Better Pay Attention Whilst Fuzzing

Shunkai Zhu
shunkaiz@zju.edu.cn
Zhejiang University
China

Jingyi Wang*
wangjyee@zju.edu.cn
Zhejiang University
China

Jun Sun
junsun@smu.edu.sg
Singapore Management University
China

Jie Yang
y@zju.edu.cn
Zhejiang University
China

Xingwei Lin
xwlin.roy@gmail.com
Ant Group
China

Liyi Zhang
l392zhan@uwaterloo.ca
Zhejiang University
China

Peng Cheng
lunarheart@zju.edu.cn
Zhejiang University
China

ABSTRACT

Fuzzing is one of the prevailing methods for vulnerability detection. However, even state-of-the-art fuzzing methods become ineffective after some period of time, i.e., the coverage hardly improves as existing methods are ineffective to focus the attention of fuzzing on covering the hard-to-trigger program paths. In other words, they cannot generate inputs that can break the bottleneck due to the fundamental difficulty in capturing the complex relations between the test inputs and program coverage. In particular, existing fuzzers suffer from the following main limitations: 1) lacking an overall analysis of the program to identify the most "rewarding" seeds, and 2) lacking an effective mutation strategy which could continuously select and mutates the more relevant "bytes" of the seeds.

In this work, we propose an approach called ATTuzz to address these two issues systematically. First, we propose a lightweight dynamic analysis technique which estimates the "reward" of covering each basic block and selects the most rewarding seeds accordingly. Second, we mutate the selected seeds according to a neural network model which predicts whether a certain "rewarding" block will be covered given certain mutation on certain bytes of a seed. The model is a deep learning model equipped with attention mechanism which is learned and updated periodically whilst fuzzing. Our evaluation shows that ATTuzz significantly outperforms 5 state-of-the-art grey-box fuzzers on 13 popular real-world programs at achieving higher edge coverage and finding new bugs. In particular, ATTuzz achieved 2X edge coverage and 4X bugs detected than AFL over 24-hour runs. Moreover, ATTuzz persistently improves the edge coverage in the long run, i.e., achieving 50% more coverage than AFL in 5 days.

ACM Reference Format:
Shunkai Zhu, Jingyi Wang, Jun Sun, Jie Yang, Xingwei Lin, Liyi Zhang, and Peng Cheng. 2021. Better Pay Attention Whilst Fuzzing. In *Proceedings of ACM Conference (Conference'22).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Fuzzing has become one of the prevailing methods for vulnerability detection. It works by generating "random" test inputs to execute the target program, aiming to trigger potential security vulnerabilities [18, 37]. Thanks to its simple and easy-to-apply concept, fuzzing has been widely adopted to test real-world programs [8, 19, 42, 47, 48]. Existing fuzzers like AFL [19] and its many variants [6, 8, 13, 20, 22, 33, 36, 41, 43, 45, 47, 56]use evolutionary algorithms (and sometimes alternative optimization algorithms [8, 12, 35]) to generate tests inputs. The optimization goal is to maximize the code coverage of the program so as to maximally reveal potential security vulnerabilities. In particular, AFL-based fuzzers instrument the program under test to monitor the coverage of each program execution, record the test inputs that cover different branches, select the promising test inputs (called seeds) and mutate the selected seeds in the hope of improving the coverage. The process repeats until a time budget is exhausted. This simple strategy often allows us to efficiently cover a large number of code blocks in the program. *Its effectiveness, however, often deteriorates over time.*

There are two main reasons. First, existing AFL-based fuzzers select test inputs which cover new branches as seeds. While such a strategy is effective initially, overtime test inputs which cover new branches become few and far in between, which renders such a strategy ineffective. To solve this problem, *we need a systematic and adaptive way of identifying the most "rewarding" (in terms of covering those un-covered branches) test inputs as seeds.* Second, after certain seeds are selected, existing AFL-based fuzzers apply a rich set of mutation operators on the seeds to generate new inputs. Similarly, such a strategy becomes ineffective overtime. After covering the easy-to-cover branches, covering the remaining

*Corresponding author.

Conference'22, May 2022, Pittsburgh, PA, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

branches often require specific mutation on specific "bytes" in the inputs. To address this problem, *we need a way of knowing where and how to apply the mutation operators in order to cover those uncovered branches.*

There are multiple attempts on addressing these two problems in the literature. To select seeds that are likely to cover new branches, LibFuzzer [47]'s heuristic is to select newly generated tests. AFLfast [8] prioritizes the seeds that can trigger the less-frequently visited pathes. Entropic [5] selects seeds that carry more program information according to certain entropy measure. Fairfuzz [32] locates and selects the seeds which trigger low-probability edges. Cerebro [34] selects seeds based on multiple factors such as code complexity, execution time and coverage. While being effective to some extent, these approaches lack a global view of the program under fuzzing and thus often miss the most 'rewarding' seeds over time. For instance, a low-probability edge may not be as "rewarding" as a high-probability edge if the latter leads to a large number of uncovered branches.

To selectively apply mutations, Steelix [33], REDQUEEN [2] and other works [9–12, 14, 23, 24, 28, 29, 38, 46, 51] proposed to perform dynamic taint analysis to determine the specific bytes in the input for solving the so-called "magic bytes" problem. MOpt [35] utilizes a customized particle swarm optimization to guide mutations scheduling. Besides, machine learning has been introduced to improve the performance of fuzzing in recent years [13, 26, 27, 33, 39, 45, 54]. For instance, RNN fuzzer [44] uses a recurrent neural networks (RNNs) to predict whether an input can reach the target program block to filter out uninteresting test cases. Based on a fairly similar idea, FuzzGuard [58] achieves good results in directed fuzzing. Neuzz [49] uses neural networks to smooth the target program and guides the input variation through gradients. However, these approaches only address the problem partially, i.e., only to identify the relevant bytes but not how to select the mutation operator (among the many choices).

For hybrid fuzzers, Driller [51] uses concolic execution to explore new paths when it gets stuck on superficial ones. QSYM [57] using dynamic binary translation to efficiently solve symbolic emulation.

In this work, we introduce ATTuzz, a novel framework to address the two problems systematically. First, in order to draw the fuzzer's attention to those test inputs that are most rewarding, ATTuzz quantifies the 'reward' of covering each basic block in the program through a lightweight global analysis. Intuitively speaking, ATTuzz estimates the reward based on the probability of covering uncovered branches and the number of them. Second, in order to draw the fuzzer's attention to specific mutations on specific bytes of the seeds that are most rewarding, ATTuzz trains a model which predicts whether a certain "rewarding" block will be covered given certain mutation on certain bytes of a seed. This is achieved by training an explainable deep learning model with attention mechanism which is learned and updated periodically.

ATTuzz has been implemented on top of AFL. We systematically evaluate ATTuzz with multiple experiments, comparing ATTuzz with 4 related state-of-the-art fuzzers on 13 real-world programs. The results show that ATTuzz significantly outperforms all existing fuzzers both at achieving higher edge coverage and finding new bugs. In particular, ATTuzz achieved 2X edge coverage and 4X bugs detected than AFL over 24 hours' run. More importantly, thanks to the seed selection strategy and the attention-based deep learning model, ATTuzz consistently improves its coverage over time (whereas existing fuzzer's effectiveness drops significantly after 24 hours), i.e., achieving 50% more coverage than AFL after 5 days of fuzzing.

In a nutshell, our technical contributions are as follows.

- We propose a lightweight global analysis to dynamically and adaptively identify the most "rewarding" test inputs as seeds during the fuzzing process.
- We propose to use explainable deep learning models with attention mechanisms learned from the massive fuzzing data to identify effective mutations on specific bytes of the identified seeds.
- We design, implement, and evaluate ATTuzz and demonstrate that it significantly outperforms 4 state-of-the-art fuzzers on a wide range of real-world programs.

2 BACKGROUND

2.1 Coverage-guided Fuzzing

We start with formalizing the grey-box fuzzing problem.

Definition 2.1. A program is a labeled transition system $\mathcal{P} = (B, init, V, \phi, GC, T)$ where

- B is a finite set of control locations[1];
- $init \in B$ is a unique entry point of the program;
- V is a finite set of variables;
- GC is a set of guarded commands of the form $[g]f$, where g is a guard condition and f is a function updating valuation of variables V. f represents a basic code block in general.
- $T : B \times GC \rightarrow B$ is a transition function.

Note that for the sake of presentation, the above definition assumes a flattened program structure without functions, classes and packages. We leave the details on how function calls are handled in the implementation section.

