 000.

TABLE 8. *p* value of a paired t-test for each pair of selected neural network designs of the agent for both RUBiS$_{POI}$ and RUBiS$_{UNI}$. We failed to reject the null hypothesis $H_{0,\text{network}}$ for all pairs except for [32, 32, 32], [64, 64] and [64, 32], [64, 64] for RUBiS$_{UNI}$. We have highlighted the cells in the table with the gray color where the null hypothesis $H_{0,\text{network}}$ was rejected.

A pair of neural network designs for comparison	p value for RUBiS$_{UNI}$	p value for RUBiS$_{POI}$
[32, 32, 32], [64, 32]	4.8e-01	4.1e-01
[32, 32, 32], [64, 64]	1.5e-03	1.1e-01
[64, 32], [64, 64]	1.3e-04	2.9e-01

TABLE 9. *p* value of a paired t-test for each pair of selected values of the *TtlSteps* parameter for both RUBiS$_{POI}$ and RUBiS$_{UNI}$. The null hypothesis $H_{0,\text{TtlSteps}}$ was rejected for all pairs except for 775 000, 1 085 000 for RUBiS$_{POI}$. We have highlighted the cells in the table with the gray color where the null hypothesis $H_{0,\text{TtlSteps}}$ was rejected.

A pair of values for comparison	p value for RUBiS$_{UNI}$	p value for RUBiS$_{POI}$
310 000, 465 000	3.7e-05	1.4e-04
310 000, 775 000	4.3e-08	2.0e-06
310 000, 1 085 000	8.7e-10	5.7e-08
465 000, 775 000	2.1e-05	1.4e-04
465 000, 1 085 000	5.8e-06	1.8e-04
775 000, 1 085 000	9.1e-04	1.9e-01

As expected, Figures 16 and 17 show that iPerfXRL identifies more relevant combinations as we increase the number of steps in RUBiS$_{UNI}$ and RUBiS$_{POI}$, respectively. Table 9 shows the p value of a paired t-test conducted for each pair of the selected values of the *TtlSteps* parameter. The null hypothesis $H_{0,\text{TtlSteps}}$ was rejected for all pairs except for 775 000, 1 085 000 for RUBiS$_{POI}$. This means that no statistical differences were found in the mean number of relevant combinations identified by iPerfXRL when we increased the value of the *TtlSteps* parameter from 775 000 to 1 085 000 for RUBiS$_{POI}$. More specifically, iPerfXRL found 2657.5 more relevant combinations, on average, as we change the value of *TtlSteps* from 775 000 to 1 085 000 for RUBiS$_{POI}$. As we have mentioned before, the relevant combinations are tightly packed together in the input space of RUBiS$_{POI}$ and the agent gets a reward only once per a relevant combination. Therefore, it becomes increasingly difficult for the agent to find the hitherto undiscovered relevant combinations incorporated among the already identified relevant combinations.

V. RELATED WORK

Performance testing is a well-investigated research topic. In this section, we focus on some of the most important and recent related works on performance testing that use machine learning and other state-of-the-art approaches such as CIT approaches. We also review some related works that use machine learning for non-functional testing, a performance analysis framework, and a symbolic execution approach for generating performance distributions for a program under test.

Many CIT approaches [38], [40], [41], [43], [44] have been proposed to generate a test suite that tests all t-way interactions among the input parameters while keeping the

size of the test suite as small as possible. A CIT approach tests the subset of the input space by limiting the degree of interactions between the values of its input parameters. In contrast, iPerfXRL is aimed at efficiently exploring all possible interactions among the input parameters. It learns and recognizes the relevant regions of the input space and concentrates the search on those regions to find input combinations that trigger performance bottlenecks.

PerfFuzz [5] is another performance testing approach. It uses mutational fuzzing to find program inputs that can reveal worst-case algorithmic complexity in different parts of the program under test. PerfFuzz begins the test generation process with a set of randomly generated inputs. Then in each iteration, it generates new inputs by mutating the previous inputs and saving the ones that increase code coverage. In comparison to PerfFuzz, iPerfXRL does not depend on the source code of the SUT and explores the input space of the SUT by using DRL.

