Optimizing decision making in concolic execution using reinforcement learning

Ciprian Paduraru
The Research Institute of the University of Bucharest (ICUB), Romania
Department of Computer Science, University of Bucharest, Romania
ciprian.paduraru@fmi.unibuc.ro

Miruna Paduraru
Department of Computer Science University of Bucharest, Romania
miruna.paduraru@drd.unibuc.ro

Alin Stefanescu
The Research Institute of the University of Bucharest (ICUB), Romania
Department of Computer Science, University of Bucharest, Romania
alin@fmi.unibuc.ro

***Abstract*—This paper presents an improvement to a new open-source testing tool capable of performing concolic execution on x86 binaries. The novelty is to use a reinforcement learning solution that reduces the number of symbolically executed states. It does so by learning a set of models that predict how efficiently it would be to change the conditions at various branch points. Thus, we first reinterpret the state-of-the-art concolic execution algorithm as a typical reinforcement learning environment, then we build estimation models used to prune states that do not look promising. The architecture of the base model is a Deep Q-Network used inside an LSTM that captures the patterns from the ordered set of branch points (path) resulted by executing the application under test with different inputs generated at runtime (experiments). Various reward functions can give automatic feedback from the concolic execution environment to define different policies. These are customizable in our open-source implementation, such that users can define their custom test targets.**

***Index Terms*—concolic execution, symbolic execution, testing, x86, tainting, reinforcement learning.**

I. Introduction

Software testing is a very important concept nowadays from multiple perspectives. First, it can save important resources for companies because finding bugs in the early stages of software development can cost much less to fix them. From the security perspective, it can find vulnerabilities in the source code that could potentially lead to system breaches. Finally, the product quality perspective is also important and in order to better satisfy the customers, the application must be well tested against different scenarios. Several strategies and tools were created to automatize software testing, as discussed in Section II. This paper describes a contribution to one of the components in our open-source software testing suite, which implements a concolic execution engine at x86 binary level. As the literature suggests, a lot of computational resources in the testing process are used by the SMT solver that tries to obtain input payloads that take different paths in the application under test. Our motivation in this work is to reduce the numbers of SMT solver calls.

Contributions. The novelty of our paper for the concolic binary x86 testing is the usage of reinforcement learning (RL) techniques to estimate the value of each possible modification in the path that an input payload takes, such that computational resources are saved and faster feedback can be obtained from the testing tool. To the best of our knowledge, this is the first work that attempts to use reinforcement learning in this context. To achieve this, the following contributions were implemented:

- an execution environment setup for using reinforcement learning with concolic execution.
- an estimation network capable of predicting how "valuable" are the branch condition changes in different states.
- a set of reward functions that could tackle different targets when performing software testing.
- a basic evaluation of the proposed mechanisms, showing its advantages in testing a software application during its development lifecycles.

River - an open source testing framework.

Our framework for automatic software testing of x86 binaries is available at https://agapia.github.io/river/. The interested reader can also find more details about it in the documentation page, describing its architecture, implementation details and future work plan. Here we briefly sketch only the components needed to understand the rest of the paper:

- *SimpleTracer* - executes a program with given input and returns a trace, represented by a list of basic blocks encountered during execution. In our work, as in many other previous papers, a basic block is a contiguous set of x86 assembler instructions ending with a jump.
- *AnnotatedTracerZ3* - similar to the one above, it executes a program with given input and returns a trace. The difference is that this execution uses dynamic taint analysis and returns as output the Z3 (the SMT solver used in our framework) serialized jump conditions for each branch in the trace that caused the move from the current basic block to the next.
- *RiverConcolic* - this component orchestrates a concolic execution process in a many-core environment. It uses the same methods explained in [1], named *Generational Search*. Additionally, in this paper, we also add reinforcement learning techniques for pruning states.

The paper is structured as follows. The next section presents some existing work in the field. Section III describes the

978-1-7281-1075-2/20/$31.00 ©2020 IEEE
DOI 10.1109/ICSTW50294.2020.00025

implementation details for the methods used to estimate the value of states using reinforcement learning. Evaluation of our tool and methods are discussed in Section IV. Finally, conclusions are given in the last section.

II. Related work

The main purpose of an automatic test data generation system for program evaluation is to generate test data that covers as many branches as possible from a program's source code, with the least usage of computational resources, with the goal of discovering as many subtle bugs as possible. One of the fundamental known technique is *fuzz testing* [2], in which the test data is automatically generated using random inputs (with different strategies) and executed against the program under test. The well-known limitation of fuzz testing is that it takes a significant effort to produce inputs that cover almost all branches of a program's source code, many of them having a very low probability to be reached only by fuzzing.

In the field of fuzzing techniques, there are currently three main categories: blackbox random fuzzing [3], whitebox random fuzzing [1], and grammar based fuzzing [4], [3]. Blackbox fuzzing methods are also augmented with other strategies for better results. For example, in [5] (also part of the RIVER tool) and AFL (American Fuzzy Lop) [6], genetic algorithms, and various heuristics are used to find vulnerabilities faster and achieve better code coverage. Autogram, mentioned in [7], learns context-free grammars given a set of inputs and dynamic tainting, i.e., dynamically observing how inputs are processed inside a program. Recent work concentrates also on learning grammars automatically, using recurrent neural networks, such as the work presented in [8] and [9].