A concrete execution (a.k.a. a test) of $\mathcal{P}$ is a sequence

$$\pi = \langle (v_0, b_0), gc_0, (v_1, b_1), gc_1, \cdots, (v_k, b_k), gc_k, \cdots \rangle,$$

where v_i is a valuation of V, $b_i \in B$, $gc_i = [g_i]f_i$ is a guarded command such that $(b_i, gc_i, b_{i+1}) \in T$, $v_i \vDash g_i$, and $v_{i+1} = f_i(v_i)$ for all i, and $v_0 \vDash \phi$ and $b_0 = init$. We use Π to denote a set of tests. We say a test $\pi \in \Pi$ covers a control location b if and only if b is in the sequence. A control location b is reachable by Π if and only if there exists a concrete execution $\pi \in \Pi$ which covers b.

Definition 2.2. **Pre-dominant Blocks** Given a basic block $b \in B$, we define the set of b's pre-dominant blocks $D_b = \{b' \mid b' \in B \,\&\, \exists\, g : (b', gb', b) \in T\}$.

Intuitively, the set of pre-dominant blocks D_b are those blocks which could transit to b in one step.

Grey-box fuzzers like AFL [19] are designed to generate a set of test inputs Π which covers as many edges of the target program as possible with the hope of triggering bugs. We summarize the overall process of such fuzzers in Figure 1. First, the target program is instrumented to obtain program coverage information during the fuzzing process. Second, to maximize reachable B_b, the fuzzer

[1]We use 'control location' and 'basic block' interchangeably throughout.

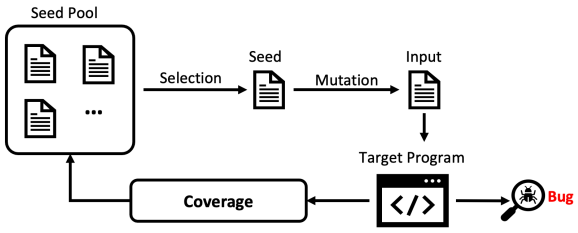

Figure 1: Coverage-guided Fuzzing Overview.

selects and mutates a test from a seed pool such that the mutated input would incur a different concrete execution of the program (covering new control locations). Afterwards, the program coverage information is updated and those tests incurring a different execution are prioritized and added into the seed pool. Afterwards, a new iteration starts.

The challenging problem to be solved by fuzzing is to identify the most 'rewarding' seeds and mutations so that program edges are covered efficiently. The problem is highly non-trivial due to the large search space, i.e., there are often many test cases which could serve as seeds and, given a seed, there are many possible mutations[2] (transform, deletion, and splicing, etc) as well. For instance, given a test seed of N bytes, the number of mutations defined by the AFL mutation operators is 29^N.

2.2 Deep Learning and Attention Mechanism

Deep learning is a class of machine learning algorithms that use multi-layer neural networks to abstract high-level features from the input data. Different from traditional machine learning algorithms, deep learning models automatically learn features from data rather than using handcrafted features. This end-to-end deep learning framework does not require complex manual feature engineering and has shown distinct advantages in various areas, including computer vision [31], natural language processing [15] and progressively applied in computer security [21].

Further, the attention mechanism enables a neural network model to distinguish the contributions of different input segments in the model decision process [3], by explicitly assigning a unique weight for each basic unit of input data, and calculating the representation of the input data using weighted sum of all basic units. Deep learning with attention mechanism is particularly relevant to the above-defined fuzzing problem for the following reasons. To identity the most rewarding mutation given a seed, we need to efficiently predict what mutations are more likely to cover certain uncovered program edges. Given the complex relation between the test inputs and program coverage, a powerful model like a deep learning model is necessary. More importantly, the attention mechanism allows us to 'understand' the coverage of particular 'magic' bytes, which provides effective guidance on future mutations.

[2]We refer the reader to [19] for details due to space limit.

2.3 Problem Definition

We summarize our core problem as how to *pay better attentions* whilst fuzzing to improve program coverage effectively and consistently from the following main aspects:

- How to pay attention to the most rewarding seeds?
- How to pay attention to the most rewarding bytes and mutations?

3 ATTUZZ FRAMEWORK

In this section, we present details of ATTuzz. An overview of ATTuzz is shown in Figure 2. ATTuzz includes four main phases: data collection, reward calculation, model training and mutation strategy updating. In the data collection phase, ATTuzz adopts a carrier fuzzer to generate inputs and records the seed files and mutations used. ATTuzz also tracks the coverage achieved by each test (step 1 and step 2 in Figure 2). Over time, the fuzzing process often gets stuck and the coverage is difficult to improve. Then, ATTuzz is activated. The core idea is to adopt deep learning with attention to predict whether a seed and mutation combination can cover certain basic block based on a basic block's coverage data. However, due to the large number of uncovered basic blocks, it is costly to train a model for each of them. Apart from that, we are not able to learn an effective model since we do not have any positive labeled data for the uncovered blocks. To address the challenges, as shown in Figure 2,in step 2, we build an abstraction of the program in the form of a (labeled) discrete time Markov Chain (DTMC). Then, step 3 aims to find out *critical blocks* based on the DTMC and step 4 prepares the respective fuzzing data. Next, step 5 aims to get the heat maps of the seed file under different mutators to provide guidance on selecting the more valuable bytes and corresponding mutators by training an attention model. The above process continues until the current bottleneck is overcome.

3.1 Data Collection

In the data collection stage, ATTuzz collects relevant information on test inputs generated by the carrier fuzzer, i.e., the seed file and mutations used. For coverage, ATTuzz records the AFL bitmap to track which basic blocks are triggered.

Example 3.1. For the program in Figure 3, AFL performs an $Arth_+$ operation on the seed file $\langle 0, 5\rangle$ with parameter 5 on the first byte, and gets the final input $\langle 5, 5\rangle$. This input can cover blocks 1, 2 and 3 in the program. So we record ($\langle 0, 5\rangle$, arth+, 5; 111000). After a series of mutation operations, we can get the same form of data, such as ($\langle -5, 0\rangle$, arth+, 5; 110101), ($\langle 0, 0\rangle$, bitflip, 5; 100101), ($\langle 0, 0\rangle$, dictionary, 0xFF; 100101).

3.2 Reward Calculation

As mentioned above, to effectively solve the fuzzing problem, we need a systematic way of identifying the most rewarding seeds and mutations. Intuitively, a test case is most rewarding if it leads a maximal improvement of the edge coverage. In the following, we present a lightweight approach which allows us to systematically compute the reward of covering a basic block. Note that the reward is then used as a guide to select seeds and mutation, i.e., those which are predicted to cover the basic blocks with highest rewards.

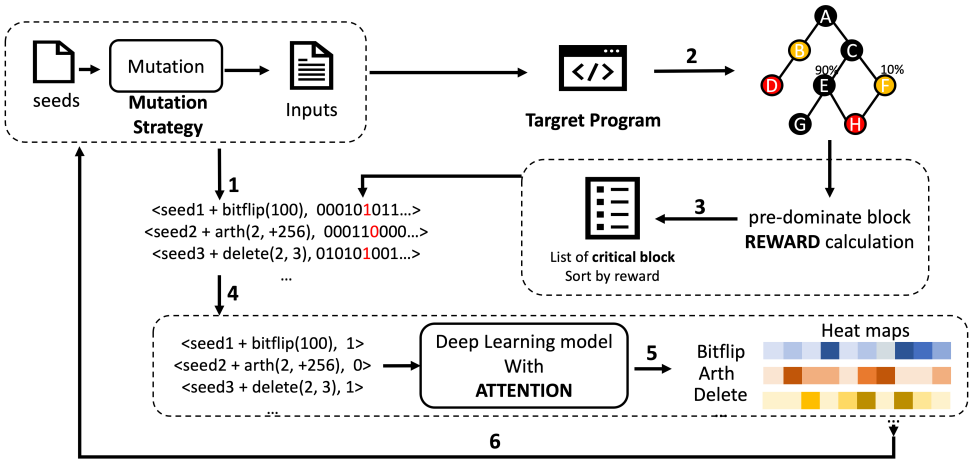

Figure 2: ATTuzz overview.

Our approach is inspired by [52, 55], which enables us to build a discrete-time Markov Chain (DTMC) abstraction of the program from the collected fuzzing data. Specifically,

Definition 3.2. A (labeled) discrete-time Markov Chain (DTMC) is a tuple $\mathcal{M} = (B, Pr, \mu)$ where B is the set of basic blocks in $\mathcal{P}$; $Pr : B \times B \rightarrow \mathbb{R}^+$ is a labeled transition probability function such that $\Sigma_{b' \in B} Pr(b, b') = 1$ for all $b \in B$; and μ is the initial probability distribution such that $\Sigma_{b \in B} \mu(b) = 1$.

