Better Pay Attention Whilst Fuzzing

Shunkai Zhu, Jingyi Wang, Member, IEEE, Jun Sun, Jie Yang, Xingwei Lin, Tianyi Wang, Liyi Zhang, and Peng Cheng

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 mutate 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 that 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 mutations on certain bytes of a seed. The model is a deep learning model equipped with an 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 6 popular real-world programs and MAGMA data sets at achieving higher edge coverage and finding new bugs. In particular, ATTUZZ achieved 1.2X edge coverage and 1.8X bugs detected than AFL++ over 24-hour runs. In addition, ATTUZZ also finds 4 new bugs in the latest version of some popular software including p7zip and openSUSE.

Index Terms—Fuzzing, deep learning, program analysis, attention model.

I. 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 [1], [2]. Thanks to its simple and easy-to-apply concept, fuzzing has been widely adopted to test real-world programs [3], [4], [5], [6], [7]. Existing fuzzers like AFL [6] and its many variants [4], [5], [8], [9], [10], [11], [12], [13], [14], [15], [16] use evolutionary algorithms (and sometimes alternative optimization algorithms [4], [17], [18]) 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 that cover new branches as seeds. While such a strategy is effective initially, over time test inputs that cover new branches become few and far 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 over time. After covering the easy-to-cover branches, covering the remaining branches often requires 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 to address these two problems in the literature. To select seeds that are likely to cover new branches, LibFuzzer [5]’s heuristic is to select newly generated tests. AFLfast [4] prioritizes the seeds that can trigger the less frequently visited paths. Entropic [19] selects seeds that carry more program information according to certain entropy measures. Fairfuzz [20] locates and selects the seeds which trigger low-probability edges. Cerebro [21] 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 [13], REDQUEEN [22] and other works [17], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33] proposed to perform dynamic taint analysis to determine the specific bytes in the input for solving the so-called “magic bytes” problem. MOpt [18] utilizes a customized particle swarm optimization to adjust the distribution used by the mutation, which does not guide the position of the mutation on the seed file. Besides, machine learning has been introduced to improve the performance of fuzzing in recent years [12], [13], [14], [34], [35], [36], [37]. For instance, RNN fuzzer [38] uses 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 [39] achieves good results in directed fuzzing. Neuzz [40] 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).

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 that 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 an 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 8 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 1.2X edge coverage and 1.8X 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 i.e., achieving 50% more coverage than AFL after 5 days of fuzzing. In addition, ATTUZZ also finds 4 new bugs in the latest version of some popular software including p7zip and openUSD.

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.

II. BACKGROUND

A. Coverage-Guided Fuzzing

We start with formalizing the grey-box fuzzing problem.

Definition 1: A program is a labeled transition system \( \mathcal{P} = (B, \text{init}, V, \phi, GC, T) \) where

- \( B \) is a finite set of control locations\(^1\);
- \( \text{init} \in B \) 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

\[
\pi = ((v_0, b_0), gc_0, (v_1, b_1), gc_1, \ldots, (v_k, b_k), gc_k, \ldots),
\]

where \( v_i \) is a valuation of \( V \), \( b_i \in B \), \( gc_i = [g]_f \) is a guarded command such that \((b_i, gc_i, b_{i+1}) \in T \), \( v_i \models g \), and \( v_{i+1} = f_i (v_i) \) for all \( i \), and \( v_0 \models \phi \) and \( b_0 = \text{init} \). We use \( \Pi \) 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 \( \Pi \) if and only if there exists a concrete execution \( \pi \in \Pi \) which covers \( b \).

Definition 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 that could transit to \( b \) in one step.

Grey-box fuzzers like AFL [6] are designed to generate a set of test inputs \( \Pi \) 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 Fig. 1. First, the target program is instrumented to obtain program coverage

---

\(^1\)We use “control location” and “basic block” interchangeably throughout.
information during the fuzzing process. Second, to maximize reachable \( B_v \), the fuzzer 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 to 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. We have consolidated additions, subtractions, and different endians into a single item, summarizing all 32 distinct mutators in Table I. Each byte can be modified using one of the mutators listed in Table I with varying parameters. Even with the most conservative estimate of only combining different mutators, a seed file of length \( N \) bytes can produce at least \( 29^N \) unique program inputs.

