RiverConc: An Open-source Concolic Execution Engine for x86 Binaries

Ciprian Paduraru[1,2], Bogdan Ghimis[1,2] and Alin Stefanescu[1,2]
[1]*Department of Computer Science, University of Bucharest, Romania*
[2]*Research Institute of the University of Bucharest, Romania*
{*ciprian.paduraru, bogdan.ghimis, alin*}*@fmi.unibuc.ro*

Keywords: Security, Threats, Vulnerabilities, Concolic, Symbolic, Execution, Testing, Tool, Binaries, x86, Tainting, Z3, Reinforcement Learning.

Abstract: This paper presents a new open-source testing tool capable of performing concolic execution on x86 binaries. Using this tool, one can find out ahead of time of potential bugs that can enable threats such as process hijacking and stack buffer overflow attacks. Although a similar tool, SAGE, already exists in literature, it is closed-sourced and we think that using its description to implement an open-sourced version of its main novel algorithm, Generational Search, is beneficial to both industry and research communities. This paper describes, in more detail than previous work, how the components at the core of a concolic execution tool, such as tracers, dynamic tainting mechanisms and SMT solvers, collaborate together to ensure code coverage. Also, it briefly describes how reinforcement learning can be used to speed up the state of the art heuristics for prioritization of inputs. Research opportunities and the technical difficulties that the authors observed during the current development of the project are presented as well.

1 INTRODUCTION

Software testing is a very important concept nowadays from multiple perspectives. Firstly, it can save important resources for companies because finding bugs in the early stages of software development can be solved faster. Secondly, from the security perspective, it can find vulnerabilities in the source code that could potentially lead to hacking and stolen information. Lastly, in terms of the product quality perspective and in order to better satisfy the customers it is also important that the application must be well tested against different scenarios.

Several strategies and tools were created to automate software testing, as discussed in Section 2. This paper describes an open-source tool that implements a concolic execution engine at x86 binary level, without using the source code, similar to the one reported in SAGE (Godefroid et al., 2012). To the best of our knowledge, it is the first one at the moment of writing this paper. The motivation for re-implementing the Generational Search strategy for concolic execution and making it open source stems from the fact that, according to authors, SAGE has had an important impact in finding issues of Microsoft's software suite over time. Note that our tool is different from (Bucur et al., 2011) or (Chen et al., 2018) in the way that we are not using LLVM at all, and the user provides us with a raw binary x86 build as input, together with the payload input address and with the execution's starting point. This is more appropriate to a real execution and testing process.

Contribution of this Paper. We believe that making available such a open-source implementation could have an important impact on both industry and research community. For industry, the repository can act as a free alternative for software testing. We think that testing and security engineers or quality assurance engineers can also benefit from such a product. For the research community, we unlock several opportunities also detailed in Section 3.4. As an example, in concolic execution, the backend framework has a queue of newly generated input. This queue requires a prioritization mechanism that is decisive in obtaining better testing results (i.e., code coverage, interesting braches) with less computational resources. By making it open-source, the research community can contribute and test new prioritization techniques that can lead to significant results in the future. Also, we describe the technical architecture and implementation in detail such that the community can extend the framework on various points. We also present some difficulties that occurred during development and our vision to continue improving the project from vari-

ous perspectives. On top of re-implementing the previously state of the art concolic execution tool and making it open-source, the framework adds an innovation on the field of x86 binary concolic execution: it replaces the inputs prioritization heuristic with a complete reinforcement learning strategy that trains agents to score the importance of states and actions (Paduraru et al., 2020). Thus, instead of having a fixed heuristic score for all inputs, as described in (Godefroid et al., 2012), our attempt is to learn the importance of changing conditions along a program execution trace by having an agent that explores an application binary code and optimizes a decision making policy. One of the contributions of our implementation for using reinforcement learning (RL) strategies is the refactoring of the original execution framework to make it friendly with RL interfaces.

