Appendix A User guide

A.1 Concolic executor and online fuzzing

For easier exemplification purposes that is uniform across our framework components, we use the same test function as in SAGE [2] - Fig. 11.

```
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. 3: Example of a simple function that the user might want to evaluate. The example is taken from SAGE paper [2].

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. 4. 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.

A.2 Offline fuzzer usage

The component need as input two folder paths and one variable:

1. A path to an existing test corpus and the generative models created by training process and grouped in clusters by input / application type. This is named *destination*.

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

Fig. 4: 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.

2. A path to the new test inputs that user wants to clusterize automatically and append to the existing test corpus.
3. A variable specifying how important is the new dataset, between 0.0 and 1.0. The existing generative models in the *destination* will be updated for a few epochs with the new dataset. A value of 1.0 would mean for example that the user wants to forget totally what the models learnt before.

At the end of this process, the *destination* will be updated with the new datasets, and the previously generative models are updated.

Appendix B Taint analysis details

The *Taint analysis* component uses two sub-components that are explained in more datails below using examples.

1. SimpleTracer - executes a program with a 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 continuous set of assembler instructions ending with a jump. For instance, the first basic block in Fig. 6 starts at address *ea*7 and ends at *ebc* with a conditional jump.

2. AnnotatedTracerZ3 - similar to the one above, it executes a program with a given input and returns a trace. 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 moved from the current basic block to the next. By using dynamic taint analysis, the conditions involve always 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. 5 based on executing the input payload "good" over the example function given in Fig. 11. The disassembly of this program is also presented in Fig. 6.

Note that in the tested function code, there are four branches (marked with red lines in Fig. 6) thus, the output contains four branch descriptions as output.

In each branch (beggining 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 offseted 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. In this case specifically, the condition translated from Z3 is equivalent to:
$if \quad payloadBuffer[1] == x61$
$then \quad jump \quad condition = TRUE$
$else \quad jump \quad condition \quad is \quad FALSE.$ (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 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. 5) using the initial input seed "good", the jump condition will be taken because byte 0 (@0) does not have value x62 (corresponding to ASCII character $'b'$). Looking at Fig. 6, the first jump (*jne* - jump if not equal) 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 input buffer content (i.e., "**b**ood'), the first jump will not be taken next time, and the counter instruction (at address *ebe*) will be executed first before moving to the next basic block. After some iterations that modified the conditions, the "bad!' content is obtained for payload input buffer and the *abort* instruction will be executed, raising an issue for the user. If the original source code test condition implies an entire tree of variables that in the end depends on parts of the payload input buffer, the corresponding bytes indices will be added in the Z3 condition matching the condition tree over values. The mechanism behind variables tracking is the dynamic taint analysis component implemented inside RIVER as explained in more detail in our previous work [29]. Note also that in a normal execution environment (not debugging), we are serializing the output and send it as binary data between components.

```
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)
```

Fig. 5: Example of a textual debugging output result by evaluating "good" payload input against the function shown in Fig. 11, whose dissassembled code is shown in Fig. 11.

Appendix C Online guided fuzzing component details

C.1 Initialization, selection and genetic operations

The initialization module generates a configurable number N of initial individuals in a genetic population G. The initial generation of individuals (input data) is obtained using a uniform random distribution for each gene (considering also the hints given by the user for different ranges of input variables). The initial population of individuals is then improved over a configurable number of generations using the usual genetic algorithm operators: selection, mutation and crossover.

Selection is based on the fitness function presented in the previous subsection. The selection method in our implementation is a mix between *Elitism* [30], *Rank-based Roulette Wheel Selection* [31] and random selection, each one with a given percentage. One of the input parameters of our test data generation is the number of individuals to preserve between population, which will be denoted by K. Then, EL will represent the percentage of individuals selected with Elitism, RW using the Rank-based Roulette Wheel Selection and RR using random selection such that $EL + RW + RR = 1$.

The idea behind elitism is to keep between consecutive generations a given percentage of individuals that have the highest fitness values (candidates known as *elite*). The advantage of this in our approach is that rare branches inside a program (i.e., with higher fitness than common branches) are difficult to find and we want to preserve them across generations. The Rank-based Roulette Wheel