### B. 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 [41], natural language processing [42] and progressively applied in computer security [43].

Further, the attention mechanism enables a neural network model to distinguish the contributions of different input segments in the model decision process [44], by explicitly assigning a unique weight for each basic unit of input data, and calculating the representation of the input data using a 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 incorporation of the attention mechanism provides a means of comprehending the importance of various bytes for the present coverage in the seed file. By analyzing the different weights assigned to the bytes in the attention layer, we gain insight into their relevance. Notably, bytes with higher weights hold a distinct significance as “magic bytes”. If mutated, these bytes may directly result in the alteration of the specific coverage.

### C. Problem Definition

We summarize our core problem as how to pay better attention 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?

### D. A Motivating Example

We start with a motivating example shown in Fig. 2. 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 checksums are not easy to be satisfied by random mutations (Challenge 1), i.e., the probability of generating a seed covering new control locations is extremely low. Second, even if we finally meet the conditions and obtain the corresponding seeds, 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 lines 2, 3, and 4 in Fig. 2, 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 the flag to false. For instance, AFL fails to cover line 8 after 12 hours and 13 million runs even a seed that 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 [12], [14], concolic execution [23], [24], [25], [45] and static analysis [12], [13] 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) [39] or use it to generate more valuable inputs [38], [40]. 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 [38] and NN [40]) are often considered lacking interpretability [46].

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 location 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 the 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 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 lines 2, 3, and 4 will be generated, thereby increasing the probability of triggering the bug in line 8.

III. ATTUZZ FRAMEWORK

In this section, we present details of ATTUZZ. An overview of ATTUZZ is shown in Fig. 3. 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 Fig. 3). Over time, the fuzzing process often gets stuck and the coverage is difficult to improve. Then, the ATTUZZ is activated. The core idea is to adopt deep learning with attention to predicting whether a seed and mutation combination can cover certain basic blocks 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 Fig. 3, 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.
Algorithm 1: Main Algorithm

1. Let $P$ be the target program;
2. Let $B$ be the set of bugs;
3. Let $\text{iter\_limit}$ be the maximum iterations;
4. $\text{ICFG} = \text{StaticAnalysis}(P)$;
5. for $\text{seed} \in \text{seed\_pool}$ do
6.   Let length be the length of seed;
7.   Let $\text{seed\_coverage}$ be the coverage of executing $P$ using seed;
8.   for iterations $\leq \text{iter\_limit}$ do
9.     if $\text{EncounterBottleneck}$ then
10.    uncovered_blocks = FindUncov(ICFG, cov);
11.    rewards = RewardCal(ICFG, cov);
12.    $B_{\text{critical}} = \text{Select(uncov_blocks, rewards, ICFG)}$;
13.    for block $\in B_{\text{critical}}$ do
14.       heatmap_list = TrainModel(dataset);
15.   for mutator $\in \text{mutator\_list}$ do
16.      for loc $\leq \text{length}$ do
17.       if $\text{EncounterBottleneck}$ then
18.         input = Guide(seed, $B_{\text{critical}}$, heatmap_list);
19.       else
20.         input = mutate(seed, mutator, parameter);
21.       cov_result = Excute($P$, input);
22.       dataset = Record(seed, mutation, cov_result);
23.       if result $==$ Crash then
24.         $B$.append(input);
25.       if HasNewCov(cov_result) then
26.         $\text{seed\_pool}$.append(input);
27. return $B$;

A. Overall Algorithm

We summarize our overall algorithm in Algorithm 1. A mutation budget for each seed is set as $\text{iter\_limit}$. We first obtain the Interprocedural Control Flow Graph (ICFG) of the target program $P$ 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 IV). 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 (lines 19-25). 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 to 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 selecting 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 $B$ (line 23). Different from existing learning-enabled fuzzing [35], [47], [48], [49], [50], [51], [52], ATTUZZ is more effective in accurately locating the bottleneck and paying better attention to those valuable bytes and mutators to break them.

Example 1: Take the program in Fig. 4 as an example. At the beginning, ATTUZZ runs AFL with its default mutation strategy and collects 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 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).