The paper is structured as follows. The next section presents some existing work. Section 3 presents the architecture, implementation details and future work plan for the framework. Evaluation of the tool and methods are discussed in Section 4. Finally, conclusions are given in the last section.

2 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 fundamentally known technique is *fuzz testing*, in which the test data is automatically generated using random inputs. The well-known limitation of fuzz testing is that it is very hard to produce good, meaningful inputs since many of them will be malformed, thus the important ones have a very low probability to happen only by fuzzing.

The three main categories of fuzz testing that currently used by the community are: blackbox random fuzzing (Sutton et al., 2007), where we use the program under test as a black box - looking only at the input and the output, whitebox random fuzzing (Godefroid et al., 2012), where we know what the program is actually doing and we craft the inputs accordingly, and grammar based fuzzing (Purdom, 1972), (Sutton et al., 2007), where we use grammars to generate new inputs. For blackbox fuzzing, since we view the program as a black box, the methods used in practice are also augmented and enhanced with other strategies for better results. As an example, in (Paduraru et al., 2017) (also part of the RIVER test suite) and AFL (american fuzzy lop)[1], genetic algorithms, and various heuristics are used to find threats faster and to achieve good code coverage. Grammar based fuzzing can be viewed as a model-based testing approach, in which, given the input grammar, the generation of inputs can be done either randomly (Sirer and Bershad, 1999), (Coppit and Lian, 2005) or exhaustively (Lämmel and Schulte, 2006). Autogram, mentioned in (Höschele and Zeller, 2016) tries to relief the user of the tedious task of manually defining the grammar and can learn CFGs given a set of inputs and observing the parts of the input that were used by the program (dynamic taint analysis). Recent work concentrates also on learning grammars automatically, using recurrent neural networks, see (Godefroid et al., 2017) and (Paduraru and Melemciuc, 2018).

Symbolic execution is another strategy used for automated software testing (King, 1976). While theoretically it can provide more code coverage, it has serious challenges because of the possible paths exponential growth. At the moment, there are two approaches for implementing symbolic execution: online symbolic execution and concolic execution. Briefly, concolic execution works by executing the input on the program under test and gathering all the branch points met during the execution together with their conditions. Its advantage over the online symbolic execution is that, at the end of each execution, a trace containing the branch points and their conditions are obtained, thus it can be used to generate offline a new set of inputs. Because of this advantage, it is more suitable than online symbolic execution when applied to large applications having hundreds of millions of instructions.

In the field of online symbolic execution, two of the most common open-source used frameworks are KLEE (Bucur et al., 2011) and S2E (Chipounov et al., 2012). Several concolic execution engines are also presented in the literature. Some early work is represented by DART (Godefroid et al., 2005) and CUTE (Sen et al., 2005), which both operate on the source code level. CRETE (Chen et al., 2018), which is based on KLEE, operates on LLVM representation, in contrast to RIVER, which operates at x86 level.

The most related tool that also works on x86 level is SAGE (Godefroid et al., 2012). We follow the same algorithms and provide an open-source implementation of it. In addition to the original work, we were able to fill up some missing pieces of the puzzle and come up with own ideas about the implementation of various components, e.g., how a task can be split in a distributed environment execution or how the tainting works at byte level in collaboration with the SMT

[1] http://lcamtuf.coredump.cx/afl/

```
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();
}
```

Figure 1: Example of a simple function that the user might want to evaluate. The example is taken from SAGE paper (Godefroid et al., 2012).

solver, thus providing a more detailed technical description for such a concolic execution tool.

Triton (Salwan and Saudel, 2015), which was mostly used in software protection against deobfuscation of source code, is a dynamic binary analysis framework containing similar components to RIVER: a symbolic execution engine using Z3 and a dynamic taint engine. Also, it provides Python bindings for easier interaction, an AST representation of the program under test, and works not only at binary x86, but also at x86-64 and ARM32 level. An open-source Python based example of reimplementation of the SAGE strategy for a small code example can be found in a Github source code example of the tool. However, our implementation works at a lower-level by providing C++ based access to main components such as tainting, works with binary x86 programs, and adds reinforcement learning capabilities in addition to SAGE.

