Testing Multi-tenant Applications using Fuzzing and Reinforcement Learning

Ciprian Paduraru*
ciprian.paduraru@fmi.unibuc.ro
Dept. of Computer Science,
University of Bucharest
Romania

Alin Stefanescu*
alin@fmi.unibuc.ro
Dept. of Computer Science,
University of Bucharest
Romania

Bogdan Ghimis*
bogdan.ghimis@fmi.unibuc.ro
Dept. of Computer Science,
University of Bucharest
Romania

ABSTRACT

Testing cloud applications has recently gained in importance since many companies migrated their operations in the cloud. To optimise resources, cloud applications may serve several users at once in a so-called multi-tenant setting. We propose a new technique for testing multi-tenant applications using reinforcement learning combined with gray-box fuzzing techniques. A preliminary evaluation using a combination of fuzzing techniques and genetic algorithms is also provided.

CCS CONCEPTS

• **Security and privacy → Software and application security**;
• **Software and its engineering → Cloud computing**.

KEYWORDS

Multi-tenancy, fuzz testing, reinforcement learning, security, cloud

ACM Reference Format:
Ciprian Paduraru, Alin Stefanescu, and Bogdan Ghimis. 2020. Testing Multi-tenant Applications using Fuzzing and Reinforcement Learning. In *Proceedings of the 1st ACM SIGSOFT International Workshop on Languages and Tools for Next Generation Testing (LANGETI '20), November 8–9, 2020, Virtual, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3416504.3424333

1 INTRODUCTION

Cloud computing [13] has become a pervasive technology and major companies invested in cloud technologies. Amazon launched Amazon Web Services (AWS) in 2006, and other tech enterprises such as Microsoft, Google, and IBM followed in AWS footsteps.

Currently, the major players in this field developed their own services, but they still offer the cloud solutions: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS) [22].

SaaS allows end-users to connect seamlessly to the deployed software and focus on the main task by abstracting away the infrastructure and the technologies used. Some of the benefits of SaaS are that applications can be developed on a platform and run on third party infrastructure and, using SaaS, can cost less in terms of software licensing and infrastructure resources [22]. Some of the main drawbacks of SaaS are the data security and low performance compared to PaaS and IaaS [24].

Multi-Tenancy Architecture (MTA) [15] is an architectural design pattern that consists of an application that is shared among multiple groups of users called tenants. These tenants use the same functionality of the application on the same server independently of one another. Sharing resources having the same underlying platform and the same single code base can be an efficient way of distributing resources, but the service must be flexible enough in order to adapt to different tenant specifications. However, testing of MTA was not addressed too much in the academic literature.

Fuzz testing is a common technique for desktop-based applications. It is a general technique in which random inputs are sent to a test program to find bugs. Fuzz testing can be classified as black-box, white-box and gray-box [8, 9]. The difference between them is the degree of knowledge of the system. The most general testing procedure is black-box, which does not assume anything about the system and tries to find problematic inputs through random input. White-box testing presumes knowledge about the system and, by knowing specific details about it, one can find interesting inputs faster. Gray-box fuzz testing is a combination of the two.

Our contribution consists in a novel way to test multi-tenancy applications by using interprocess communication backed by fuzz testing that can be repeated using the same seeds values. Our methods are enhanced by artificial intelligence (AI) techniques that involve reinforcement learning (RL), genetic algorithms, and other heuristics.

The paper is structured as follows. In the next section, we present the related work in the multi-tenancy testing field. In Section 3, we propose reinforcement learning as a technique to improve fuzz testing in our context. In Section 4, we present our preliminary evaluation. Finally, in the last section, we discuss future work.

2 RELATED WORK

In [15] the authors have identified several architecture principles that can be used while developing an MTA application such as affinity (routing tenants to the same server of the application if state that can not be cached is involved), persistence design (regarding database sharing principles), performance isolation and Quality of Service (QoS), e.g, ensuring that tenants work within their Service Level Agreement (SLA), and UI or system customization.

