Optimizing decision making in concolic execution using reinforcement learning

Ciprian Paduraru Miruna Paduraru Alin Stefanescu

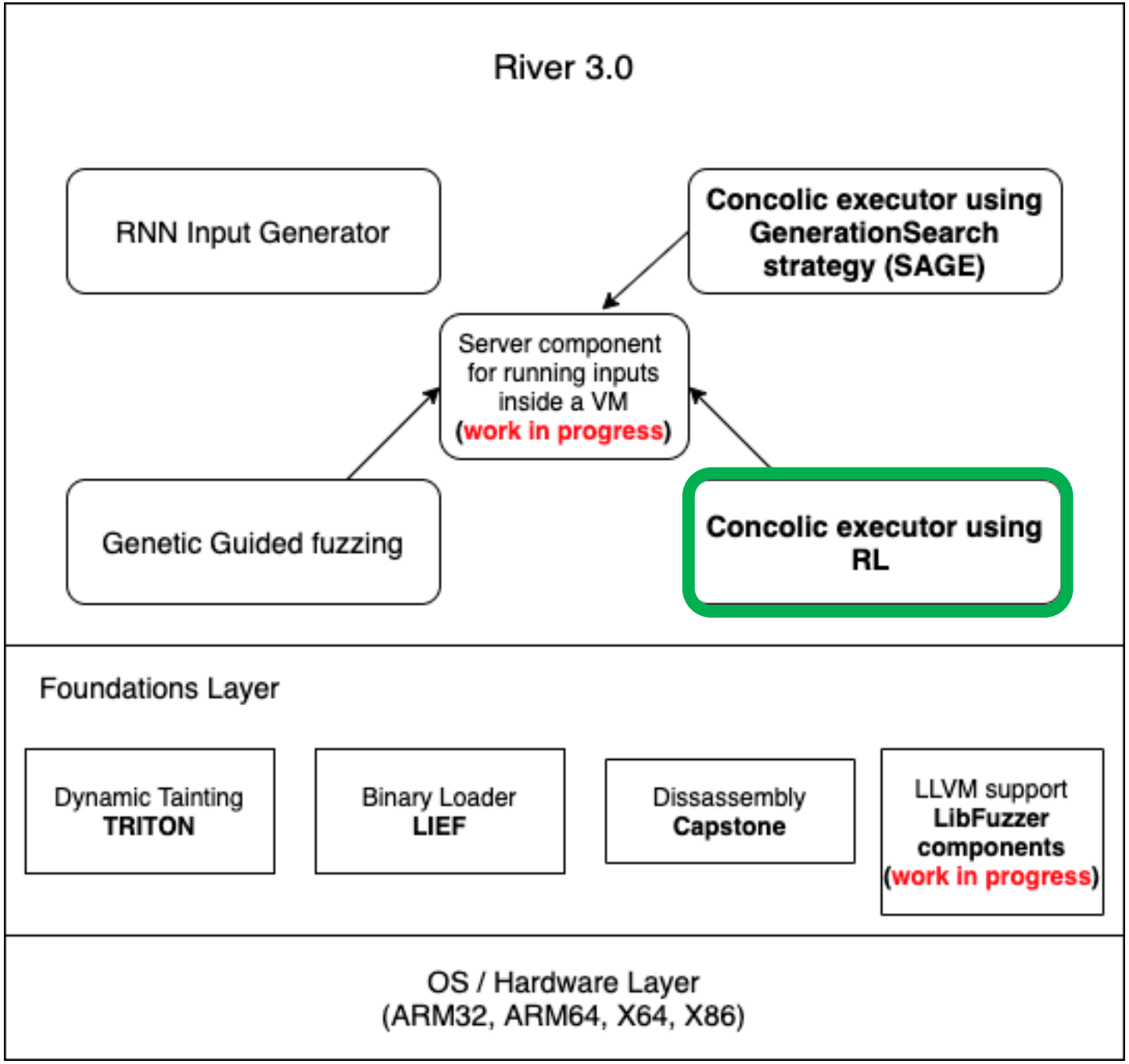
Department of Computer Science
University of Bucharest, Romania

Outline

1. Our framework - RIVER
2. Concolic execution and binary testing
3. A reinforcement learning (RL) solution for optimizing concolic execution
4. Evaluation
5. Future work

Our framework - RIVER

- An **open-source** framework for **fuzz testing** guided by **AI algorithms**
- Works at binary level: x86, x64, ARM32, ARM64.
- Cross-platform: Windows, Linux, MacOS.
- Available at https://river.cs.unibuc.ro / https://github.com/AGAPIA/river