Symbolic execution is another strategy to automate software testing [10]. While theoretically being able provide better code coverage, it has practical challenges due to the possible paths exponential growth. At the moment, there are two approaches for implementing symbolic execution in software testing: online symbolic execution and offline symbolic execution (a.k.a concolic execution). In a nutshell, concolic execution works by executing the input on the program under test, and gathering all the branch points met along the execution together with their conditions. Its advantage over online symbolic execution is that, at the end of each execution, a trace containing the branch points and their conditions are obtained, which can be used to generate offline a new set of inputs. Because of that, concolic execution is more suitable than online symbolic execution when applied to large applications having hundreds of millions of instructions and deep paths.

In the field of online symbolic execution, two of the common open-source frameworks are KLEE [11] and S2E [12]. Several concolic execution engines are also presented in the literature. Some early work is represented by DART [13] and CUTE [14], which both operate on the source code level. CRETE [15], which is based on KLEE, operates on LLVM representation in contrast to our framework RIVER, which operates at x86 binary level. In pure symbolic execution, various pruning tactics were implemented to solve the path explosion problem and get better code coverage. For example in [16], they prune out paths by bounding the number of iterations done in a loop. In [17], authors are using an iterative depth first search that increases the exploration depth gradually and discarding similar paths at the same depth. Projecting small-scale behavior into a large scale fails if there is discontinuity due to parts of the code that are activated in a scale dependent manner [18], [19]. Our paper aims at learning a policy at large scale directly. Worst case complexity finding solutions were also studied. In WISE [20] and XSTRESSOR [21], they learn the policy at a small scale then apply it at a bigger scale. The continuation is the PySE [22], which also uses Deep Q-learning (DQN) [23] to create a policy at a large scale directly, being capable of learning complex and irregular pattern in behavior, thus improving the test efficiency for large scale programs. They apply the DQN in online symbolic execution to promote longer execution paths and penalizing actions that do not lead to satisfiable constraints. In our case first, we have a different setup with symbolic offline execution, thus the execution environment, experiences gathering and training processes are different. We also use multiple models trained at once because of variable paths length. Also, in terms of features the reward targets in our case can be more generic, even made custom by users as described in the paper, not just for finding the highest complexity input to the application. In [24] authors are using RL for testing in continuous software integration processes. The RL method is used to rank a test suite, then the most promising ones are scheduled up to an estimated time limit for testing at each integration cycle. As features, for each input, they consider the duration and previous history verdict (i.e., fail or pass). Their reward function penalizes passed tests ranked before failing ones. Thus, the strategy is to reinforce the behavior of ranking the failing test first.

The most related tool in terms of features similarity, i.e., which also works on x86 level, is SAGE [1]. In comparison with their work, we use reinforcement learning methods to estimate the value of states, which has the potential to save computational resources from solving paths through the application that are not evaluated as promising. Moreover, we also provide our tool as open source.

III. Design of a Reinforcement Learning solution for concolic execution

Reinforcement Learning (RL) is a machine learning paradigm where an agent takes actions in a given environment and receives back the reward of his actions and the resulted environment's states. The user defining the optimization problem specifies the reward function according to what it wants to achieve, the environment's states, the possible set of actions, transitions rules and when an episode ends (what states are terminal, or a time limit to restart exploration). An important characteristic of this paradigm is that it needs no supervisor, the agent improving its decision making policy towards getting more rewards by exploring the environment with various actions during a limited number of episodes.

The interested user is invited to read more details about RL paradigm in [25]. The rest of the section explains how we used reinforcement learning to optimize the concolic execution of binary x86 programs.

A. Overview and motivation

The purpose of using reinforcement learning for concolic execution is to estimate the score of different actions while doing less expensive symbolic execution using an SMT solver. The idea behind is to reduce the work on paths that do not look promising. As a concrete example, consider the sample tree in Fig. 2. The method used in SAGE [1] evaluates an entire tree and eventually the last executed branch will be responsible for detecting an input that triggers an assert. Almost every node (branch point) needs to be executed symbolically to obtain that input (Fig. 2, left part). Our target is to obtain a method that sorts the available options as close to reality as possible, which in the end could lead to obtaining the same results (e.g., code coverage, detection of inputs that cause issues, etc.), but faster by evaluating symbolically a smaller subset of branch points (Fig. 2, right part). The estimation can be done using Deep Reinforcement Learning techniques [23], by creating a network that estimates the values of all actions possible from a given state, i.e., $Q(state, action)$. For instance, in the right side of Fig. 2, the action A that changes the state from $P0$ to $P1$ is estimated to have the highest value among all other actions to choose at that source state.

```
void test_simple(const unsigned char *input)
{

    int cnt=0;
    if (input[0] == 'b') cnt++;
    if (input[1] == 'a') cnt++;
    if (input[2] == 'd') cnt++;
    if (input[3] == '!') cnt++;
    if (cnt >= 4) abort();
}
```

Fig. 1: Example of a simple function that the user might want to evaluate. The example is taken from SAGE paper [1].

Next, we sketch a reinforcement learning method implementation to learn models that can predict action values in given states. This is possible in the context of concolic execution, since, as shown below, the problem can be modeled in such a way that reward feedback can be gathered automatically from the application under test and our tools. Thus, there is no need for human supervised pairs of inputs and outputs.