Although there is literature on cloud testing we have not found many papers that focus specifically on MTA. In [6], the authors

*All the authors had an equal contribution to this paper.

LANGETI '20, November 8–9, 2020, Virtual, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8123-9/20/11…$15.00
https://doi.org/10.1145/3416504.3424333

presented a general view on how testing can occur in the cloud, specifying how should test types should be done when testing different cloud components like SaaS-oriented testing, online application-based testing and cloud-based application testing. The majority of cloud vendors delegate testing to third-parties that are available online all the time. This paper also provides a comparative view of internet-based software testing and cloud-based software testing.

Information protection isolation is very important in a multi-tenancy application since a security breach of one tenant may leak other tenant's data [12]. Because of this, MTA must prevent tenants from reading and writing from one another. As noted in [12] the traditional way of achieving this is to encrypt and sign tenant's data, with each tenant storing securely their keys. For testing purposes, this layer of security makes it harder to reason about determinism in the execution of tenants actions since one must use mock data that may not reflect the real-world settings.

The authors of [7] have presented SaaS multi-tenancy testing and partitioned the process in six steps ranging from component unit testing (black-box and white-box testing) to component integration testing and tenant-based continuous testing. The difference between SaaS multi-tenancy testing and multi-tenancy testing in general is that the former has components that are shared between tenants whereas, in the latter, the tenants use the same application. One important challenge specified in the paper represents the scalability testing of a SaaS application.

MTA application testing was studied in [26], where the authors proposed a combinatorial testing algorithm, called Adaptive Reasoning (AR). They tested the algorithm on a multi-tenant application running on the SaaS platform offered from Google App Engine, where tenants could write code for their applications that was stored in a database. Other tenants could use their code and so they achieved code reusability. The AR algorithm searches for tenant configurations or applications that fails a test, and will discover other tenants that use the same configuration.

Some clients are reluctant in using SaaS because they think it is not safe, in particular MTA applications [25]. The authors of [25] have classified the main technical layers and possible security bugs that can arise when using an MTA application ranging from low-level vulnerabilities that can come from hardware processing and programming code to high-level vulnerabilities that can come from the interaction between the different components of the MTA application (web components, user data isolation, authentication and authorization).

Authors in [24] discussed the tenant-based resource allocation model for SaaS applications where VMs were allocated based on tenants needs instead of working with Amazon's default service *A*uto Scaling. They achieved this through monitoring the tenant's resources by implementing tenant-based authentication and data persistence as well as a tenant-based load balancer that efficiently mapped tenants to VMs.

Now we introduce the reinforcement learning (RL) model, which belongs to the machine learning (ML) field. RL works by defining an agent that is interacting with the environment and through a feedback loop is given a reward at each step that is dependent on its actions [14]. Through exploration of the available actions and the exploitation of previous positive actions, the agent learns a policy that tells it how to react in a given situation. By using RL, one must define the environment with which the agent will interact and it will eventually learn the best actions to perform. Training the agent usually occurs in episodes, which can end either at a fixed time or when the agent cannot take any more actions.

The usual way of testing MTA applications is to use system knowledge to generate different tenant configurations that, when run, would result in an unwanted behaviour. In this paper, we take a different route to MTA testing by using fuzz testing and reinforcement learning algorithms that generate tenant behaviours that would lead to finding faults in the MTA application or achieve better code coverage.

3 METHODS

3.1 Assumptions

We suppose that each application has a known fixed number of tenants: $U = \{u_1, u_2, ..., u_{|U|}\}$. Also, we consider that the application knows approximately the limits for the maximum of users allowed for the same tenant: $Max(u_i)$ for each $u_i \in U$. Both assumptions should not be too restrictive since each application knows its categories of users and the expected maximum number of serviced clients, according to the deployed infrastructure and application's layers. In future work, we will try to weaken these assumptions.

