Search-Based Testing of Reinforcement Learning

Martin Tappler[1,3] * , **Filip Cano Córdoba**[2] * , **Bernhard K. Aichernig**[1,3] , **Bettina Könighofer**[2,4] *

[1]Institute of Software Technology, Graz University of Technology

[2]Institute of Applied Information Processing and Communications, Graz University of Technology

[3]TU Graz-SAL DES Lab,Silicon Austria Labs, Graz, Austria

[4]Lamarr Security Research

martin.tappler@ist.tugraz.at, filip.cano@iaik.tugraz.at, aichernig@ist.tugraz.at,
bettina.koenighofer@lamarr.at

Abstract

Evaluation of deep reinforcement learning (RL) is inherently challenging. Especially the opaqueness of learned policies and the stochastic nature of both agents and environments make testing the behavior of deep RL agents difficult. We present a search-based testing framework that enables a wide range of novel analysis capabilities for evaluating the safety and performance of deep RL agents. For *safety testing*, our framework utilizes a search algorithm that searches for a reference trace that solves the RL task. The backtracking states of the search, called *boundary states*, pose safety-critical situations. We create safety test-suites that evaluate how well the RL agent escapes safety-critical situations near these boundary states. For *robust performance testing*, we create a diverse set of traces via fuzz testing. These fuzz traces are used to bring the agent into a wide variety of potentially unknown states from which the average performance of the agent is compared to the average performance of the fuzz traces. We apply our search-based testing approach on RL for Nintendo's Super Mario Bros.

1 Introduction

In reinforcement learning (RL) [Sutton and Barto, 1998], an agent aims to maximize the total amount of reward through trial-and-error via interactions with an unknown environment. Recently, RL algorithms achieved stunning results in playing video games and complex board games [Schrittwieser *et al.*, 2020].

To achieve a broad acceptance and enlarge the application areas of learned controllers, there is the urgent need to reliably evaluate trained RL agents. When evaluating trained agents, two fundamental questions need to be answered: (Q1) *Does the trained deep RL agent circumvent safety violations?* (Q2) *Does the trained deep RL agent perform well from a wide variety of states?* Testing deep RL agents is notoriously difficult. The first challenge arises from the environment, which is is often not fully known and has an immense state space, combined with the byzantine complexity of the

agent's model, and the lack of determinism of both the agent and the environment. Secondly, to evaluate the performance of a trained agent's policy, an estimation of the performance of the optimal policy is needed.

To address these challenges, we transfer well-established search-based concepts from software testing into the RL-setting. *Search algorithms* like backtracking-based depth-first search (DFS) are standard to find valid and invalid program executions. *Fuzz testing* refers to automated software testing techniques that generate interesting test cases with the goal to expose corner cases that have not been properly dealt with in the program under test. In this work, we propose a search-based testing framework to reliably evaluate trained RL agents to answer the questions Q1 and Q2. Our testing framework comprises four steps:

Step 1: Search for reference trace and boundary states. In the first step, we use a DFS algorithm to search for a *reference trace* that solves the RL task by sampling the black-box environment. This idea is motivated by the experience from AI competitions, like the Mario AI and the NetHack Challenges [Karakovskiy and Togelius, 2012; Küttler *et al.*, 2020], where best performers are symbolic agents, providing a reference solution of the task faster than neural-based agents. Furthermore, since the DFS algorithm backtracks when reaching an unsafe state in the environment, the search reveals safety-critical situations that we call *boundary states*.

Step 2: Testing for safety. To answer Q1, our testing framework computes *safety test-suites* that bring the agent into safety-critical situations near the boundary states. Based on the ability of the agent to succeed in these safety-critical situations we can evaluate the safety of agents. Our intuition is that a safe agent should not violate safety regardless of the situation it faces.

Step 3: Generation of fuzz traces. As a basis for performance testing, our testing framework applies a *search-based fuzzing method* to compute a diverse set of traces from the reference trace (Step 1) aiming for traces that gain high rewards and cover large parts of the state space.

Step 4: Testing for performance. To answer Q2, we create *performance test-suites* from the fuzz traces to bring the agent into a diverse set of states within the environment. As performance metric we propose to point-wise compare the averaged performance gained by executing the agent's policy with the averaged performance gained by executing the fuzz traces.
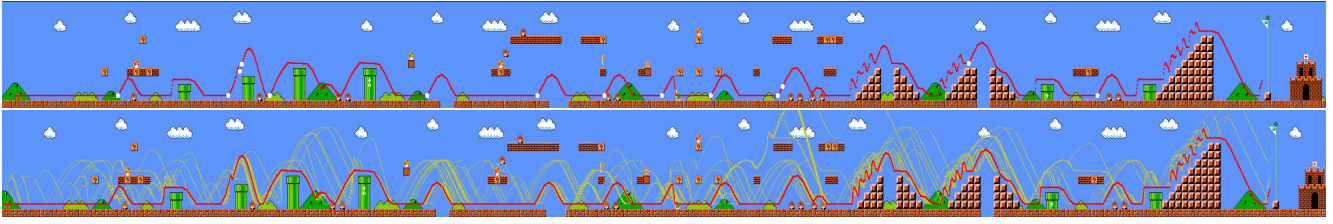
---

*Contact author

Figure 1: Super Mario Bros. Up: Reference Trace and Boundary States. Down: Reference Trace and Fuzz Traces.

Our approach is very general and can be adapted to several application areas. In settings where initial traces are given, for example, due to demonstrations by humans, such traces can be used as a basis for fuzzing. Our approach only requires to be able to sample the environment as an oracle. Even in the case of partial observability, our testing framework can be successfully applied since we only need the information whether a trace successfully completed the task to be learned, partially completed the task, or whether it violated safety. Exact state information is not required.

In our case study, we apply our framework to test the safety and performance of a set of deep RL agents trained to play Super Mario Bros. Fig. 1 shows the reference trace (red) and boundary states (white points) computed in Step 1 and the fuzz traces (yellow) from Step 3, computed in our case study. Since we consider the environment (as well as the trained agent, a learned policy may need to break ties) to be probabilistic, we execute every test case a number of times and present the averaged results.

Related Work. While RL has proven successful in solving many complex tasks [Silver *et al.*, 2016] and often outperform classical controllers [Kiran *et al.*, 2022], safety concerns prevent learned controllers from being widely used in safety-critical applications. The research on *safe RL* targets to guarantee safety during the training and the execution phase of the RL agent [García and Fernández, 2015]. Safe RL has attracted a lot of attention in the formal methods community, culminating in a growing body of work on the verification of trained networks [Ehlers, 2017; Pathak *et al.*, 2017; Corsi *et al.*, 2021]. However, all of these approaches suffer from scalability issues and are not yet able to verify industrial-size deep neural networks. An alternative line of research aims to enforce safe operation of an RL agent during runtime, using techniques from runtime monitoring and enforcement [Alshiekh *et al.*, 2018; Pranger *et al.*, 2021]. These methods typically require a complete and faithful model of the environment dynamics, which is often not available. While a large amount of work on offline and runtime verification of RL agents exists, studying suitable testing methods for RL has attracted less attention.