The intuition for using trained models when testing the same application continuously between different project phases (and ideally at each new code change) is that there will always be similar (patterns) paths composed by blocks of code and offsets in the execution binary analysis that lead to similar results. Unchanged code over time should keep the same patterns, while new code incrementally added could change by small areas over time. But even with small changes, the modules and offsets of executed blocks would stay almost the same, and we expect that the neural network used behind the model will approximate the same previous patterns. In case of bigger changes, however, online learning methods can keep the previously trained model in sync with the latest changes in the application's code. Testing duplicated code lines can also be learnt and exploited by the RL method since similar patterns would occur in the testing process, even if the blocks of code are located at different locations.

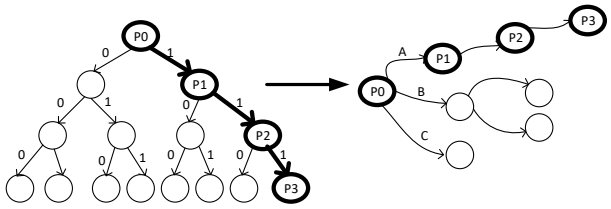

Fig. 2: The left part of the figure shows the tree obtained by running state of the art concolic execution on the source code in Fig. 1. The edges marked with 0 mean that branches are not taken, while 1s means that they are taken. The bolded nodes in the path show the one that generates the crash, and it is evaluated last. In the right side, each node is a state - a path constraint obtained by a symbolical execution for an input. A method that sorts the actions to take at each branch point by their estimated value is considered. If branches are sorted top to down by their estimated value (i.e., action A has the highest value at state $P0$), then it could execute the same red path first this time.

B. Concolic execution environment

To build a concolic execution environment, we implemented the state of the art method proposed in [1]. However, we adapted it to our open-source tools suite by adding different pieces of code to prepare it as a reinforcement learning environment. Listing 1 shows the definition of an input data type in our algorithm. The field at line 8, PC represents the **state** concept in the reinforcement learning terminology. Its name comes from "Path Constraints" because it represents an ordered collection of branch points encountered while executing an input with the application under test (gathered by one of our tracing tools, *SimpleTracer* or *Z3Tracer*), and obtaining the details at each individual branch decision: the position (module and offset), condition for each branch to take the same decision again, and if the jump was taken or not with the tested input (Eq. 1 and 2). A path constraint and details from each branch point is depicted in Fig. 1. Instead of using directly the module offsets numbers, we keep only the relative offsets from the first entry address in the path. There are two reasons behind this: (a) it is easier to learn a model with offset values (i.e., differences) rather than full number patterns, and (b) when the source code is changed, the most relative offsets between execution paths still holds the same, while the global modules' addresses are changed. Of course, there is a chance of confusion, but from our experience, very rarely in

the disassembled code execution two paths will be similar in terms of offsets. Our conclusion so far is that by using offsets instead of full numbers is beneficial to the problem. The *bound* parameter in line 10 corresponds to the starting index in the path's branch points where the conditions can be changed to take a different path. This is used to prevent backtracking, as mentioned in [1]. Intuitively, this variable prevents a state to make the same change in the path constraints as one of its ancestors.

$$PC = \{BranchDesc_i\}_{i=\overline{0,len(PC)-1}} \quad (1)$$

$$BranchDesc_i = \{ModuleID, Offset, Z3condition, taken\} \quad (2)$$

Listing 1: Definition of an input data structure.

```
1  class Input:
2    // The concrete input buffer payload for this input
3    v = null
4    // A reference to the parent input that generated this
5    parent = null
6    // The path constraint obtained after
7    // symbolically evaluating the parent input
8    PC = null
9    // The lower bound index for chosing the action
10   bound = −1
11   // The index where this input should inverse the
12   // condition (relative to PC, in [bound, PC.len−1])
13   action = −1
```

In Listing 2, function *CheckAndScore* first runs the input under a supervised process and outputs possible issues, if any. Then, it computes the heuristic score. In training mode, it additionally gathers experiences and invokes the optimization process. The heuristic score from [1] is used as a ground truth value for reward in our experiments (in the case of targeting the code coverage rate). It scores the value of a state as the number of new basic blocks discovered by the path constraints representing the state. Users can hook their own custom strategies as shown in the text below, more reward strategy ideas being presented in III-D. Function *SearchInputs* is the entry point of the concolic execution environment, both when training models for a new episode or in inference mode. The inputs generated by the environment are added in a priority queue by their estimated score (i.e., how likely taking that input and executing would move closer to user's test targets - defined in our terminology by custom reward functions). The most promising input is taken out from this queue (line 24), then the execution splits into two parts, as opposed to the implementation in the previous work. If the input payload buffer value ($input.v$) was not obtained yet (line 27), it means that the model was used to estimate the score of the input that would be obtained by inverting the condition at index $input.action$ in state $input.PC$. Note that this decision is taken in the pseudocode of Listing 3, which is explained in more detail in Section III-C.

This is a **keypoint** in understanding our optimization purpose, since it is visible from here that, if a condition inversion along a path of branch points would be classified as not a promising move by its estimated score using the model, then it would save the time needed to symbolically solve the input for it. Instead, the SMT solver will be called for it only if the state entry is taken out from the priority queue before other not so promising inputs. This is opposed to the previous work, which solves every possible condition inversion along with a newly discovered state (*PC*) and adds the resulted input and its score in the priority queue. However, the quality of the estimation and the resources additionally consumed by the method become very important for the end result and analyzed briefly in Section IV. Note also that the proposed method does not eliminate any potential inputs, it is just trying to prioritize them in a different way.