3.2 Testing Purposes

We follow the architectures and the concerns defined in the literature of the multi-tenancy domain and plan for research and implementation of test methods that cover two points of view:

(1) **Execution correctness (determinism) testing**. If a group of multiple tenants are using the application at the same time, are everybody outputs the same as if they would use the application sequentially? Are the shared resourced of the application (databases, shared files, etc) in the same state as if the application would be running sequentially?
(2) **Performance testing**. If a group of multiple tenants are accessing the application at the same time, are they still serviced without SLA/QoS violations?

3.3 High-level Description of Our Methods

The high-level plan is to train a multi-agent system that contains multiple Fuzzer instances that in turn simulates real users. The set of all these agents will be used to produce randomized new inputs, but with a probability distribution over time and actions (i.e., inputs produced) that are able to address the testing goals as close as possible.

We consider our techniques as gray-box since we have an abstract view of the input. In general, we can use a grammar to generate new inputs, a RNN if a textual input is required, or a GAN if the required input has a specific format.

For each tenant $u_i \in U$, our testing framework will train and output a Fuzzer model, $Fuzzer_{u_i}$, which can be used to produce smart test cases for the application under tests. The methods we aim at are based on guiding the fuzzing process using genetic algorithms [20], reinforcement learning methods [3], or other machine learning techniques [18] with two targets:

- Report as many issues as possible (either invalid correctness cases or performance issues).
- Cover the source code with different inputs as much as possible.

In the training or evaluation phase, at each test episode, for each tenant $u_i \in U$, our testing framework creates $Max(u_i)$ instances of $Fuzzer_{u_i}$ to simulate a real test case with multiple real users for the same tenant group, and multiple tenant groups at the same time. We denote such a Fuzzer instance as *Fuzzer agent*, and the set of all agents to be *FA*. Also, note that not only fuzzing the application with different inputs is important, but also the time when the fuzzing happens since we have multiple instances that run in parallel.

The role/output of a Fuzzer agent at time t is then Inp_{t,F_i} =Input produced by agent F_i at time t. $Inp_{t,Fi}$ is allowed to be $\emptyset$, which means that agent F_i decides to do nothing at time t.

Thus, our problem can be modeled as a decision-making problem: what should be the decision of an agent F_i at time t to test the software according to some established metrics? This kind of decision-making problem can be handled with AI techniques as the one mentioned above. In the next subsection, we go deeper into details on how to train, evaluate, and score these agents to make them learn how to produce new inputs.

3.4 Details about Model Training

When using a genetic algorithm or a machine learning approach, such as reinforcement learning, which seems like the most plausible one for our usecases, we need an *evaluation function* for our Fuzzer agents. The flow of the training process is sketched in Figure 1.

It is important to note at each point that all Fuzzer models inside a tenant group u_i are sharing the same weights. What differentiate them to produce different output is a latent input node, trainable, similar to Generative Adversarial Networks [11]. A critics eye could also say at this point that the tenants actions over the system are not independent and different subsets of actions between them could lead to different behaviors. That is true, but multi-agent systems are proven to work and converge even when considering them independent to reduce the training problem complexity [2], [21]. Also, training agents in parallel within a reinforcement learning environment was studied and proven to be effective in [17].

3.5 Evaluation Function

Eq. (1) shows the high-level scoring function of the input produced by a Fuzzer F_i at time t. It has three main components, detailed in the text below. The constants KS, KC, KR and float constants either fixed or learned from data that needs to be determined after a proper evaluation.

$$\begin{aligned}\mathbf{Eval}(Inp_{t,Fi}) = KS * \mathbf{ScoreSharedResources}(t, F_i) + \\ KC * \mathbf{CodeCoverage}(t, F_i) + \\ KR * \mathbf{ScoreRealIssuesDetected}(t, F_i)\end{aligned} \tag{1}$$

SharedResourcesScore component. Dynamic tainting [19], [23] can help tracking down what is the path of data from inputs to memory locations in the program. In the case of correctness checks, this can represent a part of the score function since having multiple