The development of RL algorithms has greatly benefited from benchmark environments for performance evaluation, including the Arcade Learning Environment [Bellemare *et al.*, 2013], and OpenAI Gym [Brockman *et al.*, 2016], Deepmind Control Suite [Tassa *et al.*, 2018], to name a few. SafetyGym [Achiam and Amodei, 2019] was especially designed to evaluate the safety of RL algorithms during exploration. Most work on testing for RL evaluates the aggregate per-

formance by comparing the mean and median scores across tasks. Recently, testing metrics addressing the statistical uncertainty in such point estimates have been proposed [Agarwal *et al.*, 2021]. We extend previous work by proposing search-based testing tailored toward (deep) RL. We use search-based methods to automatically create safety-critical test-cases and test cases for robust performance testing.

RL has been proposed for software testing and in particular also for fuzz testing [Böttinger *et al.*, 2018; Wang *et al.*, 2021; Scott *et al.*, 2021; Drozd and Wagner, 2018]. In contrast, we propose a novel search-based testing framework including fuzzing to test RL agents. Fuzzing has been applied to efficiently solve complex tasks [Aschermann *et al.*, 2020; Schumilo *et al.*, 2022]. We perform a backtracking-based search to efficiently solve the task, while fuzzing serves to cover a large portion of the state space. Related is also the work from Trujillo et al. [Trujillo *et al.*, 2020] which analyzes the adequacy of neuron coverage for testing deep RL, whereas our adequacy criteria are inspired by traditional boundary value and combinatorial testing.

We used our testing framework to evaluate trained deep-Q learning agents, i.e., agents that internally use deep neural networks to approximate the Q-function. Recent years have seen a surge in works on testing deep neural networks. Techniques, like DeepTest [Tian *et al.*, 2018], DeepXplore [Pei *et al.*, 2019], and DeepRoad [Zhang *et al.*, 2018], are orthogonal to our proposed framework. While we focus on the stateful reactive nature of RL agents, viewing them as a whole, these techniques are used to test sensor-related aspects of networks and find application in particular in image processing. Furthermore, we may consider taking into account neural-network-specific testing criteria [Ma *et al.*, 2018]. However, doubts about the adequacy of neuron coverage and related criteria have been raised recently [Harel-Canada *et al.*, 2020]. Hence, more research is necessary in this area, as has been pointed out by Trujillo et al. [Trujillo *et al.*, 2020].

Outline. The remainder of the paper is structured as follows. In Sec. 2, we give the background and notation. In the Sec. 3 to 6 we present and discuss in detail Step 1 - Step 4 of our testing framework. We present a detailed case study in Sec. 7.

2 Preliminaries

A **Markov decision process** (MDP) $\mathcal{M} = (\mathcal{S}, s_0, \mathcal{A}, \mathcal{P}, \mathcal{R})$ is a tuple with a finite set $\mathcal{S}$ of states including initial state s_0, a finite set $\mathcal{A} = \{a_1 \ldots, a_n\}$ of actions, and a *probabilistic transition function* $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$, and an *immediate reward function* $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$. For all $s \in \mathcal{S}$ the available actions are $\mathcal{A}(s) = \{a \in \mathcal{A} \mid \exists s', \mathcal{P}(s, a, s') \neq$

0} and we assume $|\mathcal{A}(s)| \geq 1$. A memoryless deterministic policy $\pi : \mathcal{S} \to \mathcal{A}$ is a function over action given states. The set of all memoryless deterministic policies is denoted by Π.

An **MDP with terminal states** is an MDP $\mathcal{M}$ with a set of *terminal states* $\mathcal{S}_T \subseteq \mathcal{S}$ in which the MDP terminates, i.e., the execution of a policy π on $\mathcal{M}$ yields a trace $exec_\pi(\pi, s_0) = \langle s_0, a_1, r_1, s_1, \ldots, r_n, s_n \rangle$ with only s_n being a state in $\mathcal{S}_T$. $\mathcal{S}_T$ consists of two types of states: *goal-states* $\mathcal{S}_G \subseteq \mathcal{S}_T$ representing states in which the task to be learned was accomplished by reaching them, and undesired *unsafe-states* $\mathcal{S}_U \subseteq \mathcal{S}_T$. A *safety violation* occurs whenever a state in $\mathcal{S}_U$ is entered. We define the set of *bad-states* $\mathcal{S}_B$ as all states that almost-surely lead to an unsafe state in $\mathcal{S}_U$, i.e., a state $s_B \in \mathcal{S}$ is in $\mathcal{S}_B$, if applying any policy $\pi \in \Pi$ starting in s_B leads to a state in $\mathcal{S}_U$ with probability 1. The set of *boundary-states* $\mathcal{S}_{BO}$ is defined as the set of not bad states with successor states within the bad states, i.e., a state $s_{BO} \in \mathcal{S}$ is in $\mathcal{S}_{BO}$ if $s_{BO} \notin \mathcal{S}_B$ and there exists a state $s \in \mathcal{S}_B$ and an action $a \in \mathcal{A}$ with $\mathcal{P}(s_{BO}, a, s) > 0$.

We consider **reinforcement learning** (RL) in which an agent learns a task through trial-and-error via interactions with an unknown environment modeled by a MDP $\mathcal{M} = (\mathcal{S}, s_0, \mathcal{A}, \mathcal{P}, \mathcal{R})$ with terminal states $\mathcal{S}_T \in \mathcal{S}$. At each step t, the agent receives an observation s_t. It then chooses an action $a_{t+1} \in \mathcal{A}$. The environment then moves to a state s_{t+1} with probability $\mathcal{P}(s_t, a_{t+1}, s_{t+1})$. The reward is determined with $r_{t+1} = \mathcal{R}(s_t, a_{t+1}, s_{t+1})$. If the environment enters a terminal state in $\mathcal{S}_T$, the training episode ends. The time step the episode ends is denoted by t_{end}. The return $\mathtt{ret} = \Sigma_{t=1}^{t_{\text{end}}} \gamma^t r_t$ is the cumulative future discounted reward per episode, using the *discount factor* $\gamma \in [0, 1]$. The objective of the agent is to learn an *optimal policy* $\pi^* : \mathcal{S} \to \mathcal{A}$ that maximizes the expectation of the return, i.e., $\max_{\pi \in \Pi} \mathbb{E}_\pi(\mathtt{ret})$. The accumulated reward per episode is $R = \Sigma_{t=1}^{t_{\text{end}}} r_t$.

