RIVER 2.0: An Open-Source Testing Framework using AI Techniques

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

Miruna Paduraru*
miruna.paduraru@drd.unibuc.ro
Dept. of Computer Science
University of Bucharest
Romania

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

ABSTRACT

This paper presents the latest updates to the RIVER open-source testing platform for x86 programs, focusing on how artificial intelligence (AI) techniques can be used to improve the automated testing processes. It is also important to mention that RIVER is the first open-source platform that offers a concolic execution engine with reinforcement learning capabilities. On the industry side, this can allow security software engineers to test their applications with fewer costs, while for the research community, it can help prototyping new ideas faster. As a secondary contribution, our work makes a summary of the AI techniques that were used for testing processes either in our previous work or other existing work in the field. The presentation describes technical aspects, challenges, and future work.

CCS CONCEPTS

• **Security and privacy** → **Software and application security**; • **Software and its engineering** → **Software testing and debugging**; **Dynamic analysis**; • **Computing methodologies** → **Machine learning**.

KEYWORDS

Reinforcement learning, neural networks, deep learning, concolic, symbolic, execution, testing, x86, tainting, SMT

ACM Reference Format:
Bogdan Ghimis, Miruna Paduraru, and Alin Stefanescu. 2020. RIVER 2.0: An Open-Source Testing Framework using AI Techniques. In *Proceedings of the 1st ACM SIGSOFT International Workshop on Languages and Tools for Next Generation Testing (LANGETI '20), November 8–9, 2020, Virtual, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3416504.3424335

1 INTRODUCTION

Security is becoming more and more pervasive in the software engineering field and people that write code must check their code for bugs, using specialized programs, in order to ensure that they do not deploy code that hackers can use to their advantage. Since this problem cannot easily be solved, computer scientists constantly try to use the lastest advances in emerging fields like machine learning (ML) to improve their testing software such that more bugs can be identified as potential threats at development time and eliminated.

RIVER 2.0 is a tool that can test x86 executable code, which uses ML algorithms in order to get better code coverage. We discuss different strategies and present several tools for automatic software testing in Section 2.

Contribution of this paper: This tool paper describes the RIVER 2.0 testing framework, focusing mainly on the progress made in the last few years over the previous version of RIVER 1.0 [21]. Overall, the development moved along two directions:

(1) Using AI techniques to generate or guide inputs propagation such as genetic algorithms, recursive neural networks or reinforcement learning.
(2) Providing open-source and ready for use components to the testing community that did not exist, were not updated, or are deprecated. Such a component that was added and provided recently as open-source software is a concolic execution engine similar to SAGE [8]. We emphasize that, from our knowledge, this is the first open-source implementation of a concolic execution for x86 binaries. On top of reimplementing the component using their documented work, we added reinforcement learning methods to score inputs and optimize the inputs' prioritization searching strategy.
(3) Adding together several components that were studied either in our previous work or related ones in the field. The purpose of this is the unification of small programs and studies such that they work in the same framework and use a unified interface.

Our new framework is available open-source for evaluation at https://river.cs.unibuc.ro. It currently offers an automatic installer for Linux users and many-core execution options. We are also working to make it available as a virtual machine (VM) image with RIVER preinstalled and an online service, such that users can evaluate it easier.

Since this paper gives only a short high-level overview, an interested reader can find more technical details in the RIVER 2.0 documentation available as an appendix in [2].

The paper is structured as follows. The next section presents some existing work in the field that had an impact on our new framework version. Section 3 presents the RIVER 2.0 architecture overview and a description of the most important components in the framework. Finally, conclusions and future work plans are given in the last section.

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

2 RELATED WORK

One of the most popular ways to test a programming application is to generate random inputs and to observe how the software will react to them. In software literature, this is called *fuzz testing*, and usually, there is a supervisor that monitors the application for crashes and then informs the user about the input that produces that crash. One way to test one's application is to look for the code coverage metric, representing the number of touched lines or basic blocks (representing a list of assembly instructions followed by a jump) that the program encountered during an automated test.