Kim *et al.* [47] presented a search-based software testing framework called GunPowder. The framework employs an RL algorithm to learn heuristics for search-based test data generation. To that extent, they trained and tested a Double Deep Q-Network [48] agent for branch coverage in functional white-box testing. Our proposed approach uses the SAC algorithm for exploratory performance testing. Our agent does not rely on the source code of the SUT and learns to concentrate the search on the most important subsets of the input space.

Mariani *et al.* [49] presented an RL approach for black-box testing of interactive applications called AutoBlackTest. The approach focuses on graphical user interfaces and it tests system-level user interactions. It uses Q-learning to learn the user interaction patterns with the application under test. It builds a model of the event sequences, and then uses the model to produce test cases. In comparison to AutoBlackTest, iPerfXRL uses DRL to generate performance tests instead of usability tests.

Koren *et al.* [50] used DRL for adaptive stress testing of autonomous vehicles aimed at finding some problematic self-driving scenarios which may lead to a collision with a moving pedestrian. Their work is similar to our previous work in [51] that finds the worst sequences of actions to maximize the resource utilization on the SUT. In contrast, iPerfXRL finds concrete input combinations to reveal performance bottlenecks in the SUT and identifies the most problematic regions of the input space.

PerfPlotter [52] provides a performance analysis framework and a symbolic execution approach for generating performance distributions. It requires the source code of the program under test along with the usage profiles of the program and uses probabilistic symbolic execution to traverse different high-probability and low-probability execution paths in the program and to generate performance distributions. The main drawback of PerfPlotter is that it does not provide a performance testing approach.

Briand *et al.* [53] used a genetic algorithm to generate tests for deadline-constrained tasks in real-time systems. The generated tests stress the system in such a way that some critical tasks miss (or nearly-miss) their estimated deadlines. In contrast, iPerfXRL does not rely on estimated deadlines. It uses DRL to reveal actual performance bottlenecks in the SUT and to identify the most problematic regions of the input space.

Xiao *et al.* [54] presented a delta inference approach to predict performance bottlenecks in graphical user interfaces. They used complexity models of multiple workloads to predict iteration counts of certain loops in the source code. In contrast, iPerfXRL does not rely on the source code of the SUT to find performance bottlenecks in the entire system, not just the graphical user interfaces.

GA-Prof [55] performs search-based application profiling to detect performance bottlenecks in the Application Under Test (AUT). It uses a genetic algorithm to guide the search process and relies on the source code of the AUT to map test inputs to different methods in the source code and then to relate the methods to different performance bottlenecks. FOREPOST [56] is a performance testing approach. It uses feedback-oriented machine learning to find performance problems. The main idea is to use a rule learning algorithm, which extracts a set of rules that map the application performance to certain input combinations. Although GA-Prof and FOREPOST seem promising and similar to iPerfXRL, there are three fundamental differences between these approaches and iPerfXRL. Firstly, these approaches look for a particular input combination which increases the elapsed execution time. In contrast, we are looking for all input combinations which increase the elapsed execution time. Secondly, the authors do not specify the input space (i.e., input parameters and their ranges) used for evaluation. Thirdly, the approaches rely on execution traces containing information about the method calls, the total number of invocations, and the total elapsed self-time for each method. In contrast, our approach is designed to work with black-box systems where we do not have access to this kind of information. Therefore, it is difficult to draw a comparison with our proposed approach.

To summarize, although performance testing is a widely-investigated research topic, none of the existing performance testing approaches efficiently explores only a small subset of multi-dimensional large input spaces without any prior domain or application knowledge and finds a near-complete set of relevant combinations that trigger performance bottlenecks.