Traces. A *trace* $\tau = \langle s_0, a_1, r_1, s_1, \ldots, a_n, r_n, s_n \rangle$ is the state-action-reward sequence induced by a policy during an episode starting with the initial state s_0. We denote a set of traces with $\mathcal{T}$. Given a trace $\tau = \langle s_0, a_1, r_1, s_1 \ldots r_n, s_n \rangle$, we use $\tau[i]$ to denote the ith state of τ ($s_i = \tau[i]$), τ^{-i} to denote the *prefix* of τ (τ^{-i} consists of all entries from τ from position 0 to i) and we denote the trace τ^{+i} to be the *suffix* of τ (τ^{+i} consists of all entries from τ from position i to n). Given a trace $\tau = \langle s_0, a_1, r_1, s_1 \ldots r_n, s_n \rangle$, we denote $|\tau| = n$ to be the length of the trace. We denote the first appearance of state s in trace τ by $d(\tau, s)$ (if $d(\tau, s) = i$ then $\tau[i] = s$). We call the action sequence resulting from omitting the states and rewards from τ an *action trace* $\tau_\mathcal{A} = \langle a_1, a_2, \ldots a_n \rangle$. $\tau_\mathcal{A}[i]$ gives the ith action, i.e., $a_i = \tau_\mathcal{A}[i]$. Executing $\tau_\mathcal{A}$ on $\mathcal{M}$ from s_0 yields a trace $exec_\tau(\tau_\mathcal{A}, s_0) = \langle s_0, a_1, r_1, s_1 \ldots r_n, s_n \rangle$ with $n = |\tau_\mathcal{A}|$.

3 Step 1 - Search for Reference Trace and Boundary States

The first step of our testing framework is to perform a *search* for a reference trace τ_{ref} that performs the tasks to be learned by the RL agent (not necessarily in the optimal way) and to detect boundary-states $\mathcal{S}'_{B0} \subseteq \mathcal{S}_{B0}$ along the reference trace.

We propose to compute τ_{ref} using a *backtracking-based,*

Algorithm 1: Search for Reference Trace τ_{ref}

input : MDP $\mathcal{M} = (\mathcal{S}, s_0, \mathcal{A}, \mathcal{P}, \mathcal{R})$, repetitions rep
output: τ_{ref}, $\mathcal{S}'_{BO}$

1 $\mathcal{V}_\mathcal{S} \leftarrow [s_0]$; $\mathcal{V}_\mathcal{A} \leftarrow [\,]$; $Explored \leftarrow \emptyset$; $success \leftarrow false$;
2 $\tau_{\text{ref}} \leftarrow [s_0]$; $\mathcal{S}'_{BO} \leftarrow \emptyset$;
3 DFS(s_0);
4 **if** $success$ **then**
5    $s_{\text{prev}} \leftarrow s_0$;
6   **for** $i \in 1, \ldots, |\mathcal{V}_\mathcal{A}|$ **do**
7       $a, s \leftarrow \mathcal{V}_\mathcal{A}[i], \mathcal{V}_\mathcal{S}[i+1]$;
8   **if** $s \notin Explored$ **then**
9          $r \leftarrow \mathcal{R}(s_{\text{prev}}, a, s)$;
10   Push($\tau_{\text{ref}}, \langle a, r, s \rangle$);
11          $s_{\text{prev}} \leftarrow s$;
12   **if** $\mathcal{V}_\mathcal{S}[i+2] \in Explored$ **then**
            /* next state is a backtracking point */
13             $\mathcal{S}'_{BO} \leftarrow \mathcal{S}'_{BO} \cup \{s\}$;
14 **Function** DFS(s):
15   **if** $s \in \mathcal{S}_U$ **then**
16       $Explored \leftarrow Explored \cup \{s\}$; **return**;
17   **if** $s \in \mathcal{S}_G$ **or** $success$ **then**
18       $success \leftarrow true$; **return**;
19   **for** $a \in \mathcal{A}$ **do**
20   **repeat** rep **times**
21   Sample s' from $\mathcal{P}(s, a)$;
22   **if** $s' \notin \mathcal{V}_\mathcal{S}$ **then**
23   Push($\mathcal{V}_\mathcal{A}, a$); Push($\mathcal{V}_\mathcal{S}, s$);
24   DFS(s');
25   **if** $\neg success$ **then** $Explored \leftarrow Explored \cup \{s\}$;

---

depth-first search (DFS) by sampling the MDP $\mathcal{M}$. For the DFS, we abstract away stochastic behavior of $\mathcal{M}$ by exploring all possible behaviors in visited states by repeating actions sufficiently often [Khalili and Tacchella, 2014]. Assuming that $p = \mathcal{P}(s, a, s')$ is the smallest transition probability greater 0 for any $s, s' \in \mathcal{S}$ and $a \in \mathcal{A}$ in $\mathcal{M}$, we compute the number of repetitions rep required to match a confidence level c via $rep(c, p) = log(1 - c)/log(1 - p)$. This ensures observing all possible states with a probability of at least c.

Example. Assume that $p = 0.1$ is the smallest probability > 0 in $\mathcal{M}$. To achieve a confidence level of 90% that the search visited any reachable state, the DFS has to perform $rep(0.9, 0.1) = 22$ repetitions of any action in any state.

Algorithm 1 gives the pseudo code of our search algorithm to compute $\tau_{\text{ref}} \in \mathcal{T}$ and a set of boundary-states $\mathcal{S}'_{B0} \subseteq \mathcal{S}_{B0}$. The list $\mathcal{V}_\mathcal{S}$ stores states that have already been visited, and $\mathcal{V}_\mathcal{A}$ stores the executed actions leading to the corresponding states in $\mathcal{V}_\mathcal{S}$. Every time the search visits an unsafe state the algorithm backtracks. A non-terminal state s is added to $Explored$ if the DFS backtracked to s from all successor states. By keeping track of visited states in the variable $\mathcal{V}_\mathcal{S}$, we ensure that we do not explore a state twice along the same trace. That is, we use $\mathcal{V}_\mathcal{S}$ to detect cycles. When visiting a goal state, DFS(s_0) terminates successfully. In this case, τ_{ref} is built from the set of visited states that were not part of a backtracking branch of the search, i.e., $s \in \tau_\mathcal{S}$ if $s \in \mathcal{V}_\mathcal{S}$ and $s \notin Explored$, with corresponding actions in $\mathcal{V}_\mathcal{A}$. States $s \in \tau_{\text{ref}}$ that have successor states $s' \in Explored$ are boundary states, i.e., $s \in \mathcal{S}'_{BO}$.