3 RIVER FRAMEWORK FOR CONCOLIC EXECUTION

The framework is available for evaluation at

http://river.cs.unibuc.ro

It currently offers an automatic installer for Linux users and many-core execution options. We are currently working on making it available as a VM image that has RIVER preinstalled and an online service that provides concolic execution on demand. In this section, we provide an overview containing a user guide, a high-level technical description of how we do taint analysis and how we use an SMT solver to get new different execution branches. Finally, we present some future plans based on the opportunities observed during the development.

```
extern "C" {
        DLL_PUBLIC unsigned char payloadBuffer[MAX_PAYLOAD_BUF];
        DLL_PUBLIC int Payload() {
                test_simple(payloadBuffer);
                return 0;
        }
};
```

Figure 2: Example of user declaration of input payload buffer and application entry point. These two symbol names will be searched by RIVER to feed new data input tests.

3.1 Overview and Usage

For easier exemplification purposes, we use the same test function as in SAGE (Godefroid et al., 2012) - Fig. 1.

Assuming that this is the code that the user wants to test and get full code coverage for it through concolic execution, the following steps must be done on the user side:

1. On top of the user's existing source code, add a symbol named *payloadBuffer* (which must be a data buffer). This will be used by RIVER to send input to the user's tested application.
2. Add a function symbol named *Payload*, which marks the entry point of the application under test.
3. Build the program for x86.
4. Run the tool with a command, optionally specifying a starting seed input (in the exemplification, the input seed is "good") and the number of processes that can be used.

The code for the first two steps can be seen in Fig. 2. The user receives live feedback through an interface about the execution status, plus folders on disk with inputs that caused issues grouped by category, such as: SIGBUS, SIGSEGV, SIGTERM, not classified, SIGABRT, and SIGFPE. For this simple example, the abort will be hit in less than 1 millisecond and the input buffer containing string "bad!" will be added to the SIGABRT folder.

3.2 Architecture and Implementation Overview

Details about RIVER architecture were presented in previous work (Stoenescu et al., 2017). Note that the previous version of the tool did not support Concolic Execution, which we introduce now as RiverConc. We kept using Z3 (De Moura and Bjørner, 2008) as SMT solver for solving branches' conditions. We also implemented a novel reinforcement learning algorithm in order to optimize the search. For this paper, we sketch the description only for the main components that help our presentation purposes:

1. SimpleTracer - executes a program with a given input and returns a trace, represented by a list of basic blocks encountered during execution. A basic block is a continuous set of assembler instructions ending with a jump. For instance, the first basic block in Fig. 4 starts at address *ea*7 and ends at *ebc* with a conditional jump.

2. AnnotatedTracerZ3 - similar to the one above. The difference between them is that this execution uses dynamic taint analysis and returns as output the Z3 serialized jump conditions for each branch in the trace that caused the move from the current basic block to the next. By using dynamic taint analysis, the conditions always involve combinations of bytes indices from the *payloadBuffer* sent to the program. It is easy then to ask Z3 solver to give an input (i.e., bytes values for the affected input part) that inverses the original jump condition value.

Since for debugging purposes we kept using a textual output, we are able to show the output of this component in Fig. 3 based on executing the input payload "good" over the example function given in Fig. 1. The disassembly of this program is also presented in Fig. 4.

```
Test: f61a9e97 - Taken f61a9eb2, NotTaken f61a9eae. Was taken ? Yes
1 0
2 f61a9e97 f61a9ef2
(= (bvnot (ite (= @0 #x62) #b1 #b0)) $f61a9e97)

Test: f61a9eb2 - Taken f61a9ec3, NotTaken f61a9ebf. Was taken ? Yes
1 1
1 f61a9eb2
(= (bvnot (ite (= @1 #x61) #b1 #b0)) $f61a9eb2)

Test: f61a9ec3 - Taken f61a9ed4, NotTaken f61a9ed0. Was taken ? Yes
1 2
1 f61a9ec3
(= (bvnot (ite (= @2 #x64) #b1 #b0)) $f61a9ec3)

Test: f61a9ed4 - Taken f61a9ee5, NotTaken f61a9ee1. Was taken ? Yes
1 3
1 f61a9ed4
(= (bvnot (ite (= @3 #x21) #b1 #b0)) $f61a9ed4)
```

Figure 3: Example of a textual debugging output result by evaluating "good" payload input against the function shown in Fig. 1, whose dissassembled code is shown in Fig. 4.