Naturally, we can abstract a program into a DTMC if we impose an initial distribution on its initial states, where each control location in the program becomes a state in the DTMC, and each edge between two control locations are associated with a conditional probability. For example, a program shown on the left side of Figure 3 can then be transformed into the DTMC on the right [55]. The key to construct the DTMC is to estimate the conditional probabilities between edges from the fuzzing records as follows:

Definition 3.3. Let $\#(b1, b2)$ denote the total number of times $b1$ transits to $b2$ in the fuzzing process and $\#b1$ represents the total number of executions of $b1$, n is the total number of outgoing edges of b_1 in the CFG. Then the conditional probability from b_1 to b_2 is then $Pr(b1, b2) = \frac{1+\#(b1,b2)}{\#b1+n}$.

The reward of a basic block is defined as follows.

Definition 3.4. Let R_b where $b \in B$ be the reward of visiting b. $t \in T$ be the block set that b can reach with one step. We build an equation system as follows.

$$R_b = \begin{cases} 1 + \Sigma_{t \in B}\{Pr(b, t) \times R_t\} & \text{if } b \notin visited \\ \Sigma_{t \in B}\{Pr(b, t) \times R_t\} & \text{if } b \in visited \end{cases}$$

With an estimated DTMC and the above equation system, we can calculate each basic block's reward with the help of the program's CFG. Note that for those indirect calls that are unable to be extracted by static analysis, we use the dynamic fuzzing data to complement the CFG of static analysis. We omit the details of solving the equation system (which is efficient, e.g., in seconds) and refer interested readers to [55] fo details.

Example 3.5. We use the program in Figure 3 as an example to illustrate the reward calculation process. As shown in Figure 3, after 998 program executions, we can estimate the DTMC from the basic blocks coverage information. The equation system can be built on the bottom of Figure 3. By solving the equation system, we have the rewards of covering each basic block as $R_1 = 0.001$, $R_2 = 0.002$, $R_3 = 0.086$, $R_4 = 1.333$, $R_5 = 0.143$, $R_6 = 0$, $R_7 = 0$ and $R_8 = 1$.

Once we know the reward of covering each uncovered basic block, ATTuzz then selects the top k percent of them as target uncovered blocks B_c (most rewarding). As mentioned, since we have no positive labeled data for the uncovered blocks, we obtain the predominant blocks of each target block as $pre(B_c)$ by static analysis. The blocks in $pre(B_c)$ are our target for fuzzing. To further reduce the number of targets (and thus reduce the number of mutants to be generated), we filter those blocks in $pre(B_c)$ which have a high probability of being reached according to the DTMC (since they hardly need much assistance). Note that among $pre(B_c)$, we prefer those which has a low probability to reach as more interesting *critical blocks*. In practice, we omit those pre-dominant blocks which has a probability higher than a threshold k'. We use $B_{critical}$ to denote the finally selected pre-dominant blocks for deep learning.

Overall, with the help of the reward mechanism, the goal becomes to generate test cases which is likely to cover $B_{critical}$. Now we know which block to cover, we further need a way of predicting which seed and mutation would cover that block.

3.3 Training Attention Models

The above two phases enable ATTuzz to pay attention to the most valuable basic blocks ($B_{critical}$). Next, we build a deep learning model to systematically predict if a seed-mutation combination is likely to cover a certain code block. We choose an attention based deep learning model which can extract features from inputs automatically and making the correct classification (of whether a block will be covered). More importantly, the attention mechanism enables our model to distinguish the impact of different mutators and parameters on each byte for different seed files. In particular, we prepare the training data which are composed of two parts: one

```
void func(int a, int b, int c, char* buf) {
1.    int Flag = 0;
2.    if(a > 100) {
3.        if(b == -1) {
4.            if(c < 0) {
5.                Flag = 1;
              }
          }
      }
6.    if(Flag) {
7.        if(buf[0] == "X") {
8.            buggy code ...;
          }
      }
9.    return
  }
```

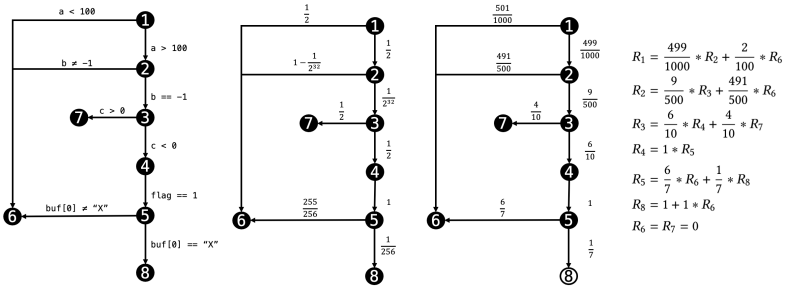

Figure 3: Program abstraction

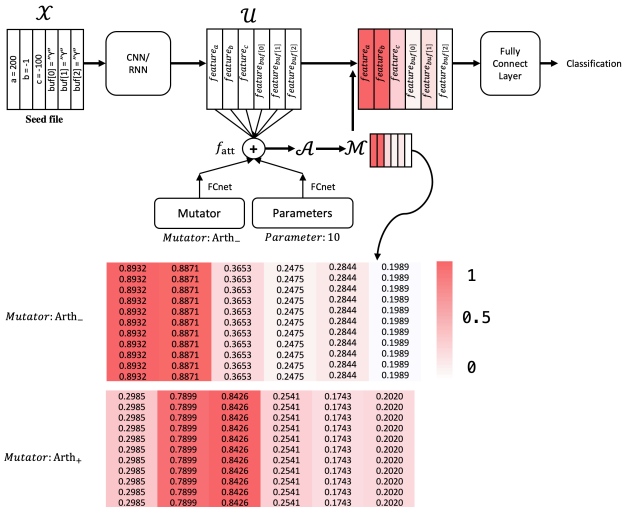

Figure 4: Attention model

is the seed file, and the other is the mutator and the corresponding mutation parameters.

Specifically, we convert the seed file into a series of vectors $\{\mathcal{X}_{D\times N} = \langle v_1, v_2, \cdots, v_N\rangle,\ v_{m_{D\times 1}},\ v_{p_{D\times 1}}\}$, where D is the dimension of bytes embeddings or pixel vector, and N is the largest size of the collected seed files. Note that for those inputs whose size is less than the maximum length, we padding the input to length N. Next, we adopt customized models for different kinds of programs to extract the relevant features depending on their inputs. For instance, for programs that take image as input, traditional convolutional neural networks (CNNs) are used. For programs that take byte sequences as input (such as inputs which are XML files or JSON string), recurrent neural networks (RNNs) are used to capture the sequential information. These two networks have been widely used in image feature and byte sequence feature extraction. After feature extraction, we can get the vector of extracted features as $\mathcal{U}_{D'\times N} = \langle v'_1, v'_2, \cdots, v'_N\rangle = f_{conv}(\mathcal{X}_{D\times N})$.

To further utilize the information of mutations, we add in the mutation operator <mutator, parameters>. Then, as shown in Figure 4, for each vector in $\mathcal{U}$, we calculate an attention weight α:

$$\alpha_i = f_{att}(\mathcal{U}_{D'\times N}, v_{m_{D\times 1}}, v_{p_{D\times 1}}) \tag{1}$$

The function f_{att} is composed of an activation function, which first merges the vectors of three elements (feature, mutator, and parameter) through a fully connected layer, and then passes through the nonlinear function softmax:

$$f_{att} = softmax(FCnet(\mathcal{U}) + FCnet(v_m) + FCnet(v_p)) \tag{2}$$

Combining all α together, we get the vector $\mathcal{A}_{1\times N} = \langle \alpha_1, \alpha_2, \cdots, \alpha_N\rangle$. We normalize $\mathcal{A}$ to matrix $\mathcal{W} = \langle \alpha'_{11}, \alpha'_{12}, \cdots, \alpha'_{1N}\rangle$ and expand the dimension of $\mathcal{W}$ to $D' \times N$ to form a mask matrix $\mathcal{M}$, which allows $\mathcal{M}$ to be applied to each vector in $\mathcal{U}$.

$$\mathcal{W} = \frac{\mathcal{A}}{\Sigma\{\alpha_i | \alpha_i \in \mathcal{A}\}} \tag{3}$$

$$\mathcal{M} = \begin{bmatrix} \mathcal{W} \\ \mathcal{W} \\ \vdots \\ \mathcal{W} \end{bmatrix}_{D'\times N} \tag{4}$$

We scale the elements in the matrix $\mathcal{M}$ to $\mathcal{U}$ to get the final attention layer output vector $\mathcal{U}'$. Next, we use a fully connected layer for classification. Through training the following attention layer, we can further visualize the information from feature $\mathcal{U}$ through a heat map, which measures the importance of the input features. In order to get the heat maps, we cluster the data set according to different mutators. For each cluster, we feed the data into the attention model, and obtain the matrix $\mathcal{A}$ from each input while calculating the accuracy. Finally, we obtain each mutator using a heat map, shown in Figure 4. The darker red positions represent the corresponding seed areas which play a more important role in the classification and vice versa.

