Policy Testing with MDPFuzz (Replicability Study)

Quentin Mazouni
Simula Research Laboratory
Oslo, Norway
quentin@simula.no

Helge Spieker
Simula Research Laboratory
Oslo, Norway
helge@simula.no

Arnaud Gotlieb
Simula Research Laboratory
Oslo, Norway
arnaud@simula.no

Mathieu Acher
Univ Rennes, Inria, INSA Rennes, CNRS, IRISA
Rennes, France
mathieu.acher@irisa.fr

ABSTRACT

In recent years, following tremendous achievements in Reinforcement Learning, a great deal of interest has been devoted to ML models for sequential decision-making. Together with these scientific breakthroughs/advances, research has been conducted to develop automated functional testing methods for finding faults in black-box Markov decision processes. In 2022, Pang et al. presented a black-box fuzz testing framework called MDPFuzz. The method consists of a fuzzer whose main feature is to use *Gaussian Mixture Models* (GMMs) to compute coverage of the test inputs as the likelihood to have already observed their results. This guidance through coverage evaluation aims at favoring novelty during testing and fault discovery in the decision model.

Pang et al. evaluated their work with four use cases, by comparing the number of failures found after twelve-hour testing campaigns with or without the guidance of the GMMs (ablation study). In this paper, we verify some of the key findings of the original paper and explore the limits of MDPFuzz through reproduction and replication. We re-implemented the proposed methodology and evaluated our replication in a large-scale study that extends the original four use cases with three new ones. Furthermore, we compare MDPFuzz and its ablated counterpart with a random testing baseline. We also assess the effectiveness of coverage guidance for different parameters, something that has not been done in the original evaluation. Despite this parameter analysis and unlike Pang et al.' original conclusions, we find that in most cases, the aforementioned ablated Fuzzer outperforms MDPFuzz, and conclude that the coverage model proposed does not lead to finding more faults.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; • **Computing methodologies → Reinforcement learning**.

KEYWORDS

Replicability, Software Testing, Reinforcement Learning

ACM Reference Format:
Quentin Mazouni, Helge Spieker, Arnaud Gotlieb, and Mathieu Acher. 2018. Policy Testing with MDPFuzz (Replicability Study). In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024)*. ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

In recent years, the combination of Machine Learning (ML) with neural networks has been effectively employed to solve complex sequential decision-making tasks of diverse nature, such as gaming [22], planning [11, 28] or system control [3, 17]. Likewise, dedicated testing techniques have been proposed to enable the safe use and deployment of these models. Despite the wide variety in the current approaches, such as search-based testing [26], metamorphic testing [4, 5] and fuzz testing [18, 23], few works are comparing them, exploring their limitations or verifying their findings. In particular, as time of writing no replicability study has been conducted.

In 2022, Pang et al. [18] presented a black-box fuzz framework called MDPFuzz. We decided to replicate this precise framework for several reasons. First, MDPFuzz is generally applicable and shown to be effective at finding functional faults in decision-making policies. Second, the method has attracted strong interest in the software testing community. For instance, several works have built upon it [9, 30]. Similarly, Li et al. [12] and Mazouni et al. [15] consider MDPFuzz as a state-of-the-art policy testing technique to evaluate their approach.

However, we identified severe bugs in the original implementation[1] as well as significant differences compared to the algorithm of the method (detailed later). We contacted the authors to discuss these discrepancies (during an email conversation [20]). Finally, we worried about its functional complexity, something critical for efficient testing. Indeed, this framework guides the testing process with coverage estimations based on Gaussian Mixture Models (GMMs) [7] which are compute-heavy tasks.

Precisely, the idea of MDPFuzz is to guide a fuzzing process [16] that generates and mutates input tests. Testing is guided by maintaining the pool of inputs with (1) ones that reveal a weakness in the model under test (i.e. robustness) and (2) ones whose test result is deemed uncovered (i.e. coverage-based guidance). Pang

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISSTA 2024, 16-20 September, 2024, Vienna, Austria
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

[1]https://github.com/Qi-Pang/MDPFuzz

et al. propose to compute input coverage over the state sequence resulting from a test case with two GMMs, which requires the computation of $1 + 2\ |M|$ probability densities (M being the length of the sequence). The additional coverage guidance aims at exercising the model under test in novel ways such that faults are more likely to be found. Pang et al. evaluated their work with four use cases on fault detection, GMMs guidance efficiency, fault analysis and policy improvement (retraining). In particular, they show that MDPFuzz finds more faults than an ablated version we refer to as Fuzzer. This framework is only guided by robustness, i.e., it maintains its pool with mechanism (1) and, consequently, does not have GMMs and does not compute coverage.

In this paper, we investigate the ability of MDPFuzz to find faults in decision-making models. Given the issues and algorithmic discrepancies found in the original implementation, we decided to conduct a two-step methodology. In the first step (later called "reproduction study"), we patch the code and use both the unfixed and fixed versions in an attempt to reproduce the original findings and determine whether the bugs were present when Pang et al. evaluated their method. In a second step (later called "replication study"), we re-implement the fuzzers, i.e., we implement the algorithm described in the original paper and extend their evaluation. Specifically, We add three new use cases and include a random testing baseline. Furthermore, we perform a parameter analysis of MDPFuzz – which is not available in the original publication – to (1) assess its sensitivity, (2) mitigate potential biases in the previous results and (3) possibly find out an optimal configuration for fault discovery.

In summary, the contributions of this paper are:

- **Reproduction** We report on the issues and difficulties in reproducing the results with the original implementation. We describe our best efforts to understand and mitigate them and identify the underlying threats to the validity of the original evaluation. We patch the code and present the results obtained with both the unfixed and fixed versions.
- **Replication** We re-implement MDPFuzz and its coverage-free version. Our results validate the fault discovery ability of MPDFuzz in the tested models, but unlike the original evaluation, we find that the ablated version outperforms its coverage-guided counterpart.
- **Comparison** We compare our replication of the two fuzzers with a random testing baseline in an extended study of seven use cases.
- **Parameter Evaluation** The coverage model integrated by MDPFuzz is parameterized by three parameters, but defining their values is not straightforward, and they were not investigated in the original evaluation. Therefore, we explore their impact on fault detection.
- **Valuable Artefacts** Following the best practices of replication studies, our replicate is freely available. In particular, we provide the core components of MDPFuzz as reusable modules to enable convenient future work.

2 BACKGROUND

In this section, we introduce policy testing for sequential decision-making and MDPFuzz.

2.1 Sequential decision-making

Roughly speaking, sequential decision-making describes any task that can be solved in a step-by-step manner by one or several models [6]. In this work, we are interested in testing these decision-making entities. Precisely, a test case consists of setting the environment of the decision task to an initial state and letting the model under test (or agent) interact with the former. This step-wise, observation-decision-action process eventually leads to a final state, which is passed to the test oracle to detect a failure. Decision-making problems are formally defined as Markov Decision Processes (MDPs), often represented as 4-tuples $\langle S, A, R, T\rangle$:

- S is a set of states. They correspond to what the agent observes at every step of an execution.
- A is the set of possible actions, which can be either continuous or discrete. They are what the agent returns to the environment.
- $R : S \times A \mapsto \mathbb{R}$ is the reward function. Rewards are the feedback received by the agent after every action.
- $T : S \times A \times S \mapsto [0, 1]$ is the transition function, which is a probability distribution over S and A. It depicts which state the MDP will transit to for a given action. The function is not known by the agent and describes the environment's dynamic.