Example. Figure 3 shows parts of an MDP $\mathcal{M}$ that was explored during a run of our search algorithm. Found unsafe states are marked red. After visiting $s_{10} \in \mathcal{S}_G$ (green circle), the search function DFS(s_0) returns with $\mathcal{V}_\mathcal{S} = [s_0, \ldots, s_{10}]$, $\mathcal{V}_\mathcal{A} = [a, a, b, a, b, b, a, a, b, b]$ and $Explored =$
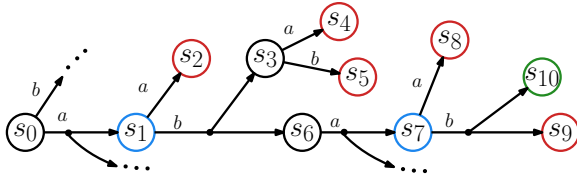
Figure 3: Run of the Search Algorithm

$\{s_2, s_3, s_4, s_5, s_8, s_9\}$. The reference trace (omitting rewards) is $\tau_{\mathrm{ref}} = \{s_0, a, s_1, b, s_6, a, s_7, b, s_{10}\}$ and the subset of boundary states is $\mathcal{S}'_{BO} = \{s_1, s_7\}$ (blue circles).

Optimizing Search. Proper abstractions of the state space may be used to merge similar states, thereby pruning the search space and enabling to find cycles in the abstract state space via the DFS. Detecting cycles speeds up the search since the DFS backtracks when finding an edge to an already visited state. An example for such an abstraction is omitting the execution time in the state space to merge states.

4 Step 2 - Testing for Safety

Based on τ_{ref} and $\mathcal{S}'_{BO}$ searched for in Step 1, we propose several *test suites* to identify weak points of a policy with a high frequency of fail verdicts, i.e., safety violations. After discussing suitable test suites, we discuss how to execute them to test the safety of RL agents.

Simple Boundary Test Suite. We use the boundary states $\mathcal{S}'_{BO}$ in τ_{ref} for boundary value testing [Pezzè and Young, 2007].We compute a simple test suite that consists of all prefixes of τ_{ref} that end in a boundary state. From these traces, we use the action traces to bring the RL agent into safety-critical situations and to test its behavior regarding safety.

Formally, let DB be the sequence of depths of the boundary states $\mathcal{S}'_{BO}$ in τ_{ref}, i.e., for any $s_{BO} \in \mathcal{S}'_{BO} : d(\tau, s_{BO}) \in DB$. Using DB, we compute a set of traces T by

$$T = \{\tau_{\mathrm{ref}}^{-DB[i]} \mid 1 \le i \le |DB|\}.$$

Omitting states and rewards from the traces in T results in a set of action traces that form a *simple boundary test suite* called ST. We say the action trace $\tau_{\mathcal{A},\mathrm{ref}}^{DB[i]} \in ST$ is the test case for the ith boundary state in $\mathcal{S}'_{BO}$.

Test Suites using Boundary Intervals. Boundary value testing checks not only boundary values, but also inputs slightly off of the boundary [Pezzè and Young, 2007]. To transfer this concept into RL testing, we introduce boundary intervals to test at additional states near the boundary.

In contrast to boundary testing of conventional software, our test cases stay in states traversed by τ_{ref}. This choice is motivated by the definition of a boundary state: a state with successor states that necessarily lead to an unsafe state. Bringing the RL agent in such a losing position will not provide additional insight concerning the learned safety objective, since the agent has no other choice than to violate safety. However, testing states of τ_{ref} within an offset of boundary states provides insights into how well the RL agent escapes safety-critical situations. Given a simple test suite ST and

an interval-size is, we create an *interval test suite* $IT(is)$ by adding additional test cases to ST, such that

$$IT(is) = \{\tau_{\mathcal{A},\mathrm{ref}}^{DB[i]+\mathit{off}} \mid \tau_{\mathcal{A},\mathrm{ref}}^{DB[i]} \in ST, -is \le \mathit{off} \le is\},$$

where $\tau_{\mathcal{A},\mathrm{ref}}$ is the reference action-trace. The test case $\tau_{\mathcal{A},\mathrm{ref}}^{DB[i]+\mathit{off}}$ tests the agent at boundary state i with offset off.

Test Suites using Action Coverage. Combinatorial testing covers issues resulting from combinations of input values. We adapt this concept by creating test suites that cover combinations of actions near boundary states, i.e., the test suite evaluates which actions cause unsafe behavior in boundary regions. Given the reference action-trace $\tau_{\mathcal{A},\mathrm{ref}}$, a simple test suite ST, and a $k \ge 1$, we generate a *k-wise action-coverage* test suite $AC(k)$ by creating $|\mathcal{A}|^k$ test cases for every test case in ST covering all k-wise combinations of actions at the kth predecessor of a boundary state. The test suite is given by

$$AC(k) = \{\tau_{\mathcal{A},\mathrm{ref}}^{DB[i]-k} \cdot ac \mid \tau_{\mathcal{A},\mathrm{ref}}^{DB[i]} \in ST, ac \in \mathcal{A}^k\}.$$

Test-case Execution & Verdict. To test the behavior of an agent regarding safety, we use a safety test-suite to bring the agent in safety critical situations. A single test-case execution takes an action-trace $\tau_{\mathcal{A}}$, an inital state s_0 and a test length l as parameters. To test an RL agent using $\tau_{\mathcal{A}}$, we first execute $\tau_{\mathcal{A}}$, yielding a trace $exec(\tau_{\mathcal{A}}) = \langle s_0, a_1, r_1, s_1, \ldots, a_n, r_n, s_n \rangle$. A test case $\tau_{\mathcal{A}}$ is *invalid* if $exec(\tau_{\mathcal{A}})$ consistently visits a terminal state in $\mathcal{S}_T$ when executed repeatedly.

Starting from s_n, we pick the next l actions according to the policy of the agent. Note that l should be chosen large enough to evaluate the behavior of the agent regarding safety. Therefore, it should be considerably larger than the shortest path to the next unsafe state in S_U. After performing l steps of the agent's policy, we evaluate the test case. A test can *fail* or *pass*: A test fails, if starting from s_n the agent reaches an unsafe state in $\mathcal{S}_U$ within l steps. Otherwise, the test passes.

To execute a test suite T, we perform every test case of T n times. During that, we compute the *relative frequency of fail verdicts* resulting from executing each individual test case.