B. 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 2: For the program in Fig. 4, AFL performs an $\text{Arith}_+$ 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 $\langle 0, 5 \rangle$, $\langle 5, 5 \rangle$; 111000).

After a series of mutation operations, we can get the same form of data, such as $\langle -5, 0 \rangle$, $\text{arith}_+$, 5; 110101), $\langle 0, 0 \rangle$, $\text{bitflip}$, 5; 100101), $\langle 0, 0 \rangle$, dictionary, 0xFF; 100101).

C. 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 to a maximal improvement of the edge coverage. In the following, we present a lightweight approach that 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 the highest rewards. Our approach is inspired by [53], [54], which enables us to build a discrete-time Markov Chain (DTMC) abstraction of the program from the collected fuzzing data. Specifically,

Definition 3: A (labeled) discrete-time Markov Chain (DTMC) is a tuple $M = (B, P_r, \mu)$ where $B$ is the set of basic blocks in $P$; $P_r : B \times B \rightarrow \mathbb{R}^+$ is a labeled transition probability function such that $\Sigma_{b' \in B} P_r(b, b') = 1$ for all $b \in B$; and $\mu$ is the initial probability distribution such that $\Sigma_{b \in B} \mu(b) = 1$. 

Authorized licensed use limited to the terms of the applicable license agreement with IEEE. Restrictions apply.
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) {
                flag = 1;
            }
        }
    }
    6. if(flag) {
        7. if(buf[8] == "x") {
            buggy code ...;
        }
    }
    9. return
}

(a)Example program
(b)Edge condition
(c)Edge execution statistic
(d)DTMC estimation
(e)Reward calculation

Fig. 4. Program abstraction.

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 is associated with a conditional probability. For example, a program shown on the left side of Fig. 4 can then be transformed into the DTMC on the right [53]. The key to construct the DTMC is to estimate the conditional probabilities between edges from the fuzzing records as follows:

Definition 4: 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 b1 in the ICFG. Then the conditional probability from b1 to b2 is then \( P(b1, b2) = \frac{1 + \#(b1, b2)}{\#b1 + n} \).

We add n to the denominator and 1 to the numerator in order to use Laplace to estimate the probability of a state transfer edge instead of 0 even if b1, b2 have not been run once thus ensuring the correct estimation of the REWARD. The reward of a basic block is defined as follows.

Definition 5: 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.

\[
R_b = \begin{cases} 
1 + \sum_{t \in T} \{ P_r(b, t) \times R_t \} & \text{if } b \notin \text{visited} \\
\sum_{t \in T} \{ P_r(b, t) \times R_t \} & \text{if } b \in \text{visited}
\end{cases}
\]

With an estimated DTMC and the above equation system, we can calculate each basic block’s reward with the help of the program’s ICFG. Note that for those indirect calls that are unable to be extracted by static analysis, we use the dynamic fuzzing data to complement the ICFG of static analysis. We omit the details of solving the equation system (which is efficient, e.g., in seconds) and refer interested readers to [53] for details.

Example 3: We use the program in Fig. 4(a) as an example to illustrate the reward calculation process. Assuming that 998 inputs have been generated and tested to activate various edges, we proceed to record the number of executions of these edges (Fig. 4c). To estimate the probability of all state transfer edges and derive the DTMC, we refer to Fig. 4(d), which demonstrates the application of definition 4. Finally, Fig. 4(e) outlines the procedure for calculating rewards for each block by following Definition 5. 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 pre-dominant 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 have a low probability to reach as more interesting critical blocks. In practice, we omit those pre-dominant blocks which have 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 are 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.

D. Training Attention Models

The above two phases lay the foundation for ATTUZZ to focus on the most valuable basic blocks, which are critical for achieving high coverage during the testing phase. To build a robust model that can predict whether a seed-mutation combination will cover a certain code block, we use a deep learning approach. We choose an attention-based deep learning model for this task, which can extract features from inputs automatically and make accurate predictions [55]. The attention mechanism employed in our model enables it to identify and distinguish the impact of different mutators and parameters on each byte for different seed files.