Listing 2: Two of the main functionalities of the conclic execution environment implementation: *CheckAndScore* - a function that checks a given input for issues and *SearchInputs* - a search function that generates new inputs starting from an initial input seed.

```
1  class RiverConcolic:
2    CheckAndScore(input, trainMode)
3      score = null
4      if input.v != null:
5        Res = execute input.v using a SimpleTracer process
6        if Res has issues:
7          output(Res)
8        score = ScoreHeuristic(input, Res)
9      if trainMode != −1:
10       NewPC = Run an AnnotatedTracerZ3 process with input.v
11       RLModule.onNewExp(input, newPC, score)
12     return score
13
14   SearchInputs (initialInput, trainMode):
15     initialInput.bound = 0
16     // A priority queue of inputs holding on each item
17     // the score and the concrete input buffer.
18     PQInputs = {(0, initialInput)}
19     Res = execute initialInput using a SimpleTracer process
20     if Res has issues:
21       output(Res)
22
23     while (PQInputs.empty() == false):
24       input = PQInputs.pop()
25       // If the input was not executed symbolically yet
26       // we run it before.
27       if input.v == null:
28         PC = input.PC
29         i = input.action
30         Solution = Z3Solver(
31                 PC[0..i−1] == same jump value as before
32                 and PC[i] == inversed jump value)
33         if Solution == null:
34           // If there is no solution we still send this further for
35           // experience gathering purposes
36           CheckAndScore(input, trainMode)
37           continue
38         input.v = overwrite Solution over input.parent
39         CheckAndScore(input, trainMode)
40
41       // Get the children of this input
42       nextInputs = Expand(input)
```

```
43        foreach newInput in nextInputs:
44          // If the new input was executed symbolically,
45          // then use the heuristics to score it
46          if newInput.v != null:
47            score = CheckAndScore(newInput, false)
48            PQInputs.push((score, newInput))
49        else: // Just use the estimated score
50            PQInputs.push((newInput.estScore, newInput)
```

Listing 3: The Expand function pseudocode using the AnnotatedTracerZ3 process to get symbolic conditions for each of the jump conditions met during the execution of the program with the given input.

```
1   class RiverConcolic:
2     Expand(input):
3       childInputs = []
4
5       // Get the Z3 conditions for each jump
6       // (branch) encountered during execution
7       // In our example, PC contains four entries
8       // one for each of the branch points.
9       PC = Run an AnnotatedTracerZ3
10                process with input
11
12      // Take each condition index and inverse
13      // only that one, keeping the prefix with
14      // the same jump value
15      for i in range(input.bound, PC.length):
16        // Do we have a model to estimate ?
17        tailSize = PC.length − input.bound
18        action = i
19        if L_min ≤ tailSize and tailSize ≤ L_max:
20          newInput.v = null
21          tailLen = PC.length − input.bound
22          newInput.estScore =
23              RLModule.Predict(PC, tailLen, action)
24          newInput.parent = input
25          newInput.PC = PC
26        else:
27          // Otherwise, use the symbolic solver
28          // Solution will contain the input byte
29          // indices and their values, which need to
30          // be changed to inverse the i'th jump condition
31
32          Solution = Z3Solver(
33              PC[0 .. i−1] == same jump value as before
34              and PC[i]== inversed jump value)
35        if Solution == null: continue
36        newInput.v = overwrite Solution over input
37        // no sense to inverse conditions again
38        // before i'th branch,
39        // i.e., prevent backtracking.
40        newInput.bound = i
41        childInputs.append(newInput)
42
43      return childInputs
```

C. Design of our reinforcement learning solution

The concept of **state** in our solution was presented in Eq. 1 and 2. At each state, a concolic execution environment could act by modifying the original input such that the Z3 condition for each branch point along the *PC* (starting from *bound* to PC.len) is inversed and the program would take a different path. Thus, in a classic Deep Q-Network, it would need to estimate the value of $Q(PC, action)$, where $action$ means inverting one of the Z3 conditions in the PC between indices $[bound, PC.len-1]$. However, there is an issue since the *PC*s can have different lengths. Various attempts have been tried, described in more detail in Section III-D. The current version is training several models at once, each one being used for a fixed action size. More specifically, we let the user provide two parameters - L_{max} and L_{min}, which denote the maximum and minimum action length considered, respectively. There are in total $L_{max} - L_{min} + 1$ models trained at the same time, one for each action length. For a PC that has a length higher than L_{max}, only the last L_{max} items are considered, as shown in Fig. 3. Anything higher or between L_{max} and L_{min} goes into one corresponding model index, both for training and inference, as show in in Listing 4 at different points: models instancing - line 4, new experiences gathering - line 12, or prediction - line 25.

It is important to note at this point, as it can be seen in Listing 3 line 19, that estimation of the values for each possible action in a given state is made only when the tail size of the current path is higher or between the two bounds. Otherwise, we fall back to the *Generational Search* method from [1], where all the actions available for the state are evaluated, and the new resulted PC is added in the priority queue.

The pseudocode in Listing 4 shows the high level management of modules and interaction with the concolic environment presented above, as it is implemented in $RiverRLModule$. Remember that **experience** gathering starts from Listing 2 at line 9, where the function *OnNewExp* is called. An experience in our solution is a pair of $(PC, action, newPC, reward)$ where:

- PC represents the current state, a path constraint as defined in Eq. 1 and 2.
- $action$ represents the index of the path constraint where the condition inversion took place.
- $newPC$ represents the new state obtained after performing the action and getting the new input using the SMT solver (Listing 2, line 9).
- $reward$ is the feedback from the system regarding the value of a transition from PC to $newPC$, explained later in more details in Section III-D.