In summary, our attention model is mainly composed of three parts. First, according to whether the input type is a picture or a byte sequence, we use three layers of CNN or RNN to extract features respectively. Second, ATTuzz introduce an attention layer described above to apply attention weight to each feature and finally we pass the weighted features to a fully connected layer for classification.

Example 3.6. Take the program in Figure 3 for example, after 1 hour's fuzzing, AFL can only cover line 2, 3, 4, 6 with the given seeds. So line 7 becomes the uncovered block of the program. Line 6 is the critical block (only 1/1000 inputs reach line 6). AFL spends

Algorithm 1: Mutation Guidance

```
1  Let seed be the seed file;
2  Let B_critical be the set of blocks selected by reward;
3  Let mutator_list be the set of mutator;
4  seed_cov = GetCoverage(seed);
5  for mutator ∈ mutator_list do
6      let threshold be the average value of seed's heat map of
         mutator;
7      for each byte b in seed do
8          let heat be the value in b's heat value;
9          if heat > threshold then
10             skip this mutation with a certain probability p;
11             continue;
12         Apply mutator on seed;
```

most of its effort on test cases which fail to reach line 6, i.e., 90% of the generated test cases cannot arrive at line 6. ATTuzz introduces an attention model to predict whether a certain mutation on a certain input can cover line 6. For instance, the influence of mutator $Arth_-$ on the seed file {<a=200, b=-1, c=-10, buf[0]="Y", buf[1]="Y", buf[2]="Y" > can be visualized as the heat map at the bottom of Figure 4. We could see that variables a and b are important to the coverage with very high weight, and the $Arth_-$ operation on other variables does not affect the coverage of program execution. This is consistent with the logic of the program. For the > and == operations corresponding to variables a and b, reducing the number of them will likely lead to a failure of the comparison, and the coverage of the corresponding program will also change. Similarly, for $Arth_+$, variables b and c are important to the coverage of the seed file, and adding variable a will not affect the result of comparison >.

3.4 Mutation Guidance

With the attention model, we can not only learn whether an input can reach a specific block, but also get the heat maps for each mutator. ATTuzz then uses the heat maps to guide the fuzzer to generate inputs that can reach the selected critical blocks. In the heat map, *the level of heat quantifies the importance of the corresponding position of a seed file in the classification result under a certain mutator.* The hotter it is, the more important it is in the classification, which means those 'hot bytes' (with heat values larger than a threshold) determine whether an input can reach the critical blocks of interest. If the hot bytes are mutated with the specific mutation operator, there is a high probability that the coverage will be changed.

Algorithm 1 shows the details of how ATTuzz efficiently generates a large amount of inputs that can reach the critical blocks with the guidance of the heat maps. We first obtain the seed file coverage (line 4). If the seed file can cover any of the critical blocks that we select, then for each mutator, we skip the hot bytes (determined at line 9) to avoid change in the coverage (line 10). We remark that a seed file may cover multiple critical blocks. In this case, as long as a byte is not a hot byte for all the blocks, it is mutated. Besides, ATTuzz still adopts some randomness to improve the diversity of mutation, i.e., choosing to mutate the hot bytes with a small probability p.

Algorithm 2: Main Algorithm

```
1  Let P be the target program;
2  Let B be the set of bugs;
3  Let iter_limit be the maximum iterations;
4  CFG = StaticAnalysis(P);
5  for seed ∈ seed_pool do
6      Let length be the length of seed;
7      Let seed_coverage be the coverage of executing P using
         seed;
8      for iterations ≤ iter_limit do
9          if EncounterBottleneck then
10             uncovered_blocks = FindUncov(CFG, cov);
11             rewards = RewardCal(CFG, cov);
12             B_critical = Select(uncov_blocks, rewards, CFG);
13             for block ∈ B_critical do
14                 heatmap_list = TrainModel(dataset);
15         for mutator ∈ mutator_list do
16             for loc ≤ length do
17                 if EncounterBottleneck then
18                     Guide(heatmap_list, mutator, loc);
19                 input = mutate(seed, mutator, parameter);
20                 cov_result = Excute(P, input);
21                 dataset =
                     Record(seed, mutation, cov_result);
22                 if result == Crash then
23                     B.append(input);
24                 if HasNewCov(cov_result) then
25                     seed_pool.append(input);
26 return B;
```

Example 3.7. Following the example in Section 3.3, the seed file can cover line 6 and we introduce our mutation guidance strategy. Take $Arth_-$ as an example whose heat map is shown in Figure **??**. We set the 'hot' threshold in Algorithm 1 as the average of all the heat values of the seed file, i.e., 0.4794 in this example. Variables a, b both have heat values that are greater than the threshold. So, for the $Arth_-$ operation, we avoid the subtraction of a and b and mutate the remaining variables with high probability. Similarly, for the $Arth_+$ operation, we avoid adding variables b and c because their heats are great than the average heat 0.5123.

3.5 Overall Algorithm

Putting all together, we summarize our overall algorithm in Algorithm 2. A mutation budget for each seed is set as $iter\_limit$. We first obtain the CFG of the target program $\mathcal{P}$ at line 4. Then, for each seed in the seed pool, within the mutation budget (line 8), we first determine whether a bottleneck is met at line 9 (details explained later in Section 4). Note that this is often not the case in

the initial phase of fuzzing. So, ATTuzz will initially execute the carrier fuzzer according to the default strategy (line 15-25). During the process, ATTuzz mutates the seed (line 19), executes the program (line 20) and collects the data (line 21). If an input incurs new edge coverage (line 24), it will be added into the seed pool (line 25). After a while, a bottleneck might be met (line 9), this is when ATTuzz starts to work by finding the uncovered blocks (line 10), evaluating the rewards of covering them (line 11) and select the critical blocks from their pre-dominant blocks (line 12). Then for each critical block, we train an attention model for each mutator using the dataset collected (line 14). After the models are trained, ATTuzz goes on to guide the subsequent fuzzing process at line 18. Note that whenever a crash is triggered (line 22), we add the input to the bug-triggering inputs $\mathcal{B}$ (line 23). Different from existing learning-enabled fuzzing [4, 7, 25, 26, 40, 50, 50, 53], ATTuzz is more effective in accurately locating the bottleneck and paying better attention to those valuable bytes and mutators to break them.

Example 3.8. Take the program in Figure 3 as an example. At the beginning, ATTuzz runs AFL with its default mutation strategy and collect the data. In the early stage of fuzzing, AFL was able to successfully cover line 2 to 6. But after 1 hour, AFL meets the bottleneck. We identify the uncovered blocks, which is the line 7. By reward calculation and computing the pre-dominant block, we select line 6 as the critical block for learning. ATTuzz uses the previously collected data to train the attention model and obtain the heat maps. ATTuzz prioritizes seed files that can reach line 6 and further guides the mutation according to the heat map so that a large number of inputs can be generated setting $flag$ to 1. The strategy enables ATTuzz to break the bottleneck efficiently and reach line 7 (triggering the bug) within 10 minutes (while AFL fails in over 24 hours).

4 IMPLEMENTATION DETAILS

Data Collection In the initial stage, ATTuzz runs the carrier fuzzer with the default configuration. As mentioned before, the data we collect are seed files, mutators, mutation parameters and the basic block coverage of each program execution. We use AFL++ [20] as our carrier fuzzer. Note that compared with AFL, AFL++ provides accurate coverage information (without hash collision) as well as the above-mentioned additional information which are required by our approach.

Unfortunately, in terms of detailed coverage information, a fuzzer like AFL, which uses edges as coverage statistics, does not provide accurate coverage information. Instead of comprehensively recording the complete execution paths, AFL uses a compact hash bitmap to store code coverage. This compact bitmap is highly efficient but not informative enough for our purpose. When the program is executed, AFL can only know whether a new edge is triggered or not but does not have any idea of the specific position of the edge in the program's CFG and due to hash collision, it is infeasible to identify which edge is covered. There are some mature instrumentation methods, such as LLVM's sanitizer coverage to obtain the coverage of each execution. However, we found that these methods have a noticeable overhead for program execution, which leads to a significant decrease in the fuzzing speed. To solve this problem, we try to only use the information that AFL provides.