River 3.0
RNN Input Generator
Concolic executor using GenerationSearch strategy (SAGE)
Server component for running inputs inside a VM (work in progress)
Genetic Guided fuzzing
Concolic executor using RL
Foundations Layer
Dynamic Tainting TRITON
Binary Loader LIEF
Dissassembly Capstone
LLVM support LibFuzzer components (work in progress)
OS / Hardware Layer
(ARM32, ARM64, X64, X86)

Concolic execution for binaries

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.

```
Search(inputSeed){
  inputSeed.bound = 0;
  workList = {inputSeed};
  Run&Check(inputSeed);
  while (workList not empty) {//new children
    input = PickFirstItem(workList);
    childInputs = ExpandExecution(input);
    while (childInputs not empty) {
      newInput = PickOneItem(childInputs);
      Run&Check(newInput);
      Score(newInput);
      workList = workList + newInput;
    }
  }
}
```

```
ExpandExecution(input) {
  childInputs = {};
  // symbolically execute (program,input)
  PC = ComputePathConstraint(input);
  for (j=input.bound; j < |PC|; j++) {
    if((PC[0..(j-1)] and not(PC[j]))
                    has a solution I){
      newInput = input + I;
      newInput.bound = j;
      childInputs = childInputs + newInput;
  }
  return childInputs;
}
```

Example

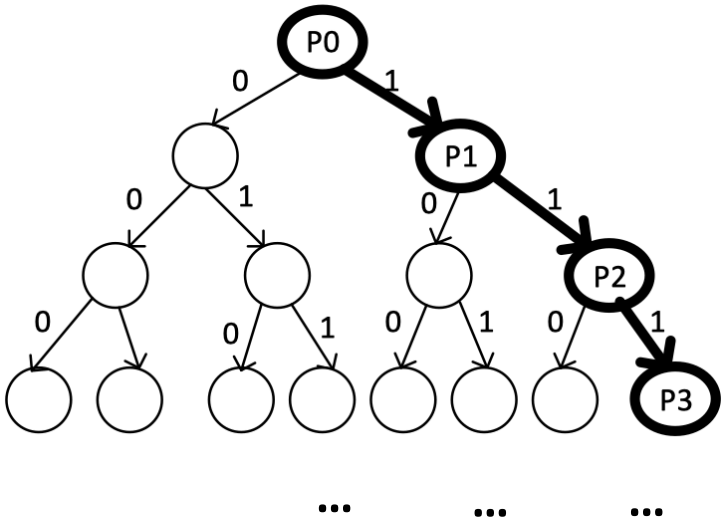
```
void test_simple(const unsigned char *input)
{

    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();
}
```

test_simple(“good”)

test_simple(“bad!”) - **abort**

An execution tree for obtaining the *abort* instruction
(0 on edge means branch not taken, 1 means taken)



Above, the branch causing problems is found **last**. **Can we optimize this?**

Optimization

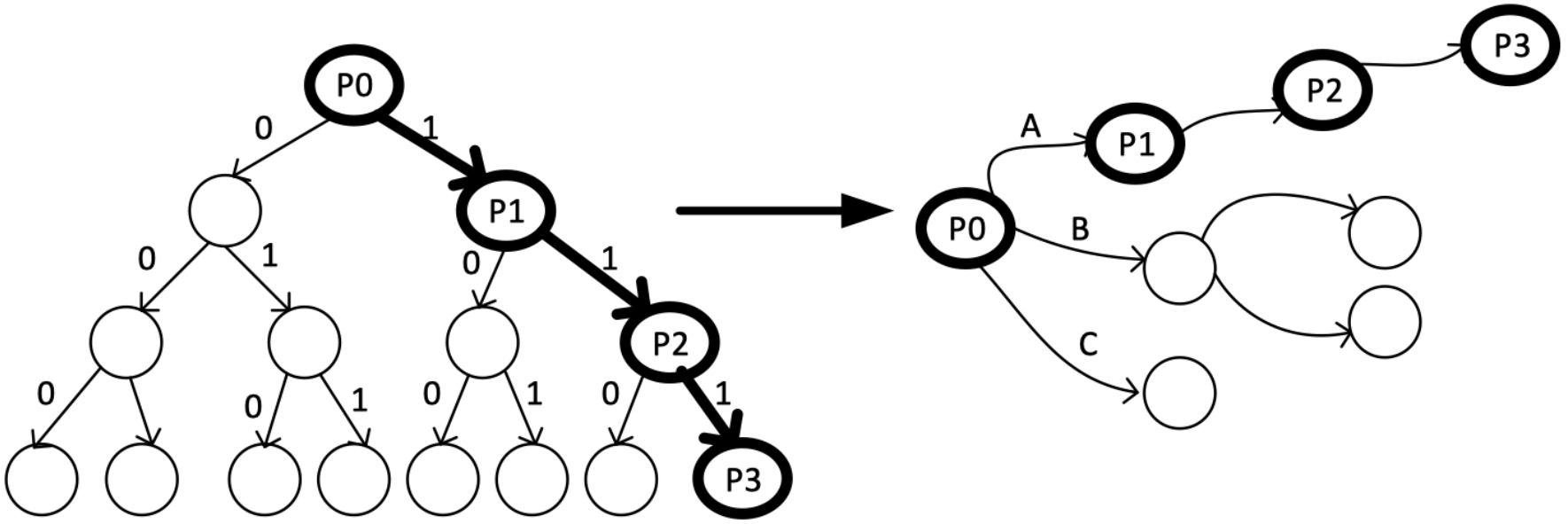
- **Empirical observation**: in concolic (and symbolic) execution, most of the execution time (>60-70%) is spent on the SMT evaluation of branch conditions