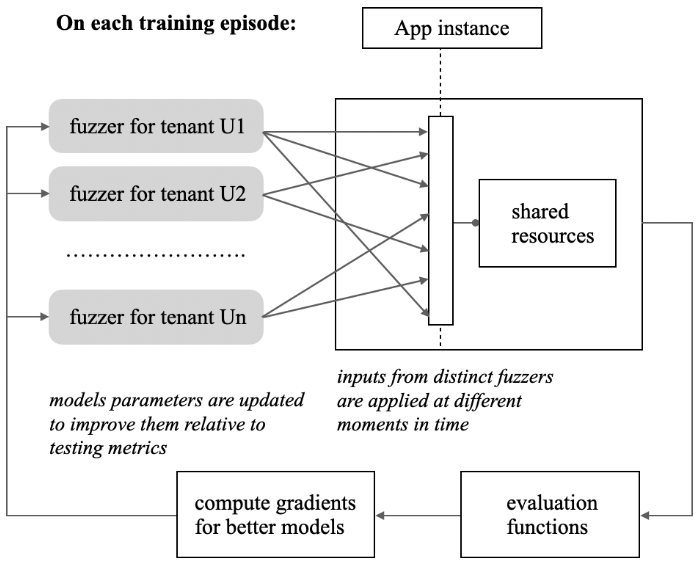
On each training episode:
App instance
fuzzer for tenant U1
fuzzer for tenant U2
....
fuzzer for tenant Un
shared resources
models parameters are updated to improve them relative to testing metrics
inputs from distinct fuzzers are applied at different moments in time
compute gradients for better models
evaluation functions

Figure 1: The flow of the training procedure for the Fuzzers in FA set inside our framework. In each training episode, we collect the feedback using existing policies, compute the gradients for producing better models, and finally improve the existing models. The arrows from Fuzzers to *App instance* represent some certain moments of time when they produce an input different than $\emptyset$.

fuzzer instances using the same memory locations at the same time increases the chance of seeing non-deterministic behaviours.

We define the amount of shared resources of the application used by a set of agents at any time t as in 2. By shared resources, we understand databases, files, buffers in memory, etc. is to be determined later through evaluation, if different kinds of resources need to be factored differently. For now, we measure the number of bytes used in common by the agents.

$$\begin{aligned}SharedResources(t, AgentsSet) = \\ bytes\Big(\bigcap_{agent \in AgentSet} ResourcesUsed(agent)\Big)\end{aligned} \tag{2}$$

Then, the score for the evaluation of the shared resources can be obtained by checking with how many bytes would the shared resources increase using the input produced by agent F_i at time t, Eq. (3).

$$\begin{aligned}ScoreSharedResources(t, F_i) = \\ SharedResources(t, FA) - \\ SharedResources(t, FA \setminus F_i).\end{aligned} \tag{3}$$

CodeCoverage component. Code coverage rewards for a training process can be represented by the number of new blocks that the Fuzzer touched in the run. It is important to always try new paths inside the source code, otherwise the method will get stuck in a local optimum ((4)).

$$CodeCoverage(t, F_i) = \text{number of new basic blocks of code detected by using the input from } F_i \text{ at time t.} \quad (4)$$

RealIssuesDetected component. Concrete issues detected represent the most important aspect of the testing process. The two scores above are only guiding the Fuzzer agents towards detecting these. On this evaluation metric, the formula is dividing depending on the testing target:

- **Performance testing.** We consider that each agent F_i, being part of tenant, has an agreed SLA/QoS. We will suppose that we know the SLA, but even if we do not, we can find an approximation of it if we will run the system on full load and detect how the application is handling the requests [5]. The motivation behind this scoring function is to promote inputs that violate the SLA/QoS between users as much as possible. We define the scoring function in this case for a given set of agents as in Eq. 5. The formula counts the total penalization for SLA violations of the agents in the set.

$$RealIssuesDetected(t, AgentsSet) = \sum_{agent \in AgentsSet} Penalization(max(0, ResponseTime(agent) - SLA(agent))) \quad (5)$$

  Finally, the scoring of the input produced by agent F_i at time t is described by Eq. 6. Basically, the formula used is based on how much penalization would the input of agent F_i would produce at the moment of time t, versus not using it at all.