Note that in the tested function code, there are four branches (marked with red lines in Fig. 4) thus, the output contains four branch descriptions as output. In each branch (beginning with "Test" in the textual output), the first line contains in order: the address of the tested jump instruction in the binary, the basic block addresses to go if the branch is taken or not taken, and a boolean representing whether the jump was taken or not with the current input given. Note that because the user code is loaded inside RIVER process at runtime, the disassembly code's addresses are offset in the output. The second line contains the number of bytes indices used by the jump condition from the initial payload input buffer, along with the indices of those bytes. The last line is important since it shows the Z3 textual output condition needed for each branch point to take the same value as in the input given. If we would like the program to take a different path than before, Z3 solver can be asked to give values for the negated condition. Each condition is based on the initial input payload buffer indices. Note for example the @1 symbol in the second test, suggesting that the condition there is over the byte index 1 from the input buffer.

```
00000ea7 <test_simple>:
 ea7:   55                      push   %ebp
 ea8:   89 e5                   mov    %esp,%ebp
 eaa:   83 ec 18                sub    $0x18,%esp
 ead:   c7 45 f4 00 00 00 00    movl   $0x0,-0xc(%ebp)
 eb4:   8b 45 08                mov    0x8(%ebp),%eax
 eb7:   0f b6 00                movzbl (%eax),%eax
 eba:   3c 62                   cmp    $0x62,%al
 ebc:   75 04                   jne    ec2 <test_simple+0x1b>
 ebe:   83 45 f4 01             addl   $0x1,-0xc(%ebp)
 ec2:   8b 45 08                mov    0x8(%ebp),%eax
 ec5:   83 c0 01                add    $0x1,%eax
 ec8:   0f b6 00                movzbl (%eax),%eax
 ecb:   3c 61                   cmp    $0x61,%al
 ecd:   75 04                   jne    ed3 <test_simple+0x2c>
 ecf:   83 45 f4 01             addl   $0x1,-0xc(%ebp)
 ed3:   8b 45 08                mov    0x8(%ebp),%eax
 ed6:   83 c0 02                add    $0x2,%eax
 ed9:   0f b6 00                movzbl (%eax),%eax
 edc:   3c 64                   cmp    $0x64,%al
 ede:   75 04                   jne    ee4 <test_simple+0x3d>
 ee0:   83 45 f4 01             addl   $0x1,-0xc(%ebp)
 ee4:   8b 45 08                mov    0x8(%ebp),%eax
 ee7:   83 c0 03                add    $0x3,%eax
 eea:   0f b6 00                movzbl (%eax),%eax
 eed:   3c 21                   cmp    $0x21,%al
 eef:   75 04                   jne    ef5 <test_simple+0x4e>
 ef1:   83 45 f4 01             addl   $0x1,-0xc(%ebp)
 ef5:   83 7d f4 03             cmpl   $0x3,-0xc(%ebp)
 ef9:   7e 10                   jle    f0b <test_simple+0x64>
 efb:   83 ec 0c                sub    $0xc,%esp
 efe:   68 de 0f 00 00          push   $0xfde
 f03:   e8 fc ff ff ff          call   f04 <test_simple+0x5d>
 f08:   83 c4 10                add    $0x10,%esp
 f0b:   90                      nop
 f0c:   c9                      leave
 f0d:   c3                      ret
```

Figure 4: Disassembly code for the function under test in Fig. 1. The lines marked with red underlines are jump conditions ending basic blocks.