5 Step 3 - Generation of Fuzz Traces

Our testing framework evaluates the performance of RL agents using fuzz traces. The traces are used to compare gained rewards as well as to bring the agent in a variety of states and to evaluate the performance from these onward. In this section, we discuss the fuzz-trace generation for performance testing. For this purpose, we propose a search-based fuzzing method [Zeller *et al.*, 2021] based on genetic algorithms. The goal is to find action traces that (1) cover a large portion of the state space while (2) accomplishing the task to be learned by the RL agent.

Overview of Computation of Fuzz Traces. Given the reference trace τ_{ref} that solves the RL task (i.e., $s_n \in \mathcal{S}_G$) and parameter values for the number of generations g and the population size p, the *fuzz traces* are computed as follows:

1. Initialize $\mathcal{T}_0$, the trace population: $\mathcal{T}_0 := \{\tau_{\mathcal{A},\mathrm{ref}}\}$.
2. For $i = 1$ **to** g generations do:
   (a) Create p action traces (called *offspring*) from $\mathcal{T}_{i-1}$ to yield a new population $\mathcal{T}_i$ of size p by:

- *either mutating* a single parent trace from $\mathcal{T}_{i-1}$,
- *or* through *crossover* of two parents from $\mathcal{T}_{i-1}$ with a specified crossover probability.

  (b) Evaluate the *fitness* of every offspring trace in $\mathcal{T}_i$.

3. Return $\mathcal{T}_{fit}$ containing the fittest trace of each generation

The *fitness* of a trace is defined in terms of state-space coverage and the degree to which the RL task is solved. The computation of the fuzz traces searches iteratively for traces with a high fitness by choosing parent traces with a probability proportional to their fitness. To promote diversity, we favor mutation over crossover, by setting the crossover probability to a value < 0.5. The set of the fittest traces $\mathcal{T}_{fit}$ will be used in Step 4 for performance testing. Using the single fittest trace from every generation helps enforcing variety.

Fitness Computation. We propose a fitness function especially suited for testing RL agents. For an action trace $\tau_{\mathcal{A}}$, the fitness $F(\tau_{\mathcal{A}})$ is the weighted sum of three normalized terms:

- The *positive-reward term* $r_{pos}(\tau_{\mathcal{A}}, s_0)$ is the normalized positive reward gained in $exec_\tau(\tau_{\mathcal{A}}, s_0)$.
- The *negative-reward term* $r_{neg}(\tau_{\mathcal{A}}, s_0)$ is the normalized inverted negative reward gained in $exec_\tau(\tau_{\mathcal{A}}, s_0)$.
- The *coverage fitness-term* $fc(\tau_{\mathcal{A}}, s_0)$ describes the number of newly visited states by $exec_\tau(\tau_{\mathcal{A}}, s_0)$, normalized by dividing by the maximum number of newly visited states by any action traces in the current population.

Positive rewards correspond to the degree as to which the RL task is solved by $\tau_{\mathcal{A}}$. *Negative rewards* often correspond to the time required to solve the RL task. Hence if $\tau_{\mathcal{A}}$ solves the task fast, it would be assigned a small negative reward. We invert the negative reward to have only positive fitness terms. We normalize r_{pos}/r_{neg} by dividing it by the highest r'_{pos}/r'_{neg} in the current generation. The *coverage fitness-term* depends on all states visited in previous populations. Assume that the current generation is i. Let Cov_{ppop} be the set of all states visited by the previous populations $\bigcup_{j<i} \mathcal{T}_j$ and let $Cov(\tau_{\mathcal{A}})$ be the visited states when executing an action trace $\tau_{\mathcal{A}}$, i.e., $Cov(\tau_{\mathcal{A}}) = \bigcup_{k \le n}\{s_k\}$, where $exec_\tau(\tau_{\mathcal{A}}, s_0) = \langle s_0, a_0, r_0, s_1, \ldots, s_n\rangle$.

The *coverage fitness-term* $fc(\tau_{\mathcal{A}})$ is then given by

$$fc(\tau_{\mathcal{A}}) = \frac{|Cov_{ppop} \setminus Cov(\tau_{\mathcal{A}})|}{\max_{\tau'_{\mathcal{A}} \in \mathcal{T}_i} |Cov_{ppop} \setminus Cov(\tau'_{\mathcal{A}})|}.$$

$fc(\tau_{\mathcal{A}})$ is a normalized value ≤ 1 and changes during fuzzing as more states are covered. The *fitness* $F(\tau_{\mathcal{A}})$ is given by

$$F(\tau_{\mathcal{A}}) = \lambda_{cov} fc(\tau_{\mathcal{A}}) + \lambda_{pos} r_{pos}(\tau_{\mathcal{A}}) + \lambda_{neg}(1 - |r_{neg}(\tau_{\mathcal{A}})|),$$

where the factors λ_j are weights and $r_{pos}(\tau_{\mathcal{A}})$ and $r_{neg}(\tau_{\mathcal{A}})$ are normalized rewards gained when executing $\tau_{\mathcal{A}}$.

Mutation & Crossover. To generate a new action trace, we perform either a crossover of two parent traces or mutate a single parent trace. For *crossover*, we create a new offspring trace splitting two parent traces and concatenating the resulting subtraces. Let $\tau_{\mathcal{A},1}$ and $\tau_{\mathcal{A},2}$ be the parent action-traces. To create an offspring trace, we uniformly select a random *crossover point* $i \in \{1, \ldots, \min(|\tau_{\mathcal{A},1}|, |\tau_{\mathcal{A},2}|) - 1\}$. The offspring is the concatenation of $\tau_{\mathcal{A},1}^{-i}$ and $\tau_{\mathcal{A},2}^{+i}$. For *mutation*, we repeatedly apply mutation operators. Given a parent trace $\tau_{\mathcal{A}}$ and a parameter ms defining the potential effect size of a mutation, we create an offspring $\tau'_{\mathcal{A}}$ as following:

Algorithm 2: Performance Testing with Fuzz Traces

input : $\mathcal{M} = (\mathcal{S}, s_0, \mathcal{A}, \mathcal{P}, \mathcal{R})$, policy π, fuzz traces $\mathcal{T}_{fit}$, # episodes n_{ep}

output: Avg. accumulated rewards of the agent R_a and the fuzz traces R_t

1   **return** $R_t \leftarrow$ EvalTraces$(\mathcal{T}_{fit}, s_0, n_{ep})$, $R_a \leftarrow$ EvalAgent(π, s_0, n_{ep});

2   **Function** EvalTraces$(\mathcal{T}_{fit}, s_0, n_{ep})$:

3   **for** $\tau_{\mathcal{A}} \in \mathcal{T}_{fit}$ **do**

4   **for** $i \leftarrow 1$ **to** n_{ep} **do**

5         $\tau_i \leftarrow exec_\tau(\tau_{\mathcal{A}}, s_0) = \langle s_0, a_1, r_1 \ldots s_n \rangle$;

6         $R_{t,\tau_{\mathcal{A}},i} \leftarrow \Sigma_{k=1}^n r_k$ with $r_k \in \tau_i$

7   **return** $R_t = (\Sigma_{\tau_{\mathcal{A}} \in \mathcal{T}_{fit}} \Sigma_{i=1}^{n_{ep}} R_{t,\tau_{\mathcal{A}},i})/(n_{ep} \cdot |\mathcal{T}_{fit}|)$

8   **Function** EvalAgent(π, s_0, n_{ep}):

9   **for** $i \leftarrow 0$ **to** n_{ep} **do**

10      $\tau_i \leftarrow exec_\tau(\pi, s_0) = \langle s_0, a_1, r_1 \ldots s_n \rangle$ with $s_n \in \mathcal{S}_T$;

11      $R_{a,i} \leftarrow \Sigma_{k=1}^n r_k$ with $r_k \in \tau_i$;

12   **return** $R_a = (\Sigma_{i=1}^{n_{ep}} R_{a,i})/n_{ep}$

Algorithm 3: Robust Performance Testing

input : $\mathcal{M} = (\mathcal{S}, s_0, \mathcal{A}, \mathcal{P}, \mathcal{R})$, policy π, fuzz traces $\mathcal{T}_{fit}$, # tests n_{test}, # episodes n_{ep}, step width w

output: Avg. accumulated rewards R_t^{pl} and R_a^{pl}

1   $pl \leftarrow w$;

2   **repeat**

3   **for** $i \leftarrow 1$ **to** n_{test} **do**

4       $\tau_{\mathcal{A}} \leftarrow$ random action trace $\in \mathcal{T}_{fit}$;

5       $\tau^{-pl} \leftarrow exec_\tau(\tau_{\mathcal{A}}^{-pl}, s_0) = \langle s_0 \ldots s_{pl} \rangle$;

6       $R^- \leftarrow \Sigma_{t=1}^{pl} r_t$ with $r_t \in \tau^{-pl}$;

7       $R_{t,i}^{pl} \leftarrow R^- +$ EvalTraces$(\{\tau_{\mathcal{A}}^{pl+}\}, s_{pl}, n_{ep})$;

8       $R_{a,i}^{pl} \leftarrow R^- +$ EvalAgent(π, s_{pl}, n_{ep});

9     $R_t^{pl} \leftarrow (\Sigma_{i=1}^{n_{test}} R_{t,i}^{pl})/n_{test}$;

10     $R_a^{pl} \leftarrow (\Sigma_{i=1}^{n_{test}} R_{a,i}^{pl})/n_{test}$;

11     $pl \leftarrow pl + w$;

12   **until** $|\{\tau_{\mathcal{A}} \in \mathcal{T}_{fit} : |\tau_{\mathcal{A}}| \ge pl\}| < n_{test}$ // too few traces of length pl;

13   **return** $\bigcup_{pl}\{pl \mapsto R_t^{pl}\}, \bigcup_{pl}\{pl \mapsto R_a^{pl}\}$

---

1. Uniformly sample $x \in \{1, \ldots, ms\}$.
2. Chose a mutation operator parametrized with x and perform the mutation on $\tau_{\mathcal{A}}$ to create an action trace $\tau'_{\mathcal{A}}$.
3. Stop with a probability $p_{mstop} \in (0, 1]$ and return $\tau'_{\mathcal{A}}$. Otherwise, set $\tau_{\mathcal{A}} \leftarrow \tau'_{\mathcal{A}}$ and continue with Step 1.

The applied mutation operators are (1) Insert, (2) Remove (3) Change, and (4) Append. Each performs its eponymous operation on an action sequence of length x at a randomly chosen index in the parent trace, except for Append.

6 Step 4 - Testing for Performance

In the final step, we evaluate the performance of trained RL agents. The evaluation compares the accumulated reward gained from applying the agent's policy with the accumulated reward gained by executing fuzzed traces. Especially in RL settings where the maximal expected reward is unknown, the rewards gained by the fuzz traces serve as a benchmark for the agents' performance. Furthermore, the fuzz traces are used to test the agents' performance from a diverse set of states.

Performance Testing. Simple performance testing starts in a fixed initial state and compares the average accumulated reward of the agent with the average accumulated reward resulting from the execution of fuzz traces. Given the policy π of an agent under test, the fuzz traces $\mathcal{T}_{fit}$, an initial state s_0, and a number of episodes n_{ep}, Algo. 2 returns the averaged accumulated rewards of the agent R_a and the fuzz traces R_t.

Robust Performance Testing. Robust performance testing

targets checking the robustness of learned policies in potentially unknown situations. For this purpose, we use the fuzz traces to bring the agent into a diverse set of states and apply the policy of the agent from these states onward until a terminal state is reached. To cover states close to the initial state as well as close to the goal, we actually use fuzz trace prefixes of increasing length. The averaged accumulated rewards of the agent traces and the fuzz traces serve as performance metric.

Let π be the policy of the RL agent under test, $\mathcal{T}_{fit}$ be the fuzz traces, s_0 be an initial state, w be a step width to increase the fuzz trace prefix-length, and n_{test} and n_{ep} be numbers of tests and episodes, Algo. 3 implements our robust performance testing approach. Starting from prefix length $pl = w$, for each pl the amount of executed tests is n_{test}. To do so, we first select a random trace (line 4) and execute its prefix of length pl to arrive at a state s_{pl}. From s_{pl}, we compute the accumulated reward of the fuzz trace (line 6) and of the agent averaged over n_{ep} episodes (line 8) and add the accumulated reward of the common prefix R^- to both. We average the accumulated rewards over all tests for each pl individiually (lines 9 and 10) and finally return all results in Line 13.

7 Experimental Evaluation

We evaluate our testing framework on trained deep RL agents for the video game Super Mario Bros., trained with varying numbers of training episodes and different action sets.

Setup for RL. We use a third-party RL agent that operates in the OpenAI gym environment and uses double deep Q-Networks [Feng *et al.*, 2020]. Details on the learning parameters along with more experiments and the source code are included in the technical appendix. To evaluate different agents, we test agents at varying stages of learning having three different action sets: (1) *2-moves:* fast running and jumping to the right. (2) *basic:* 2-moves plus slow running to the right and left. (3) *right-complex:* basic without running left, but actions for pressing up and down, resulting in the largest action set. Unless otherwise noted, we present results from training for $80k$ episodes. We stopped training at this point, since we observed only little improvement from $40k$ to $80k$ episodes, which is also twice as long as suggested [Feng *et al.*, 2020].