The internal model would use this set of information, and at each new T experiences gathered it will update that model weights using back-propagation, such that over time the models become better at estimating the value of states. This mechanism is shown in Listing 4 line 40. The training starts by calling the *Train* function at line 30. A user specified variable named $NumMaxEpisodes$ denotes the number of episodes to train the set of models. On each iteration, the environment is initialized with a new input seed. This one can be chosen randomly between a set of possible input seeds, or generated randomly on each new iteration (the first method is preferred usually since it can provide valid inputs that are able to create long PCs since the beginning of the training process). For in-

ference purposes or online learning, the *RiverRLModule* is first instanced, then method *OnInit* is called with the *restore* option parameter, such that the previously saved trained module is loaded first. An episode can finish from two possible reasons: (a) The queue in function *RiverConcolic.SearchInput* has no more inputs to process, or (b) the maximum number of training updates has been reached (line 45). This second parameter was added because the number of states can grow exponentially in applications and instead of blocking the algorithm in exploring a local optima, it might be helpful to restart at some point and start again with different seeds.

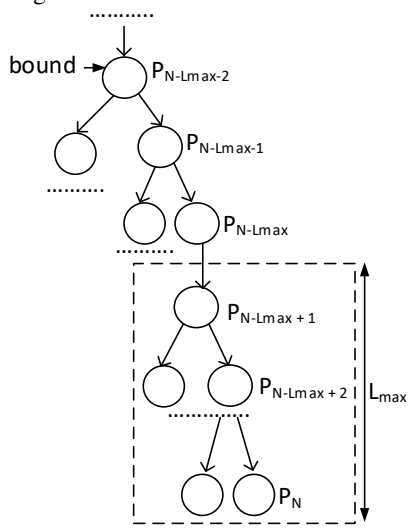

Fig. 3: A subtree starting from the bound index node of a hypothetical *PC* resulted by tracing the application under test with an input payload. The resulted *PC* is represented by the right-most branch, while the other nodes and edges represent possible different execution paths that the program could take by modifying the input such that the conditions on each node are satisfied. The index of each node in the *PC* (level) is shown on each node's right side. As shown in the figure, only the patterns represented by the last L_{max} items can be considered.

Listing 4: Pseudocode of the RiverRLModule responsible for coordinating the training and inference processes of the proposed methods.

```
1   class RiverRLModule:
2     // These are instances of models that estimate the score
3     // depending on the size of the tail
4     RiverModel models[L_max − L_min + 1];
5
6     def OnInit(restore):
7       if restore:
8         Load previous saved models
9
10    // Gathering a new experience.
11    def OnNewExp(input, newPC, targetScore, estimatedScore):
12      tailLen = input.PC.len − input.bound
13      // Ignore too short sequences
14      K = min(L_max, tailLen) − L_min
15      if K < 0:
16        return
17
18    // Add the new exprience to the data store of the
19    // corresponding model
20    experience = (input.PC, newPC, input.action,
21                  targetScore, estimatedScore)
22    models[K].AddExperience(experience)
23
24   def Predict(input, tailLen, action):
25     // Choose the available model
26     assert tailLen ≥ L_min
27     K = min(tailLen, L_max) − L_min
28     models[K].Predict(input, action)
29
30   def Train(NumMaxEpisodes):
31     for episode in range(NumMaxEpisodes):
32       input = seed new input
33       RiverConcolic.SearchInputs(input, true)
34       Save model and update training statistics
35
36  class RiverModel:
37    def AddExperience(experience):
38      memory.add(experience)
39      experiencesSinceUpdate++
40      if experiencesSinceUpdate ≥ T:
41        experiencesSinceUpdate = 0
42        batch = memory.selectBatch(N)
43        optimize RiverDQN using batch
44        num_optimizations++
45        if num_optimizations > MaxUpdatesPerEpisodes:
46          terminate episode
47
48    def Predict(input, action):
49       scores = model.Predict(input)
50       return scores[action]
51
52    ExperienceReplay memory
53    RiverDQN model
```

D. Estimation model architecture, discussion of difficulties and other attempts

An $LSTM$ architecture [26] is at the core of the network that estimates the value of each action in a given state (i.e., the instance at line 53 in Listing 4). This architecture is used since it is capable of learning the long term dependencies between input values and outputs. This fits the needs of our problem state representation since it is structured as a sequence of the last L items, with $L_{min} \leq L \leq L_{max}$, as presented in Eq. 2 and Figure 4. Note that the field *Z3condition* does not play any role in the input, and it is left out when using the state as input for the model. The $ModuleID$ field, which the tracer components of RIVER framework outputs as a module name (e.g., "libc.so", "userCustom.so"), is converted into one-hot encoding since there is no correlation between modules to represent them as numeric labels. The dimensionality of the one-hot encoding can be optimized in each kind of application by checking first the list of dependencies for the application under test, instead of using an exhaustive list of all operating system's and user's binaries. The output of the network is

obtained by a linear combination between the hidden states of each LSTM cell and a learnable weight matrix (Eq. 3).