To prepare the training data, we split them into two parts: the seed file and the mutator along with the corresponding mutation parameters. Specifically, we convert the seed file into a series of vectors \( XD \times N = (v_1, v_2, \cdots, v_N) \), \( v_{m,d+i} \), \( v_{p,d+i} \), where \( D \) is the dimension of byte embeddings, and \( N \) is the maximum size of the collected seed files. If the input size is less than the maximum length, we pad it to the length \( N \). Next,
we use customized models for different kinds of programs to extract relevant features based on their inputs. For instance, for programs that take images as input, we use traditional convolutional neural networks (CNNs). On the other hand, for programs that take byte sequences as input (such as XML files or JSON strings), we use recurrent neural networks (RNNs) to capture the sequential information in the input.

After feature extraction, we get the vector of extracted features as $\mathcal{U} \times D^t \times N = \langle v_1, v_2, \cdots, v_N \rangle$. These networks have been widely used for image feature and byte sequence feature extraction. By leveraging the extracted features, our model can make predictions about whether a seed-file will fail to reach line 6, i.e., 99.9% of the generated test cases fail to reach line 6. AFL spends most of its effort on test cases that fail to reach line 6, i.e., 99.9% 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 $\text{Arith}$ on the seed file $\{a=200, b=-1, c=-10, bu[0]="Y", bu[1]="Y", bu[2]="Y\}$ can be visualized as the heat map at the bottom of Fig. 5. We could see that variables $a$ and $b$ are important to the coverage with very high weight, and the $\text{Arith}$ operation on other variables does not affect the coverage of program execution. This is consistent with the logic of the program.

In summary, our attention model consists of three main parts. Firstly, depending on whether the input type is a picture or a byte sequence, we use either three layers of CNN or RNN to extract features. Secondly, we introduce an attention layer to apply attention weights to each feature. Finally, the weighted features are passed through a fully connected layer for classification.

Example 4: Take the program in Fig. 4(a) for example, after 1 hour’s fuzzing, AFL can only cover lines 2, 3, 4, and 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 most of its effort on test cases that fail to reach line 6, i.e., 99.9% 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 $\text{Arith}_-$ on the seed file $\{a=200, b=-1, c=-10, bu[0]="Y", bu[1]="Y", bu[2]="Y\}$ can be visualized as the heat map at the bottom of Fig. 5. We could see that variables $a$ and $b$ are important to the coverage with very high weight, and the $\text{Arith}_-$ 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 $\text{Arith}_+$, variables $b$ and $c$ are important to the coverage of the seed file, and adding variable $c$ will not affect the result of comparison $>$. The attention weight for each mutator $\text{Arith}_-$ is calculated as follows:

$$\alpha_i = f_{att}(\mathcal{U}^{D^t \times N}, v_{m,i}, v_p)$$

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

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

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

$$\mathbf{W} = \sum_{\alpha_i \in \mathcal{A}}$$

$$\mathcal{M} = \begin{bmatrix} \mathbf{W} \\ \vdots \\ \mathbf{W} \end{bmatrix}$$

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 Fig. 5. 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 consists of three main parts. Firstly, depending on whether the input type is a picture or a byte sequence, we use either three layers of CNN or RNN to extract features. Secondly, we introduce an attention layer to apply attention weights to each feature. Finally, the weighted features are passed through a fully connected layer for classification.
the fuzzer to generate inputs that can reach the selected critical blocks. In the heat map, the heat level 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 2 shows the details of how ATTUZZ efficiently generates a large number of inputs that can reach the critical blocks with the guidance of the heat maps. We first obtain the seed file coverage (line 6). If the seed does not cover any critical block at all, we skip this seed and pick the next seed in the seed queue. Otherwise, for each mutator (line 9), we skip the hot bytes (determined at line 13) to avoid a change in the coverage (line 14). 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 $p$.

**Example 5:** Following the example in Section III-D, the seed file can cover line 6 and we introduce our mutation guidance strategy. Take $Arith_-$ as an example whose heat map is shown in Fig. 5. 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 $Arith_-$ operation, we avoid the subtraction of $a$ and $b$ and mutate the remaining variables with high probability. Similarly, for the $Arith_+$ operation, we avoid adding variables $b$ and $c$ because their heat is greater than the average heat of 0.5123.