```
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
```

Fig. 6: Disassembly code for the function under test in Fig. 11. The lines marked with red lines are jump conditions ending basic blocks.

Selection can add some variety by selecting probabilistically the individuals according to their rank in fitness value, while the random selection just adds some more variety of individuals by performing selection ignoring their fitness.

After the selection phase, mutation and crossover operations are applied to the $NL = N-K$ individuals left out of selection. Because at each new generation we keep the same number of individuals N, we create NL new individuals, which are created using the ones unselected. The mutation operation implies selecting a set of individuals to be mutated, each with a chosen probability P_m, then changing in each individual/chromosome one or more genes (bytes) to a new random value. The crossover operation selects pairs of individuals, with probability P_c. For each pair of individuals (A, B), a random number k for the "cut-point" for crossover is generated. The initial individuals A and B are replaced by the individuals $[A_0 \dots A_{k-1} B_k \dots B_S]$ and $[B_0 \dots B_{k-1} A_k \dots A_S]$, where S is the size of the input length used (and $0 \leq k \leq S$).

The number of generations has an upper bound given as a parameter `maxNumberOfGenerations`, but, before this limit is reached, a *plateau effect* is likely to be observed, i.e., when the individuals over a couple of generations do not improve significantly their fitness functions. More precisely, for two consecutive generations the change is checked as follows:

$$\sum_{x \in N} \big[\, \text{Fitness}\big(G_i[x]\big) - \text{Fitness}\big(G_{i-1}[x]\big) \,\big]^2 < \varepsilon.$$

One way to overcome this effect is to increase the parameters P_m and P_c temporarily until more diversity is added to the population of individuals [32]. In our implementation, if the average fitness is not improved in the last NG generations, then P_m and P_c are gradually increased until they get to P_{mMax} and P_{cMax} after a specified number of generations. If the plateau still occurs, the algorithm is restarted because there is a low probability that it can find any better tests from this point.

Figure 7 shows the pseudocode of the entire genetic algorithm described in this subsection.

G_0 := random population of N individuals
for i from 0 to `maxNumberOfGenerations` do
  S_i := select K individuals from G_{i-1} with probab. EL, RW, RR
  $Others_i$:= generate $N - K$ individuals from $G_{i-1} \backslash S_i$
    using mutations and crossover products
  $G_i := S_i \cup Others_i$
  Check for plateau and increase P_m and P_c if needed
  If plateau still occurs after a number of generations then STOP

Fig. 7: The genetic algorithm

In our experiments, we used $P_m = 0.2$, $P_c = 0.2$, $P_{mMax} = P_m + 0.3$, $P_{cMax} = P_c + 0.3$, $\varepsilon = 0.0001$, $K = N \times 0.2$, $EL = 0.8$, $RW = 0.1$, $RR = 0.1$.

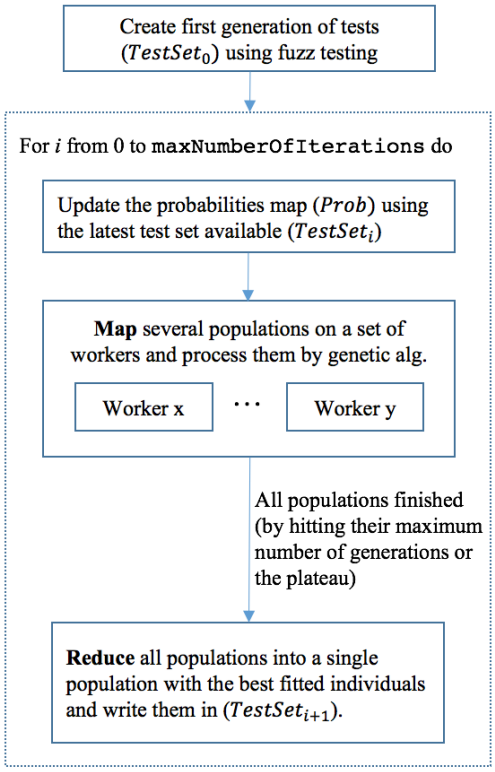

Fig. 8: The workflow for computing the generations

Also, the maximum number of generations produced by the genetic algorithm from the initial population `maxNumberOfGenerations` was set to 50.

C.2 The automated test data generation process

The first generation of tests and traces mentioned in Subsection is obtained using fuzz testing, i.e., random tests. The automated test data generation process runs continuously, and at each iteration, it produces a pair containing a set of test data and a set of traces obtained by executing these: ($TestSet_i$, $TraceSet_i$). The probabilities map is dynamically updated, i.e., at each new iteration it is updated with the latest traces obtained in the set TraceSet_i. This is an important optimization, since paths that are uncommon at some point during the process may become common later with the new tests obtained. The overall process is depicted in Figure 8.

The user running the process has access to all the test data generated at any point during its execution: $TestSet := \cup_i \{TestSet_i\}$, which can be valuable for regression testing.

The maximum number of iterations, `maxNumberOfIterations`, was set to 100 in our simulations.

C.3 Evaluation

The component can be evaluated from two points of view:

Research question 1 is the genetic algorithm approach indeed as parallelizable as announced in above; and also

Research question 2 does the genetic algorithm cover more branching conditions of the evaluated program than fuzz testing.

For both research questions, we used as programs to be tested two open-source libraries (which were compiled to x86 binaries): `http-parser`[3] (library for parsing HTTP requests/responses) and `libxml`[4] (library for parsing and creating XML files).

For the first research question the usual metric in evaluating parallel programs is the throughput of the serial versus parallel implementation. The throughput in our application is the number of new tests obtained and evaluated per unit of time. The hardware platform used for testing was a cluster of 8 PCs, each one with 12 physical cores, totaling 96 physical cores of approximately same performance - the type of processor used was Intel Core i7-5930K 3.50 Ghz. A total of $NumWorkers = 480$ workers were instantiated (a factor of the 96 workers, to create enough tasks per physical workers as suggested in the text explaining the implementation, avoiding this way the idle times for physical workers that hit the plateau before finishing the maximum number of allocated generations). The population size for each `Worker` object was $PopulationSize = 100$ individuals each with a length of 80 genes (bytes). `Worker` objects were let to optimize the internal population within a maximum of $maxNumberOfGenerations = 50$ generations, checking for plateau at each 5 generations. Technically speaking, the maximum number of tests expected for evaluation in this configuration is $MaxTests = PopulationSize * maxNumberOfGenerations * NumWorkers = 2,400,000$.