The condition translated from Z3 is equivalent to:

```
if payloadBuffer[1] == x61
  then jump condition = TRUE
  else jump condition = FALSE
```

(note that x61 is the hex ASCII code for the $'a'$ character, so the test corresponds exactly to the second *if* condition in the tested user function). Solving for the

negated condition for any of the four tests and leaving the rest intact would potentially get a new path in the application. For example, in the first test (as seen in Fig. 3) using the initial input seed "good", the jump condition will be taken because byte 0 (@0) does not have value x62 ('*b*', as ASCII). Looking at Fig. 4, the first jump (*jne*) means that if the compare condition is not true, then it will go to the next condition block (*if* statement in the original source code). If the condition is changed and Z3 solver is asked to give a value for byte index 0 such that the condition is inversed and it will give value x62. With the new payload content ("**b**ood'), the first jump will not be taken next time, and the counter instruction (at address *ebe*) will be executed. After some iterations that modified the conditions, the "bad!' content is obtained and the *abort* instruction will be executed.

The branching conditions that depend on the input buffer will eventually be added to the Z3 conditions. The mechanism behind variables tracking is the dynamic taint analysis component implemented inside RIVER explained better in (Paduraru et al., 2019).

3. RiverConc - this new component orchestrates the concolic execution process. The system acts as a centralized distributed system, and the processes will be used for spawning tracer components of type *SimpleTracer* and *AnnotatedTracerZ3*. The *AnnotatedTracerZ3* component takes significantly more time and usually, we tend to spawn more processes in order to get the results faster. The "master" process will be a "RiverConc" process, while tracer processes will be "slaves". The communication is done using sockets, with components exchanging binary data messages.

RiverConc uses the same algorithm explained in (Godefroid et al., 2012), named *Generational Search*. As mentioned in that paper, their search over input solution space is adapted for applications with very deep paths and large input spaces. The algorithm adapted to our components and architecture is shown in Listings 1 and 2. Note that the variable "bound" is used to avoid redundancy in the search, while the heuristic that scores inputs in the priority queue tries to promote block coverage maximization: it basically counts how many unseen blocks the evaluated input has generated. Intuitively, as the input successfully generates new code blocks, it can go further and discover some other new ones.

Limitations. Because some standard or operating system functions produce divergences (i.e., if the same input is executed multiple times against the same program it can give different results), we evaluated these and replaced their code at initialization time in RIVER environment with empty stubs. This is called in the literature *imperfect symbolic execution*.

Listing 1: The main search function that generates new inputs implemented in RiverConc component. It is called with the initial input seed specified by user, or a random input if no starting seed is specified.

```
SearchInputs (initialInput):
 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()
  // Execute the input and get the branch conditions.
  // For each branch, we can get a new child as shown
  // in the Expand function's pseudocode
  nextInputs = Expand(input)
  foreach newInput in nextInputs:
    Res = execute newInput using a SimpleTracer process
    if Res has issues: output(Res)
    score = ScoreHeuristic(newInput)
    PQInputs.push((score, newInput))
```

Listing 2: 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.

```
Expand(input):
 childInputs = []
 // Get the Z3 conditions for each jump (branch) encountered
 // during execution. In our example, PC contains four entries
 // one for each of the branch points.
 PC = Run an AnnotatedTracerZ3 process with input

 // Take each condition index and inverse  only that one,
 // keeping the prefix with the same jump value
 for i in range(input.bound, PC.length):
   // Solution will contain the input byte indices and their
   // values, which need to be changed to inverse the i'th
   // jump condition
   Solution = Z3Solver (
       PC[0..i-1] == same jump value as before and
       PC[i]== inversed jump value)
   if Solution == null: continue
   newInput = overwrite Solution over input
   // no sense to inverse conditions again before i'th branch,
   // i.e., prevent backtracking.
   newInput.bound = i
   childInputs.append(newInput)

 return childInputs
```

3.3 RiverConc with Reinforcement Learning Techniques

This subsection describes how our open-source concolic execution engine uses RL techniques to optimize the number of calls to the SMT solver for testing the target specified by the user. The interested party can read (Paduraru et al., 2020) for more theoretical information on the used methods and their evaluation.