$$ScoreRealIssuesDetected(t, F_i) = max(0, RealIssuesDetected(t, FA) - RealIssuesDetected(t, FA \setminus F_i)). \quad (6)$$

- **Correctness testing, determinism of outputs, and shared resources state.** $ScoreRealIssuesDetected(t, F_i)$ = the absolute sum of differences in bytes between the outputs buffers produced by the agents (only the ones expected to be deterministic) plus the shared resources contents difference for all each agent by running: **serially the actions up to time t, versus running in parallel the same inputs and up to the same time t**. The motivation of this scoring is to produce inputs that are trying to break the determinism of the actions as much as possible in the multi-agent environment.

3.6 Inference and Transfer Learning

Having a set of trained fuzzers FA, their owner can then use the latent variable that randomizes the output results to produce new input each time, but all of these are guaranteed to be as close as possible to the testing purposes goals. Transfer learning, i.e. transferring the fuzzer learned models to a different application or to the same application that is developed incrementally over time was studied in [3] and proved to be effective.

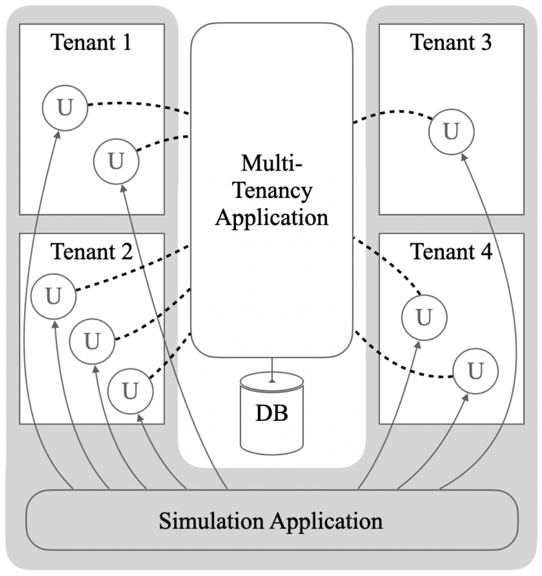

Figure 2: Figure describing the evaluated system using a multi-tenancy application and a simulator that controls the interaction between the users and the main application. The system was deployed as a Virtual Machine (VM) in Google Cloud Platform (GCP), the main application is written in Python using Flask framework for the web requests. The generic code that simulates the users is written in C++ and it is integrated with libFuzzer [16] to generate random input. Finally, the simulator is written in Python and interacts with the main application and spawns a program for each user that in turn will communicate directly with the MTA. For simplicity, the database is an instance of SQLite, containing a database as a file (but this is not a PoC limitation).

4 EVALUATION

To test our idea, we have implemented a simple web application, in which each tenant has access to individual data fields. Each user that belongs to a tenant can modify those fields by sending a request to the web application - see Figure 2.

Since we want to demonstrate a proof of concept (PoC), the data fields can contain either an integer or a string.

4.1 Simulator

The simulator is responsible to generate a random number of tenants and for each tenant to generate a random number of users. The simulator will also choose for each tenant one of the following possible modifications:

- Integer addition modulo n, in which the fuzzer for each user will use its first 4 bytes, add them together with the tenant's data, apply the modulo operator, and save the field to the database. One possible problem with this approach is integer overflow, when the product of the 4 bytes is bigger than *sizeof(int)*;
- Integer multiplication modulo n, same as integer addition;

- String addition with sorting, in which the fuzzer will use the first 4 bytes of the input, add them together with the tenant's field data from the database and then sort the input;
- String addition with reverse sorting;
- String addition with sorting and duplicates removal;
- String addition with reverse sorting and duplicates removal.

In order to better simulate the possible bugs that can arise in shared applications, we have used commutative functions that allow us to check if the system performed correctly or if there are bugs due to the order of operations in the system.

Timing interactions. The simulator is responsible for setting the interaction time for each user. In order to do this, it will choose the delay for each user to wait before interacting with the main application.