We let the algorithm execute in serial (a single process on one of the PCs) and in parallel on the configuration mentioned above, and stopped after 9 hours. The number of tests evaluated is shown in Table 1. The list of `Worker` objects was scheduled (on the 96 physical cores available) by Spark using its `parallelize` function, as shown in Figure 7. This setup with many instances of `Workers` did not affect the serial performance at all since the list of instanced `Workers` was executed one by one in this approach. Also, the parallel speedup and efficiency is expected to remain constant in all types of applications, despite the different costs of computing the traces. This is indeed the case for both examples as seen in Table 2. This is because in our framework each `Worker` has its own private tracer process spawned and this means that there is no contention between different physical `Workers`.

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

Table 1: Serial vs parallel throughput comparison

Library evaluated	Serial	Parallel
http-parser	1,593	130,843
libxml	446	34,891

Table 2: Speedup (i.e., parallel over serial throughput) and efficiency (i.e., speedup over number of physical processes used) metrics

Library evaluated	Speedup	Efficiency
http-parser	82.13	0.85
libxml	78.23	0.81

The memory footprint of our application is not a concern, and should not generally be, unless the fitness computation process would require significant memory (note that, if the fitness computation process would take N bytes, then creating PY processes on the same machine would require a minimum of $N * PY$ bytes of memory available).

For research question 2, we want to compare our genetic algorithms for automated testing data that finds rare paths inside programs' execution. In this case, we would like to see how much our approach helped us compared to fuzz testing (i.e. generating random tests). We executed the algorithm for 18 times, and computed different statistical metrics related to the number of different branch instructions encountered when testing `http-parser` and `libxml`, respectively, in an interval time of 1 hour. The results are provided in Tables 3 and 4.

Our genetic algorithms performs better in both cases, although by a small margin. Even if the difference in the number of instructions found is not big (4% and 6.5%, respectively, more branching executed, based on the mean metric), but one should take into account that usually rare paths and instructions are more difficult to find as the number of paths increases.

Although the results of our experiments look positive for both research questions (especially the first one), there are several threats to validity. First, we only run the experiments on two programs. This was due to the fact that our internal tracer module still misses some functionality and cannot be applied to any x86 program, but in the next period this will be fixed such that we can run the experiments on the cybersecurity grand challenge benchmark [**?**]. Moreover, we have not yet applied a full battery of statistical tests (including t-test and U-tests) as suggested in [33].

Table 3: Statistical results for the http-parser library

Metric	Fuzz testing	Genetic testing
mean	215.27	229.72
median	217	230
variance	75.03	1.03
mean absolute deviation	9.63	1.07
min	201	228
max	228	231

Table 4: Statistical results for the libxml library

Metric	Fuzz testing	Genetic testing
mean	1219.94	1270.11
median	1225	1270
variance	118.64	0.81
mean absolute deviation	14.16	1.31
min	1202	1269
max	1231	1272

Appendix D Offline guided fuzzing component details

The existing content of the test corpus can be updated online in both directions: either adding new files of existing types, or adding new file types. This is an important requirement since the main requirements from software security companies (such as the one we collaborated with, Bitdefender) are: (1) to be able to learn and produce new inputs of different kinds for many different programs with the purpose of security evaluation, and (2) to automatically and dynamically collect data from users, i.e. new input tests are added online and used to improve the trained model).

D.1 The training pipeline

Given the path to an existing corpus folder (*data*), the training pipeline writes its output in two folders:

1. *data_preprocesses*
2. *data_models*

Folder (1) stores the clusterized and preprocessed corpus data. Since the types of the files in there is unknown, our first target is to cluster them by identifying the type of each file in the corpus then put them in a different subfolder corresponding to each file type. As an example, if the corpus folder (*data*) contains four different input file types such as XML, PDF, JSON and HTTP