Our idea with RiverConc is to improve the existing score heuristic by learning an estimation method using reinforcement learning in order to estimate the score of different actions, using the SMT sparsely. We do this because we want to speed up the process by reducing uninteresting inputs. Ideally, the estimation method should sort the available options by im-

portance as close to reality as possible. This could lead to obtaining the same results faster by evaluating symbolically a smaller subset of branch points (e.g., detection of inputs that cause issues, code coverage). In our implementation, we use Deep RL techniques (van Hasselt et al., 2015), by using a network that estimates $Q(state, action)$. The *state* is represented by a sequence of branch points resulted by running the *AnnotatedTracerZ3* component against a given input. At each branch point it can understand from the output in which module and offset the branch decision occured, what is the Z3 condition for that branch, and if the jump was taken or not with the given input (Eq. 1). Then, Z3 can be asked to solve the condition and give a new input that reverses any of the branch decision taken by the application with the initial input.

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

$$Branch_i = \{ModuleID, Offset, Z3\_cond, taken\} \quad (2)$$

The *action* is represented by the index of the branch condition to be inversed. Thus, the network estimates how important it is to inverse any of the branch decisions in the current state. If it is good enough, then many of the unpromising changes are pruned and the Z3 solver will be sparingly called. The training process is organized in *episodes*, which can end either if a maximum number of iteration has been reached, or if there are no more inputs in the queue to estimate and process further. At the beginning of each episode, the seed is being randomized or chosen randomly from a set of user given set of seeds. The *reward* function can be adjusted by the user depending on the test targets. For example, if the target is to get better code coverage, one can count how many new different blocks of x86 instructions were obtained by starting from a given state and performing a given action (i.e., choosing a certain branch point condition in the current state and inversing it to get a new input). More sample reward functions can be found in (Paduraru et al., 2020).

3.4 Future Plans, Lessons Learned and Research Opportunities

One of the most important things, in order to find reasonable code coverage in a short time, is to implement efficient input prioritization strategies. We use the same greedy strategy and prioritization formula as given in the SAGE paper (described above). Our plan is to investigate better strategies, especially those based on both classic AI methods and RL. We would like to focus on improving the time and finding a relationship about how to find a better mapping from the x86 code to the decision making process.

We also plan to move from the many-core implementation to a completely distributed environment using the state-of-the-art framework Apache Spark. This is motivated by at least three facts:

- nowadays, both industry and research have access to large distributed computational clusters;
- concolic execution is very resource demanding and parallelization on a single computer with many cores cannot scale very well for applications with deep paths and large inputs;
- it is difficult and error-prone to ensure resilience using sockets communications in such environments, since the slave processes can crash constantly due to bugs in the original source code.

Through profiling, we noticed that the Z3 solver was the bottleneck for the *RiverConc* component. While using reinforcement learning techniques can partially solve this problem, a topic of interest for our team in the short future is to reuse some of KLEE's optimization strategies, i.e., use a decorator pattern and optimize Z3 queries (e.g., the solver will be executed as the last resort if the expression is not already in the cache or is not redundant in the given context).

Also, we plan to experiment with more heuristics in order to increase the tool coverage. In particular, we will investigate how the reversible operations in our executions as implemented in the RIVER tool (Stoenescu et al., 2017) can be used in conjunction with heuristics that make use of state restoration, such as RFD (river formation dynamics) (Rabanal et al., 2017).

4 EVALUATION

So far, the evaluation of our solution was done using three applications: an XML parser[2], JSON parser[3], and HTTP parser[4]. On top of their code, we added only the symbols declared in Section 3.1 to inject the payload buffer and mark the entry code of the binary resulted after building the solution. The experiments described below were done on a 6-Core Intel i7 processor with 16 GB RAM and an Nvidia RTX 2070 video card, on Ubuntu 16.04.