An important factor in fuzz testing is the automatic creation of meaningful inputs, that are not rejected in the early execution of the program, but will allow the program to dive deeper into its code. Using only fuzz testing by generating random inputs, good code coverage cannot be achieved, since inputs usually follow some specification, like headers, and if they are malformed, the program will exit prematurely.

Fuzz testing can be divided in three categories: white-box random fuzzing, black-box random fuzzing [23], and grammar-based fuzzing [19], [23]. Each of these contains different methods to cope with the main limitation of fuzz testing.

In white-box fuzzing, we know what the program is supposed to do, and because of this we can generate valid parts, for example for the headers and use random bits in the rest. In black-box fuzzing, we have access only to the input and to the output and in these cases there exists augmented methods such as [17] and [1], which that use genetic algorithms or heuristics to handle cases where one does not know what exactly is being tested.

In grammar-based testing, one supplies the program with a grammar describing the input (manually [20], [6], or automatically, like Autogram [11], where a CFG grammar is learnt). Recent work involves using machine learning, especially recurrent neural networks (RNNs) for the automatic learning of the grammar.

Another approach for doing software testing is by using symbolic [12], [3], [5] and concolic [8] execution of the program. Through symbolic execution, we can assign for each variable a symbolic value and if we want to see how we can get to a certain place, we can solve the aggregated conditions along that path. Hypothetically, this is a good solution for the code coverage problem, but the number of possible paths grows exponentially with the input. One way to get better results is to use concolic execution, meaning that we keep some variables as symbolic, but we also use the concrete value of other variables that we can get at runtime.

RIVER 2.0 works at x86 binary level and requires only the executable, the address of the function from the executable to be tested, and an input, Based on them, it offers information about the basic blocks that were encountered during that specific execution. In the latest version, we use machine learning algorithms, especially reinforcement learning in order to find inputs that reach better code coverage.

3 RIVER 2.0 FRAMEWORK OVERVIEW

The architecture of RIVER 2.0 is shown in Fig. 1. Bolded rectangles in the top-right of the figure show the new components added from the previous major release. The interested reader can check the user guide in [2].

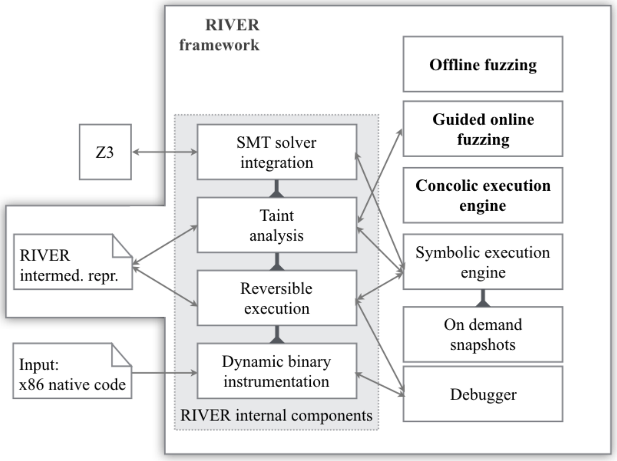

Figure 1: The architecture of *RIVER 2.0* is composed of smaller components that interact with one another in order to produce traces used for code coverage analysis. RIVER takes as an input a binary x86 program, which is passed to an internal *Dynamic binary instrumentation* tool, which allows us to find the basic blocks (assembly blocks followed by a jump) needed by the high-level components presented in this paper, namely *Offline fuzzing*, *Guided online fuzzing*, *Concolic execution engine*. As a side note, RIVER framework uses Z3 as the SMT solver, which is in turn used by the *Concolic execution engine*.

3.1 Architecture and Implementation Overview

Details about the first RIVER version architecture were presented in [21]. Meanwhile, some components were slightly adapted to support concolic execution, and we detailed these changes on the project's repository address and in [2]. We kept using Z3 [7] as the SMT solver for solving branch conditions. For this paper, we only briefly present the previous components and try to focus on the new components given the space limit.