ATTuzz extracts the ICFG of the program by statically analyzing the program's instrumented binary file and calculates the hash for each edge in the same way as AFL. When fuzzing the program, whenever AFL gets the hash of the program execution path, we can look up the dictionary and get its corresponding basic block. Based on the dictionary, we can further obtain the exact program coverage from the existing records of AFL with little overhead. We remark that for large programs, it is inevitable that the edges will have hash collisions (while for small programs, this problem is negligible). In this case, a hash value may correspond to multiple edges, making the coverage of each execution biased. This problem has been discussed in [22], which proposes a new hash calculation method to mitigate path collisions while trying to preserve low instrumentation overhead. Note that ATTuzz is flexible to incorporate such method to obtain more accurate coverage information.

Heat Map Acquisition In practice, the coverage of critical blocks tends to be polarized, i.e., they are either covered by almost every input, or they are only covered by few. To solve this problem, we under-sample the data to ensure it has a balance distribution. Thanks to the fast speed of fuzzing, although the probability of covering certain interesting blocks is often relatively low, we are able to collect millions of fuzzing data, which is enough for us to train a reasonable model. We randomly select the same amount of positive and negative labeled data to keep the training data as balanced as possible.

We train deep learning models using PyTorch (version 1.6.0). For feature extraction, we use 3 layers CNN or RNN for picture or byte sequence respectively. We generate a mask at the attention layer by performing non-linear operation $softmax$ on the mutator and parameter and then adding them together. We use a fully connected layer for classification after multiplying the normalized mask by the corresponding pixel/byte. We use cross-entropy loss function in our model and Adam optimizer [16] with learning rate 0.001 to help the model converge quickly. The attention model is trained for 60 epochs to achieve high accuracy (about 85% on average). After model training, we cluster the data set according to the mutator. By feeding the data into the trained model and calculating the attention mask, we can generate each mutator's heat map for different seed files. Note that it takes 40 to 60 minutes to train the model and obtain the heat map on average, which could be paralleled with the normal fuzzing process.

Mutation Guidance As mentioned in section 3.4, we choose not to mutate the hot bytes with high probability. We use the average heat value to determine whether a byte is hot (larger than the average heat). Meanwhile, we choose to mutate these hot bytes with a probability of 5% to introduce some randomness in the mutation.

Bottleneck Judgement Intuitively, fuzzing encounters a bottleneck if the coverage does not increase over certain amount of time. In practice, we check whether a bottleneck is met every hour (which aligns to the model training time) by calculating if the coverage increase in the last hour is smaller than a threshold, in our case, 5%.

Programs		Size	Version
Class	name		
JPEG	libjpeg	1.1M	9c
TTF	harfbuzz	6.1M	2.0.0
PDF	mupdf	45.8M	1.12.0
XML	libxml	9.2M	20907
ZIP	zlib	441k	1.01b
BIN	readelf	3.4M	2.30
	objdump	13.1M	
	size	9.2M	
	strip	10.7M	

Table 1: Programs in the experiments.

Programs	ATTuzz	NEUZZ	AFL	Driller	Vuzzer	AFLfast
libjpeg	**5260**	3879	2696	3555	1794	2812
harfbuzz	**9268**	5440	4344	4838	2449	3810
mupdf	**2039**	1701	1520	1694	1062	1274
libxml	**3221**	2870	1786	2097	1413	1883
zlib	**1617**	854	786	543	691	224
readelf	**7534**	5398	2142	4140	1346	2461
size	**3847**	2848	1995	2848	1077	2336
strip	**3818**	3122	1852	2685	1617	1828
base64	**144**	**144**	111	121	138	125
md5sum	**512**	319	368	328	211	257
who	**525**	461	176	226	204	290
uniq	**155**	142	134	140	106	119

Table 2: Comparing edge coverage for 24 hours' runs.

5 EVALUATION

In this section, we evaluated ATTuzz from two aspects. First, we compare the performance of ATTuzz with state-of-the-art fuzzers including: AFL, AFLfast [8], Vuzzer [45], Driller [51] and NEUZZ [49] to assess its effectiveness and efficiency. AFLfast performs seed file scheduling through statistics of execution path to optimize AFL. Vuzzer uses taint analysis to determine the relevant bytes in the input to improve the mutation efficiency. Driller as the state-of-the-art hybrid fuzzer, solve the path constraint by symbolic execution to improve coverage. NEUZZ introduces neural network to smoothing programs, helping fuzzer build inputs efficiently. These Baseline fuzzers have its own advantages in some respects, which can reflect ATTuzz's effectiveness in seed file and mutation scheduling as a hybrid fuzzer. Then, we evaluate the usefulness of each component of ATTuzz.

Specifically, we aim to answer the following research questions through our experiments. [3]

RQ1: Does ATTuzz improve code coverage effectively?
RQ2: Does ATTuzz allow us to break the bottleneck effectively?
RQ3: Are reward computation and attention guidance complementary to each other?
RQ4: Does ATTuzz allow us to discover more bugs?

Experiment Settings

Target Programs: We evaluate ATTuzz on two different types programs: (i) 9 real-world programs, as shown in Table 1, (ii) LAVA-M dataset [17]. To demonstrate the performance of ATTuzz, we compare 1) the edge coverage and 2) the number of bugs detected.

Experimental Setup: The host used in the experiment has 16-core CPU (Intel R core (TM) i9-9900k CPU @ 3.60GHz), 32G GPU (Tesla V100-SXM2) and 64G main memory. Note that each fuzzing task only takes one CPU core. We run each fuzzer for the same time budget with the same initial seed corpus and compare their edge coverage achieved and the number of bugs found. We compare the average coverage results of different fuzzers given the same time budget: 24 hours with 5 times. Since NEUZZ needs to run AFL for an hour to generate the initial seed corpus in advance, for a fair comparison, we also spend 1 hour in advance to learn the initial heat map when the first bottleneck is encountered. For other fuzzers, we add the seed files obtained within the first hour to the initial seed file pool in advance.

[3] ATTuzz is available at https://github.com/ICSE-ATTuzz/ATTuzz

RQ1. Does ATTuzz improve code coverage effectively?

We first discuss the overall coverage improvement from two aspects: 1) the total number of new edges found (Table 2) and 2) the rate of improving the edge coverage over time (Figure 5). Our experimental settings are consistent with the recommendation from [30]. As shown in Table 2, the solid lines represent the average data of the 5 times repeated fuzzing data and the shadows represents the confidence intervals of 95%. For all programs, ATTuzz outperforms AFL, Driler, Vuzzer and AFLfast in terms of overall coverage, and outperforms NEUZZ in all programs. On average, ATTuzz achieves 100% more coverage than AFL, 90% more than Driller and 20% more than NEUZZ. Furthermore, *for programs including base64, libjpeg, md5sum, mupdf, and readelf, ATTuzz achieves the same or even more coverage in the first two hours than that of AFL, AFLfast and Driller in 24 hours.* This shows the superior performance of ATTuzz in improving program coverage than state-of-the-art approaches. A closer look reveals that these programs share a common feature, that is, theirs inputs are highly structured, e.g., the JPEG format or the ELF format, which are suitable for deep learning with attentions to extract useful features and select the right bytes and mutators for mutation. Besides, we could observe from Figure 5 that the baseline fuzzers tend to get stuck in the bottleneck after some period of fuzzing while ATTuzz can consistently improve coverage in the long run, which is clearly evidenced in programs including harfbuzz, libjpeg, libxml, mupdf, readelf, size, strip, who and zlib. We further run libjpeg using ATTuzz and AFL for a long time, i.e., 5 days to show that ATTuzz is consistently improving coverage. The result shows that ATTuzz achieves about 50% more coverage than AFL in the end (5026 V.S. 3297).

RQ2. Does ATTuzz allow us to break the bottleneck effectively?