**F. Bottleneck Judgement**

Intuitively, fuzzing encounters a bottleneck if the coverage does not increase over a certain amount of time. We do not perform training and mutation guidance at all times. This is because, on the one hand, guiding requires additional computational resources. If fuzzing can quickly discover new coverage in the current state, then there is no need to introduce ATTUZZ. On the other hand, during the rapid growth period of coverage, the rewards of basic blocks will change rapidly, so we cannot select the target critical block for further training and mutation guidance. To sum up, we record the current rate of coverage growth in real time. Once the hourly growth rate of coverage falls below $k\%$, we consider that fuzzing has encountered a bottleneck and we further introduce ATTUZZ. In practice, we check whether a bottleneck is met every hour (which aligns with the model training time) by calculating if the coverage increase in the last hour is smaller than a threshold, in our case, 5%.

**IV. IMPLEMENTATION DETAILS**

**A. 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++ [8] as our carrier fuzzer. Note that compared with AFL provides accurate coverage information (without hash collision) as well as the above-mentioned additional information which is 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 block 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 ICFG. 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. To minimize the impact of data collection on fuzzing, we employ a lightweight method for collecting coverage information. First, we statically analyze the program after it has been compiled with afl-clang-fast/++, emulating the way AFL++ computes edges to generate an “edge-basic block” dictionary. This dictionary enables us to analyze block coverage in parallel with the fuzzing process, allowing us to record only the bitmap while fuzzing. Additionally, we intervene only when we encounter a bottleneck, rather than recording fuzzing data at every instance. When ATTUZZ detects there is a bottleneck, it analyzes the raw edge bitmap recorded while fuzzing and looks up the dictionary to get its corresponding basic block coverage.

Besides, to achieve a comprehensive and accurate simulation of program execution records in AFL++, it is necessary to consider the call relationships between program functions. While normal control flow graph (CFG) analysis tools like IDApro

```plaintext
Algorithm 2: Mutation Guidance
1 Let seed be the seed file;
2 Let $B_{\text{crit}}$ be the set of blocks selected by reward;
3 Let heatmap list be the set of heatmap;
4 Function GUIDE(seed, $B_{\text{crit}}$, heatmap list)
5   for mutator $\in$ heatmap 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;
13   return program input;
```
and the LLVM command “opt” can generate separate CFGs for each function, they fail to directly merge the call relationships between CFGs. This not only causes the loss of numerous edges but also leads to incorrect edge recording due to function calls. Thus, to address this limitation, we introduce the use of the Interprocedural Control Flow Graph (ICFG), which enables us to model program execution with greater fidelity and achieve more reliable results. This approach further reduces the impact of data collection and model training on fuzzing.

B. Seed File Scheduling

Within the seed file pool, there may exist numerous seed files that can cover the target critical block. The challenge lies in selecting these seed files in a preferred manner. In our specific implementation, we determine whether to skip the current seed by assessing whether its coverage contains a critical block. This approach offers several benefits, including the ability to seamlessly adapt the seed scheduling policy to all AFL++ native implementations when there are multiple seed files that can cover the target critical block.

C. 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 a few. To solve this problem, we under-sample the data to ensure it has a balanced 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. Specifically, we selected a total of 10,000 target inputs as the training dataset. 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 use a sequence of raw binary bytes as input to the neural network and choose different types of neural networks for different input file types in order to better extract the input features. For file formats similar to images (jpeg, png), we interpret the binary data as a sequence of pixel values and reshape it into an image with a fixed width and height, and use CNN for input feature extraction, because CNN is naturally designed to extract features for image type inputs. For other program inputs such as zip, xml, lua, we choose RNN as the feature extraction layer. Because RNNs are specifically designed to work with sequential data, such as speech and text. It can learn to recognize patterns and relationships within sequences and can use this information to generate predictions.

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 the cross-entropy loss function in our model and Adam optimizer [56] with a learning rate of 0.001 to help the model coverages 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.

D. Mutation Guidance

As mentioned in section III-E, 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). Further, if a seed file can cover multiple critical blocks, then we merge the corresponding multiple heat maps into one for mutation guidance.

Meanwhile, we choose to mutate these hot bytes with a probability of 5% to introduce some randomness in the mutation.