Solutions to MDPs are mapping functions $\pi : S \times A \mapsto [0, 1]$ called policies, which choose the action for a given state. The most commonly used approach to train policies is Reinforcement Learning (RL), a sub-field of Machine Learning which consists of learning from the rewards/penalties returned by the MDP [24]. Precisely, RL learns an optimal, or near-optimal, policy that maximises the total expected discounted cumulative reward $R_t = \sum_{t>0} \gamma^{t-1} r_t$, where $\gamma \in [0, 1)$ is the discount factor. The latter controls how π considers future rewards: a small value emphasizes short-term gains, whereas values close to 1 lead the agent to focus on maximising long-term rewards.

In this work, we consider black-box policy testing, meaning that, during a test case execution, we can only observe the interaction between MDP and agent, but have no access to any of their internals. MDPFuzz further assumes that the executions are limited to M steps, and targets *deterministic* decision models. As such, given the model under test $\pi : S \mapsto A$, we denote the execution of input $i \in S$ as $MDP(i, \pi)$. The result is composed of the cumulative reward r and the sequence of states $\{s_t\}_{t\in 0..M\text{-}1}$.

2.2 MDPFuzz

2.2.1 *Preliminaries.* MDPFuzz is based on fuzzing, which here means that the inputs (or initial states) used to initialize the MDP are first randomly generated and then mutated. To that end, the method records the inputs tested in a pool along with two measurements: sensitivity (also called "energy") and their coverage (or "density").

Sensitivity. Sensitivity estimates the robustness of π against MDPFuzz's mutations on the states. The sensitivity of the state s is defined as the absolute reward difference between $MDP(s, \pi)$ and $MDP(s + \Delta s, \pi)$, where Δs is a small random perturbation [18]. The higher the sensitivity of a state, the more diverse the results of its mutations. The sensitivities are used to bias the input selection during fuzzing, which we discuss below.

Coverage. This measure aims at fostering the novelty of the state sequences collected after test executions. It aims at favoring the discovery of new behaviors by keeping low-covered inputs back in the pool. Precisely, the coverage d_s of an initial state s is defined as the following joint probability density:

$$d_s = p(s_0) \times \prod_{t=0}^{M-2} \frac{p(s_t, s_{t+1})}{p(s_t)} \quad (1)$$

where $\{s_t\}_{t\in 0..M\text{-}1}$ is the state sequence returned by $MDP(s, \pi)$. Pang et al. [18] propose to estimate the probability densities $p(.)$ with two Gaussian Mixture Models (GMMs) [7], dealing with single states s_t and concatenated states s_t, s_{t+1}, respectively. A common approach to approximate the parameters of GMMs is Expectation Maximization [2]; however, this method requires their re-computation after every test case execution. A key feature of MDPFuzz is *DynEM*, an algorithm that dynamically updates the current parameters at a constant cost. For further details, interested readers can refer to the original work [18].

2.2.2 *Method.*

Algorithm 1 illustrates the high-level working of MDPFuzz. The method inputs a number of initial states N, the model under test π as well as three parameters: the threshold to update the GMMs (with DynEM) τ, the update weight γ and K, the number of components of the GMMs.

Initialization. This stage is described at lines 1-15, and aims at feeding the pool with inputs and their sensitivities and coverage. It begins by initializing the parameters of the GMMs and the dynamic update algorithm DynEM. Then, N initial states are randomly sampled in the observation space of the MDP. MDPFuzz iterates the states to compute and record their respective sensitivity and coverage. When the latter is lower than τ, the parameters of the GMMs are updated with DynEM.

Fuzzing. This second and main stage of the framework starts at line 17. It consists of a loop in which three key steps are performed. First, an input s is selected from the pool and mutated (lines 19-21). As mentioned above, the selection is biased with respect to the sensitivity of the states. Precisely, the probability of selecting the input s_i of sensitivity e_i is $\frac{e_i}{\sum_{i=0}^{n-1} e_i}$, where n is the current size of the pool. Then, π is tested with the mutated input (line 22). The latter is saved in the solution set in case of a failure. In the third step, the mutated input is added to the pool if (1) it induces a lower reward than its original counterpart or (2) its coverage is lower than τ. As in the previous stage, in such a case the parameters of the GMMs are updated with DynEM.

In their work, Pang et al. compare MDPFuzz to what we refer as the "Fuzzer", which is an ablated version that does not compute coverage. As such, this framework only measures sensitivities and maintains its pool with (1), i.e., mutated inputs that provoke lower rewarded executions than their original counterparts. Note that by being coverage-free, Fuzzer is not parameterized.

Algorithm 1 MDPFuzz

```
Input: $N$: initial size of the pool, $\pi$: model under test, $\tau$: coverage
       threshold, $\gamma$: update weight, $K$: number of components for the
       2 GMMs
Output: $R$: crash-revealing initial states
 1: $Pool, R \leftarrow \emptyset, \emptyset$
 2: ▷ initialize GMMs' and DynEM's parameters ◁
 3: $GMM^s, GMM^c \leftarrow$ init_DynEM$(K, \gamma)$
 4: ▷ sample N initial states to feed the pool ◁
 5: for $i = 1$ to $N$ do
 6:     $s_i \leftarrow$ sample_initial_state$()$
 7:     $e_i \leftarrow$ sensitivity$(s_i)$
 8:     $r_i, \{s_t\}_{t\in 0..M\text{-}1} \leftarrow$ MDP$(s_i, \pi)$
 9:     $d_i \leftarrow$ coverage$(\{s_t\}_{t\in 0..M\text{-}1}, GMM^s, GMM^c)$
10:     ▷ update GMMs' parameters if low coverage ◁
11:     if $d_i < \tau$ then
12:         $GMM^s, GMM^c \leftarrow$ DynEM$(\{s_t\}_{t\in 0..M\text{-}1}, GMM^s, GMM^c)$
13:     end if
14:     $Pool.add(s_i, r_i, e_i, d_i)$   ▷ feed the pool
15: end for
16: ▷ fuzz until test budget is consumed ◁
17: while test budget do
18:     ▷ sensitivity-biased random selection ◁
19:     $s, r \leftarrow Pool.select()$
20:     ▷ mutate and execute the state ◁
21:     $s' \leftarrow$ mutate$(s)$
22:     $r', \{s_t\}_{t\in 0..M\text{-}1} \leftarrow$ MDP$(s', \pi)$
23:     $d' \leftarrow$ coverage$(\{s_t\}_{t\in 0..M\text{-}1}, GMM^s, GMM^c)$
24:     ▷ detect and record fault ◁
25:     if crash$(\{s_t\}_{t\in 0..M\text{-}1})$ then
26:         $R.add(s')$
27:         ▷ maintain the pool if lower reward or low coverage ◁
28:     else if $r' < r$ or $d' < \tau$ then
29:         $e' \leftarrow$ sensitivity$(s')$
30:         $Pool.add(s', r', e', d')$
31:         ▷ dynamic update of GMMs' parameters ◁
32:         $GMM^s, GMM^c \leftarrow$ DynEM$(\{s_t\}_{t\in 0..M\text{-}1}, GMM^s, GMM^c)$
33:     end if
34: end while
35: return $R$
```

3 RESEARCH QUESTIONS

Our primary goal is to verify the ability of MDPFuzz to find functional faults in the models under test, both with the original implementation (provided by Pang et al.) and with our re-implementation. As detailed in introduction, we followed this two-step methodology after experiencing issues with the original code.

Our secondary goal is to investigate the effects of the parameters K, τ and γ on MDPFuzz's performance. In short, this work answers the following research questions:

RQ1 Reproduction study: can we reproduce the results with the original implementation?

RQ2 Replication study: can our replicate discover faults in the original use cases, in other use cases? How does it compare to Random Testing and Fuzzer (also re-implemented)?

RQ3 Parameter analysis: is MDPFuzz sensitive to its parameters? Can we find optimal configurations for fault detection?