requests, then the first step will create (if not already existing) four clusters (folders) and add each input file to the corresponding one. Currently, the classification of files to clusters is done using the $file - l$ command in Unix, and getting the output string of the command (we plan to improve this classification in the future work by using unsupervised learning and perform clusterization based on common identified features). Since at each training epoch the entire sequence of character in each file must be processed, and considering that seek operations on disk can be expensive, the strategy used by our training pipeline is to concatenate together all files in each cluster (folder) in a single file to make the training process faster. Thus, each of the four folders in the concrete example above will contain a single file with the aggregated context from the initial ones. The neural-network model of each cluster is trained by splitting the aggregated file content ($C_{Content}$) in multiple training sequences of a fixed size L, which can be customized by user. Thus, the i^{th} training sequence contains $t_i = C_{Content}[i * L \ : \ (i+1) * L]$ (where F[a : b] denotes the subsequence of characters in F between indices a and b). For each of these training sequences, the expected output that the network is trained against is the input one shifted by 1 position to the right, i.e, $o_i = C_{Content}[i*L+1 \ : \ (i+1)*L+1]$. The model is then trained with all these input/output sequences from a cluster's content and using backpropagation to correct the weights, it learns the probability map of next characters having a given context (prior sequence). This previous context is modeled with the hidden state layer.

However, we need a generic way to mark the beginning and ending of an individual file content, such that the sampling method knows how to start and when to stop. At this moment, the beginning marker is a string *BEGIN#CLUSTERID*, while the end marker is a string *END#CLUSTERID*, where $CLUSTERID$ is an integer built using a string to integer mapping heuristic. The input string used for mapping is the full classification output string given by the $file \ -l$ command when the file was classified in a cluster. A supervisor map checks if all hashcodes are unique and tries different methods until for each cluster there is a unique identifier. The equation below (Source: [17]) shows the content of a cluster's aggregated file, where the $\sum$ and $+$ operators acts as concatenation of strings, and C is a given cluster type.

$$Identifier(C) = GetUniqueClusterIdentifier(C)$$
$$Cluster(C) = \sum_{each file F \in C} (\text{“}BEGIN'' + Identifier(C) + FileContent(F) + \text{“}END'' + Identifier(C))$$

The tool uses Tensorflow [34] for implementing both learning and sampling processes. Each cluster will have its own generative model, saved in *data_models* folder. In the example given above, four models will be created, one for each XML, PDF, JSON, and HTTP input types. A mapping from $CLUSTERID$ to the corrensponding model will be created (and stored on disk) to let the sampling process know where to get data from. In the network built using Tensorflow implementation we use LSTM cells for avoiding the problems with exploding or

vanishing gradients [35]. By default, the network built has two hidden layers each with 128 hidden states. However, the user can modify this network using expert knowledge per cluster granularity as stated in Section D.4 (the starting point of the process described in this section is defined in *generateModel.py* script, which has a documented set of parameters as help). Tensorflow is also able to parallelize automatically the training/sampling in a given cluster. On a high-level view, the framework allows users to customize a network and its internal compiler / executer decides where to run tasks with the scope of optimizing performance (e.g. minimize communication time, GPU-CPU memory transfer, etc).

Our tool takes advantage of the checkpointing feature available in the Tensorflow framework, i.e. at any time the learned model up to a point can be saved to disk. This helps users by letting them update the generative models if new files were added dynamically to the clusters after the initial learning step. This way, the learned weights in the neural network are reused and if the new files are not completely different in terms of features from the initial ones, the training time scales proportionally to the size of the new content added. At the implementation level, an indexing service keeps the track of the new content in each cluster and informs a service periodically to start the generative models updating for each of the modified clusters. Another advantage of the checkpoint feature is that it allows users to take advantage of the intermediate trained models. Although not optimal, these can be used in parallel with the training process (until convergence) to generate new test data.

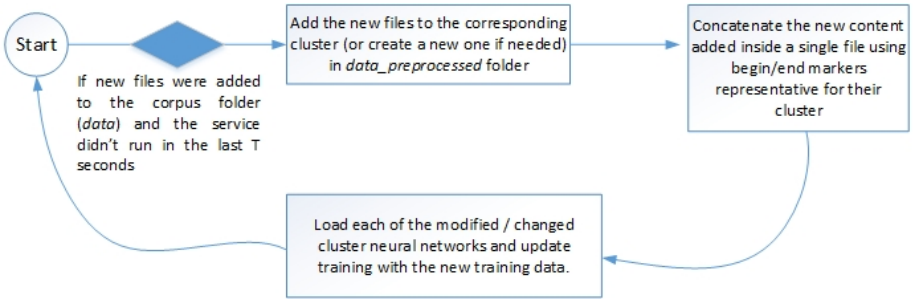

Fig. 9: The process of updating the generative models. Source: [17]

D.2 New inputs generation

The pseudocode in the listing below shows the method used to generate a new input test. The function receives as input a cluster type (considering that there exists a trained generative model for the given cluster), and a policy functor pointing to one of the four policies defined in the previous section. The first step is to get the custom parameters and the begin/end marker strings for the

given cluster. The next step is to feed the entire begin marker string (starting with a zero set hidden layer) and get the resulted hidden state. This will capture the context learned from the training data at the beginning of the files in that cluster. Then, the code loops producing output characters one by one using the probability distribution map (P) returned by the $FeedForward$ function in the current state (h_state). At each iteration, as seen in Figure 2a, the last produced output character and state are given as parameters to find the probability distribution map over vocabulary. The loop ends when the last part of the output (suffix) is exactly the end marker string (or until a certain maximum size was produced to avoid blocking if the training was not good enough to get to the end marker). The starting point of the concrete implementation can be found in the script file named *sampleModel.py*.