- **Speed-up improvement**: reduce the number of explored branches by choosing earlier the branches that could lead to a problematic state

- **Optimisation idea**:
  - assign **scores** to the decisions to take (i.e., which branch input to reverse)
  - prioritize the most relevant decisions and inputs, based on the scores.

- Our key contribution: use Reinforcement Learning (RL) to implement the above

Assigning scores to decision branches



If we assign **scores** A, B, and C, such that

$$A > B > C$$

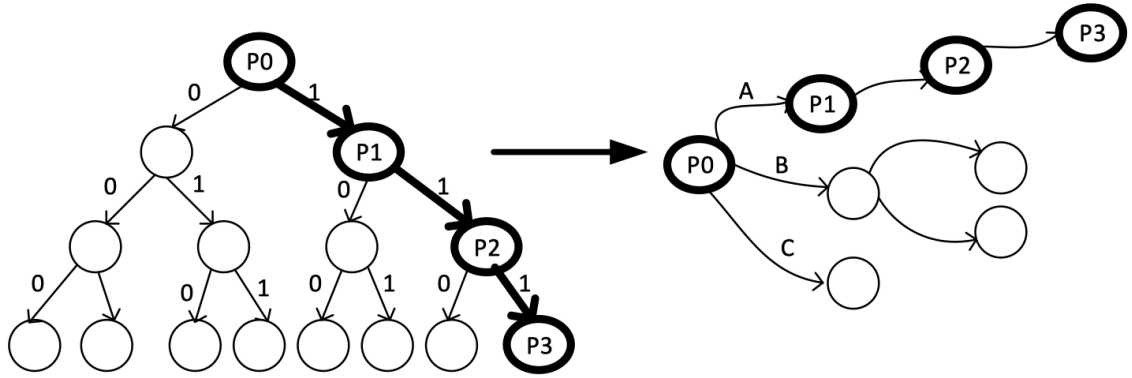
Then the firstly explored branch will be A to P1

so we move “faster” towards the target state

A RL solution for optimizing concolic execution

- We want to know in a **state,** i.e., an execution path constraint, which is the best **decision (action)** to take, i.e., which branch to inverse, to get higher **rewards (**e.g., new basic blocks, or whatever your testing target is)

- Can we estimate the value of an **action**?



- In RL terminology, estimate the State-Action value: **Q (state, action)**

Our RL setup

- The environment in our case is the program under test itself.

- When we submit a series of inputs $I_1, I_2, \ldots\ldots, I_N,$ we preserve the state of the environment (i.e., we reset everything between generation of inputs to have determinism).