VI. THREATS TO VALIDITY

The first threat to the internal validity of the experiments is that we randomly injected artificial bottlenecks into the subject application. Thus, there is a threat that we may achieve different results by running our approach against a system with real bottlenecks. However, by following this experimental design, we managed to evaluate iPerfXRL in a controlled setting effectively.

iPerfXRL, like many other machine learning approaches, is sensitive to its parameters, for instance, a suitable set of

parameter values for one problem environment might not work well for others. For the SAC algorithm, we selected the parameter values according to the practical experiences reported by the authors of the algorithm [16]. Since our approach does not require access to the source code of the SUT nor domain knowledge, in a real-world setting, one only needs to adjust few parameters (e.g., the performance threshold $\mathcal{L}$) for a new environment or SUT. Moreover, we have conducted an empirical study in Section IV-E to investigate how different parameter values affect the performance of iPerfXRL in finding performance bottlenecks.

The main threat to external validity is that we used only one SUT in our evaluation. Therefore, the results of our experiment may differ for systems that have different architectures or different input spaces. To mitigate this threat, we have evaluated our approach against eight alternative approaches on two bottleneck distributions and have shown that our approach has outperformed the alternative approaches. This threat can be reduced further by performing additional experiments using different applications. However, to the best of our knowledge, there are no publicly available performance benchmark applications which closely represent the real-world applications.

Another threat is the selection of URLs listed in Section IV-A. The only reason we have chosen those URLs is that we are focusing on numerical input parameters in this work. Nevertheless, our approach is agnostic to input parameter types and can be applied to different types of inputs (e.g., string) by updating the action and input spaces accordingly.

A final threat to external validity is that we have compared iPerfXRL with PerfXRL [9], DGS, random testing, and CIT approaches in order to evaluate the efficiency of iPerfXRL. The evaluation might seem subjective because we have not compared iPerfXRL with other performance exploration approaches from the literature. However, as we have discussed in Section V, we could not find any approach similar to iPerfXRL that finds performance bottleneck in a black-box system by exploring only a subset of the input space of the SUT. Further, random testing provides a good unbiased benchmark because it has proved to be more effective than many other systematic approaches, especially when we test a black-box SUT [27]–[29].

VII. CONCLUSION

We presented iPerfXRL, a novel approach to find performance bottlenecks in a software system without any prior domain knowledge using deep reinforcement learning. We can apply our approach to any software system where we can interactively execute different input combinations while monitoring their performance impact on the SUT. In our approach, a learning agent non-exhaustively explores a multi-dimensional large input space and finds relevant combinations (i.e., input combinations which are most likely to trigger performance bottlenecks in the system under test). The results presented in this paper show that iPerfXRL managed to find up to 9 times more relevant combinations than the alternative approaches. Moreover, Our results show that iPerfXRL can be used to identify the regions of the input space which contain relevant combinations. For our future work, we aim to integrate various KPIs such as CPU and memory usage into our reward function. Further, we plan to extend our approach to identify potential root causes of the bottlenecks.

ACKNOWLEDGMENT

This work has received funding from the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement number 737494. This Joint Undertaking receives support from the European Unions Horizon 2020 research and innovation programme and Sweden, France,Spain, Italy, Finland, the Czech Republic.

REFERENCES

[1] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in *Proc. 33rd ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, New York, NY, USA, 2012, pp. 77–88, doi: 10.1145/2254064.2254075.

[2] E. J. Weyuker and F. I. Vokolos, "Experience with performance testing of software systems: Issues, an approach, and case study," *IEEE Trans. Softw. Eng.*, vol. 26, no. 12, pp. 1147–1156, 2000.

[3] I. Molyneaux, *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. Newton, MA, USA: O'Reilly, 2009. [Online]. Available: http://www.oreilly.de/catalog/9780596520663/index.html

[4] B. M. Subraya and S. V. Subrahmanya, "Object driven performance testing of Web applications," in *Proc. 1st Asia–Pacific Conf. Qual. Softw.*, 2000, pp. 17–26.

[5] C. Lemieux, R. Padhye, K. Sen, and D. Song, "PerfFuzz: Automatically generating pathological inputs," in *Proc. 27th ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA)*, New York, NY, USA, 2018, pp. 254–265, doi: 10.1145/3213846.3213874.

[6] P. Zhang, S. Elbaum, and M. B. Dwyer, "Automatic generation of load tests," in *Proc. 26th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Washington, DC, USA: IEEE Computer Society, Nov. 2011, pp. 43–52, doi: 10.1109/ASE.2011.6100093.