```
SampleNewTest(Cluster, PolicyType):

  Params = GetParams(Cluster)
  BeginMarker, EndMarker = GetMarkers(Params)

  foreach c in BeginMarker:
    h_state, P = FeedForward(h0, internalRNN, c)
    lastChar = c

  output = ""

  while the suffix of output != EndMarker :
    lastChar = Policy(PolicyType, P, lastChar)
    output += lastChar
    h_state, P = FeedForward(h_state,
                        internalRNN, lastChar)
return output
```

Source: [17]

A pseudocode defining sampling policies is presented in the listing below. Roulette-wheel based random selection is used with the Sample policy, and with the SampleSpace one when the previous character generated was a whitespace. If SampleSpace is used but still inside a word, or if SampleFuzz sampling method is used and the random value drawn is higher than the fuzz threshold, then the character with the highest probability from the vocabulary is chosen. Instead, if the random value is smaller than fuzz threshold, the character with the lowest probability is chosen in an attempt to trick the program under test.

```
Policy(PolicyType, P, C):
  switch Type:
     case NoSample:
        return argmax(P)
```

```
case SampleSpace:
   if C == " " return roulettewheel(P)
   else return argmax(P)
case Sample:
   return roulettewheel(P)
case SampleFuzz:
   if rand < FuzzThreshold:
       return argmin(P)
   else
       return argmax(P)
default:
   assert "no such policy"
```

Source: [17]

D.3 Producing generative models for applications with binary inputs

Considering N samples of test examples for a specific aplication Figure (2b), and by using using our *RIVER* tool (https://github.com/bitdefender/, [1]), we can find out which parts of the inputs can affect the branching decision of a program through taint analysis ([36]). The intuition is that to achieve efficient result, we should concentrate fuzzing more on those specific areas rather than the whole input. Considering this, our approach is to create generative models for each of the sample and contignous green region. At inference time, we choose one of the sample, duplicate it in a memory region, call each green area model, and fuzz the rest of the input a little bit. The process is depicted in the listing below.

```
GetNewBinaryInput(AppType A)
  NewInput = GetRandomInputSampleByAppType( A )
  for each model M in NewInput
     PartialInput = SampleNewTest(M)
     Replace the bytes covered by M (green areas) in NewInput with PartialInput