- This setup is known as **contextual bandits** in Reinforcement Learning, because:
  - Consecutive actions do not affect the state of the environment
  - It has a state at each point (i.e., it is contextual)

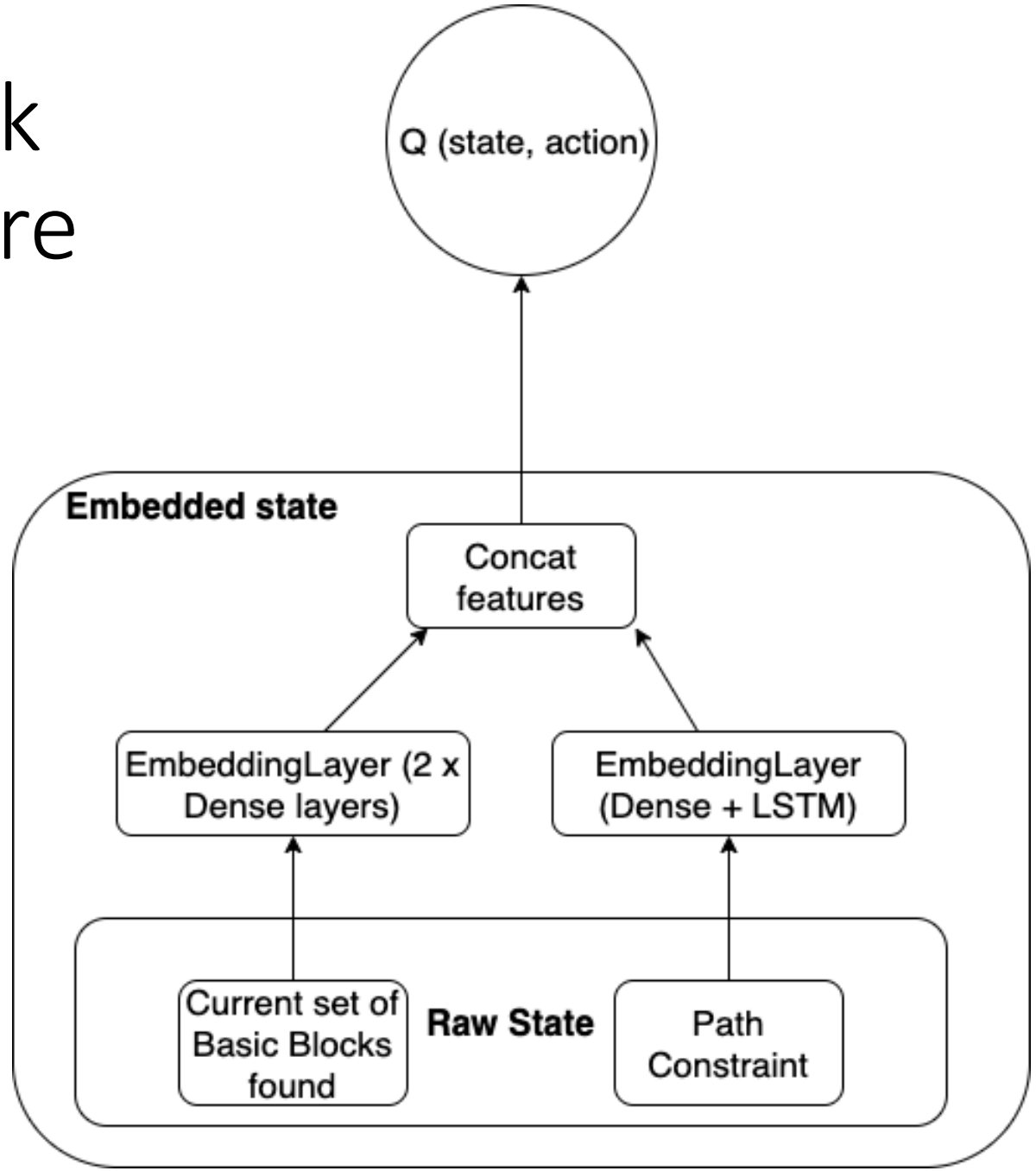
Our RL setup

- We estimate **Q (state, action)** in the environment (program under test), where:
  - ***state*** = (Embedding(basic blocks found so far) + Embedding(a start path_constraint found))
  - ***action*** = numeric value between [0 … length(path_constraint)-1]
  - ***reward*** = various functions (see next slides). E.g., the number of new basic blocks detected by taking an action in state

- We choose an episodic environment with online batches updating the model.

- Mathematically, we try to learn the value of actions in given states of the testing process, relative to testing goals.