[7] P. Bourque, R. E. Fairley, and I. C. Society, *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*, 3rd ed. Washington, DC, USA: IEEE Computer Society Press, 2014.

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[9] T. Ahmad, A. Ashraf, D. Truscan, and I. Porres, "Exploratory performance testing using reinforcement learning," in *Proc. 45th Euromicro Conf. Softw. Eng. Adv. Appl. (SEAA)*, Aug. 2019, pp. 156–163.

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

[11] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 1998.

[12] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *Proc. Int. Conf. Mach. Learn. (ICML)*, 2018, pp. 1–14.

[13] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org

[14] Y. Bengio, "Deep learning of representations for unsupervised and transfer learning," in *Proc. Workshop Unsupervised Transf. Learn.*, vol. 27, I. Guyon, G. Dror, V. Lemaire, G. Taylor, and D. Silver, Eds. Bellevue, WA, USA: PMLR, Jul. 2012, pp. 17–36. [Online]. Available: http://proceedings.mlr.press/v27/bengio12a.html

[15] L.-J. Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Mach. Learn.*, vol. 8, nos. 3–4, pp. 293–321, May 1992, doi: 10.1007/BF00992699.

[16] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine, "Soft actor-critic algorithms and applications," 2018, *arXiv:1812.05905*. [Online]. Available: http://arxiv.org/abs/1812.05905

[17] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, "Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software," in *Proc. 10th Joint Meeting Found. Softw. Eng. (ESEC/FSE)*, New York, NY, USA: Association for Computing Machinery, 2015, pp. 307–319, doi: 10.1145/2786805.2786852.

[18] R. S. Barr, B. L. Golden, J. P. Kelly, M. G. C. Resende, and W. R. Stewart, "Designing and reporting on computational experiments with heuristic methods," *J. Heuristics*, vol. 1, no. 1, pp. 9–32, Sep. 1995, doi: 10.1007/BF02430363.

[19] Amza, Chanda, Cox, Elnikety, Gil, Rajamani, Zwaenepoel, Cecchet, and Marguerite, "Specification and implementation of dynamic Web site benchmarks," in *Proc. IEEE Int. Workshop Workload Characterization*, Nov. 2002, pp. 3–13.

[20] Google Scholar. *RUBiS Citations*. Accessed: Apr. 2, 2020. [Online]. Available: http://scholar.google.com/scholar?q=Specification+and+Implementation+of+Dynamic+Web+Site+Benchmarks

[21] Apache. *HTTP Sever Project*. Accessed: Apr. 2, 2020. [Online]. Available: https://httpd.apache.org/

[22] PHP. *Scripting Language*. Accessed: Apr. 2, 2020. [Online]. Available: https://www.php.net/

[23] MySQL. *Sever*. Accessed: Apr. 2, 2020. [Online]. Available: https://www.mysql.com/products/community/

[24] M. Evans, N. Hastings, and B. Peacock, *Statistical Distributions*, 3rd ed. Hoboken, NJ, USA: Wiley, Jun. 2000. [Online]. Available: http://www.worldcat.org/isbn/0471371246

[25] T. Ahmad. (2020). *Data Related to iPerfXRL, RUBiS(uni), and RUBiS(poi)*. [Online]. Available: https://gitlab.abo.fi/tahmad/iperfxrl-data

[26] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, *arXiv:1412.6980*. [Online]. Available: http://arxiv.org/abs/1412.6980

[27] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "Experimental assessment of random testing for object-oriented software," in *Proc. Int. Symp. Softw. Test. Anal. (ISSTA)*, New York, NY, USA, 2007, pp. 84–94, doi: 10.1145/1273463.1273476.

[28] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," *IEEE Trans. Softw. Eng.*, vols. SE–10, no. 4, pp. 438–444, Jul. 1984.

[29] D. Hamlet, "When only random testing will do," in *Proc. 1st Int. Workshop Random Test. (RT)*, New York, NY, USA, 2006, pp. 1–9, doi: 10.1145/1145735.1145737.