E. Bottleneck Judgement

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

V. Evaluation

We have implemented ATTUZZ on top of AFL++, which has been released as an open-source toolkit2. The neural network models are implemented by pytorch and require numpy and torch packages. In this section, we evaluated ATTUZZ from two aspects. First, we compare the performance of ATTUZZ with state-of-the-art baseline fuzzers including: AFL++ [8], AFLfast [4], MOpt [18], Angora [14], QSYM [45] and NEUZZ [40] to assess its effectiveness and efficiency. Then, we evaluate the usefulness of each component of ATTUZZ.

The chosen baselines and the reasoning behind them are as follows. These Baseline fuzzers are designed for different goals, which in turn reflects the ATTUZZ’s effectiveness by a fair comparison.

- AFLfast performs seed file scheduling through statistics of execution path to optimize AFL. We aim to show the effectiveness of the reward mechanism in ATTUZZ by comparing it with it.
- MOpt utilizes a customized algorithm to find the optimal selection probability distribution of mutation to accelerate the fuzzing speed. MOpt’s idea of mutator selection coincides with the generation of different heat maps for different mutators in ATTUZZ. We use the “-d” option to skip the deterministic stage.
- Angora uses byte-level taint tracking to determine the magic bytes, while ATTUZZ uses the attention model to reflect magic bytes in the heatmap.
- NEUZZ also introduces a neural network to smooth the programs and help the fuzzer construct inputs efficiently.

2https://github.com/IS2Lab/ATTuzz
TABLE II
PROGRAMS IN THE EXPERIMENTS

| Programs     | name                        | Size   | Version |
|--------------|-----------------------------|--------|---------|
| JPEG        | libjpeg(g@?)                | 1.1M   | 9c      |
| TIF          | harfbuzz(g@?)               | 6.1M   | 2.00    |
| PDF          | mupdfinfo(g@?)              | 45.8M  | 1.12.0  |
| XML          | libxml(g@?)                 | 9.2M   | 2.9007  |
| ZIP          | zlib(x@?)                   | 441K   | 1.01b   |
| BIN          | readdir-a(g@?)              | 5.4M   | 2.30    |
|             | size@g@|                    | 9.2M   | 10.7M   |
| MAGMA Dataset| tiff_read_rgba_fuzzer      | 2.2M   |         |
| LUA          | lua                         | 1.6M   | 1.2.1   |
| PDF          | pdf_fuzzer                  | 25.1M  |         |
| SQ           | sqlite3_fuzzer              | 6.6M   |         |
| PNG          | libpng_read_fuzzer          | 821K   |         |

TABLE III
FUZZER OPTIONS

| Fuzzer       | Options                         |
|--------------|---------------------------------|
| ATTuzz       | -m 200M -L 0 -p fast            |
| AFL++        | -m 200M -L 0 -p fast            |
| AFLfast      | -m 200M -L 0                   |
| MOpt         | -d -m 200M -L 0                |
| NEUZZ        | -i MAX_SEED_LEN                 |
| QSYM         | -M afl-master -m 200M          |
| Angora       | -M 200                          |

- QSYM integrates a concolic execution engine with the native execution using dynamic binary translation to support hybrid fuzzing.

Through the experiments, we aim to answer the following research questions.

**RQ1:** Does ATTuzz improve code coverage effectively and efficiently?

**RQ2:** Does ATTuzz allow us to break the bottleneck effectively?

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

**RQ4:** How the use of a CNN vs. RNN affect the performance of ATTuzz over different input formats?

**RQ5:** Does ATTuzz allow us to discover more bugs?

A. Experiment Settings

1) Target Programs: We evaluate ATTuzz on various different types of programs which have been widely adopted by previous works [4], [12], [13], [14], [18], [40], [57], [58], [59] (i) 8 real-world programs, as shown in Table II. These programs cover several representative types of program input, including plain text formats, other highly structured input formats (including images, PDF, ZIP, BIN), (ii) MAGMA dataset, as detailed in the authoritative publication by Hazimeh et al. [60], is a remarkable collection of open-source libraries with a long history of security-critical bugs and vulnerabilities, widely utilized across various domains. With the aim of improving fuzzer evaluation, it has front-porting bugs from prior bug reports into the latest versions of these libraries. To facilitate real-time tracking of fuzzer progress, MAGMA has introduced in-line (source-code-level) instrumentation for each ported bug to gather ground-truth data on reached bugs (buggy code executed) and triggered faults (condition satisfied by input). This instrumentation enables a monitoring utility to measure fuzzer progress continuously, contributing to enhanced testing and vulnerability detection. Furthermore, the magma dataset features a diverse set of program inputs, encompassing binary images, PDFs, and plain text formats, which can verify the robust adaptability of ATTuzz’s neural network to various input types.