We further discuss the trend of the coverage improvement to show that existing fuzzers get stuck after a while whereas ATTuzz is able to break the bottlenecks during the fuzzing process in a measurable way as follows. We calculated the coverage growth rate in each hour and show the results in Figure 6 (we only show the

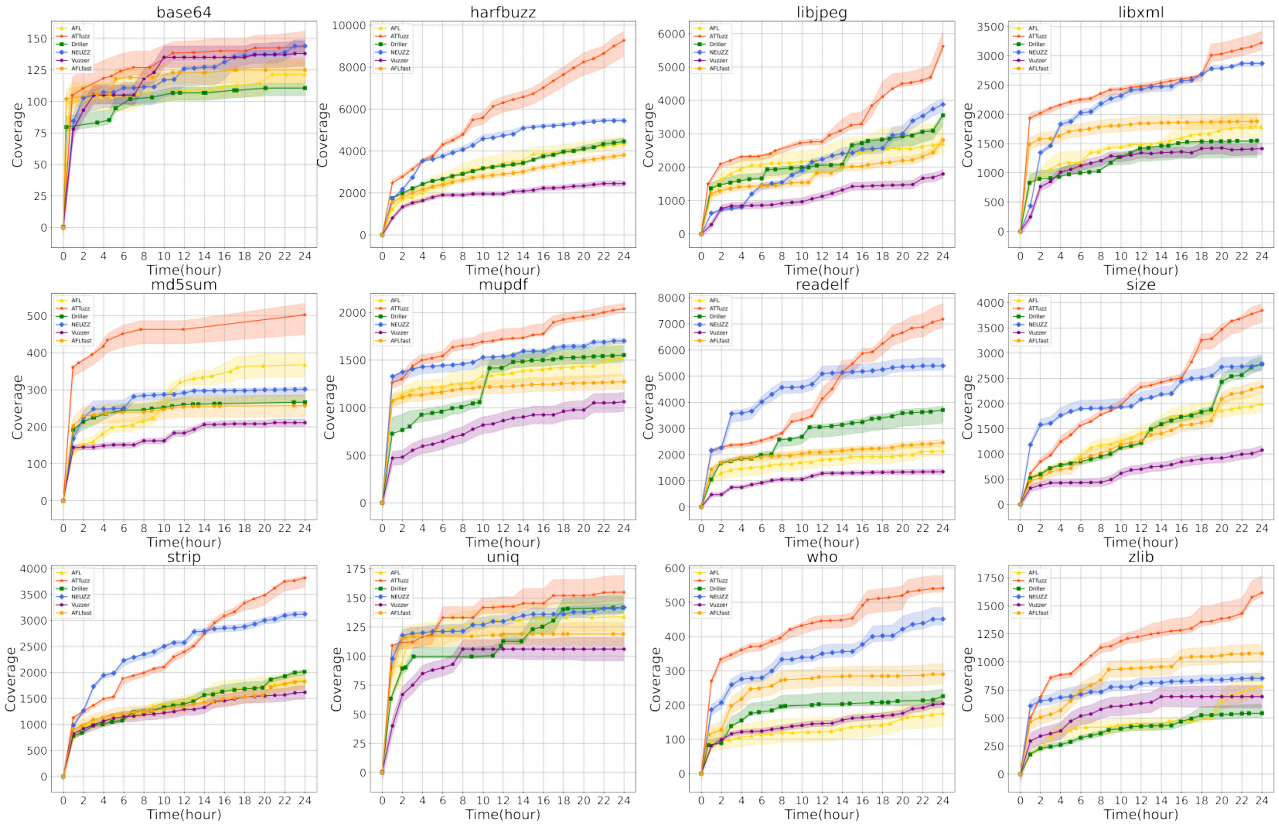

Figure 5: Coverage growth through 24 fuzzing.

comparison of AFL and ATTuzz for the sake of space[4]). We could observe that at the beginning, both fuzzers do not encounter any bottlenecks and have a high coverage growth rate. However, in the later stage of fuzzing, AFL struggles to achieve new coverage, i.e., having a significantly lower coverage growth than ATTuzz. Thanks to reward calculation and attention guidance, ATTuzz is able to repeatedly break the bottlenecks (even in the very late hours especially for libjpeg and size) and achieve the highest coverage in 24 hours compared to baseline fuzzers. More comprehensive results can be found at our github site, which consistently shows the effectiveness of ATTuzz in breaking the bottlenecks.

Evaluation of mutation guidance. Recall that the heat maps are used by ATTuzz to guide the mutation on the right bytes of the input with right mutators to better reach the critical blocks and further break the bottleneck. We use the ratio of inputs to trigger the critical blocks by different fuzzers to show the effectiveness of our heat map based mutation guidance. The results are shown in Figure 7. We have performed mutation ratio statistics on the AFL-based Fuzzer. We observe that the input generated by AFL and AFLfast has a very sparse coverage ratio of the critical blocks, i.e., only about 10% of the inputs can cover them. In other words, most fuzzing effort is spent in program branches that are less useful to induce new coverage. Note that to keep certain randomness (in the same spirit of MCMC [1]) in the exploration, ATTuzz also mutates the hot bytes with a certain small probability (5% in our experiment and flexible to adjust). Overall, guided by the heat maps, ATTuzz pays good attention to those critical blocks, i.e., about 75% of the

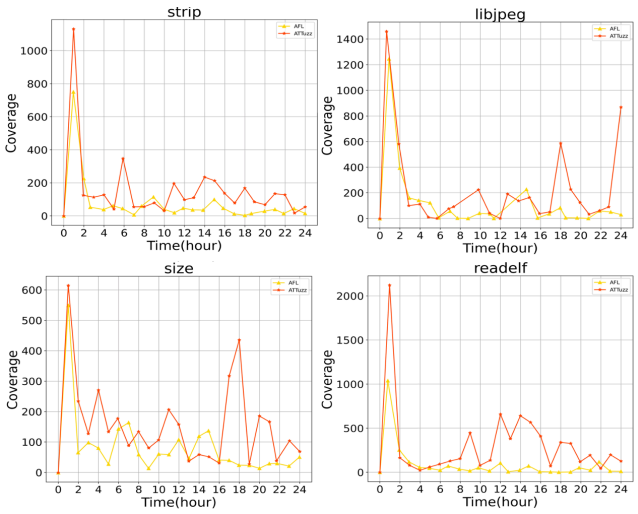

Figure 6: The normalized derivative average value.

[4]Comparisons on the other programs share a similar trend and are available at https://github.com/ICSE2022/ATTuzz

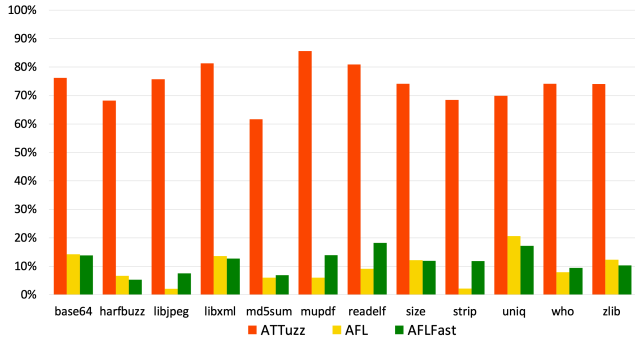

Figure 7: Critical block coverage ratio.

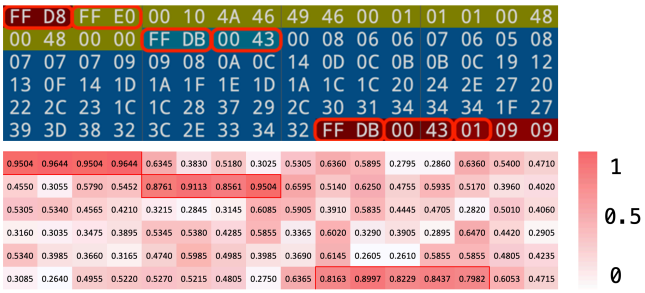

Figure 8: A fragment of the JPEG file format.

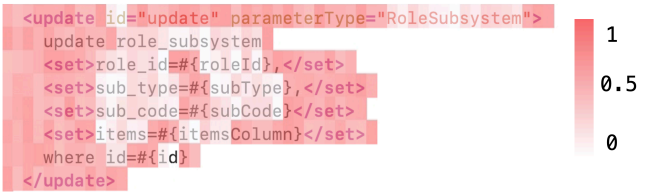

Figure 9: A fragment of the xml file format.

input generated by ATTuzz can reach the target critical blocks, which greatly increases the chance of breaking the bottlenecks.

Case study of heat map. Take libjpeg as an example. The upper part of Figure 8 is a beginning fragment of a jpeg format file opened in 010EDITOR, where different colors represent different areas and structures of the file. Specifically, the parts in the red circle represent a special field identifier. If any of these bytes is changed, the file cannot be recognized as a jpeg file normally, while mutating the rest bytes will not directly lead to such a problem. The correspondence between the program coverage and byte mutation is successfully reflected in the heat map (the lower part of Figure 8), i.e., the bytes in the red circles mostly have high heat values (hot bytes). ATTuzz avoids mutating these bytes to ensure that the generated inputs are still valid. Similarly, we observe that bytes that are critical for the validity of XML syntax are successfully identified as hot-bytes. Hot bytes in Figure 9 indicate those words which are closely related to

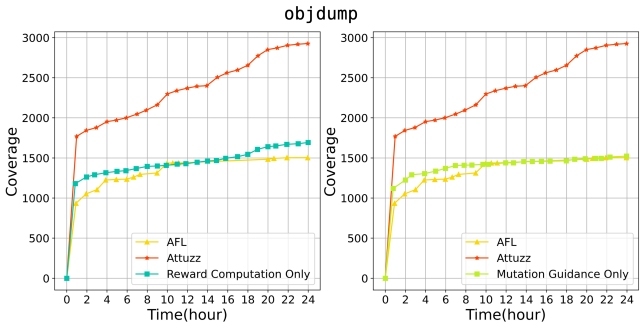

Figure 10: Coverage result with individual component.

the xml syntax, and mutating them will directly cause the xml file fail to execute (invalid inputs).