Answering the two first research questions will let us conclude on the general fault discovery ability of MDPFuzz and Fuzzer. In particular, the replicate, i.e., RQ2, will compare these techniques against Random Testing, which has not been done in the original evaluation. The last question examines the sensitivity of MDPFuzz to its parameters, and how they affect coverage guidance for fault detection. Besides, it will let us check that the original evaluation as well as our reproduction study (RQ1, in Section 4) have not been biased by weak parameters.

From now on, we distinguish the results of the reproduction study (using the original implementation) from the replication using our re-implementation, (cf. Section 5) by applying the suffix "-O" and "-R", respectively.

4 REPRODUCTION STUDY

In the following, we explain our methodology to attempt to reproduce the results of Pang et al. with the original implementation. Indeed, we encountered difficulties in using the implementation, and we had to add and change some parts of the code.

The first subsection documents the two most significant differences we found between the implementation and the original specification of MDPFuzz. The second presents the findings of our thorough code review where we cover all the bugs identified as well as the missing or inconsistent elements that prevented us from running the experiments. The third subsection details the subsequent additions and modifications we had to perform to successfully execute the experiments, comments on our results and their deviations from the original results.

4.1 Algorithmic Differences

The two most significant discrepancies compared to the specification of MDPFuzz defined in the original publication concern coverage computation and parameter update (DynEM). In the following, we present those changes and the authors' explanation.

Coverage computation. The authors consider the *probability densities* (computed by the GMMs) as *probabilities*. Indeed, instead of implementing the definition of Equation (1), the authors have implemented the following definition:

$$d_s = p(s_0) \times \prod_{t=0}^{M-2} \boldsymbol{min}(\mathbf{1}, \frac{p(s_t, s_{t+1})}{p(s_t)})$$

We think that the introduction of the *min* function corresponds to a wanted simplification (i.e., real variables in $[0, 1]$).

GMMs' parameters update. DynEM updates the parameters of the GMMs with all the states of a given sequence [18]. However, we found that the mixture models for the single and concatenated states are updated with the first state and 10% of the states, respectively. The authors argued that it is a good trade-off for efficiency and accuracy since the GMMs aim at approximating correlations between the states [20]. They further detailed that the mixture models are not powerful enough to estimate long sequences, which could lead to overestimation.

4.2 Code Review

In the following, we present the unintentional bugs we found in the original code and the missing or inconsistent elements that made us unable to perform the original evaluation without addressing them (i.e., changing the code).

Bugs in Coop-Navi. We found two severe bugs in the implementation of the Coop-Navi case study. First, the pool is fed with *references* of the initial states instead of copies. Because the state of the MDP evolves during test execution, the pool ends up being fed with *final* states, i.e., a failure or solved state. In any case, mutating such states is likely to produce very short executions. The second issue concerns the computation of inputs' sensitivity during the sampling phase (described in Subsection 2.2). We found that, even though the perturbed inputs are *computed*, they are not then *used* to reset the MDP. In other words, if the first flaw were fixed, then the sensitivities would always be null (same input re-executed).

Flawed mutation in Bipedal Walker. The instructions that ensure that the mutated and input states are different are put in comments (i.e., they are not executed). We empirically computed the probability of null mutation as 20%.

Missing Fuzzer baselines. The Fuzzer baseline is only implemented for the Bipedal Walker case study, in which coverages are still computed but not used, thus artificially decreasing its efficiency. Such biased implementation follows what Pang et al. suggested to do when we asked why the aforementioned baselines were missing: to simply change the condition to maintain the pool during fuzzing (i.e., line 28 in Algorithm 1).

Different initialization requirements. The initialization procedure implemented requires a number of inputs to sample, as specified in Algorithm 1, line 5. However, the one defined in the evaluation section of the original work consists of 2 hours of sampling. We asked the authors for clarification and they confirmed that the sampling was performed for 2 hours. They didn't explain to us why it is not the case in the code.

Different Hyperparameters. While the original publication does not mention the parameters of MDPFuzz evaluated, in a preprint[2] their values are $K = 10$ and $\tau = \gamma = 0.01$. Surprisingly though, the code implements different values (summarized in Table 1). As before, we sought clarification from the authors and they confirmed that they used the values $K = 10$, $\tau = 0.01$ and $\gamma = 0.01$. They added that the code has been later modified for testing/debugging purposes.

Besides, we note that the underlying MDPs of the Bipedal Walker and CARLA use cases include randomness and are thus stochastic. To that regard, the authors could have decided to reset the seed of the random generator for Bipedal Walker, while – as far as we know – there is no way to entirely control the randomness of the CARLA simulator[3]. If considering stochastic $MDP(.)$ is not a bug, it might lead to unstable results and disables replicable test case execution. Hence, in our replication study (RQ2, in Section 5), we fix the random parameters of the underlying MDP (i.e., given an input i, $MDP(i, \pi)$ returns the same results).

[2]Available here: https://arxiv.org/abs/2112.02807v3.

[3]In particular, the next direction of the vehicles when they enter an intersection is randomly chosen and cannot be controlled.

Use Case	K	τ	γ
ACAS Xu	5	0.1	0.1
Bipedal Walker	10	0.02	0.1
CARLA	10	0.1	0.1
Coop Navi	1	0.01	0.1
Paper [18]	10	0.01	0.01

Table 1: Parameters of MDPFuzz found in original implementation for each use case. The last row shows the values confirmed by the authors.

4.3 Reproduction Results

Implemented Changes. To account for the authors' answers and the bugs discovered, we forked the original code and made the following changes. First, we added logging methods, since no script to analyze the initial logs is provided. Second, we set back the GMM's and DynEM's parameters to the values confirmed by the authors [20], namely $K = 10$, $\tau = 0.01$, and $\gamma = 0.01$. Similarly, we changed the initialization functions such that the fuzzers sample for 2 hours. Third, we added the Fuzzer for all the use cases, by copy-pasting the exact code of MDPFuzz and putting in comments all the instructions related to coverage guidance and computation. Eventually, we fixed the three bugs detailed in Subsection 4.2. As a last note, even though we expected the original implementation to contain all the material to reproduce Pang et al.' experiments, we emphasize that the artefacts provided are labelled as "available" (and not, for instance, as "reproducible") and note that Pang et al. kindly answered our interrogations.

Experimental Design. To assess how the bugs threaten the validity of the original evaluation, we consider both the unmodified and the modified versions of MDPFuzz-O and Fuzzer-O in the reproduction results. Precisely, for Bipedal Walker, we evaluate the untouched mutation function and the fixed one (later referred as "Mutation"), for which null mutation is not possible anymore. For the Coop Navi case study, we compare three versions: original, "Sampling" (where the pools are fed with the initial states) and "Fixed" (the two bugs are addressed). We did not study the fourth case, which would isolate the second issue, because of fatal errors. Indeed, not perturbing initial states systematically implies null sensitivity; this corner case is not handled by the implementation.

We follow the experimental procedure defined in the original paper and account for the randomness effect by running each execution three times. All the material of the experiments is available online[4]. We ran the experiments on a Linux machine (Ubuntu 22.04.3 LTS) equipped with an AMD Ryzen 9 3950X 16-Core processor and 32GB of RAM. Due to long execution times, we doubled the sampling and fuzzing times for the CARLA use case compared to the original paper.

Figure 1 reports on the number of faults found during fuzzing, in a similar fashion as the original publication. To improve readability, we show the results of each use case in separate plots.