3.2 Taint Analysis

A *SimpleTracer* sub-component executes a program P with a given input *test* and returns a trace, representing an ordered list of branch instructions that a program encountered in the execution:
SimpleTracer(P, *test*) $= B_0 B_1 \ldots B_n$. Because a program can make calls to other libraries or system executables, each branch is a pair of the module name and offset where the branch instruction occurred: $B_i = (module, offset)$. Note that a program is divided into basic blocks, which are sequences of x86 instructions that contain exactly one branch instruction at its end. The *AnnotatedTracer* component [15] is similar to *SimpleTracer*, excepting that it adds dynamic taint analysis by linking the inputs given to program P to where it is used in the x86 code, and returns as output the Z3 serialized jump conditions for each branch in the trace that caused the process to move from the current basic block to the next. By using dynamic taint analysis, the conditions involve always combinations of bytes

indices from the input sent to the program. It is then possible to ask Z3 solver to solve the conditions, which in turn will give a new input, i.e., the values of the bytes in the input that affect the jump condition, which inverses the original jump condition value. [2] completes the explanation given about the taint analysis component with concrete examples.

3.3 Online Guided Fuzzer

In [17] we presented ideas for using a genetic algorithm that guides the fuzzing process. A fitness function is proposed, based on the "probabilities" that certain branch conditions occur in some order and use them to guide the tests towards areas not yet explored. Assume we have an initial set of test inputs *TestDataSet* that produces (using the *SimpleTracer* module) a set of traces *Traces* of the program P. Then we can evaluate the probability of one branching instruction B_i to occur immediately after another given B_j as follows. Informally, this probability, denoted by $\mathrm{Prob}(B_i, B_j)$ is the number of all occurrences of sequence (B_i, B_j) in traces in the set *Traces* divided by the number of all occurrences in *Traces* of sequences (B_i, B) for any branching B. Formally, we distinguish two cases:

(1) If there exists at least one occurrence of (B_i, B_j) in *Traces*, then

$$\mathrm{Prob}(B_i, B_j) := \frac{\text{no\_of\_occurrences}(B_i, B_j, \mathit{Traces})}{\text{no\_of\_occurrences}(B_i, B, \mathit{Traces})}$$

where $\text{no\_of\_occurrences}(B_i, B_j, \mathit{Traces})$ is the number of distinct occurrences of (B_i, B_j) in the set of traces: $\sum_{trace \in Traces} \mathrm{card}(\{\, k \mid (T_k, T_{k+1}) = (B_i, B_j) \text{ in } \mathit{trace} \,\})$, and
$\text{no\_of\_occurrences}(B_i, B, \mathit{Traces})$ is the number of distinct occurrences of
(B_i, B) for any B in the set of traces: $\sum_{trace \in Traces} \mathrm{card}(\{\, k \mid (T_k, T_{k+1})$
in *trace* and $T_k = B_i \,\})$.

(2) If there is no occurrence of (B_i, B_j) in *Traces*, we want to avoid probabilities with value 0, by using a value smaller than the minimum of the probabilities for pairs that appear in *Traces*:

$$\mathrm{Prob}(B_i, B_j) = \min(\{\mathrm{Prob}(B_x, B_y) \mid (B_x, B_y) \text{ in } \mathit{Traces}\}) * F$$

where F is a factor between $[0.1, 0.5]$ (we used $F = 0.2$ in our experiments). This trick helps us differentiate between the fitness of two different traces that both contain an edge with no occurrence in *Traces*.

Now we can use $\mathrm{Prob}(B_i, B_j)$ to define the fitness of a trace $\mathit{trace} := B_0 B_1 \ldots B_n$, $n > 0$ produced by an individual as follows:

$$\mathrm{Fitness}(\mathit{trace}) := 1 - \prod_{(B_i, B_{i+1}) \in \mathrm{Distinct}(\mathit{trace})} \mathrm{Prob}(B_i, B_{i+1})$$

where $\mathrm{Distinct}(B_0 B1 \ldots B_n) := \{(B_i, B_{i+1}) \mid 0 \le i < n\}$, i.e., the set of pairs from the trace with no repeated elements.