[30] H. Niederreiter, *Random Number Generation and Quasi-monte Carlo Methods*. Philadelphia, PA, USA: SIAM, 1992. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611970081

[31] S. Kucherenko and Y. Sytsko, "Application of deterministic low-discrepancy sequences in global optimization," *Comput. Optim. Appl.*, vol. 30, no. 3, pp. 297–318, Mar. 2005, doi: 10.1007/s10589-005-4615-1.

[32] J. H. Halton, "On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals," *Numerische Math.*, vol. 2, no. 1, pp. 84–90, Dec. 1960.

[33] I. M. Sobol, "On the distribution of points in a cube and the approximate evaluation of integrals," *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki*, vol. 7, no. 4, pp. 784–802, 1967.

[34] C. Haselgrove, "A method for numerical integration," *Math. Comput.*, vol. 15, no. 76, pp. 323–337, 1961.

[35] S. Galanti and A. Jung, "Low-discrepancy sequences: Monte Carlo simulation of option prices," *J. Derivatives*, vol. 5, no. 1, p. 63, 1997.

[36] J. Feinberg and H. P. Langtangen, "Chaospy: An open source tool for designing methods of uncertainty quantification," *J. Comput. Sci.*, vol. 11, pp. 46–57, Nov. 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877750315300119

[37] M. Grindal, J. Offutt, and S. F. Andler, "Combination testing strategies: A survey," *Softw. Test., Verification Rel.*, vol. 15, no. 3, pp. 167–199, 2005. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.319

[38] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG system: An approach to testing based on combinatorial design," *IEEE Trans. Softw. Eng.*, vol. 23, no. 7, pp. 437–444, Jul. 1997.

[39] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, "Software fault interactions and implications for software testing," *IEEE Trans. Softw. Eng.*, vol. 30, no. 6, pp. 418–421, Jun. 2004.

[40] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG/IPOG-D: Efficient test generation for multi-way combinatorial testing," *Softw. Test., Verification Rel.*, vol. 18, no. 3, pp. 125–148, Sep. 2008. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.381

[41] M. Forbes, J. Lawrence, Y. Lei, R. N. Kacker, and D. R. Kuhn, "Refining the in-parameter-order strategy for constructing covering arrays," *J. Res. Nat. Inst. Standards Technol.*, vol. 113, no. 5, pp. 287–297, 2008, doi: 10.6028/jres.113.022.

[42] R. Kuhn, Y. Lei, and R. Kacker, "Practical combinatorial testing: Beyond pairwise," *IT Prof.*, vol. 10, no. 3, pp. 19–23, May 2008.

[43] B. J. Garvin, M. B. Cohen, and M. B. Dwyer, "Evaluating improvements to a meta-heuristic search for constrained interaction testing," *Empirical Softw. Eng.*, vol. 16, no. 1, pp. 61–102, Feb. 2011, doi: 10.1007/s10664-010-9135-7.

[44] J. Petke, M. B. Cohen, M. Harman, and S. Yoo, "Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection," *IEEE Trans. Softw. Eng.*, vol. 41, no. 9, pp. 901–924, Sep. 2015.

[45] R. Freund, D. Mohr, and W. Wilson, *Statistical Methods*. Amsterdam, The Netherlands: Elsevier, 2010. [Online]. Available: https://www.sciencedirect.com/book/9780123749703/statistical-methods

[46] R. A. Reese, "Boxplots," *Significance*, vol. 2, no. 3, pp. 134–135, 2005.

[47] J. Kim, M. Kwon, and S. Yoo, "Generating test input with deep reinforcement learning," in *Proc. 11th Int. Workshop Search-Based Softw. Test. (SBST)*, New York, NY, USA, 2018, pp. 51–58, doi: 10.1145/3194718.3194720.

[48] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double Q-learning," in *Proc. AAAI*, vol. 2, Phoenix, AZ, USA, 2016, p. 5.

[49] L. Mariani, M. Pezze, O. Riganelli, and M. Santoro, "AutoBlackTest: Automatic black-box testing of interactive applications," in *Proc. IEEE 5th Int. Conf. Softw. Test., Verification Validation*, Apr. 2012, pp. 81–90.