Deterministic behavior of the multi-tenancy application. To test the deterministic behaviour, we first log each user's actions by letting the simulation run for a given amount of time. Then, we check if the application's output is the same as when we run each action sequentially. If the result is different at the end of the simulation, since we have chosen commutative field operations, it must be that some other bug was present.

4.2 Users

We simulated the users with the same *generic_user* program that will use command-line arguments to have a different instantiation. The simulator will send each user the following arguments: *username*, *password*, *tenant_id*, *tenant_operation* and *user_action_time*.

The first two parameters are needed by the user to login to the application. The third one (*tenant_id*) is needed for logging purposes. The fourth one (*tenant_operation*) is needed by the user to know how to interact with the tenant's data field. The fifth one (*user_action_time*) is needed to set the time when the users should interact with the application.

Currently, the users are based on the vanilla libFuzzer [16], which employs genetic and heuristic algorithms in order to generate better inputs. As future work, we are planning to use the newer methods in the field that use Concolic Reinforcement Learning [3], which are not hard to implement since the components are loosely coupled. We also expect better evaluation results from such methods, because the training methods used in RL could optimize faster the parameters involved in guiding the fuzzer in comparison to heuristics and genetic algorithms [1, 10]. One improvement over libFuzzer worth mentioning is the newly proposed set of metrics and optimization methods as described in Section 3.

4.3 Generating and Finding Bugs

We followed LAVA Benchmark's [4] approach by injecting different concurrency bugs into the shared application code that can happen due to the multi-tenancy setting of the application. There are two types of possible injected bugs: a non-determinism issue and a deadlock situation. Concrete examples of injected bugs in the source code can be found at https://github.com/AGAPIA/river/tree/master/MTA_testing.

Non-deterministic bugs are injected when a given set of users are logged in at the same time and a certain pattern occurs in a random tenant's field. As an example, consider that the following four users log in at the same time: User 1 and User 2 are in Tenant 1, User 3 is in Tenant 2, while User 4 is in Tenant 3. We randomly select one of the tenants (let's say Tenant 1) and if the accessed data field contains a certain pattern, such as a specific part of the input contains a hardcoded value, we inject a bug in the source code to cause non-determinism. Another type of bug in the same category is the overflow error.

An injected bug for a deadlock situation can occur when the users will try to lock the tenant's field before reading and writing to it, simulating an atomic operation. This type of bugs is solved in the real-world implementations using stored procedures. However, a flawed design can still cause bugs, which can be found by fuzzing.

4.4 Reproducing Bugs

To make results reproducible we generate log files with the seeds that were used by the internal random generator and the timing when the users interacted with the main web application. Using the same seeds one can potentially find the circumstances in which the error had happened.

4.5 Results

We present our preliminary results in Table 1. Each row represents the results for 100 simulations, where we count how many times one of the following happened:

- no bug was detected;
- an injected bug was detected;
- integer overflow has occurred.

In our preliminary evaluation, we did not discover any deadlocks. At the start of the simulation, we choose a random number of users (between 1 and 4) for each tenant, a number that will stay constant over all simulations.

The first column represents the number of tenants that were available during each simulation of the 100 total simulations. For the first 6 simulations, we used a 4 tenant configuration similar to Figure 2. The second column contains the number of bugs we injected represented as a percentage relative to the maximal number of possible bugs (here, the maximal number of bugs is the number of ways one can group users from all tenants such that the list will contain at least two users from different tenants).

The bug frequency column gives the used pattern (*int_pattern*, *string_pattern*). We simulate issues by checking the input data patterns. For example, we inject a bug in the *int* case, if the tenant's data field value modulo 10 is 0. Similarly, for the *string* case, we inject a bug, if the input contains at least 3 vowels. Thus, when using for *int* the value of 100, and for *string* the value of 5 vowels, i.e., (100, 5) in the table columns, there is a much lower probability to inject a bug than when using (10, 3).

Note that for injected bugs and overflow errors we made a summation over all tenants' fields and, because of that, the sum of the last three fields can be bigger than 100.