RL network architecture

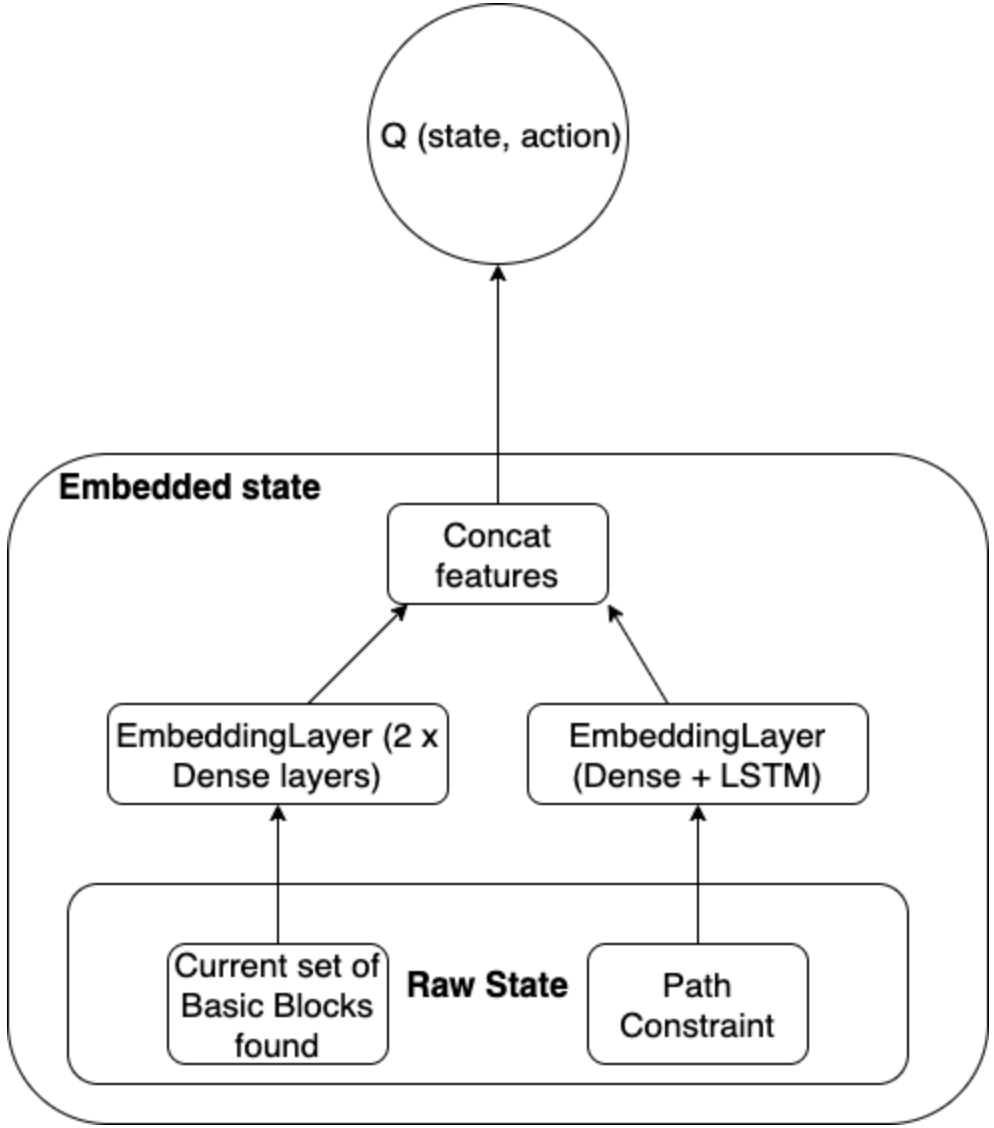


RL training approach

- Get a set of **online gathered experiences**,
  { (*State*, *Action*, *Reward*) },
  where *Reward is* obtained by the
  user's reward formula
  (with the testing objectives in mind)

- **Loss function** is MSE (mean square error) between *Reward* and the estimated values of *Q (State, Action)*

- Do backpropagation **at runtime** to optimize the network's output towards the real values



Example of a reward function - 1

Target: *Increase code coverage in the shortest time possible.*

This could be used in continuously developed software (CI/CD) when engineers submit source code and ideally immediate feedback should be received.

$$R(S, action, SNext) = E1 * (B(SNext) - B(S)) + E2 * (S.len - a + 1)$$

E1 and E2 parameters control the **tradeoff** between promoting the **new number of basic blocks found** in a new state resulting by applying an action and **level of parallelism** (applying an action at the top of the path constraint gives more parallelism opportunity later).

Example of a reward function - 2

Target: *Increase code coverage in coldspots, i.e., a collection of basic blocks that are known as very susceptible to issues.*

The reward function weights the importance of each newly discovered block by how often issues appeared at a given module and range of addresses in it.