Guiding fuzzing with genetic algorithms was also studied in [1] with heuristics that obtain even better code coverage. Our current implementation uses Apache Spark [24] to drive the search in a distributed environment and has the potential to do fuzzing at a large scale. We also implemented a web-like interface to show progress and control various variables.[2] gives more details about this component for the interested reader.

3.4 Offline Guided Fuzzer

Starting from a set of given input examples, this component addresses the problem of creating generative models that are able to produce more similar inputs. The problem is also studied in [9], [14], and [16]. The motivations for this method comes from the fact that large input file types (such as the ones used for text editors, e.g. PDFs, XMLs) have a certain structure, and it would be hard for a generic fuzzing mechanism to produce tests that are not rejected early in the execution, because of an incorrect format, thus it would be difficult to achieve high coverage. The generative models are based on a *seq2seq* [22] architecture using Recurrent Neural Networks (RNNs) and LSTM cells. The training process uses the sequences of bytes from the input examples. In the test generation process (inference time) instead of producing deterministic outputs from the learned model, the method uses a little fuzzing and decides with a fixed probability if it should take the output from the model or just output a random byte. This probability can be understood as exploration versus exploitation and plays an important role in the tradeoff between catching issues in the software under test and producing correct inputs (Fig. 2a). The implementation also follows our previous work in [14], which analyzes which parts of the inputs are actually used for branching decisions and tries to produce generative models only for the pieces of inputs that tend to be important in the branching process. Figure 2b show how this method works. The motivation for guiding the offline generative model this way is that in many cases, applications such as image or text processing only have a few areas in their structure that are used for branching, i.e., playing a role in obtaining code coverage. Learning models that would produce similar images or text content might not be of interest in the testing process, but changing variables that decide what to do when the image has different sizes or channels could present more interest. On top of the methods mentioned above, our current implementation also supports unsupervised clustering of inputs by first studying their features and grouping similar ones in the same category. This is important since usually, companies could have a large corpus of test files clustered by applications and versions. From time to time, this set is upgraded with new smaller datasets. Using our method, one can simply append the new input examples to existing clusters or add new test input classes. Since the generative RNNs models can be trained online, the user conducting the training process has the option to say how important is the new dataset in comparison with the old dataset, i.e. the learning rate used for adding the new data on top of the previously trained model). More details about this component can be found in [2].

3.5 Concolic Execution Engine

RIVER 2.0 contributes to the concolic execution engines field with two important aspects:

(1) It provides an open-source implementation of a concolic execution engine at x86 binary level, similar to the one reported in SAGE [8]. From the authors' knowledge, it is the first open-source available solution for x86 binaries [13]. The

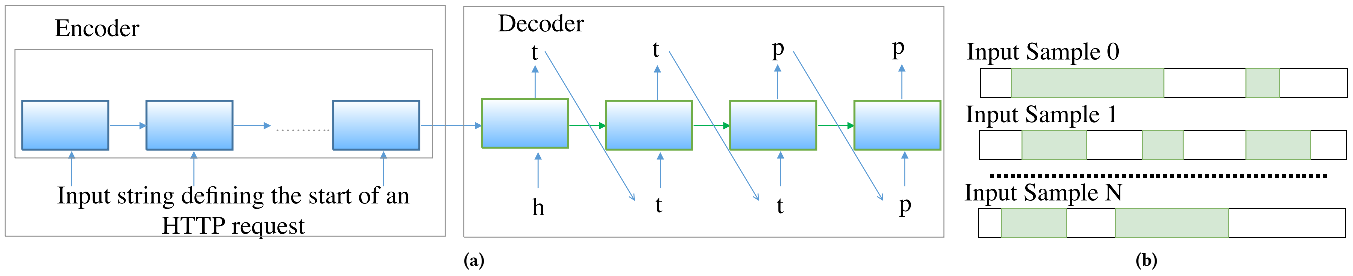

Figure 2: (a) The architecture of the *seq2seq* encoder-decoder network that learns to generate text for HTTP requests. In the generation process, the input is fuzzed with a small probability to inject intentioned mistakes, which could lead to issues in the application (b) Example of input samples are portions (green colors) that are used for branching. Our generative models learn the pattern in those areas. When the method is used to produce new inputs, one of the structures is sampled and the set of generative models learnt from that structure are used to produce a new input. Structures are grouped by similarity to avoid explosion of sampling possibilities and generative models.