[4]https://anonymous.4open.science/r/9D82

Results. We find consistent results to the original evaluation for CARLA (third column of Figure 1) as well as Coop Navi when the latter is not entirely fixed (last column, bold and dashed lines). To that regard, we note that the number of test cases executed for the "fixed" version (dotted lines, same column) is less than half of the unfixed ones. This was expected since these flawed versions feed the fuzzers with terminal states. For all the other fixed use cases except CARLA, we observe that Fuzzer-O outperforms MDPFuzz-O, which is not the case in the original study. In particular, the long time spent in computing coverage prevents MDPFuzz-O from testing as many inputs as the Fuzzer-O. Precisely, as shown in Figure 1, for almost all the cases studied, Fuzzer-O needs less than 6 hours to perform the same number of test cases as MDPFuzz-O; the only exception being CARLA, for which it needs 16 hours (of 24 hours, i.e., two thirds of the testing time).

Furthermore, the GMMs only guide the search towards faults for CARLA; otherwise Fuzzer-O discovers more failures throughout testing. This can be seen in Figure 1, where the red lines are consistently above the blue ones. In Bipedal Walker, fixing the mutation operation does not impact the results.

Analysis. Even though MDPFuzz finds faults in the models under test, its lighter counterpart Fuzzer outperforms the former for 3 of the 4 (fixed) use cases studied. If MDPFuzz's performance for CARLA is consistent with the original results found by Pang et al., sharing the same conclusions for the two unfixed versions of Coop Navi indicates that the flaws were indeed present when Pang et al. evaluated their proposal[5]. Our results show that Fuzzer is more applicable and more efficient than MDPFuzz (by avoiding computation coverage). More importantly, we find that MDPFuzz's additional coverage guidance does not in general favor fault discovery. Indeed, for all the experiments except CARLA, the baseline detects more faults as soon as fuzzing starts.

We identify several reasons why we do not observe the same results between the two fuzzers as the original evaluation. First, note that different absolute numbers of faults are not abnormal, since Pang et al. define the testing budget in time, which makes experiments hardware dependent. Then, we suppose that the Fuzzer executes significantly more test cases since it does not compute coverages. Still, we recognize that it does not explain why we find MDPFuzz outperforming its unguided counterpart only for CARLA. Our hypothesis is that the authors evaluated the Fuzzer by executing the code of MDPFuzz with the line to maintain the pool with low-covered inputs put in comments (as they suggested during our email conversation [20]).

Conclusion. We confirm that MDPFuzz can detect faults in the decision models tested, with and without coverage guidance. However, in general we don't find that its additional coverage guidance leads to more faults. This observation is our main motivation to further our investigations and replicate the work. Our second motivation is the overall poor quality of Pang et al.' implementation. Beyond the bugs spotted and the lack of Fuzzer's implementation, the code design can be improved with more modularity and usability. Indeed, the use cases embed their fuzzer, instead of relying on

[5]One could have hoped the bugs to be introduced after the evaluation, as the authors acknowledged to have changed the code thereafter.

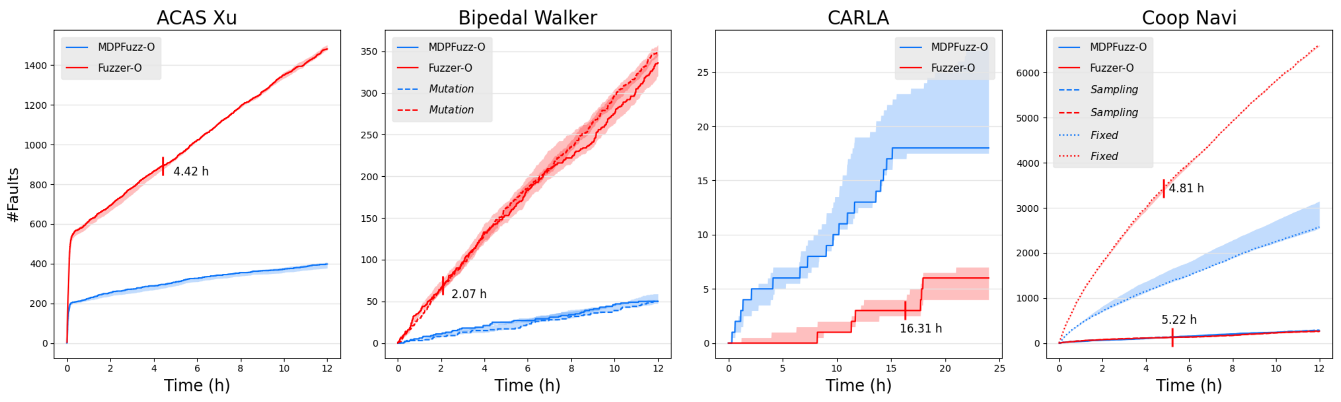

Figure 1: Summary of the reproduction results as the number of faults found over time. The lines show the median results over 3 executions, and the shaded areas draw the interquartile ranges (IQRs). The results of the threats to validity studied are plotted with dashed and dotted lines. The timestamps indicate the time by which Fuzzer-O tests as many states as MDPFuzz-O.

a single implementation. As such, it's not easy to adapt it to new use cases, limiting the adoption by researchers and practitioners and making comparisons to and extensions of MDPFuzz difficult.

Summary of RQ1 (Reproduction)

We identified mismatches between the original implementation and the algorithmic description. The provided MDPFuzz and the Fuzzer both find faults in the decision models under test. However, Fuzzer is significantly better than Pang et al.' findings, as it outperforms MDPFuzz in 3 of the 4 cases studied. Similarly, in general the coverage guidance does not drive testing towards more faults, which are at the odds of the original results.

5 REPLICABILITY STUDY

In this section, we present how we replicate the original work to answer RQ2 and RQ3. Through the replication process, we aim to clarify the doubts cast by the reproduction study, to gain insights into the applicability of MDPFuzz and to provide the policy testing research community with usable, modular artefacts.

5.1 Experimental Design

We re-implemented from scratch the algorithm proposed by Pang et al.. Note that we rigorously followed the pseudo-code specified in the paper and did not introduce any adaptations as done by the authors (cf. Section 4.1). We conducted experiments with the corrected versions of the original use cases and three new decision tasks (detailed below). We compared our replicate of MDPFuzz and Fuzzer (namely, “MDPFuzz-R” and “Fuzzer-R”) with Random Testing (RT), which acts as a baseline and allows us to assess the difficulty of the testing tasks. We measured the number of faults found during testing (as in Section 4) and analyzed the time distribution of the three methods.

5.1.1 Additional Use Cases. We extended the original evaluation with three commonly used decision-making environments [27] for Reinforcement Learning. They complement the study with diverse testing conditions, varying in complexity and sequence length, and observation spaces, including dimensions and subset types (i.e., $\mathbb{N}$ or $\mathbb{R}$).

Cart Pole. In this control problem, a pole is attached to the top of a cart that can move along a horizontal track. The goal is to keep the pole balanced by pushing the cart to the left and the right. The initial states specify cart's and pole's position and velocity.

Lunar Lander. The decision task consists of safely landing a spacecraft on the Moon. The starting position of the spacecraft is always at the top centre of the space and, likewise, the landing pad is always at the centre of the ground. The initial situations differ in the shape of the Moon (surrounding the landing pad) and the initial force applied to the space vehicle. The policy controls the main and orientation engines of the spacecraft. Note that, as for Bipedal Walker, we consider deterministic executions and thus fix the random seed of the simulator. As such, the input space is reduced to the initial force applied to the agent (i.e., the landscape is always the same).

Taxi. This problem depicts a grid world in which the policy picks up passengers and drop them off at their destinations [3]. As such, an initial situation defines the passenger's position and destination as well as the position of the taxi. At every step, the decision model has six possible actions: moving the taxi (going north, south, east or west) or interacting with the passenger (pick it up or drop it off). We enlarge the original version of the environment with a 18x13 map to disable the possibility to enumerate all MDP's possible states.

5.1.2 Mutation. For Cart Pole and Lunar Lander, we add a small random perturbation to the inputs. In Taxi, a mutation changes either the position of the car, the position of the passenger, or the destination.