We can see that when we increase the number of tenants there is a higher probability that they will interact with one another, even at a relatively small percentage of bugs. Each simulation took between 2 and 4 minutes, depending on the number of users that interacted with the system. Running 100 simulations took around

Table 1: Preliminary results

No. of tenants	Bug percent	No. of injected bugs	Max no. of bugs	Bug freq.	Successful	Injected bugs found	Overflow error
4	0.00001	1	991	(10, 3)	100	0	0
4	0.001	1	997	(10, 3)	24	83	0
4	0.01	2	115	(10, 3)	13	104	20
4	0.00001	1	487	(100, 5)	90	3	7
4	0.001	3	2019	(100, 5)	84	10	6
4	0.01	5	487	(100, 5)	51	33	16
8	0.00001	672	67108787	(100, 5)	0	181	39
8	0.001	132	131021	(100, 5)	58	34	13
8	0.01	5243	524239	(100, 5)	0	211	9

3 hours. Our simulator and results can be found at: https://github.com/AGAPIA/river/tree/master/MTA_testing.

5 FUTURE WORK

The techniques presented in this paper can help to identify problems of non-determinism and deadlocks. Also, SLA and QoS domains can be addressed using the same architecture by injecting bugs depending on the timing of user interaction with the application and the time spent on different kinds of operations. Further work that we are considering is using a taint engine to detect reads and writes calls to the database objects. Last, but not least, we are working on finalizing the implementation of the reinforcement learning algorithms proposed in Section 3 (getting also inspiration from recent work such as [1]). Once we have a working prototype, we will test it on real cloud applications.

ACKNOWLEDGMENTS

This work was supported by a grant of Romanian Ministry of Research and Innovation CCCDI-UEFISCDI, project no. 17PCCDI/2018.

REFERENCES

[1] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. 2018. Deep Reinforcement Fuzzing. In *2018 IEEE Security and Privacy Workshops*. IEEE Computer Society, 116–122.

[2] Nicolas Carion, Nicolas Usunier, Gabriel Synnaeve, and Alessandro Lazaric. 2019. A Structured Prediction Approach for Generalization in Cooperative Multi-Agent Reinforcement Learning. In *NeurIPS 2019*. 8128–8138.

[3] Miruna Paduraru, Ciprian Paduraru, Alin Stefanescu. 2020. Optimizing decision making in concolic execution using reinforcement learning. In *A-MOST'20, workshop affiliated to ICST'20*. IEEE, 52–61.

[4] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. 2016. LAVA: Large-Scale Automated Vulnerability Addition. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 110–121.

[5] Vincent C. Emeakaroha, Rodrigo N. Calheiros, Marco Aurélio Stelmar Netto, Ivona Brandic, and César A. F. De Rose. 2010. DeSVi: An Architecture for Detecting SLA Violations in Cloud Computing Infrastructures. In *2nd Int. Conf. on Cloud Computing (CloudComp 2010)*. 1–20.

[6] Jerry Gao, Xiaoying Bai, and Wei-Tek Tsai. 2011. Cloud testing-issues, challenges, needs and practice. *Software Engineering: An International Journal* 1 (2011), 9–23.

[7] Jerry Gao, Xiaoying Bai, Wei-Tek Tsai, and Tadahiro Uehara. 2013. SaaS Testing on Clouds - Issues, Challenges and Needs. In *7th Int. Symp. on Service-Oriented System Engineering (SOSE'13)*. IEEE, 409–415.

[8] Patrice Godefroid. 2007. Random Testing for Security: Blackbox vs. Whitebox Fuzzing. In *2nd Int. Workshop on Random Testing, co-located with the 22nd IEEE/ACM Int. Conf. on Automated Software Engineering (ASE'07)*. ACM, 1.

[9] Patrice Godefroid. 2020. Fuzzing: hack, art, and science. *Commun. ACM* 63, 2 (2020), 70–76.

[10] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&Fuzz: machine learning for input fuzzing. In *32nd IEEE/ACM Int. Conf. on Automated Software Engineering (ASE'17)*. IEEE Computer Society, 50–59.