RQ3. Are reward computation and attention guidance complementary to each other?

In the following, we show that either techniques by itself is not impressive using two sets of experiments. First, we keep using reward calculation to guide fuzzer to select seed files that can reach the critical blocks, but we do not use the attention models to guide mutations on these seed files to show the effectiveness of the mutation guidance. Second, we omit reward calculation and only randomly select k% of all uncovered blocks and train attention model on their pre-dominant blocks. We conduct mutation guidance towards covering these blocks to show the effectiveness of reward calculation. The results are shown in Figure 10. The figure on the left and right shows the result of the first and second set of experiments respectively. We observe from the left figure that selecting the valuable seed file only has a slight improvement in the performance of fuzzing (compared to AFL). And the effect of randomly selecting blocks for mutation guidance is almost negligible as evidenced by the right figure. In summary, the results show that either technique by itself is not impressive to improve the performance of fuzzing while combining them will significantly boost fuzzing's performance. The probable reason is that even if the seed file is selected correctly, random mutation will destroy the effectiveness of the input, while random selection of blocks will lead to a high probability of selecting non-rewarding blocks. (the generated input would have a high probability of triggering this node, or this node has little child nodes). Similar results are obtained on other programs as well.

RQ4. Does ATTuzz discover more bugs?

The main purpose of fuzzing is to find as many bugs as possible. In this experiment, we evaluate each fuzzer's capability of detecting bugs in the software binaries. We summarize the results in Table 3. Note that we omit those software that all fuzzers fail to find any bug. For the remaining programs, ATTuzz is able to find 9X, 7X, 4X, 4X and 1.5X times of bugs than Vuzzer, Driller, AFLFast, AFL and NEUZZ respectively, which shows the usefulness of improved

Programs	ATTuzz	NEUZZ	AFL	AFLfast	Driller	Vuzzer
Bug Detection						
mupdf	**8**	0	0	0	0	0
readelf	29	16	3	5	7	5
objdump	**8**	**8**	6	6	2	0
size	2	**6**	0	4	0	1
strip	**20**	**20**	7	5	2	2
Bug type						
interger overflow	✓	✓	×	×	×	✓
heap overflow	✓	✓	×	✓	×	×
invalid pointer	✓	×	✓	✓	✓	×
out-of-memory	✓	✓	✓	✓	✓	✓
assertion crash	✓	✓	×	×	×	×
Total	**67**	50	16	20	10	8

Table 3: Bug Detection.

coverage by ATTuzz in discovering more bugs. Besides, ATTuzz can not only find the maximum number of bugs, but also trigger more kinds of bugs than other fuzzers in baseline.

6 CONCLUSION

We present ATTuzz, an efficient fuzzer which pays better attention on 1) the valuable blocks identified using a reward calculation mechanism based on the fuzzing data, and 2) the valuable bytes and mutators using a deep learning network with attention mechanism to focus on those critical seed files and mutations. We further demonstrate how attention models with heat maps can be utilized to guide more effective test inputs generation to help the fuzzer break the bottlenecks met. We extensively evaluate ATTuzz on 13 real-world programs comparing to 5 state-of-the-art fuzzers of 3 different categories, and show that ATTuzz achieves over 50% more coverage and detects up to 11X bugs. ATTuzz shows the potential of paying attentions with the help of deep learning whilst fuzzing and extends the possibility of detecting more vulnerabilities by fuzzing with attentions.

REFERENCES

[1] Christophe Andrieu, Nando De Freitas, Arnaud Doucet, and Michael I Jordan. An introduction to mcmc for machine learning. *Machine learning*, 50(1):5–43, 2003.

[2] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *NDSS*, volume 19, pages 1–15, 2019.

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[4] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Synthesizing program input grammars. *ACM SIGPLAN Notices*, 52(6):95–110, 2017.

[5] Marcel Böhme, Valentin JM Manès, and Sang Kil Cha. Boosting fuzzer efficiency: An information theoretic perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 678–689, 2020.

[6] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, 2017.

[7] Richard H Byrd, Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal on scientific computing*, 16(5):1190–1208, 1995.

[8] M. Böhme, V. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2019.

[9] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[10] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: preliminary assessment. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 1066–1071. IEEE, 2011.

[11] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[12] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *2015 IEEE Symposium on Security and Privacy*, pages 725–741. IEEE, 2015.

[13] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.

[14] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. Grey-box concolic testing on binary code. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 736–747. IEEE, 2019.

[15] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of machine learning research*, 12(ARTICLE):2493–2537, 2011.

[16] Kingma Da. A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[17] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121. IEEE, 2016.

[18] Shawn Embleton, Sherri Sparks, and Ryan Cunningham. Sidewinder: An evolutionary guidance system for malicious input crafting. *Black Hat USA*, 2006.

[19] Julian Fietkau. American fuzzy lop. https://lcamtuf.coredump.cx/afl/.

[20] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. Afl++: Combining incremental steps of fuzzing research. In *14th {USENIX} Workshop on Offensive Technologies ({WOOT} 20)*, 2020.

[21] Ivan Firdausi, Alva Erwin, Anto Satriyo Nugroho, et al. Analysis of machine learning techniques used in behavior-based malware detection. In *2010 second international conference on advances in computing, control, and telecommunication technologies*, pages 201–203. IEEE, 2010.

[22] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696. IEEE, 2018.

[23] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[24] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[25] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 50–59. IEEE, 2017.

[26] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 49–64, 2013.

[27] Mark Harman and Phil McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering*, 36(2):226–247, 2009.

[28] Sarfraz Khurshid, Corina S Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568. Springer, 2003.

[29] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[30] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2018.

[31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[32] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 475–485, 2018.

[33] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 627–637, 2017.

[34] Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. Cerebro: context-aware adaptive fuzzing for effective vulnerability detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 533–544, 2019.

[35] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1949–1966, 2019.

[36] Valentin JM Manès, Soomin Kim, and Sang Kil Cha. Ankou: guiding grey-box fuzzing towards combinatorial difference. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1024–1036, 2020.

[37] Barton P Miller, David Koski, Cjin Pheow Lee, Vivekandanda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. Fuzz revisited: A re-examination of the reliability of unix utilities and services. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1995.
[38] David Molnar, Xue Cong Li, and David A Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *USENIX Security Symposium*, volume 9, pages 67–82, 2009.
[39] Matthias Neugschwandtner, Paolo Milani Comparetti, Istvan Haller, and Herbert Bos. The borg: Nanoprobing binaries for buffer overreads. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 87–97, 2015.
[40] Nicole Nichols, Mark Raugas, Robert Jasper, and Nathan Hilliard. Faster fuzzing: Reinitialization with deep neural models. *arXiv preprint arXiv:1711.02807*, 2017.
[41] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.
[42] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2155–2168, 2017.
[43] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Aflnet: a greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 460–465. IEEE, 2020.
[44] Mohit Rajpal, William Blum, and Rishabh Singh. Not all bytes are equal: Neural byte sieve for fuzzing. *arXiv preprint arXiv:1711.04596*, 2017.
[45] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.
[46] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. *ACM SIGSOFT Software Engineering Notes*, 30(5):263–272, 2005.
[47] K. Serebryany. libfuzzer – a library for coverage-guided fuzz testing. http://llvm.org/docs/LibFuzzer.html.
[48] Kostya Serebryany. Oss-fuzz-google's continuous fuzzing service for open source software. 2017.
[49] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program smoothing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 803–817. IEEE, 2019.
[50] Suphannee Sivakorn, George Argyros, Kexin Pei, Angelos D Keromytis, and Suman Jana. Hvlearn: Automated black-box analysis of hostname verification in ssl/tls implementations. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 521–538. IEEE, 2017.
[51] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.
[52] Jingyi Wang, Xiaohong Chen, Jun Sun, and Shengchao Qin. Improving probability estimation through active probabilistic model learning. In *International Conference on Formal Engineering Methods*, pages 379–395. Springer, 2017.
[53] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 579–594. IEEE, 2017.
[54] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*, pages 497–512. IEEE, 2010.
[55] Xinyu Wang, Jun Sun, Zhenbang Chen, Peixin Zhang, Jingyi Wang, and Yun Lin. Towards optimal concolic testing. In *Proceedings of the 40th International Conference on Software Engineering*, pages 291–302, 2018.
[56] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 2307–2324, 2020.
[57] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 745–761, 2018.
[58] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 2255–2269, 2020.