5.1.3 Test Oracles. For Cart Pole, the policy fails if the absolute angle of the pole or the position of the cart exceeds threshold values. In Lunar Lander, a failure occurs if the lander moves outside the viewport or crashes into the ground. For the Taxi environment, we detect a fault in case of an invalid action (e.g., dropping the passenger while the taxi is still empty) or collision.

5.1.4 *Models Tested.*

For Lunar Lander, we test a policy of the freely available Stable-Baselines3 repository [21] (from which also originates the model tested in Bipedal Walker). For the Cart Pole and Taxi use cases, we train agents via Q-Learning [31].

5.1.5 *Implementation.*

We implement Lunar Lander and Taxi in the original code of Bipedal Walker. We use the Cart Pole version of gymnasium 0.29 [27] and implement the test executions with Gimitest [8], a recent tool for testing RL policies. We set the maximum number of time steps per simulation M to 400, 1000 and 200 for Cart Pole, Lunar Lander and Taxi, respectively. To improve the stability of the executions and to enable reproducibility of the experiments, we fix the random parameters of the underlying MDPs in Lunar Lander and Bipedal Walker. All the code of the experiments is freely available[6]. As for the fuzzers and RT, we have developed them in an independent Python package, also released online[7]. This tool enables the test of any policy with MDPFuzz, Fuzzer or RT for further research.

We ran the three methods with a budget of 5000 tests, and a sampling phase of 1000 iterations for the fuzzers. By fairness, we deduce from the testing budget of the fuzzers the number of test cases executed during sampling. We account for randomness effects by repeating all the executions with 5 random seeds (3 for CARLA). As before, we report the median results as well as the interquantile ranges.

5.2 RQ2: Fault Discovery

We use the same settings as in the reproduction study, i.e. MDPFuzz-R is configured with $K = 10$ and $\tau = \gamma = 0.01$.

Results. Figure 2 shows the evolution of the faults found during testing. Except for Lunar Lander, where the three methods perform equally, the fuzzers find more faults than Random Testing. We observe a significant difference between the fuzzers in ACAS Xu and CARLA. For ACAS Xu, Fuzzer-R improves on average MDPFuzz-R's results by 175% at the end of testing. Likewise, Fuzzer-R finds 33% more faults than MDPFuzz-R in CARLA.

The fuzzing techniques also differ in terms of total execution times. Figure 3 shows the running time of the three methods. For each case study, it details the total, testing (π-environment interactions) and coverage times (for MDPFuzz-R). We can see that while Fuzzer-R and RT spend most of their time executing test cases, MDPFuzz-R suffers from an overhead which is significant for all the cases studied except CARLA. For these cases, the total times increase more than 100% compared to Fuzzer-R, meaning that MDPFuzz-R actually spends more time computing coverage than testing the models. Besides, Fuzzer-R seems to be very sensitive to randomness in ACAS Xu, as denoted by the large shaded areas in Figure 2 (first column). Also, for this use case we measure a substantial overhead for Random Testing. We looked closely at the data and found out that the latter was caused by the input generation function (borrowed from the original implementation), which requires more time than mutating inputs.

[6]https://anonymous.4open.science/r/C882
[7]https://anonymous.4open.science/r/622C

Analysis. The results of our replicate confirm the ability of the fuzzers to find faults. Indeed, except for Lunar Lander, the fuzzers beat the random testing baseline, highlighting thus their relevance as dedicated techniques for testing policies. Likewise, as found previously, we observe a better efficiency of the Fuzzer compared to MDPFuzz. This was shown in the reproduction study with more test cases for the same time; here with lesser time to perform the same number of executions. Though, we note that the coverage overhead is not always significant, as shown by the CARLA use case, where executing a test case is longer than coveraging its state sequence. However, if in the reproduction study the coverage overhead was offset by better MDPFuzz's performance (third column in Figure 1), here Fuzzer-R reveals systematically more faults.

We further analyze the results for Lunar Lander, since the similarities in the performance of the fuzzers and the random baseline might indicate a possible limitation of the work proposed. We investigated the results obtained by closer analysing the fault distribution in the input space. Our hypothesis was that a draw with RT is likely to occur if the faults are uniformly distributed. To answer our assumption, we thoroughly test the policy with 4.10^6 evenly sampled inputs. Figure 4 shows the fault distribution in the input space ($[-1000, 1000]^2$). We can see that the faults are not uniformly distributed. Therefore, we would have expected the fuzzers to outperform RT by driving the search towards dense regions (such as the lower right corner in Figure 4). While a deeper investigation is out of the scope of this paper, these results reveal a limitation of the mutation-based solution space exploration of the fuzzers. In any case, we stress that this use case is by far the most difficult testing task, for which π has a failure rate of approximately of 0.5%.

Summary of RQ2 (Replication)

By outperforming the random testing baseline in 6 of the 7 cases studied, MDPFuzz and Fuzzer show their relevance as testing approaches tailored to policies. In particular, our replicate further confirms Fuzzer's high efficiency (already found in RQ1), with no significant method overhead. However, while the reproduction study reveals some usefulness of the coverage guidance (e.g., CARLA), we find that the Fuzzer always finds at least as many failures as MDPFuzz. As such, we conclude that the coverage guidance proposed by Pang et al., the key feature of MDPFuzz, is not needed.

5.3 RQ3: Parameter Analysis

This last research question aims at investigating the impact of the parameters K, τ and γ on MDPFuzz. In particular, we wonder about its potential sensitivity and whether we can identify a fault-driving configuration which would justify coverage guidance. Besides, this study will reduce biases that might stem from parameter selection in the previous experiments.

Unfortunately, there is no straightforward way to define them. We thus explore different values around the ones used so far. To make the study tractable, we first isolate the parameter τ, since we suspect a little to no impact. Indeed, we observed that most of the coverage values are either close to zero or infinity, but not spread

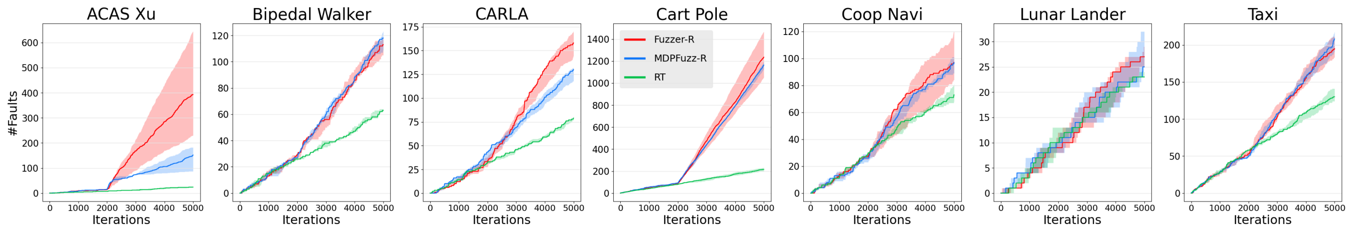

Figure 2: Evolution of the number of faults found with the replicate (Fuzzer-R and MDPFuzz-R) and Random Testing (RT). The lines show the median results over 5 executions (3 for CARLA), with their relative IQRs as shaded areas.