  Random fuzzing 5% of uncovered areas in NewInput
  return NewInput
```

D.4 Expert knowledge

Different clusters might need different parameters for optimal results. For example, training PDF objects might require more time to get to the same loss result than the threshold set for learning HTTP requests. The optimal parameters can differ starting from simple thresholds to the configuration of the neural network structure, i.e. the number of hidden layers or states. The tool allows users to inject their own parameters for both learning and sampling new results, by using a map data structure that looks more like an expert system. If custom data is

available in that map (e.g. ["$HTTP\ request\ cluster$", $num\ hidden\ layers$] = 1) for a particular cluster and parameters, then those are used instead of the default ones. Another example is the customization of the beginning/end markers used to know when a certain input data starts and ends. For well-known types, the user can override our default method for assigning the markers with the correct ones (e.g. PDF objects start with "obj" and end with "endobj"). Also, since Tensorflow can provide graphical statistics added by users (Tensorboard) during both training and sampling, the tool allows users to insert customized logs and graphics per cluster type using the function hooks provided.

D.5 EVALUATION

As the previous work in the field [16] already evaluated the training efficiency of the core method, i.e. learning a generative model with RNNs and do inference over it to find new inputs, using PDF file types, we evaluate our tool using two more parser applications: XML parser [5] and HTTP parser [6]. However, we use our own mark system for beginning/ending of a file, which works for generic (any kind of) file types as mentioned in Section **??**. The two new mentioned test applications were used to compare the results directly against the work in [6], which uses random fuzz testing driven by a genetic algorithm to get better coverage over time, and the same two programs for evaluation.

Experiment setup and methodology The experiments described below involved a cluster of 8 PCs, each one with 12 physical CPU cores, totaling 96 physical cores of approximately the same performance (Intel Core i7-5930K 3.50 Ghz). Each of the PC had one GPU device, an Nvidia GTX 1070. The user should note that adding more GPUs into the system could improve performance with our tool since the benchmarks show that the GPU device was in average about 15 times faster than the CPU both for learning models and generating new tests.

In our tests, we ultimately care about the coverage metric of a database of input tests: how many branches of a program are evaluated using all the available tests, and how much time did we spend to get to that coverage? Our implementation uses a tool called Tracer that can run a program P against the input test data and produce a trace, i.e., an ordered list of branch instructions $B_0, ..., B_n$ that a program encountered while executing with the given input test: $Tracer(P, test) = B_0B_1...B_n$. 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 we divide our program in basic blocks, which are sequences of x86 instructions that contain exactly one branch instruction at its end. We used a tracer tool developed by Bitdefender company, which helped us in the evaluation process, but there are also open-source tracer tools such as Bintrace [7]. Having a set

[5] http://xmlsoft.org
[6] https://github.com/nodejs/http-parser
[7] https://bitbucket.org/mihaila/bintrace

of input test files, we name coverage the set of different instructions (pairs of $(module, offset)$) encountered by Tracer when executing all those tests. We are interested in maximizing the size of this set usually, and/or minimizing the time needed to obtain good coverage.

Specifically, when training generative models, another point of interest is how efficient is the trained model with different setups, i.e. how many newly generated tests are correctly compiled by the HTTP and XML parsers (*Pass Rate* metric) ? This could help us make a correlation between the Pass Rate and coverage metrics.

Training data and generation of new tests The training set consisted of XML and PDF files that were taken using web-crawling different websites. A total of 12.000 files were randomly selected and stored for each of these two categories. For HTTP requests, we used an internal logger to collect 100.000 of such request. The folder grouping all these inputs is named in our terminology *corpus test set*. A metric to understand how well does the trained model learn is named *Pass Rate*. This estimates (using the output from *grep* tool) the percent of tests (from the generated suite) that are well formatted for the parser under test. As Figure 10 shows, and as expected, the quality of trained model grows with the number of epochs used for training (i.e. the number of full passes over the entire training data set). Randomizing only on spaces (i.e. using SampleSpace) gives better results for Pass Rate metric since more data is used as indicated as being optimal by the trained model. Tensorflow was used for both training and inference, and the hardware system considered was the one described at the beginning of this section.

Table 5 shows the time needed to perform model learning over the entire corpus folder of 12.000 PDF and XML files, and 100.000 HTTP requests using a different number of epochs. Other parameters are also important, the user should also take a look at the description of those inside the tool's repository and try to parametrize with expert knowledge for more optimizations when dealing with new file types. Table 6 shows the timings for producing 10.000 new inputs for PDF and XML files, and 50.000 of HTTP requests. As expected, since there is only inference through a learned model, the timings are almost equal between all models (we do not even show the difference between Sample and SampleSpace since the difference is negligible). Actually, from profiling the data tests generation it takes more time to write the output data (i.e. input tests) on disk rather than spending cycles on inference.

Main takeaway: The time needed to train the model is fixed, depending on the number of epochs and a few other parameters. After the training phase, the tool can create huge databases of new inputs (valid ones) quickly, which in the end can provide better code coverage than existing fuzzing methods. Those do not need the training phase, but the new tests generated are often rejected from early tests inside the program because of their incorrect format.

Coverage evaluation