[11] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative Adversarial Nets. In *Advances in Neural Information Processing Systems 27*. Curran Associates, Inc., 2672–2680.

[12] Chang J. Guo, Wei Sun, Ying Huang, Zhi H. Wang, and Bo Gao. 2007. A Framework for Native Multi-Tenancy Application Development and Management. In *9th IEEE Int. Conf. on E-Commerce Technology and The 4th IEEE Int. Conf. on Enterprise Computing, E-Commerce and E-Services (CEC-EEE'07)*. 551–558.

[13] Brian Hayes. 2008. Cloud Computing. *Commun. ACM* 51, 7 (2008), 9–11.

[14] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. 1996. Reinforcement Learning: A Survey. *J. Artif. Int. Res.* 4, 1 (1996), 237–285.

[15] Rouven Krebs., Christof Momm., and Samuel Kounev. 2012. Architectural Concerns in multi-tenant SaaS applications. In *2nd Int. Conf. on Cloud Computing and Services Science (CLOSER'12)*. SciTePress, 426–431.

[16] libFuzzer. 2020. https://llvm.org/docs/LibFuzzer.html. Accessed: 2020-08-07.

[17] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous Methods for Deep Reinforcement Learning *(Proc. of Machine Learning Research (PMLR))*, Vol. 48. 1928–1937.

[18] Ciprian Paduraru and Marius-Constantin Melemciuc. 2018. An Automatic Test Data Generation Tool using Machine Learning. In *13th Int. Conf. on Software Technologies, (ICSOFT'18)*. SciTePress, 506–515.

[19] Ciprian Paduraru, Marius-Constantin Melemciuc, and Bogdan Ghimis. 2019. Fuzz Testing with Dynamic Taint Analysis based Tools for Faster Code Coverage. In *14th Int. Conf. on Software Technologies (ICSOFT'19)*. SciTePress, 82–93.

[20] Ciprian Paduraru, Marius-Constantin Melemciuc, and Alin Stefanescu. 2017. A distributed implementation using apache spark of a genetic algorithm applied to test data generation. In *GECCO'17 Companion Material Proc.* ACM, 1857–1863.

[21] Ciprian Ionut Paduraru and Gheorghe Stefanescu. 2020. Adaptive Virtual Organisms: A Compositional Model for Complex Hardware-software. *Fundam. Inform.* 173, 2-3 (2020), 139–176.

[22] Shyam Patidar, Dheeraj Rane, and Pritesh Jain. 2012. A Survey Paper on Cloud Computing. In *2nd Int. Conf. on Advanced Computing & Communication Technologies (ACCT '12)*. IEEE CS, 394–398.

[23] Teodor Stoenescu, Alin Stefanescu, Sorina Predut, and Florentin Ipate. 2016. RIVER: A Binary Analysis Framework Using Symbolic Execution and Reversible x86 Instructions. In *21st Int. Symp. on Formal Methods (FM'16) (LNCS)*, Vol. 9995. 779–785.

[24] Wojciech Stolarz and Marek Woda. 2014. Performance Aspect of SaaS Application Based on Tenant-Based Allocation Model in a Public Cloud. In *9th Int. Conf. on Dependability and Complex Systems (Advances in Intelligent Systems and Computing)*, Vol. 286. Springer, 423–432.

[25] Takeshi Takahashi, Gregory Blanc, Youki Kadobayashi, Doudou Fall, Hiroaki Hazeyama, and Shin'ichiro Matsuo. 2012. Enabling secure multitenancy in cloud computing: Challenges and approaches. In *2012 2nd Baltic Congress on Future Internet Communications*. 72–79.

[26] Wei-Tek Tsai, Qingyang Li, Charles J. Colbourn, and Xiaoying Bai. 2013. Adaptive Fault Detection for Testing Tenant Applications in Multi-tenancy SaaS Systems. In *IEEE Int. Conf. on Cloud Engineering (IC2E'13)*. IEEE, 183–192.