motivation for implementing such a tool 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 [3] or [4] 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 and closer to a real execution and testing process. We believe that making available such an open-source engine could have an important impact on both industry and research community. For industry, the repository can act as a free alternative to do software testing. We think that testing and security engineers or quality assurance engineers can benefit from such a product.

(2) On top of the open-source implementation, as an innovation to the field, we proposed a reinforcement learning (RL) technique 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 [18]. Practically, we plan to reduce the set of SMT solver calls made for obtaining inputs for paths that are not promising. 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:
   (a) The execution environment described in [8] was refactored and modeled to support a reinforcement learning environment.
   (b) An estimation network capable of predicting how "valuable" are the branch condition changes in different states. The estimation can be done using deep reinforcement learning techniques [10], by using a Deep Q-Network that estimates the values of all actions possible from a given state, i.e., $Q(state, action)$. In our context, the *state* is a path constraint object obtained by executing the program under test symbolically using an input with *AnnotatedTracer*. A path constraint is defined in Eq. 1: an ordered list of blocks, where each element contains the module and the offset where the branch instruction occurred, the Z3 condition at each branch, and a flag that suggests if the condition was taken or not with the input used. The *action* describes which of the Z3 conditions in the *PC* between indices $[bound, PC.len - 1]$ is to be inverted (the *bound* index variable is used to prevent backtracking and it is stored in the *PC* state). At the beginning of each *episode*, 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 *PC*s since the beginning of the training process). An episode can finish due to two possible reasons: (a) there are no more inputs to process in the execution queue, or (b) the maximum number of iterations has been reached.

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

$$BranchDesc_i = \{ModuleID, Offset, Z3condition, taken\} \tag{2}$$

   The internal architecture of the network estimation is a Long short-term memory network (LSTM) that tries to understand the patterns of sequences between blocks in the code, and how important is one path relative to the others. By *exploration*, the agent behind the RL method tries new actions that could lead to discovering new branches or issues in the code, while by *exploatation* the agent is able to use the current learned policy that it is known to match the testing targets as defined by the reward functions.
   (c) A set of reward functions that could tackle different targets when performing software testing. For example, to get as much code coverage as possible from the testing process, one could use Eq. 3, listed in [2]. Considering that $B(State)$ is a function that gives the set of different blocks that *State* touches in the tested application (using *SimpleTracer* for example), and that *SNext* is the next state of the agent after applying *action* in state *S*, 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. 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.

$$R(S, action, SNext) = E1 * (B(SNext) - B(S)) + E2 * (S.len - a + 1) \tag{3}$$

(d) A basic evaluation of the proposed mechanisms, showing its advantages in testing a software application during its development lifecycles.

At the implementation level, the concolic execution feature supports parallel execution using a centralized distributed system architecture. The processes will be used for spawning tracer components of type *SimpleTracer* and *AnnotatedTracer*. The *AnnotatedTracer* component takes significantly more time, thus the recommendation is to spawn more processes executing them. The "master" process will be the component process itself, while tracer processes will be "slaves". The communication is done using sockets, with components exchanging binary data messages.

More details about our reinforcement learning implementation and the pseudocode behind the implementation can be found in [2]. We present here some tables from [2] containing the evaluation of our framework.

Table 1: Time in hours to train models on different number of epochs and using 12.000 files for PDF and XML, and 100.000 HTTP requests as training dataset.

Num epochs	HTTP	XML	PDF objects
50	8h:25	7h:19	9h:11
40	6h:59	5h:56	8h:04
30	5h:35	4h:20	6h:15
20	3h:48	3h:42	4h:17
10	2h:10	1h:12	3h:02

Table 2: The average time needed to produce 10.000 new inputs for PDF and XML files, and 50.000 new HTTP requests.