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

where Stats is a dictionary with scores for each block address range that caused issues in the past:

$$Stats(ModuleID, [Offset_{Start}, Offset_{End}] = \frac{Issues\ in\ (ModuleID, [Offset_{Start}, Offset_{End}])}{Num\ issues\ reported\ in\ total}.$$

Example of a reward function - 3

Target: *Increase path size and/or execution time.*

There are two possible interesting cases here:

(a) if the path is feasible, i.e., the next state Snext can be reached, the function weights between the lengths of the paths and the time needed to trace the application and get that path.

(b) Otherwise, a user-defined parameter $P_{notSatisfiable}$ is applied to penalize the network for choosing actions that lead to unsatisfiable states.

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

Evaluation RQ1

Research Question 1: *Is the estimation function efficient? Is our method faster in obtaining a certain level of code coverage in comparison with the version without reinforcement learning?*

We let both methods running until they reached 100 different basic blocks on both HTTP and JSON parser programs.

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

Evaluation RQ2

Research Question 2: *Is the same model efficient between small source code changes?*

To evaluate this, we considered three different consecutive code commits (with small code fixed, between 10-50 lines modified) on both applications and averaged the time needed to reach again 100 different basic blocks.

The RiverConcolicRL method was trained on the base code, then the evaluation was done using the binary application built at the next code committed into the application's repository.

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

Evaluation RQ3

Research Question 3: *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 further with the binary application built at the second version for another 1h.

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

Some current drawbacks

Model training:

- Training takes time and resources.
- Previously shown results were trained for an initial time of 24h on a strong GPU Nvidia 2080 Ti.

Model calibration:

- Model hyperparameters might need to be calibrated depending on the application under test.
- We will investigate if clustering approaches and parameters could be more generic

Future work

RL techniques for concolic execution look promising!

- Try different RL algorithms
- Adapt to other types of problems and applications (e.g., IoT, cloud)
- Extend to stand-alone software that retains its state between inputs

Thank you!

Code snippets

```
def Predict(input, tailLen, action):
  // Choose the available model
  assert tailLen ≥ L_min
  K = min(tailLen, L_max) − L_min
  models[K].Predict(input, action)

def Train(NumMaxEpisodes):
  for episode in range(NumMaxEpisodes):
    input = seed new input
    RiverConcolic.SearchInputs(input, true)
    Save model and update training statistics

class RiverModel:
  def AddExperience(experience):
    memory.add(experience)
    experiencesSinceUpdate++
    if experiencesSinceUpdate ≥ T:
      experiencesSinceUpdate = 0
      batch = memory.selectBatch(N)
      optimize RiverDQN using batch
      num_optimizations++
      if num_optimizations > MaxUpdatesPerEpisodes:
        terminate episode

  def Predict(input, action):
    scores = model.Predict(input)
    return scores[action]

  ExperienceReplay memory
  RiverDQN model
```

```
class RiverConcolic:
  CheckAndScore(input, trainMode)
    score = null
    if input.v != null:
      Res = execute input.v using a SimpleTracer process
      if Res has issues:
        output(Res)
      score = ScoreHeuristic(input, Res)
    if trainMode != −1:
      NewPC = Run an AnnotatedTracerZ3 process with input.v
      RLModule.onNewExp(input, newPC, score)
    return score

  SearchInputs (initialInput, trainMode):
   initialInput.bound = 0
   // A priority queue of inputs holding on each item
   // the score and the concrete input buffer.
   PQInputs = {(0, initialInput)}
   Res = execute initialInput using a SimpleTracer process
   if Res has issues:
     output(Res)

   while (PQInputs.empty() == false):
    input = PQInputs.pop()
    // If the input was not executed symbolically yet
    // we run it before.
    if input.v == null:
      PC = input.PC
      i = input.action
      Solution = Z3Solver(
              PC[0..i−1] == same jump value as before
              and PC[i] == inversed jump value)
      if Solution == null:
       // If there is no solution we still send this further for
       // experience gathering purposes
       CheckAndScore(input, trainMode)
       continue
      input.v = overwrite Solution over input.parent
      CheckAndScore(input, trainMode)

    // Get the children of this input
    nextInputs = Expand(input)
    foreach newInput in nextInputs:
     // If the new input was executed symbolically,
     // then use the heuristics to score it
     if newInput.v != null:
       score = CheckAndScore(newInput, false)
       PQInputs.push((score, newInput))
    else: // Just use the estimated score
       PQInputs.push((newInput.estScore, newInput)
```