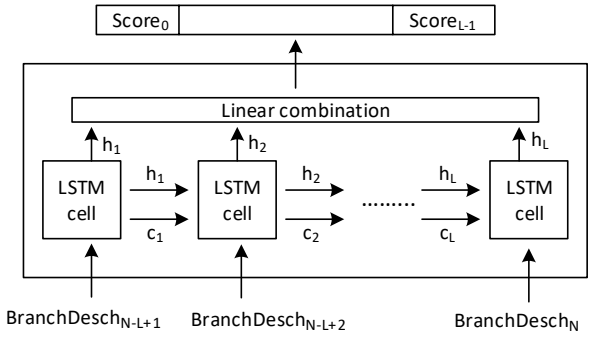

Fig. 4: The internal $LSTM$ based model for value estimation of a state with a length N. The model is supposed to learn the patterns only for the last L items in the state. Thus, the last L branch description entries from the evaluated state are used as input. The output is an array of L items, representing for each of the last L branch points in the state, what would be the value if the algorithm tries to get a new input that inverses the condition at that point. The c_i values denote the cells state vector - a kind of memory that selectively remembers computation up to that point. The h_i values represent feature vector containing dependency information between inputs (history). For more information about $LSTM$s, interested readers may refer to [26].

$$Scores = W_s \, [h_0 \, h_1 \quad .... \quad h_{L-1}]^T \tag{3}$$

Reward functions To get feedback from the environment and promote certain test targets, the algorithm needs a reward function, i.e., a score for transitioning from a state S, to a new one $SNext$ by using a given $action$. We imagined three possible categories of using automated software testing, each with a different reward function. However, in our open source framework, the user can mix these or set their own custom functions.

(a) Increase code coverage in the shortest time possible. This kind of target could be used in continuously developed software when engineers submit source code and ideally immediate feedback should be provided. In this case, the reward function in Eq. 4 is used. $B(State)$ is a function that gives the set of different blocks that $State$ touches in the tested application. The first part of the equation computes the cardinal of the set difference, resulting in how many new (different) blocks of code does the new state achieves. Note that the number of new basic blocks discovered is relative to the source state. In the implementation, for the training phase only, the method keeps a dictionary in each state that contains this information. This is different from [1], which used a single global dictionary to compute the score heuristic, but using a local/decentralized data structure is a hard requirement for the reinforcement learning method since the states would be visited in non-deterministic order each time. The second part of the equation penalizes actions that are too far from the beginning of the state. The intuition is that by using actions as close to the beginning, more concurrent work could be done if the platform executing the test process is a distributed one. Parameters $E1$ and $E2$ are used to trade-off one part or another.

$$\begin{aligned} R(S, action, SNext) =& E1 * (B(SNext) - B(S)) + \\ & E2 * (S.len - a + 1) \end{aligned} \tag{4}$$

(b) Increase code coverage in hot-spots, i.e., a collection of basic blocks that are known as very susceptible to issues. A dictionary with scores for each block address range that caused issues in the past is supposed to be stored for this kind of feedback, Eq. 5. The proposed reward function, in this case, is presented in Eq. 6, which is a modified reward function from Eq. 4 that weights the importance of each newly discovered block by how often issues appeared at a given module and range of addresses in it.

$$\begin{aligned} &Stats(ModuleID, [Offset_{Start}, Offset_{End}] = \\ &\frac{Issues\ in\ (ModuleID, [Offset_{Start}, Offset_{End}])}{Num\ issues\ reported\ in\ total}. \end{aligned} \tag{5}$$

$$\begin{aligned} R(S, action, SNext) =& E1 * \sum_{\boldsymbol{b} \in B(SNext) - B(S)} Stats[\boldsymbol{b}] \\ & + E2 * (S.len - a + 1) \end{aligned} \tag{6}$$

(c) Increase path size and/or time, used for finding the highest complexity path of an application under test. There are two possible interesting cases in this case: (a) if the path is feasible, i.e. the next state can be solved by obtaining an input with the Z3 condition inversed at index $action$ in state S, the formula in Eq. 7 simply weights between the lengths of the paths and the time needed to trace the application and get that path. If not, a user defined parameter $P_{notSatisfiable}$ is applied to penalize the network for choosing actions that lead to unsatisfiable states.

$$R(S, action, SNext) = \begin{cases} E1 * (SNext.len - S.len) + \\ E2 * (time(SNext) - time(S)), \\ \text{if SNext is not null} \\ P_{notSatisfiable}, \text{ otherwise} \end{cases} \tag{7}$$

There are other kinds of reward targets that we have also imagined, but not discussed because of space constraints. One that is worth to be briefly mentioned is to promote finding inputs that push the resources consumption in the system up to the limit.

Epsilon strategy. It is typical to reinforcement learning algorithms to explore other actions too, not only the best one according to the currently trained policy [23]. In our case, we would like to emphasize the importance of letting the algorithm explore new paths, even at an episode close to the finish of the training process. Finally, we use exponential weight decay for controlling the parameter but keep the ϵ probability interval high enough.

Other attempts. It is important to take a step back and remember that the algorithm uses $L_{max} - Lmin + 1$ instances of the models described above since we needed to use fixed action lengths. We made several attempts to solve this issue and make a model independent of the actions space size that might be worth mentioning for future research. $LSTM$ architectures can handle a variable number of input parameters (as known from their application in natural language processing), so the variable input given by the length of states is solvable. The problem that we could not solve efficiently up to now is the output part since there is also a variable set of actions that are possible on each state. One attempt was to incorporate the action in the input, i.e., concatenate the set of $BranchDesc_i$ with the action index, as a single number. However, the network did not succeed to understand the importance of the single number representing the action index based on our experiments. As future work, we consider to incorporate other state-of-the art mechanisms from machine learning domain such as attention models [27] or prioritizing the experiences [28].