[50] M. Koren, S. Alsaif, R. Lee, and M. J. Kochenderfer, "Adaptive stress testing for autonomous vehicles," in *Proc. IEEE Intell. Vehicles Symp. (IV)*, Jun. 2018, pp. 1898–1904.

[51] T. Ahmad, D. Truscan, and I. Porres, "Identifying worst-case user scenarios for performance testing of Web applications using Markov-chain workload models," *Future Gener. Comput. Syst.*, vol. 87, pp. 910–920, Oct. 2018.

[52] B. Chen, Y. Liu, and W. Le, "Generating performance distributions via probabilistic symbolic execution," in *Proc. 38th Int. Conf. Softw. Eng. (ICSE)*, New York, NY, USA, 2016, pp. 49–60.

[53] L. C. Briand, Y. Labiche, and M. Shousha, "Stress testing real-time systems with genetic algorithms," in *Proc. Conf. Genetic Evol. Comput. (GECCO)*, New York, NY, USA, 2005, pp. 1021–1028, doi: 10.1145/1068009.1068183.

[54] X. Xiao, S. Han, D. Zhang, and T. Xie, "Context-sensitive delta inference for identifying workload-dependent performance bottlenecks," in *Proc. Int. Symp. Softw. Test. Anal. (ISSTA)*, M. Pezzè and M. Harman, Eds., Lugano, Switzerland, Jul. 2013, pp. 90–100.

[55] D. Shen, Q. Luo, D. Poshyvanyk, and M. Grechanik, "Automating performance bottleneck detection using search-based application profiling," in *Proc. Int. Symp. Softw. Test. Anal. (ISSTA)*, New York, NY, USA, 2015, pp. 270–281.

[56] Q. Luo, A. Nair, M. Grechanik, and D. Poshyvanyk, "FOREPOST: Finding performance problems automatically with feedback-directed learning software testing," *Empirical Softw. Eng.*, vol. 22, no. 1, pp. 6–56, Feb. 2017, doi: 10.1007/s10664-015-9413-5.

TANWIR AHMAD received the master's degree in computer science from Government College University, Pakistan, in 2011, and the master's degree in software engineering from Åbo Akademi University, Finland, in 2014, where he is currently pursuing the Ph.D. degree. His main research interests include developing and investigating the different methodologies for performance exploration, testing of software systems using machine learning, and evolutionary algorithms.

ADNAN ASHRAF received the M.Sc. and M.S. degrees in computer science from Mohammad Ali Jinnah University, Islamabad, Pakistan, in 2003 and 2006, respectively, and the Ph.D. degree in software engineering from Åbo Akademi University, Finland, in 2014. From 2005 to 2010, he was a Full-Time University Lecturer, Pakistan. He is currently a Research Associate and a Lecturer with Åbo Akademi University. His research interests include cloud computing, autonomous systems, search-based software engineering, and software testing.

DRAGOS TRUSCAN is currently a Senior Lecturer in software engineering with Åbo Akademi University. His main emphasis of work was on deploying model-based testing techniques to industrial settings. He has participated in several industry-driven national and European level projects. His research interests include model-based techniques for testing functional and non-functional properties of software intensive systems.

ANDI DOMI received the B.S. and M.S. degrees in finance and accounting from the University of Elbasan, Albania, the B.S. degree in information technology, and the M.S. degree in software engineering from Åbo Akademi University, Finland. From 2018 to 2019, he was a Research Assistant with Åbo Akademi University. His research interests include developing and gathering of performance testing requirements, developing automated test scenarios and environments for performance testing, and analysing existing tools and techniques to improve testing efficiency.

IVAN PORRES is currently a Professor in software engineering with Åbo Akademi University. He has published more than 100 scientific articles in software engineering. His current research interests include design, verification, validation of mission critical software-intensive systems, algorithms and applications for search-based software engineering, using approximate algorithms and heuristics based on genetic algorithms, ant colony optimisation, and more recently machine learning.