Setup for Search and Fuzzing. The search for the reference traces uses the *2-moves* actions, which is sufficient to complete most levels in Super Mario Bros. We compute fuzz traces for each action set of the different RL agents. We fuzzed for 50 generations with a population of 50, used a mutation stop-probability $p_{mstop} = 0.2$ with effect size $ms = 15$ and fitness weights $\lambda_{cov} = 2$, $\lambda_{pos} = 1.5$, and $\lambda_{neg} = 1$, to focus on exploration. With a crossover probability of 0.25, we mostly rely on mutations. The search and fuzzing were performed in a standard laptop in a few minutes and a few hours, respectively. Compared to the training that took several days on a dedicated cluster, the computational effort for testing is relatively low. For safety testing, we use 10 repetitions and a test length of $l = 40$ and for performance testing we use $n_{test} = n_{ep} = 10$ and step width $w = 20$.

Safety Testing. Fig. (3) shows the relative number of fail verdicts averaged over all tests at all boundary states for agents with different action sets. For any agent, all test suites
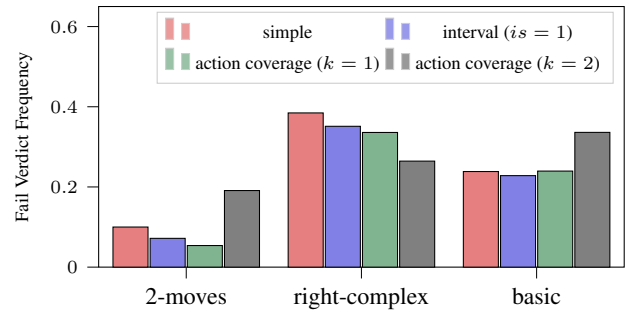

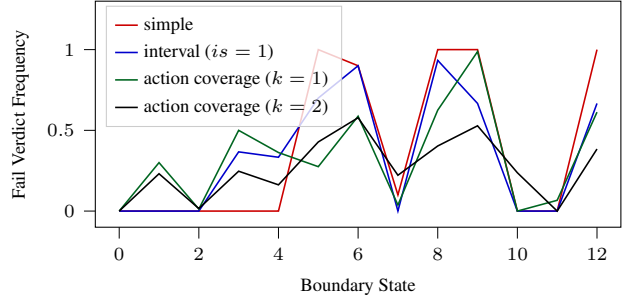Figure 3: Safety Testing: Relative frequency of fail verdicts

Figure 4: Safety Testing of the *right-complex* agent: Relative frequencies of fail verdicts at boundary states

found safety violations. For instance, the simple test suite produces fail verdicts in about 38% of the cases when testing the *right-complex* RL agent. The agent with 2-moves is tested to be the safest agent, failing only 10% of test cases of the simple test suite. For *right-complex*, the least safe agent, Fig. (4) depicts the relative number of fail verdicts distributed over the boundary states, when executing all test suites. Note that the results are affected by stochasticity and they are normalized, so that all results are within $[0, 1]$ even though the extended test suites perform more tests. We can see that the early boundary states that are explored the most cause the least issues. Furthermore, we observe that the boundary interval test suite finds safety violations not detected by the simple test suite, e.g., at the boundary states 3 and 4.

Robust Performance Testing. We perform robust performance testing on all agents trained for $20k$ and $80k$ episodes, respectively. Fig. (5) shows the average accumulated rewards (y-axis) gained by the agents and the fuzz traces when performing fuzz trace prefixes of the length given by the x-axis. It can be seen that initially only the well-trained *2-moves* agent surpasses the performance benchmark set by the fuzz traces. Training for $20k$ episodes is not enough for any agent to achieve rewards close to the fuzz traces and the *basic* agent does not improve much with more training. Hence, a larger action space may hurt robustness. The ability to move left of the *basic* agent also increases the state space of the underlying MDP, since only moving right induces a DAG-like structure. This explains the poor performance of the *basic* agents.

Relationship between Safety and Performance. Finally, we investigate whether performance expressed via rewards implies safety. Fig. (6) shows the average number of safety vi-
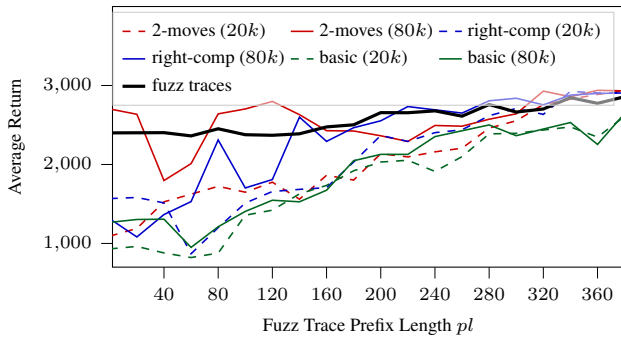
Figure 5: Robust Performance Testing: Average accumulated rewards of fuzz traces and agents trained for $20k$ and $80k$ episodes.

Figure 6: Average frequency of safety violations and average accumulated rewards during testing with the simple test suite.

olations and the average accumulated rewards gained during testing with a simple test suite with all agents trained for $20k$ and $80k$ episodes, respectively. Comparing the agents with low amount of training, the safest agent (*basic*) is also the one that gains the lowest reward. For well-trained agents, the safest agent (*2-moves*) also receives the most reward. The large negative reward assigned to safety violations (losing a life) may not be sufficient to enforce safe behavior of *right-complex*. Hence, our testing method may point to issues in reward function design. However, computing the Pearson correlation coefficient between fail verdict frequency and mean accumulated reward for all agents at four stages of training with all test suites reveals a moderate negative correlation of -0.7, thus high reward often implies low fail frequency.

8 Concluding Remarks

We present a search-based testing framework for safety and robust-performance testing of RL agents. For safety testing, we apply backtracking-based DFS to identify relevant states and adapt test-adequacy criteria from boundary value and combinatorial testing. For performance testing, we apply genetic-algorithm-based fuzzing starting from a seed trace found by the DFS. We show both testing methodologies on an off-the-shelf deep RL agent for playing Super Mario Bros, where we find safety violations of well-trained agents and analyze their performance and robustness. To the best of our knowledge, we propose one of the first testing frameworks tailored toward RL. For future work, we will instantiate our framework for more RL tasks, where solutions can be found through search and other domain-specific approaches.