IV. EVALUATION

Environment setup. The evaluation of our solution was done using two open-source applications: a JSON parser [1], and an HTTP parser [2], both with source code implemented in C++ language. On top of their code, we added only the symbols needed to inject the payload buffer and mark the entry code of the binary resulted after building the solution. The experiments described below (inference) were done on a 6-Core Intel i7 processor with 16 GB RAM on Ubuntu 16.04. The training process also used a RTX 2070 video card. For this evaluation, we used only the reward in Eq. 4 with the target of comparing the code coverage obtained by using our solution against the classic *GenerationalSearch* method and their block score heuristic as presented in [1]. We call these methods further in the text as "RiverConcolic" and "RiverConcolicRL", since the methods were implemented in our framework repository (RIVER). The coverage metric of a set of input tests generated counts how many different basic blocks of an application are evaluated using all the available tests generated by a testing tool. The time is an important criterion, because, ideally, the software must be tested continuously at each new code submitted to the application's source code repository. If the testing process is not fast enough, it might not scale with the speed of development. Table I shows the common parameters that have been used in the evaluation of both applications and methods. The training process was left running for 24h continuously, but the ϵ probability goes down to its lower bound in a fixed number of 100 episodes. The lower bound used in our experiments (0.3) seems unreasonably high for typical reinforcement learning problems, but in our case, it helped to get diversity and discover new branches faster. It is again something that users can tune depending on their application. During experiments, we concluded that penalizing too much $P_{notSatisfiable}$ does not give good results since there are many cases when inputs are not solvable, but this depends on the problem. Parameters *E1* and *E2* could have been learned, but we fixed them to test on a single client as a baseline. We plan for a distributed architecture in the future. Experiences added in an episode are unique to avoid bias to particular experiences. Before selecting them in a training batch, they are randomized to break the correlations between experiences and time, thus reducing the variance of the updates.

TABLE I: Common parameters used in evaluation

Parameter	**Value**
N (batch size)	64
L_{max}-L_{min} range	[8-4]
MaxUpdatesPerEpisodes	512
T (the number of new experiences gathered to trigger a new training batch)	32
ϵ (exploration) probability range	From 1.0 to 0.3, exponential decay in 100 episodes
E1	1.0
E2	0.0
$P_{notSatisfiable}$	-1
Learning rate	0.95
Optimizer	RMSProp

Based on the setup described above we are interested in three research questions.

Research Q1: is the estimation function efficient? We are interested to see if a trained model can obtain faster a certain level of code coverage in comparison with the version without reinforcement learning. In this case, we let both methods running until they reached 100 basic block on both HTTP and JSON parser. The comparative times are shown in Table II.

This proves that the trained reinforcement learning based model is able to get to the same code coverage results faster than the previous method (34% faster for HTTP parser, and 29% faster for JSON parser). Note that to have a fair evaluation, we used different sets of seed inputs in training versus evaluation. Even though the model was trained for 24h before, this is still valuable because it can be used as a better starting point for a testing process, or it can be re-used between small code changes.

[1]https://github.com/nlohmann/json

[2]https://github.com/nodejs/http-parser

TABLE II: Comparative time in hours and seconds to reach 100 different block code coverage on the two tested applications.

Model	HTTP parser	JSON parser
RiverConcolic	2h:10m	2h:53m
RiverConcolicRL	1h:37m	2h:14m

Research Q2: is the same model efficient between small source code changes? To evaluate this, we considered three different consecutive code submits (with small code fixed, between 10-50 lines modified) on both applications and averaged the time needed to reach again 100 basic blocks. The *RiverConcolicRL* method was trained on the base code, then evaluation was done using the binary application built at the next code submit on the application's repository. Results are shown in table III .

TABLE III: Averaged comparative time in hours and seconds to reach again 100 different block code coverage on the two tested applications, using three different consecutive code submits.

Model	HTTP parser	JSON parser
RiverConcolic	1h:56m	2h:47m
RiverConcolicRL	1h:31m	2h:15m

These results suggest that the estimation models can be reused between consecutive code changes efficiently, keeping in our tests an advantage of 27%, respectively 24% over the version without reinforcement learning.

Research Q3: how fast can the model adapt to bigger code changes ? In this case, we considered the application under test between two random versions on the repository, but this time with significant code changes (one year difference between them). Online learning was used in this case by reloading the model weights trained on the base version for the same initial training time of 24h, then training it in continuation with the binary application built at the second version for 1h. The comparative results shown in Table IV suggest that a previously trained model can quickly adapt to get fair results despite important code changes.

TABLE IV: Averaged comparative time in hours and seconds to reach again 100 different block code coverage on the two tested applications, using significant code differences between two submits. The RiverConcolicRL was reloaded and trained with the latest code change for 1h.

Model	HTTP parser	JSON parser
RiverConcolic	2h:05m	2h:38m
RiverConcolicRL	1h:39m	2h:07m

However, an extensive study is needed for evaluation in the future. We let the following ideas as future work:

- How different kinds of application perform, e.g., text versus binary based inputs.
- A graph between training time needed to get different levels of performance.
- How different options and parameters affect performance.
- Evaluate the other proposed reward functions.