For the coverage evaluation tables below, we considered only the model trained with 30 epochs, which was the winner in terms of performance versus

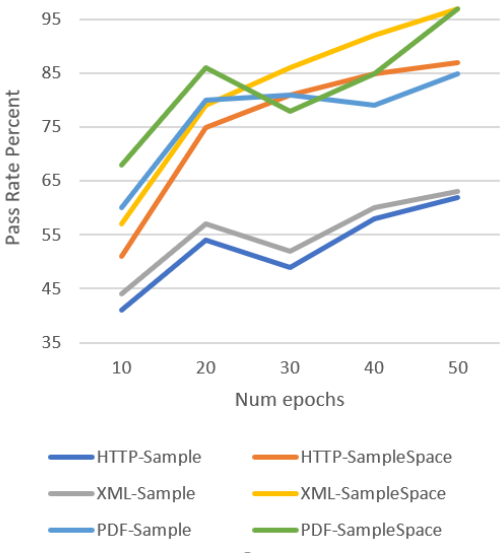

Fig. 10: Pass Rate metric evaluation for different number of epochs and models used to generate new tests.

Table 5: 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

training cost. Using 40 or 50 epochs increased just with a few new lines the coverage over time, but the training time is significantly higher. Of course, the user should experiment and find the optimal number of epochs depending on training data size for example, and their budget time limit allocated for training.

Tables 7 and 8 show the coverage for XML and HTTP file types by using three different evaluation methods. The first one, XML-fuzz+genetic / HTTP - fuzz + genetic, considers the fuzzing method driven by genetic algorithms as explained in [6]. The Sample and SampleSpace are the two models used for sampling defined above in this paper, and which uses our tool. The main observation is that with simple fuzzing (i.e. no use of generative models), the coverage value converges quickly to a value, without necessarily growing by having more time allocated. This happens mainly because the random fuzzing methods produce many times inputs that are not correct, being rejected by early outs, or difficult to deviate

Table 6: 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

from a few common branches inside a program even when adopting different policies to guide fuzzing ([6], [16]). However, fuzzing without learning the input context techniques have their own advantage: they are simple to implement and require no training time. For instance, if smoke tests [**?**] are needed after changing the user application's source code and input grammar, quick random fuzzing methods are very efficient since they do not require any training time. Learning a generative model is not feasible in this situation due to the limited time needed to respond to the new code change. Actually, techniques can be combined: classic fuzzing can be used for smoke tests, while fuzzing with generative models such as the one presented in this paper can be used to perform longer and more performant tests.

Table 7: The number of branch instructions touched in comparison between random fuzzing driven by genetic algorithms, Sample and SampleSpace models for XML files.

Model	9h	15h	24h	72h
XML-fuzz+genetic	1271	1279	1285	1286
XML-Sample	1290	1364	1455	1549
XML-SampleSpace	1291	1375	1407	1553

Table 8: 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

In 72 hours using the system described in the setup, the system was able to get approximately 20% more coverage than the best documented model on the XML and HTTP cases. Also, please note again that the two models evaluated

were chosen to compare against other documented results. Our tool is able to produce generative models and training tests after training on any kind of user inputs formats (e.g. HTML, DOC, XLS, source code for different programming languages, etc). An interesting aspect is that the Sample method has better results than SampleSpace one, although the Pass Rate metric shows inverse results. Remember that by sampling each character according to the probability distribution in the generative model, it has a higher rate of making inputs incorrect (Figure 10). One possible explanation for this is that having a high rate of correct inputs can make the program avoid some instructions that were verifying the code's correctness in more detail. Thus, those instructions might be encountered by Tracer only when the inputs given are a mix between correct and (slightly) invalid. In [16] there is also a discussion about performing random fuzzing over the inputs learned using RNN methods, but similar to our evaluation, the results are not better than the Sample method. The other technique presented in [15] that learns the grammar of the input through dynamic tainting and applicable currently only to Java programs, could not be evaluated since the tool is not (yet) open-source and could not be retrieved in any other way.

Appendix E Concolic execution engine using Reinforcement learning

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 [37]. The rest of the section explains how we used reinforcement learning to optimize the concolic execution of binary x86 programs.

E.1 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. 12. The method used in SAGE [2] 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. 12, 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. 12, right part). The estimation can be done using Deep Reinforcement Learning techniques [28], 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. 12, 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. 11: Example of a simple function that the user might want to evaluate. The example is taken from SAGE paper [2].

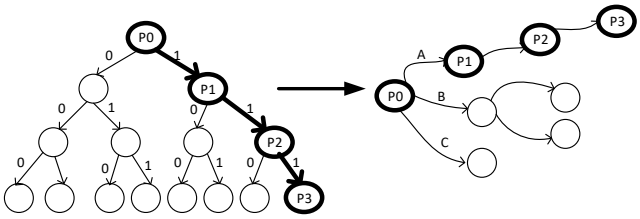

Fig. 12: The left part of the figure shows the tree obtained by running state of the art concolic execution on the source code in Fig. 11. The edges marked with 0 mean that branches are not taken, while 1s means that they are taken. The bolded nodes in the path show the one that generates the crash, 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.

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.

E.2 Concolic execution environment

To build a concolic execution environment, we implemented the state of the art method proposed in [2]. 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 1.1 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. 4 and 5). A path constraint and details from each branch point is depicted in Fig. 11. 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 [2]. Intuitively, this variable prevents a state to make the same change in the path constraints as one of its ancestors.

$$PC = \{BranchDesc_i\}_{i=\overline{0,len(PC)-1}} \tag{4}$$

$$BranchDesc_i = \{ModuleID, Offset, Z3condition, taken\} \tag{5}$$

Listing 1.1: 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 1.2, 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 [2] 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 E.4. 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 1.3, which is explained in more detail in Section E.3.

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 C.3. Note also that the proposed method does not eliminate any potential inputs, it is just trying to prioritize them in a different way.

Listing 1.2: 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
```

```
      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)
```