We also plan to investigate different fuzzing approach, e.g., fuzzing on policy level rather than trace level.

References

[Achiam and Amodei, 2019] Joshua Achiam and Dario Amodei. Benchmarking safe exploration in deep reinforcement learning. *Preprint. Under review*, 2019.

[Agarwal *et al.*, 2021] Rishabh Agarwal, Max Schwarzer, Pablo Samuel Castro, Aaron C Courville, and Marc Bellemare. Deep reinforcement learning at the edge of the statistical precipice. In *NeurIPS*, volume 34, pages 29304–29320, 2021.

[Alshiekh *et al.*, 2018] Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. Safe reinforcement learning via shielding. In *AAAI-18*, pages 2669–2678, 2018.

[Aschermann *et al.*, 2020] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *IEEE SP 2020*, pages 1597–1612, 2020.

[Bellemare *et al.*, 2013] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *J. Artif. Intell. Res.*, 47:253–279, 2013.

[Böttinger *et al.*, 2018] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. Deep reinforcement fuzzing. In *IEEE SP Workshops 2018*, pages 116–122, 2018.

[Brockman *et al.*, 2016] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI gym. *arXiv preprint arXiv:1606.01540*, 2016.

[Corsi *et al.*, 2021] Davide Corsi, Enrico Marchesini, and Alessandro Farinelli. Formal verification of neural networks for safety-critical tasks in deep reinforcement learning. In *UAI*, volume 161 of *Proceedings of Machine Learning Research*, pages 333–343, 2021.

[Drozd and Wagner, 2018] William Drozd and Michael D. Wagner. FuzzerGym: A competitive framework for fuzzing and learning. *CoRR*, abs/1807.07490, 2018.

[Ehlers, 2017] Rüdiger Ehlers. Formal verification of piecewise linear feed-forward neural networks. In *ATVA 2017*, volume 10482 of *LNCS*, pages 269–286, 2017.

[Feng *et al.*, 2020] Yuansong Feng, Suraj Subramanian, Howard Wang, and Steven Guo. Train a mario-playing RL agent. https://pytorch.org/tutorials/intermediate/mario_rl_tutorial.html, 2020. A PyTorch tutorial [online], accessed: 2022, January 07.

[Garcıa and Fernández, 2015] Javier Garcıa and Fernando Fernández. A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research*, 16(1):1437–1480, 2015.

[Harel-Canada *et al.*, 2020] Fabrice Harel-Canada, Lingxiao Wang, Muhammad Ali Gulzar, Quanquan Gu, and Miryung Kim. Is neuron coverage a meaningful measure for testing deep neural networks? In *ESEC/FSE*, pages 851–862. ACM, 2020.

[Karakovskiy and Togelius, 2012] Sergey Karakovskiy and Julian Togelius. The mario AI benchmark and competitions. *IEEE T-CIAIG*, 4(1):55–67, 2012.

[Khalili and Tacchella, 2014] Ali Khalili and Armando Tacchella. Learning nondeterministic Mealy machines. In *ICGI 2014*, volume 34 of *JMLR Workshop and Conference Proceedings*, pages 109–123, 2014.

[Kiran *et al.*, 2022] B Ravi Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A Al Sallab, Senthil Yogamani, and Patrick Pérez. Deep reinforcement learning for autonomous driving: A survey. *IEEE Transactions on Intelligent Transportation Systems*, 23(6):4909–4926, 2022.

[Küttler *et al.*, 2020] Heinrich Küttler, Nantas Nardelli, Alexander H. Miller, Roberta Raileanu, Marco Selvatici, Edward Grefenstette, and Tim Rocktäschel. The NetHack Learning Environment. In *NeurIPS*, pages 7671–7684, 2020.

[Ma *et al.*, 2018] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. DeepGauge: multi-granularity testing criteria for deep learning systems. In *ASE*, pages 120–131. ACM, 2018.

[Pathak *et al.*, 2017] Shashank Pathak, Luca Pulina, and Armando Tacchella. Verification and repair of control policies for safe reinforcement learning. *Applied Intelligence*, 48:886–908, 2017.

[Pei *et al.*, 2019] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. DeepXplore: automated whitebox testing of deep learning systems. *Commun. ACM*, 62(11):137–145, 2019.

[Pezzè and Young, 2007] Mauro Pezzè and Michal Young. *Software testing and analysis - process, principles and techniques*. Wiley, 2007.

[Pranger *et al.*, 2021] Stefan Pranger, Bettina Könighofer, Martin Tappler, Martin Deixelberger, Nils Jansen, and Roderick Bloem. Adaptive shielding under uncertainty. In *ACC*, pages 3467–3474, 2021.

[Schrittwieser *et al.*, 2020] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

[Schumilo *et al.*, 2022] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. Nyx-net: network fuzzing with incremental snapshots. In *EuroSys*, pages 166–180, 2022.

[Scott *et al.*, 2021] Joseph Scott, Trishal Sudula, Hammad Rehman, Federico Mora, and Vijay Ganesh. Banditfuzz: Fuzzing SMT solvers with multi-agent reinforcement learning. In *FM*, volume 13047 of *LNCS*, pages 103–121, 2021.

[Silver *et al.*, 2016] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[Sutton and Barto, 1998] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning - an introduction*. Adaptive computation and machine learning. MIT Press, 1998.

[Tassa *et al.*, 2018] Yuval Tassa, Yotam Doron, Alistair Muldal, Tom Erez, Yazhe Li, Diego de Las Casas, David Budden, Abbas Abdolmaleki, Josh Merel, Andrew Lefrancq, et al. DeepMind control suite. *arXiv preprint arXiv:1801.00690*, 2018.

[Tian *et al.*, 2018] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. DeepTest: automated testing of deep-neural-network-driven autonomous cars. In *ICSE*, pages 303–314. ACM, 2018.

[Trujillo *et al.*, 2020] Miller Trujillo, Mario Linares-Vásquez, Camilo Escobar-Velásquez, Ivana Dusparic, and Nicolás Cardozo. Does neuron coverage matter for deep reinforcement learning?: A preliminary study. In *ICSE '20 Workshops*, pages 215–220, 2020.

[Wang *et al.*, 2021] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V. Krishnamurthy, and Nael B. Abu-Ghazaleh. Syzvegas: Beating kernel fuzzing odds with reinforcement learning. In *USENIX Security*, pages 2741–2758, 2021.

[Zeller *et al.*, 2021] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. The fuzzing book. https://www.fuzzingbook.org/, 2021. accessed: 2022, January 07.

[Zhang *et al.*, 2018] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems. In *ASE*, pages 132–142. ACM, 2018.