V. CONCLUSIONS

This paper made a step further in advancing the component of our testing framework components which is responsible for concolic execution of x86 binaries. By using reinforcement learning techniques, the value of different changes in the input that affect the branch conditions of the application under test can be estimated without needing to solve the SMT conditions for states that do not look promising. Some ideas for reward functions, methods to optimize the current state, and more evaluation study ideas were also presented in the paper. Looking at the evaluation results, we believe that reinforcement learning can be successfully used in the future for testing at binary level the software applications during their development life-cycles faster than before.

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

REFERENCES

[1] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: Whitebox fuzzing for security testing," *Queue*, vol. 10, no. 1, pp. 20:20–20:27, Jan. 2012.
[2] P. Godefroid, "Random testing for security: blackbox vs. whitebox fuzzing," in *RT '07*, 2007.
[3] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.
[4] P. Purdom, "A sentence generator for testing parsers," *BIT Numerical Mathematics*, vol. 12, no. 3, pp. 366–375, Sep 1972.
[5] C. Paduraru, M. Melemciuc, and A. Stefanescu, "A distributed implementation using apache spark of a genetic algorithm applied to test data generation," in *Genetic and Evolutionary Computation Conference, Berlin, Germany, July 15-19, 2017, Companion Material Proceedings*, P. A. N. Bosman, Ed. ACM, 2017, pp. 1857–1863.
[6] "Afl," in *http://lcamtuf.coredump.cx/afl/*, 2018. [Online]. Available: http://lcamtuf.coredump.cx/afl/
[7] M. Höschele and A. Zeller, "Mining input grammars from dynamic taints," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016, 2016, pp. 720–725.
[8] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: machine learning for input fuzzing," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, G. Rosu, M. D. Penta, and T. N. Nguyen, Eds. IEEE Computer Society, 2017, pp. 50–59.
[9] C. Paduraru and M. Melemciuc, "An automatic test data generation tool using machine learning," in *Proceedings of the 13th International Conference on Software Technologies, ICSOFT 2018, Porto, Portugal, July 26-28, 2018.*, L. A. Maciaszek and M. van Sinderen, Eds. SciTePress, 2018, pp. 506–515.
[10] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.
[11] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, "Parallel symbolic execution for automated real-world software testing," in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys '11, 2011, pp. 183–198.
[12] V. Chipounov, V. Kuznetsov, and G. Candea, "The S2E platform: Design, implementation, and applications," *ACM Trans. Comput. Syst.*, vol. 30, no. 1, pp. 2:1–2:49, 2012.
[13] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, vol. 40, pp. 213–223, 06 2005.
[14] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13, 2005, pp. 263–272.

[15] B. Chen, C. Havlicek, Z. Yang, K. Cong, R. Kannavara, and F. Xie, "CRETE: A versatile binary-level concolic testing framework," in *Fundamental Approaches to Software Engineering, 21st International Conference, FASE 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, ser. Lecture Notes in Computer Science, A. Russo and A. Schürr, Eds., vol. 10802. Springer, 2018, pp. 281–298.

[16] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USA: USENIX Association, 2008, p. 209–224.

[17] P. Zhang, S. Elbaum, and M. B. Dwyer, "Automatic generation of load tests," in *2011 26th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE'11)*, 2011, pp. 43–52.

[18] B. Zhou, M. Kulkarni, and S. Bagchi, "Vrisha: Using scaling properties of parallel programs for bug detection and localization," 01 2011, pp. 85–96.

[19] B. Zhou, J. Too, M. Kulkarni, and S. Bagchi, "WuKong: automatically detecting and localizing bugs that manifest at large system scales," in *The 22nd Int. Symp. on High-Performance Parallel and Distributed Computing, HPDC'13*, M. Parashar, J. B. Weissman, D. H. J. Epema, and R. J. O. Figueiredo, Eds. ACM, 2013, pp. 131–142.

[20] J. Burnim, S. Juvekar, and K. Sen, "Wise: Automated test generation for worst-case complexity," in *2009 IEEE 31st Int. Conf. on Software Engineering*, 2009, pp. 463–473.

[21] C. Saumya, J. Koo, M. Kulkarni, and S. Bagchi, "Xstressor : Automatic generation of large-scale worst-case test inputs by inferring path conditions," in *2019 12th IEEE Conf. on Software Testing, Validation and Verification (ICST)*, 2019, pp. 1–12.

[22] J. Koo, C. Saumya, M. Kulkarni, and S. Bagchi, "Pyse: Automatic worst-case test generation by reinforcement learning," in *12th IEEE Conf. on Software Testing, Validation and Verification, ICST 2019*. IEEE, 2019, pp. 136–147. [Online]. Available: https://ieeexplore.ieee.org/xpl/conhome/8725538/proceeding

[23] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," *In Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, 2016. URL http://arxiv.org/abs/1509.06461*.

[24] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement learning for automatic test case prioritization and selection in continuous integration," in *Proc. of the 26th ACM SIGSOFT Int. Symp. on Software Testing and Analysis (ISSTA'17)*, 2017.

[25] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018.

[26] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, 1997.

[27] A. Manchin, E. Abbasnejad, and A. van den Hengel, "Reinforcement learning with attention that works: A self-supervised approach," *CoRR*, vol. abs/1904.03367, 2019. [Online]. Available: http://arxiv.org/abs/1904.03367

[28] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," published in Proceedings of International Conference on Learning Representations (ICLR) 2016.