[2]http://xmlsoft.org

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

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

4.1 Evaluation using the Same Heuristics Described in (Godefroid et al., 2012)

We focused on the coverage metric keeping a database of input tests: how many different basic blocks of a program were evaluated using all the available tests generated by the tool and how much time did we spend to get to that coverage. Time is an important criterion, because the software must be tested continuously; if the testing process is slow, it might not scale with the speed of development. The number of different basic blocks was obtained using *Simpletracer*, after spending intervals of 30 minutes, 1h, 2h, and 3h of generating new inputs in two ways:

- Using the tool presented in this paper - the River concolic executor (*RiverConc* component)
- Previous work combining dynamic taint analysis with fuzz based on genetic algorithms (Paduraru et al., 2019) (with performances similar to AFL) on all three applications under test.

At each interval we stored in different folders the inputs generated by each tool, then we run *SimpleTracer* to tell how many basic blocks were obtained on each individual application, time interval, and method used. The comparative results are presented in Tables 1, 2, 3.

Table 1: The number of basic blocks touched in comparison between the two methods on the XML parser application.

Model	30m	1h	2h	3h
River fuzzing	621	765	783	815
RiverConc	291	402	809	863

Table 2: The number of basic blocks touched in comparison between the two methods on the HTTP parser application.

Model	30m	1h	2h	3h
River fuzzing	73	89	97	111
RiverConc	28	52	88	109

Table 3: The number of basic blocks touched in comparison between the two methods on the JSON parser application.

Model	30m	1h	2h	3h
River fuzzing	50	79	94	97
RiverConc	32	61	90	104

The results show that the concolic execution tool has the potential to get more coverage if left running for longer. This is somewhat expected because tracing an application symbolically demands more computational effort. But the tradeoff is that for some of the branches, the SMT solver will give the right condition instead of rolling the dice until it gets in.

We plan to evaluate the new tool as soon as possible on other popular benchmarks, such as (Dolan-Gavitt et al., 2016). Also, after we migrated to a massively distributed environment, we plan to test the performance and analyze different distribution strategies.

4.2 Evaluation using RL Strategies

Firstly, we are interested to see how efficient the estimation function is after training a model for 24h. We test if such a model can obtain faster a certain level of code coverage in comparison with the version without RL. In this case, we let both methods running until they reached 100 basic block on both the HTTP and JSON parser. The comparative times are shown in Table 4. The results show that the trained RL 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). Even if the model was pre-trained for 24h, this still matters because it can be used as a better starting point and can be reused between code changes.

Table 4: Comparative time to reach 100 different block code coverage on the two tested applications.

Model	HTTP parser	JSON parser
RiverConc	2h:10m	2h:53m
RiverConcRL	1h:37m	2h:14m

As a secondary test, we want to check how efficient is a trained model between small code changes. Thus, we considered three different consecutive code submits (with small code fixed, between 10-50 lines modified) averaging the time needed to reach again 100 basic blocks. The *RiverConcRL* 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 5. These results suggest that the estimation models can be re-used between consecutive code changes efficiently.

Table 5: Averaged comparative time to reach again 100 different block code coverage on the two tested applications, using three different consecutive code submits.

Model	HTTP parser	JSON parser
RiverConc	1h:56m	2h:47m
RiverConcRL	1h:31m	2h:15m

5 CONCLUSIONS

We presented an open-source framework for concolic execution of programs at binary level. The paper described its architecture, implementation details and our perspective for future work ideas. We hope that this will help the community and industry to test their strategies for concolic execution easier than before and also that we will get contributions, support, and feedback on our source code repository. By having the first framework that is able to do concolic execution for x86 binaries, we were able to optimize the number of SMT queries using RL strategies. New research opportunities and ideas that occurred during development were also presented.

ACKNOWLEDGEMENTS

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

REFERENCES

Bucur, S., Ureche, V., Zamfir, C., and Candea, G. (2011). Parallel symbolic execution for automated real-world software testing. In *Proc. of the EuroSys'11*, pages 183–198. ACM.