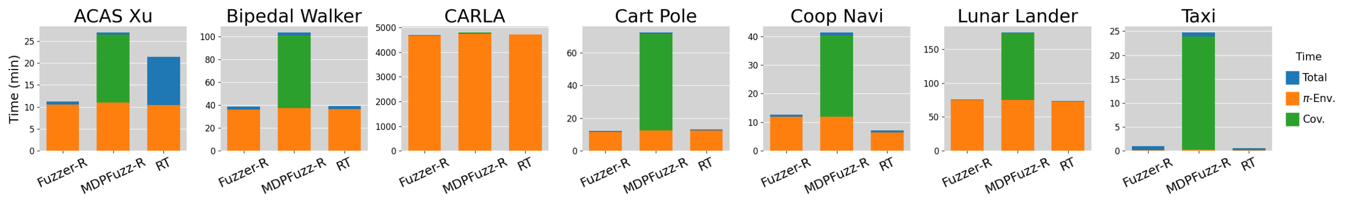

Figure 3: Running time of the replication study. We report the total, testing (π-environment interactions) and coverage times of the three methods for each use case. The bars show the median values of 5 executions (3 for CARLA), in minutes.

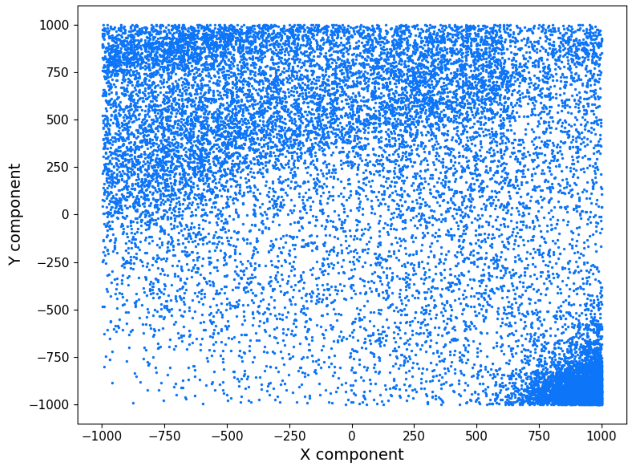

Figure 4: Fault distribution of π for the Lunar Lander use case, where the inputs describe the initial force applied to the agent. The horizontal and vertical axes show the components of the forces in the x and y directions, respectively. The input space ($[-1000, 1000]^2$) has been evenly sampled with 4.10^6 points.

over a wide range that would be strongly affected by a threshold value.

5.3.1 Impact of τ. Figure 5 shows the performance of our replicate (i.e., MDPFuzz-R) for three different τ values: [0.01, 0.1, 1.0]. Even though we observe some differences for ACAS Xu (first column), they are not substantial enough to be more deeply explored. Besides, we note that we do not aim at fine-tuning the parameters but, rather, at evaluating their effects. Therefore, we conclude that the exact value of τ is not critical, as long as the latter distinguishes high coverage values from small ones. We thus decided to keep the same coverage threshold as before, i.e. $\tau = 0.01$.

5.3.2 Impact of K and γ. We investigate a total 20 (K, γ) configurations, with $K \in [6, 8, 10, 12, 14]$ and $\gamma \in [0.05, 0.1, 0.015, 0.2]$. Figure 6 summarizes our findings, where each column shows the results for a use case. Then, the rows describe different K values, and every curve of a plot corresponds to the median number of faults found with a particular γ.

Results. We do not observe a best single configuration. In fact, in most cases, we do not find significant differences between them. Now, if we go into details, we see that (K, γ) pairs of smaller values seem to marginally increase the performance (e.g., 8-0.05 for ACAS Xu, 6-0.05 for CARLA and Coop Navi). Yet, these improvements depend on the case studied.

Summary of RQ3 (Parameter analysis)

Our analysis did not reveal significant sensitivity of MDPFuzz to its parameters. Similarly, we did not find a best single configuration for MDPFuzz that would justify its use of GMMs (and DynEM) as a guidance mechanism. We thus question the need of guiding the fuzz framework proposed by Pang et al., despite their original findings.

6 THREATS TO VALIDITY

In the following, we discuss the threats of our methodology and experimental evaluation.

Internal Threats. We acknowledge that our re-implementation might be biased by the reproduction study. We decided to take that risk because we wanted to include the original use cases in our

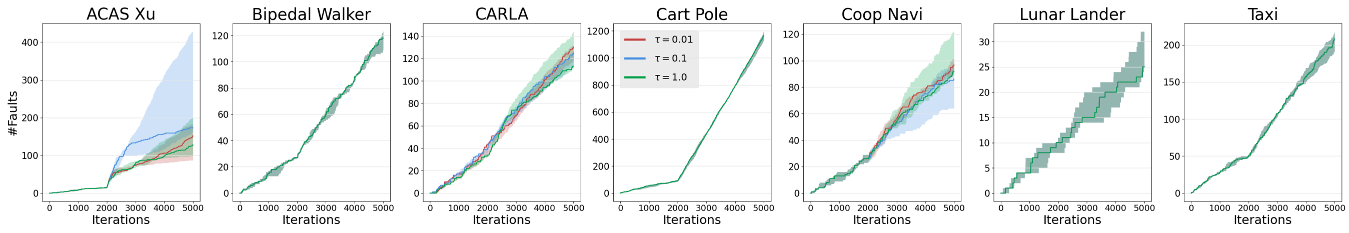

Figure 5: Analysis of the parameter τ on the ability of MDPFuzz-R to find faults in the models under test. We use the same settings for K and γ, and explore $\tau \in [0.01, 0.1, 1.0]$ (red, blue, green). Each plot details the results of a use case. The lines are the median values of 5 executions (3 for CARLA) and the shaded areas represent the IQRs.

evaluation. As explained in Subsection 4.2, the implementation of the latter and MDPFuzz are deeply intricate (especially for CARLA), and we had thus to review the code to set apart the testing techniques from the use cases. Fortunately, this deep investigation let us identify bugs as well as changes in MDPFuzz (not mentioned in the original work). We mitigate the bias of this prior knowledge by strictly following the algorithms of the original paper and its supplemental material [18], without making the aforementioned changes (detailed in Subsection 4.1).

External threats. As most empirical works, our results are inherently bound to the cases studied. We mitigate the possible biases by evaluating our replicate with the original and new use cases. To that regard, we consider testing tasks of diverse nature (planning and system control), complexity and settings (observation spaces S and sequence lengths M). Moreover, we carefully include in our experiments a random testing baseline, to ensure the relevance of the policy testing methods to begin with. Similarly, we investigated uncovered configurations for MDPFuzz, to further confirm our conclusions. Yet, we recognize that one could have selected different parameters, models under test and/or use cases.

7 RELATED WORK

In recent years, reproducibility in Machine Learning (ML) has become a major research topic. For instance, in her keynote [19] at ICSE 2019, Joelle Pineau invited the Software Engineering community to build more reproducible, reusable, and robust ML-based software systems. Likewise, Henderson et al.[10] listed different research questions and factors, such as hyperparameters and programming languages, that may affect the generalization of results in the field of Reinforcement Learning (RL). Our work embraces this research requirement by being, to the best of our knowledge, the first replicability study in RL-based policy testing.

In 2023, Mazouni et al. [14] reviewed the validation and verification techniques for decision-making models, where these authors highlighted the need to explore the limitations of the current results. Broader studies of ML-based models such as [1, 25, 32] revealed the difficulty of guaranteeing reproducible studies in the field of testing learned-based policies for Markov Decision Processes (MDP).

Policy testing has been approached in several ways, like by using genetic algorithms to reveal faults [33], reinforcement learning [13, 29] or search-based techniques [26]. Similarly as MDP-Fuzz, Steinmetz et al. [23] and Eniser et al. [5] have proposed to use fuzzing techniques to generate test inputs. Besides, Li et al. [12] and Mazouni et al. [15] compare their approach with MDPFuzz. It shows that MDPFuzz had a deep impact on the community of scientists who want to find ways to detect faults in complex learned policies for MDP testing.

Several works have adopted MDPFuzz [9, 30] as their main testing approach. However, none of these approaches relies on the coverage guidance proposed in the original paper. Precisely, Wang et al. [30] optimize MDPFuzz's efficiency by aborting the execution of test cases whose state sequence is not diverse enough. Diversity is inferred at every "checkpoint" time step by a sequence diversity model, which is trained before fuzzing.