File type	Time in minutes
XML	49
HTTP	25
PDF	51

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.*

Table 3: The number of branch instructions touched in comparison between random fuzzing driven by genetic algorithms, Sample and SampleSpace models for HTTP requests.

Model	9h	15h	24h	72h
HTTP-fuzz+genetic	229	230	230	232
HTTP-Sample	238	249	257	271
HTTP-SampleSpace	241	245	269	279

Table 4: The number of branch instructions touched in comparison between random fuzzing driven by genetic algorithms, Sample and SampleSpace models for HTTP requests.

Model	9h	15h	24h	72h
HTTP-fuzz+genetic	229	230	230	232
HTTP-Sample	238	249	257	271
HTTP-SampleSpace	241	245	269	279

4 CONCLUSIONS AND FUTURE PLANS

This paper presented RIVER 2.0 framework and an overview of how AI techniques can be used to help increase automation in software testing. We hope that this will help the community and industry to test their strategies for things such as concolic execution easier than before and also that we will get contributions, support, and feedback on our source code repository.

During development we found that Z3 solver takes a lot of computational time on the *Riverexp* component, even if the output Z3 expressions were optimized on the *AnnotatedTracerZ3* processes using $Z3\_simplify$. In this perspective, our next version will detach *Riverexp* from solving Z3 tasks. The solver will run on dedicated processes. We also plan to use some of KLEE's strategies that 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). We also plan to invest time in providing ready-to-use VMs and online services to evaluate our framework easier without local installation.

We plan to reintegrate the reversible execution capabilities inside RIVER, as explained in more detail in [21]. This would allow us to do more online symbolic execution with fewer resources used, since our unique framework feature allows us to reverse to a previous state without having to snapshot the entire state of the program, just by keeping a shadow stack of registers being destroyed by the current execution path.

ACKNOWLEDGMENTS

This work was supported by the Romanian National Authority for Scientific Research and Innovation (UEFISCDI) through ATLAS project no. 17PCCDI/2018 and ITEA3 MEASURE project no. PN-III-P3-3.5-EUK-2016-0020.

REFERENCES

[1] 2018. AFL. In *http://lcamtuf.coredump.cx/afl/*. http://lcamtuf.coredump.cx/afl/
[2] 2020. *Technical details for RIVER 2.0 framework.* Appendix, 44 pp. Available online at https://github.com/AGAPIA/river/blob/master/appendix.pdf.

[3] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. 2011. Parallel Symbolic Execution for Automated Real-world Software Testing. In *Proceedings of the Sixth Conference on Computer Systems* (Salzburg, Austria) *(EuroSys '11)*. ACM, New York, NY, USA, 183–198. https://doi.org/10.1145/1966445.1966463

[4] Bo Chen, Christopher Havlicek, Zhenkun Yang, Kai Cong, Raghudeep Kannavara, and Fei Xie. 2018. 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 (Lecture Notes in Computer Science)*, Alessandra Russo and Andy Schürr (Eds.), Vol. 10802. Springer, 281–298. https://doi.org/10.1007/978-3-319-89363-1_16

[5] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2012. The S2E Platform: Design, Implementation, and Applications. *ACM Trans. Comput. Syst.* 30, 1 (2012), 2:1–2:49. https://doi.org/10.1145/2110356.2110358

[6] David Coppit and Jiexin Lian. 2005. Yagg: An Easy-to-use Generator for Structured Test Inputs. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering* (Long Beach, CA, USA) *(ASE '05)*. ACM, New York, NY, USA, 356–359. https://doi.org/10.1145/1101908.1101969

[7] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

[8] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing: SAGE Has Had a Remarkable Impact at Microsoft. *Queue* 10, 1 (Jan. 2012), 20–27. https://doi.org/10.1145/2090147.2094081

[9] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. 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*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 50–59. https://doi.org/10.1109/ASE.2017.8115618

[10] Hado van Hasselt, Arthur Guez, and David Silver. 2016. Deep Reinforcement Learning with Double Q-Learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence* (Phoenix, Arizona) *(AAAI'16)*. AAAI Press, 2094–2100.