7 SUPPLEMENTARY MATERIAL

A Motivating Example

We start with a motivating example shown in Figure 11. First, the program checks a nested condition determined by three variables a, b, and c. The $Flag$ is set to 1 if the nested condition is satisfied. Then, the program determines if the corresponding bytes in the buffer are equal to X. If yes, a bug is triggered.

Such a simple program brings several challenges to coverage-guided fuzzers like AFL. First, *the nested conditions and checksum are not easy to be satisfied by random mutations (Challenge 1),* i.e., the probability to generate a seed covering new control locations is extremely low. Second, even if we finally meet the conditions and obtain the corresponding seed, *the effectiveness of the seed will be easily destroyed by the random mutation mechanisms (Challenge 2).* For example, after a period of fuzzing, AFL finally met the conditions of line 2, 3 and 4 in Figure 11, and obtained a seed file that can set $Flag$ to 1. In order to further improve coverage, AFL mutates this seed to generate more inputs. Unfortunately, such random mutations are hardly useful as they very likely set flag to false. For instance, AFL fails to cover line 8 after 12 hours and 13 million runs even a seed which could set $Flag$ to 1 is given.

To tackle the above challenges, different works have been proposed which can be roughly divided into two categories. The first category used program analysis techniques such as dynamic taint analysis [13, 45], concolic execution [9, 29, 51, 57] and static analysis [33, 45] to identify the most relevant bytes to mutate (a.k.a. solving the magic byte problem) directly. While program analysis based approaches can accurately obtain the value of the magic byte (with a powerful solver at hand), the subsequent mutation may still destroy such "magic". The other category introduced machine learning to filter less meaningful inputs (potentially with no coverage gain) [58] or use it to generate more valuable inputs [44, 49]. However, existing machine learning based approaches have limited effectiveness in selecting control locations that can maximize the coverage reward. For instance, Neuzz's gradient guided mutation strategy can only keep 10% of newly generated inputs reaching line 7 with seeds setting $Flag$ to 1. Besides, the adopted machine learning models (e.g., LSTM [44] and NN [49]) are often considered lacking of interpretability.

In this work, we aim to address both challenges systematically by deepening our understanding from two sides, i.e., the program and the fuzzing process, and their interactions. First, we use a carrier fuzzer (e.g., AFL) to generate a number of tests and run the target program with them. Note that the program is instrumented to collect the execution trace of each input, which contains more fine-grained information than coverage. After a while, ATTuzz determines that covering some of the blocks that are yet to be covered will be more rewarding than others by calculating a *coverage reward* for each uncovered control locations based on the abstraction of the program estimated from the fuzzing data. In the running example, line 7 will have the highest reward. ATTuzz then selects the target control locations for machine learning by prioritizing those with high rewards. In the machine learning phase, ATTuzz adopts state-of-the-art deep learning models with attention mechanism, which allows us not only to predict whether a specific combination of seed and mutator can reach the target location, but also to obtain the "explanation" (in form of heat maps) which informs us the importance of different bytes (under a mutator) on the program coverage (e.g., can reach line 7 or not). With the explanation, ATTuzz then guides the fuzzer to mutate the corresponding position of the input with different probabilities. In the example, if the seed input can already set $Flag$ to 1, we will avoid performing $bitflip$ on b, and

```
void func(int a, int b, int c, char* buf) {
1.    int Flag = 0;
2.    if(a > 100) {
3.        if(b == -1) {
4.            if(c < 0) {
5.               Flag = 1;
                 return 0;
              }
          }
      }
6.    if(Flag) {
7.        if(buf[0] == "X") {
8.            buggy code ...;
          }
      }
9.    return 1;
  }
```

Figure 11: A motivating example

allow *arth* to operate on *c*. In this way, the effectiveness of the seeds will be preserved and many more inputs satisfying the nested conditions in line 2, 3 and 4 will be generated, thereby increasing the probability of triggering the bug in line 8.

Derivative comparison

Table 4 shows the normalized average derivative through 24 fuzzing. The greater the derivative, the more efficient the fuzzy can break through the bottleneck and increase the program coverage. Since ATTuzz continue to break through the bottleneck of in the later stage of fuzzing, its average derivative is nearly 1.5 times that of the second place.

Programs	ATTuzz	NEUZZ	AFL	Driller	Vuzzer	AFLfast
base64	**0.263**	0.194	0.135	0.178	0.09	0.141
harfbuzz	**0.307**	0.18	0.144	0.16	0.081	0.126
libjpeg	**0.219**	0.183	0.164	0.182	0.114	0.137
libxml	**0.243**	0.216	0.135	0.158	0.106	0.142
md5sum	**0.343**	0.181	0.167	0.115	0.147	0.048
mupdf	**0.327**	0.234	0.093	0.18	0.058	0.107
readelf	**0.257**	0.19	0.133	0.19	0.072	0.156
size	**0.256**	0.209	0.124	0.18	0.108	0.123
strip	**0.188**	0.180	0.142	0.155	0.176	0.16
uniq	**0.257**	0.16	0.184	0.164	0.106	0.129
who	**0.279**	0.245	0.094	0.12	0.108	0.154
zlib	**0.195**	0.178	0.168	0.176	0.133	0.149

Table 4: Normalized derivative through 24h's fuzzing

Case Study

We explain the effectiveness of ATTuzz by triggering BUG in Figure 12 in mupdf which is only triggered by ATTuzz in our experiment. As shown in Figure 12, after a period of fuzzing, AFLpp can cover lines 1-5 without the help of ATTuzz but it was unable to cover line 6-7. In this case, ATTuzz intervenes to help Fuzzing improve overall coverage. First, ATTuzz builds CFG by static analysis of the program, and uses the dynamic data generated by Fuzzing to supplement the CFG, and then the edge probability of the program Markov chain is estimated. According to the reward mechanism, ATTuzz calculates the reward of basic blocks and finds the line 6 has a high Reward (8.214 in this case). By tracing pre-dominant block, ATTuzz considers line 5 as a critical block, and integrate the collected fuzzing data to prepare for the next learning stage. Thanks to the attention mechanism, ATTuzz's model can learn the heat maps of different mutators under different seed files while classifying the inputs. Figure 13 shows some of the heat maps. When selecting seed files, we first filter out those that can reach line 5, so as to increase the possibility that the generated input can still reach the critical block. When mutating the seed file, we skip the mutation of high heat bytes according to different mutators and corresponding heat maps, so as to ensure that more input can be generated on the premise of reaching line 5. Under the guidance of ATTuzz, AFLpp triggered the bug of line 7 and further explored other branches of the program.

```
1.  fz_append_display_node(
        fz_context *ctx,
        fz_device *dev,
        fz_display_command cmd,
        int flags,
        const fz_rect *rect,
        const fz_path *path,
        const float *color,
        fz_colorspace *colorspace,
        const float *alpha,
        const fz_matrix *ctm,
        const fz_stroke_state *stroke,
        const void *private_data,
        int private_data_len)
    {
    ...
2.      if (list->len + size > list->max)
     {
     ...
3.      list->list = fz_realloc_array(  //allocated here
            ctx, list->list,
            newsize,
            fz_display_node);
4.      list->max = newsize;
            ...
     }
        ...
5.    if (private_off)
     {
6.       char *out_private = (char *)(void *)(&node_ptr[private_off]);
7.      memcpy(out_private, private_data, private_data_len);
         //overflow
     }
8.     list->len += size;
    }
```

Figure 12: A code fragment in mupdf 1.12.0

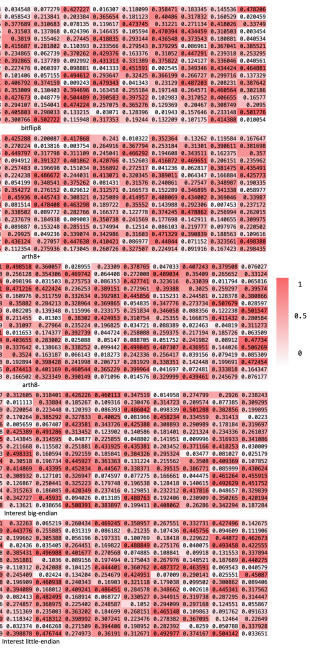

Figure 13: some of the heat maps.