He et al. [9] propose a variant of MDPFuzz called CureFuzz, which switches the coverage model with a curiosity score, and balances novelty and fault discovery with a multi-objective input selection (instead of sensitivity). Anyhow, the analysis of current related work shows that MDPFuzz has played a strong role in guiding research towards interesting results in the field of MDP policy testing. It is thus crucial to replicate and reproduce its findings in order to ensure the significance of its results.

8 CONCLUSION

In this paper, we studied a black-box fuzz testing framework for models solving MDPs called MDPFuzz [18], through a two-step methodology. In the first step, we tried to faithfully reproduce the experiment results described in the paper, by using the available open-source Pang et al.' implementation. However, we found that (1) the coverage-free version of MDPFuzz, i.e., Fuzzer, was not implemented, and (2) the initialization function input a number of states N instead of a duration of 2-hours, thus encouraging us to adapt the code. Furthermore, we identified unintentional bugs, different values for the parameters of MDPFuzz (compared to the settings of the original evaluation) as well as significant differences with the MDPFuzz methodology specified in the paper. We contacted the authors who kindly replied and provided us with explanations regarding the issues found in the implementation. We eventually executed both the unfixed and fixed versions of the implementation in an attempt to reproduce the original findings and determined whether the bugs were present when Pang et al. evaluated their method.

On one hand, we confirmed the ability of MDPFuzz to discover faults in the models under test. On the other hand, we found that its ablated counterpart, Fuzzer, outperforms MDPFuzz in three of the four use cases studied while being less resource-demanding.

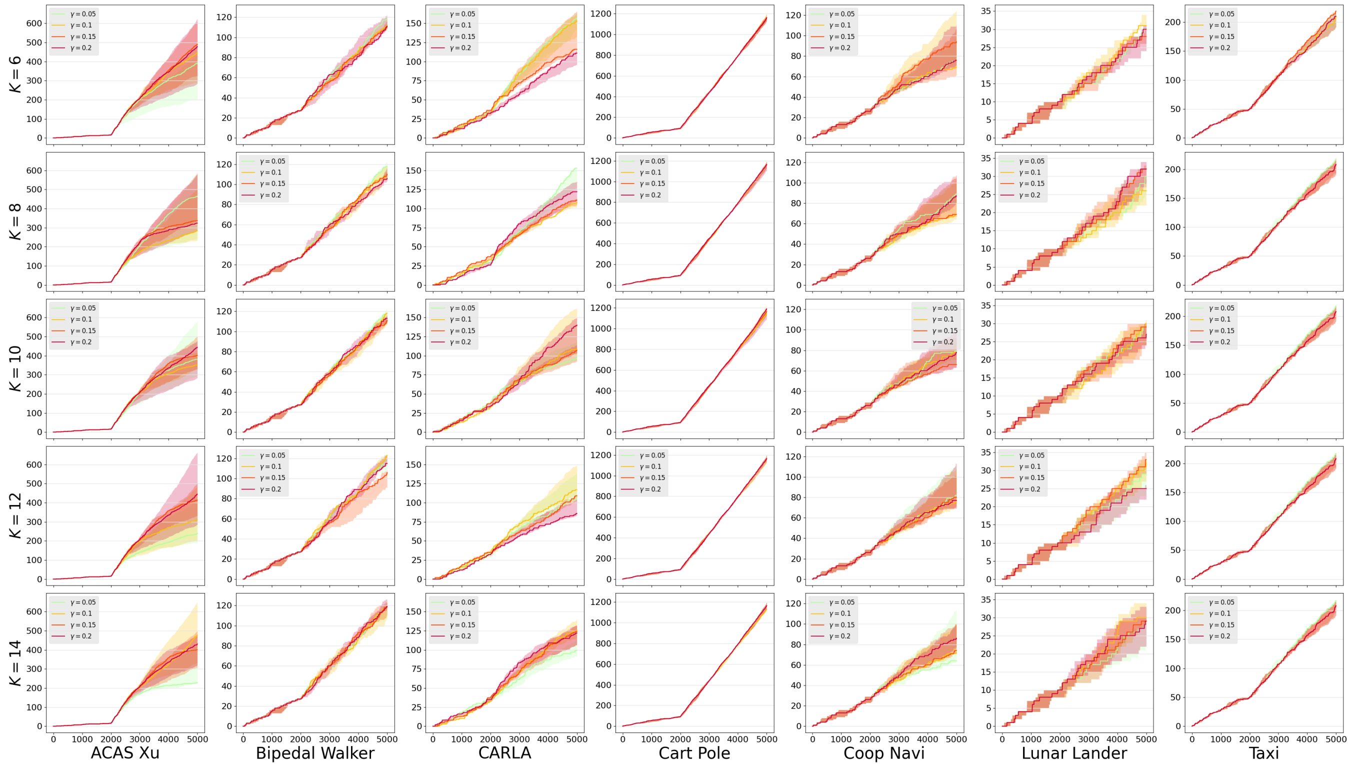

Figure 6: Parameter analysis of MDPFuzz with our replicate (i.e., MDPFuzz-R). The cases studied span over the columns, while the rows correspond to different K values ([6, 8, 10, 12, 14]). Then, each plot details the effect of the parameter γ ([0.05, 0.1, 0.015, 0.2]). We report the median values of 5 executions (3 for CARLA) and show their IQRs as shaded areas.

Not only these results are at odds with the original ones, but they also made us question the relevance of coverage-guidance, the key feature of MDPFuzz.

As such, in a second step, we replicate the fuzzers, i.e., implementing the algorithm described in the original paper. We strengthened the original evaluation with three new use cases and included a random testing baseline. Furthermore, we conducted a parameter analysis of MDPFuzz, to assess its sensitivity and possibly find out an optimal configuration for fault discovery. The results of our replication are aligned with the ones of the reproduction study. Precisely, we found that the Fuzzer systematically outperforms MDPFuzz, both in terms of fault detection and test efficiency (despite the parameter analysis).

Therefore, the main takeaway of this paper is that we do not recommend the coverage model of MDPFuzz. Yet, further research is needed to generalize this conclusion to coverage guidance for fuzz-based frameworks. Instead, we recommend addressing fault discovery with simpler approaches like Fuzzer, which have little computation overhead. The second takeaway of this work is about replicability studies. Indeed, our investigations highlighted the need to compare and reproduce the peers' work in the field of MDP policy testing. To that regard, we provide the policy testing community with re-usable artefacts, to foster the research efforts for fuzz-based testing.

9 DATA AVAILABILITY

All the material of two empirical studies conducted in this paper are publicly available online[8][9]. It includes the data and instructions to reproduce the experiments. The implementation of three methods is also available[10].

REFERENCES

[1] Anthony Corso, Robert Moss, Mark Koren, Ritchie Lee, and Mykel Kochenderfer. 2022. A Survey of Algorithms for Black-Box Safety Validation of Cyber-Physical Systems. *J. Artif. Int. Res.* (2022). https://doi.org/10.1613/jair.1.12716

[2] A. P. Dempster, N. M. Laird, and D. B. Rubin. 1977. Maximum Likelihood from Incomplete Data Via the EM Algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)* 39, 1 (1977), 1–22. https://doi.org/10.1111/j.2517-6161.1977.tb01600.x arXiv:https://rss.onlinelibrary.wiley.com/doi/pdf/10.1111/j.2517-6161.1977.tb01600.x

[3] Thomas G Dietterich. 2000. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of artificial intelligence research* 13 (2000), 227–303.