[11] Matthias Höschele and Andreas Zeller. 2016. Mining Input Grammars from Dynamic Taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE 2016)*. ACM, New York, NY, USA, 720–725. https://doi.org/10.1145/2970276.2970321

[12] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394. https://doi.org/10.1145/360248.360252

[13] Ciprian Paduraru, Bogdan Ghimis, and Alin Stefanescu. 2020. RiverConc: An Open-source Concolic Execution Engine for x86 Binaries. In *Proceedings of the 15th International Conference on Software Technologies, ICSOFT 2020, Lieusaint, Paris, France, July 7-9, 2020*, Marten van Sinderen, Hans-Georg Fill, and Leszek A. Maciaszek (Eds.). ScitePress, 529–536. https://doi.org/10.5220/0009953905290536

[14] Ciprian Paduraru and Marius-Constantin Melemciuc. 2018. 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.*, Leszek A. Maciaszek and Marten van Sinderen (Eds.). SciTePress, 506–515. https://doi.org/10.5220/0006836605060515

[15] Ciprian Paduraru, Marius-Constantin Melemciuc, and Bogdan Ghimis. 2019. Fuzz Testing with Dynamic Taint Analysis based Tools for Faster Code Coverage. In *Proceedings of the 14th International Conference on Software Technologies, ICSOFT 2019, Prague, Czech Republic, July 26-28, 2019*, Marten van Sinderen and Leszek A. Maciaszek (Eds.). SciTePress, 82–93. https://doi.org/10.5220/0007921300820093

[16] Ciprian Paduraru, Marius-Constantin Melemciuc, and Miruna Paduraru. 2018. Automatic Test Data Generation for a Given Set of Applications Using Recurrent Neural Networks. In *Software Technologies - 13th International Conference, ICSOFT 2018, Porto, Portugal, July 26-28, 2018, Revised Selected Papers (Communications in Computer and Information Science)*, Marten van Sinderen and Leszek A. Maciaszek (Eds.), Vol. 1077. Springer, 307–326. https://doi.org/10.1007/978-3-030-29157-0_14

[17] Ciprian Paduraru, Marius-Constantin Melemciuc, and Alin Stefanescu. 2017. A distributed implementation using apache spark of a genetic algorithm applied to test data generation. In *Genetic and Evolutionary Computation Conference, Berlin, Germany, July 15-19, 2017, Companion Material Proceedings*, Peter A. N. Bosman (Ed.). ACM, 1857–1863. https://doi.org/10.1145/3067695.3084219

[18] Ciprian Paduraru, Miruna Paduraru, and Alin Stefanescu. 2020. Optimizing decision making in concolic execution using reinforcement learning. In *13th IEEE International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2020, Porto, Portugal, October 24-28, 2020*. IEEE, 52–61. https://doi.org/10.1109/ICSTW50294.2020.00025

[19] Paul Purdom. 1972. A sentence generator for testing parsers. *BIT Numerical Mathematics* 12, 3 (01 Sep 1972), 366–375. https://doi.org/10.1007/BF01932308

[20] Emin Gün Sirer and Brian N. Bershad. 1999. Using Production Grammars in Software Testing. *SIGPLAN Not.* 35, 1 (Dec. 1999), 1–13. https://doi.org/10.1145/331963.331965

[21] Teodor Stoenescu, Alin Stefanescu, Sorina Predut, and Florentin Ipate. 2017. Binary Analysis based on Symbolic Execution and Reversible x86 Instructions. *Fundam. Inform.* 153, 1-2 (2017), 105–124. https://doi.org/10.3233/FI-2017-1533

[22] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2* (Montreal, Canada) *(NIPS'14)*. MIT Press, Cambridge, MA, USA, 3104–3112.

[23] Michael Sutton, Adam Greene, and Pedram Amini. 2007. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional.

[24] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proc. of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012*. USENIX, 15–28.