Chen, B., Havlicek, C., Yang, Z., Cong, K., Kannavara, R., and Xie, F. (2018). CRETE: A versatile binary-level concolic testing framework. In *Proc. of FASE 2018*, volume 10802 of *LNCS*, pages 281–298. Springer.

Chipounov, V., Kuznetsov, V., and Candea, G. (2012). The S2E platform: Design, implementation, and applications. *ACM Trans. Comput. Syst.*, 30(1):2:1–2:49.

Coppit, D. and Lian, J. (2005). yagg: An easy-to-use generator for structured test inputs. In *Proc, of ASE'05*, pages 356–359. ACM.

De Moura, L. and Bjørner, N. (2008). Z3: An efficient SMT solver. In *Proc. of TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer.

Dolan-Gavitt, B., Hulin, P., Kirda, E., Leek, T., Mambretti, A., Robertson, W., Ulrich, F., and Whelan, R. (2016). LAVA: Large-scale automated vulnerability addition. In *SP'16*, pages 110–121. IEEE Computer Society.

Godefroid, P., Klarlund, N., and Sen, K. (2005). DART: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223.

Godefroid, P., Levin, M. Y., and Molnar, D. (2012). SAGE: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27.

Godefroid, P., Peleg, H., and Singh, R. (2017). Learn&fuzz: machine learning for input fuzzing. In *Proc. of ASE'17*, pages 50–59. IEEE Computer Society.

Höschele, M. and Zeller, A. (2016). Mining input grammars from dynamic taints. In *Proc. of ASE'16*, pages 720–725. ACM.

King, J. C. (1976). Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394.

Lämmel, R. and Schulte, W. (2006). Controllable combinatorial coverage in grammar-based testing. In *Proc. of TestCom'06*, volume 3964 of *LNCS*, pages 19–38. Springer.

Paduraru, C. and Melemciuc, M. (2018). An automatic test data generation tool using machine learning. In *Proc. of ICSOFT'18*, pages 506–515. SciTePress.

Paduraru, C., Melemciuc, M., and Ghimis, B. (2019). Fuzz testing with dynamic taint analysis based tools for faster code coverage. In *Proc. of ICSOFT'19*, pages 82–93. SciTePress.

Paduraru, C., Melemciuc, M., and Stefanescu, A. (2017). A distributed implementation using Apache Spark of a genetic algorithm applied to test data generation. In *Proc. of GECCO'17 workshops*, pages 1857–1863. ACM.

Paduraru, C., Paduraru, M., and Stefanescu, A. (2020). Optimizing decision making in concolic execution using reinforcement learning. In *Proc of ICST'20 workshops*, pages 52–61. IEEE.

Purdom, P. (1972). A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12(3):366–375.

Rabanal, P., Rodríguez, I., and Rubio, F. (2017). Applications of river formation dynamics. *Journal of Computational Science*, 22:26–35.

Salwan, J. and Saudel, F. (2015). Triton: A dynamic symbolic execution framework. In *Symp. sur la sécurité des tech. de l'inform. et des comm.*, pages 31–54. Online at http://triton.quarkslab.com. SSTIC.

Sen, K., Marinov, D., and Agha, G. (2005). CUTE: A concolic unit testing engine for C. In *Proc. of ESEC/FSE'13*, pages 263–272. ACM.

Sirer, E. G. and Bershad, B. N. (1999). Using production grammars in software testing. *SIGPLAN Not.*, 35(1):1–13.

Stoenescu, T., Stefanescu, A., Predut, S., and Ipate, F. (2017). Binary analysis based on symbolic execution and reversible x86 instructions. *Fundamenta Informaticae*, 153(1-2):105–124.

Sutton, M., Greene, A., and Amini, P. (2007). *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional.

van Hasselt, H., Guez, A., and Silver, D. (2015). Deep reinforcement learning with double Q-learning. In *Proc. of AAAI'16*, pages 2094–2100. AAAI Press.