[4] Jan Eisenhut, Álvaro Torralba, Maria Christakis, and Jörg Hoffmann. 2023. Automatic Metamorphic Test Oracles for Action-Policy Testing. In *International Conference on Automated Planning and Scheduling*. https://api.semanticscholar.org/CorpusID:259638200

[5] Hasan Ferit Eniser, Timo P. Gros, Valentin Wüstholz, Jörg Hoffmann, and Maria Christakis. 2022. Metamorphic Relations via Relaxations: An Approach to Obtain Oracles for Action-Policy Testing. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. https://doi.org/10.1145/3533767.3534392

[8] https://anonymous.4open.science/r/9D82
[9] https://anonymous.4open.science/r/C882
[10] https://anonymous.4open.science/r/622C

[6] Keith Frankish and William M. Ramsey (Eds.). 2014. *The Cambridge Handbook of Artificial Intelligence.* Cambridge University Press, Cambridge, UK.

[7] D. N. Geary. 2018. Mixture Models: Inference and Applications to Clustering. *Journal of the Royal Statistical Society Series A: Statistics in Society* 152, 1 (12 2018), 126–127. https://doi.org/10.2307/2982840 arXiv:https://academic.oup.com/jrsssa/article-pdf/152/1/126/49758778/jrsssa_152_1_126.pdf

[8] Dennis Gross, Quentin Mazouni, and Helge Spieker. 2024. Gimitest: A framework for evaluating reinforcement learning agents. https://github.com/DennisGross/gimitest. GitHub repository.

[9] J. He, Z. Yang, J. Shi, C. Yang, K. Kim, B. Xu, X. Zhou, and D. Lo. 2024. Curiosity-Driven Testing for Sequential Decision-Making Process. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE).* IEEE Computer Society, Los Alamitos, CA, USA, 949–949. https://doi.ieeecomputersociety.org/

[10] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. 2018. Deep reinforcement learning that matters. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 32.

[11] Rushang Karia and Siddharth Srivastava. 2020. Learning Generalized Relational Heuristic Networks for Model-Agnostic Planning. *CoRR* (2020).

[12] Z. Li, X. Wu, D. Zhu, M. Cheng, S. Chen, F. Zhang, X. Xie, L. Ma, and J. Zhao. 2023. Generative Model-Based Testing on Decision-Making Policies. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE Computer Society, Los Alamitos, CA, USA, 243–254. https://doi.org/10.1109/ASE56229.2023.00153

[13] Chengjie Lu, Yize Shi, Huihui Zhang, Man Zhang, Tiexin Wang, Tao Yue, and Shaukat Ali. 2023. Learning Configurations of Operating Environment of Autonomous Vehicles to Maximize their Collisions. *IEEE Transactions on Software Engineering* (2023). https://doi.org/10.1109/TSE.2022.3150788

[14] Quentin Mazouni, Helge Spieker, Arnaud Gotlieb, and Mathieu Acher. 2023. A Review of Validation and Verification of Neural Network-based Policies for Sequential Decision Making. In *Rencontres des Jeunes Chercheurs en Intelligence Artificielle (RJCIA).* https://pfia23.icube.unistra.fr/conferences/rjcia/Actes/RJCIA2023_paper_5.pdf

[15] Quentin Mazouni, Helge Spieker, Arnaud Gotlieb, and Mathieu Acher. 2024. Testing for Fault Diversity in Reinforcement Learning. In *2024 IEEE/ACM International Conference on Automation of Software Test (AST).* https://doi.org/10.1145/3644032.3644458

[16] Barton P. Miller, Lars Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33, 12 (dec 1990), 32–44. https://doi.org/10.1145/96267.96279

[17] Hai Nguyen and Hung La. 2019. Review of Deep Reinforcement Learning for Robot Manipulation. In *2019 Third IEEE International Conference on Robotic Computing (IRC).* 590–595. https://doi.org/10.1109/IRC.2019.00120

[18] Qi Pang, Yuanyuan Yuan, and Shuai Wang. 2022. MDPFuzz: Testing Models Solving Markov Decision Processes. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis.* https://doi.org/10.1145/3533767.3534388

[19] Joelle Pineau. ICSE'19 Keynote. Building Reproducible, Reusable, and Robust Machine Learning Software. https://2019.icse-conferences.org/details/icse-2019-Plenary-Sessions/20/Building-Reproducible-Reusable-and-Robust-Machine-Learning-Software.

[20] Qi Pang. 2023,2024. Inquiry about MDPFuzz. Personal communication. Email correspondence.

[21] Antonin Raffin. 2020. RL Baselines3 Zoo. https://github.com/DLR-RM/rl-baselines3-zoo.

[22] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* (2018). https://doi.org/10.1126/science.aar6404

[23] Marcel Steinmetz, Daniel Fišer, Hasan Ferit Eniser, Patrick Ferber, Timo P. Gros, Philippe Heim, Daniel Höller, Xandra Schuler, Valentin Wüstholz, Maria Christakis, and Jörg Hoffmann. 2022. Debugging a Policy: Automatic Action-Policy Testing in AI Planning. *Proceedings of the International Conference on Automated Planning and Scheduling* (2022). https://doi.org/10.1609/icaps.v32i1.19820

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

[25] Florian Tambon, Gabriel Laberge, Le An, Amin Nikanjam, Paulina Stevia Nouwou Mindom, Yann Pequignot, Foutse Khomh, Giulio Antoniol, Ettore Merlo, and François Laviolette. 2022. How to Certify Machine Learning Based Safety-Critical Systems? A Systematic Literature Review. *Automated Software Engineering* (2022). https://doi.org/10.1007/s10515-022-00337-x

[26] Martin Tappler, Filip Cano Córdoba, Bernhard K. Aichernig, and Bettina Könighofer. 2022. Search-Based Testing of Reinforcement Learning. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, Luc De Raedt (Ed.). ijcai.org, 503–510. https://doi.org/10.24963/IJCAI.2022/72

[27] Mark Towers, Jordan K. Terry, Ariel Kwiatkowski, John U. Balis, Gianluca de Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Arjun KG, Markus Krimmel, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Andrew Tan Jin Shen, and Omar G. Younis. 2023. Gymnasium. https://doi.org/10.5281/zenodo.8127026

[28] Sam Toyer, Sylvie Thiébaux, Felipe Trevizan, and Lexing Xie. 2020. Asnets: Deep learning for generalised planning. *Journal of Artificial Intelligence Research* 68 (2020).

[29] Fitash Ul Haq, Donghwan Shin, and Lionel C. Briand. 2023. Many-Objective Reinforcement Learning for Online Testing of DNN-Enabled Systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE).* 1814–1826. https://doi.org/10.1109/ICSE48619.2023.00155

[30] K. Wang, Y. Wang, J. Wang, and Q. Wang. 2023. Fuzzing with Sequence Diversity Inference for Sequential Decision-making Model Testing. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE).* IEEE Computer Society, Los Alamitos, CA, USA, 706–717. https://doi.org/10.1109/ISSRE59848.2023.00041

[31] Christopher J. C. H. Watkins and Peter Dayan. 1992. Q-learning. *Machine Learning* 8, 3 (01 May 1992), 279–292. https://doi.org/10.1007/BF00992698

[32] Jin Zhang and Jingyue Li. 2020. Testing and verification of neural-network-based safety-critical control software: A systematic literature review. *Information and Software Technology* (2020). https://doi.org/10.1016/j.infsof.2020.106296

[33] Amirhossein Zolfagharian, Manel Abdellatif, Lionel C. Briand, Mojtaba Bagherzadeh, and Ramesh S. 2023. A Search-Based Testing Approach for Deep Reinforcement Learning Agents. *IEEE Transactions on Software Engineering* 49, 7 (2023), 3715–3735. https://doi.org/10.1109/TSE.2023.3269804

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009