Listing 1.3: 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.

```
class RiverConcolic:
  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):
      // Do we have a model to estimate ?
      tailSize = PC.length − input.bound
      action = i
      if L_min ≤ tailSize and tailSize ≤ L_max:
        newInput.v = null
        tailLen = PC.length − input.bound
        newInput.estScore =
            RLModule.Predict(PC, tailLen, action)
        newInput.parent = input
        newInput.PC = PC
      else:
        // Otherwise, use the symbolic solver
        // 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
```

```
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
```

E.3 Design of our reinforcement learning solution

The concept of **state** in our solution was presented in Eq. 4 and 5. 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 PCs can have different lengths. Various attempts have been tried, described in more detail in Section E.4. 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. 13. 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 1.4 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 1.3 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 [2], 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 1.4 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 1.2 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. 4 and 5.
- *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 1.2, 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 E.4.

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 1.4 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 *PC*s since the beginning of the training process). For inference 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.

Listing 1.4: 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)
```

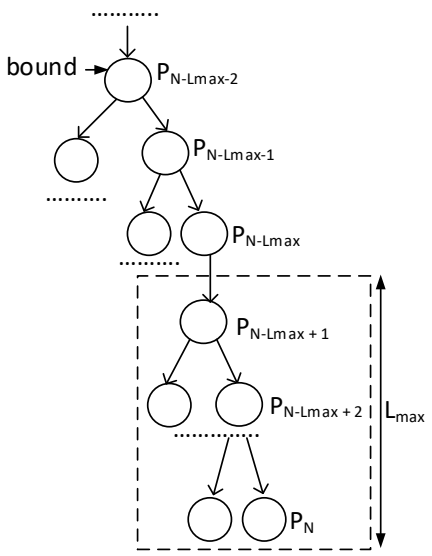

Fig. 13: A subtree starting from the bound index node of a hypothetical PC resulted by tracing the application under test with an input payload. The resulted PC is represented by the right-most branch, while the other nodes and edges represent possible different execution paths that the program could 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.

```
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
```

E.4 Estimation model architecture, discussion of difficulties and other attempts

An $LSTM$ architecture [38] 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 1.4). 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. 5 and Figure 14. 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. 6).

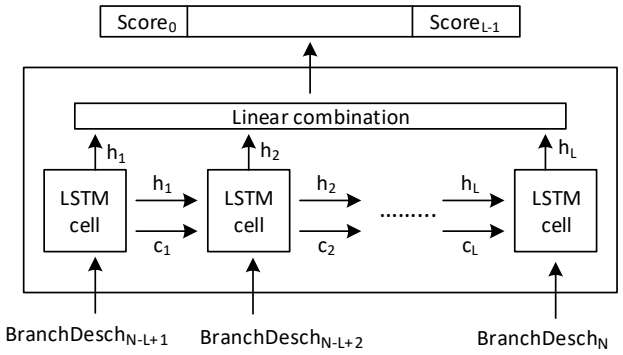

Fig. 14: The internal $LSTM$ based model for value estimation of a state with a length N. The model is supposed to learn the patterns only for the last L items in the state. Thus, the last L branch description entries from the evaluated state are used as input. The output is an array of L items, representing 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 [38].

$$Scores = W_s \, [h_0 \, h_1 \quad .... \quad h_{L-1}]^T \tag{6}$$

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. 7 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 [2], 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.

$$\begin{aligned} R(S, action, SNext) = & E1 * (B(SNext) - B(S)) + \\ & E2 * (S.len - a + 1) \end{aligned} \tag{7}$$

(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. 8. The proposed reward function, in this case, is presented in Eq. 9, which is a modified reward function from Eq. 7 that weights the importance of each newly discovered block by how often issues appeared at a given module and range of addresses in it.

$$\begin{aligned} & Stats(ModuleID, [Offset_{Start}, Offset_{End}] = \\ & \frac{Issues\ in\ (ModuleID, [Offset_{Start}, Offset_{End}])}{Num\ issues\ reported\ in\ total}. \end{aligned} \tag{8}$$

$$\begin{aligned} R(S, action, SNext) = & E1 * \sum_{\boldsymbol{b} \in B(SNext) - B(S)} Stats[\boldsymbol{b}] \\ & + E2 * (S.len - a + 1) \end{aligned} \tag{9}$$

(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. 10 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.

$$R(S, action, SNext) = \begin{cases} E1 * (SNext.len - S.len) + \\ E2 * (time(SNext) - time(S)), \\ \text{if SNext is not null} \\ P_{notSatisfiable}, \text{ otherwise} \end{cases} \quad (10)$$

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 [28]. 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 [39] or prioritizing the experiences [40].

E.5 Evaluation

Environment setup. The evaluation of our solution was done using two open-source applications: a JSON parser [8], and an HTTP parser [9], 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. 7 with the target of comparing the code coverage

[8] https://github.com/nlohmann/json
[9] https://github.com/nodejs/http-parser

obtained by using our solution against the classic *GenerationalSearch* method and their block score heuristic as presented in [2]. 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 9 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 9: 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 10.

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.

Table 10: 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 11 .

Table 11: 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 re-used 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 12 suggest that a previously trained model can quickly adapt to get fair results despite important code changes.

Table 12: 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.