2) Experimental Setup: The host used in the experiment has 64-core CPU (Intel(R) Xeon(R) Gold 6226R CPU @ 2.90GHz), 8 GPU (RTX3090), 384 main memory and OS ubuntu 18.04. Note that each fuzzing task only takes one CPU core.

3) Fuzzer Setup: All AFL-based baselines turned off the deterministic mutation stage (by default in ATTuzz and AFL++, for MOptAFL and AFLfast, -d is used to turn it off, and QSYM, Neuzz is evaluated using the guided option on their Github pages). We have listed the more detailed parameter options for each fuzzer in Table III. 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.

A. 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 (Fig. 6) and 2) the edge coverage growth over time (Fig. 6). Our experimental settings align with the recommended approach outlined in [61], which advocates for consistent time budgets and seed corpora across different fuzzers. Specifically, we run each fuzzer for a fixed time budget, utilizing the same initial seed corpus, and compare their achieved edge coverage and number of detected bugs. To ensure accurate and reliable results, we repeated each experiment ten times and averaged the outcomes, while the shadow lines represent the 95% confidence intervals for the results. Notably, ATTuzz’s model training and fuzzing are performed concurrently, meaning that ATTuzz will still utilize the default AFL++ strategy until it has encountered a bottleneck or until the model is fully trained. As a result, the training time is included in the overall timeline. Our results demonstrate that for 12 out of 13 programs tested, ATTuzz outperforms other state-of-the-art fuzzers, such as AFL++, AFLfast, Neuzz, MOpt, and Angora in terms of mean coverage. On average, ATTuzz achieves 20% more mean coverage than AFL++, 70% more than AFLfast, 35% more than Angora, 30% more than Neuzz, 40% more than QSYM, and 20% more than MOpt. Furthermore, for programs such as libpng, libtiff, libxml, poppler, libjpeg, mupdf, and readelf, ATTuzz achieves the same or even more coverage in the first two hours than that of AFL++ and AFLfast in 24 hours. This superior performance of ATTuzz in enhancing program coverage is especially remarkable considering that the tested programs often have highly structured inputs, such as the JPEG format or the ELF format. These structured inputs are highly amenable to deep learning techniques,
such as attention mechanisms that extract valuable features and facilitate the selection of appropriate bytes and mutators for mutation. Additionally, as illustrated in Fig. 6, while baseline fuzzers tend to become stuck in bottlenecks after a certain period of fuzzing, ATTUZZ consistently enhances coverage in the long run, which is particularly evident in programs such as libpng, libtiff, libxml, poppler, libjpeg, libxml, mupdf, readelf, size, strip, and zlib.

**Answer to RQ1:** On 12/13 benchmarks, ATTUZZ achieves higher mean coverage and on 7 benchmarks, ATTUZZ achieves max coverage faster than the baseline.

**B. 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, which is the average change in coverage per hour, and show the results in Fig. 7. During the early stage of fuzzing, we observed that all the fuzzers had a high coverage growth rate and did not encounter any bottlenecks. However, in the later stage, in the test of the program poplar, harfbuzz, libjpeg, readelf, size, strip, ATTUZZ shows multiple small spikes in coverage growth. This is because ATTUZZ is able to cover the critical block based on the critical block to those forward large uncovered basic blocks to improve the overall coverage to break the bottleneck. For the remaining programs, the difference in coverage growth rate is not significant though. However, according to Fig. 6, we can still get that ATTUZZ ended up in first place in terms of average coverage among the 12/13 programs, thus reflecting that ATTUZZ is in the lead in terms of average growth rate.

1) **Evaluation of Mutation Guidance**

Recall that the heat maps serve as a powerful tool for ATTUZZ to guide the mutation process on the right bytes of the input with the right mutators, in order to better reach the critical blocks and further break the bottleneck. To showcase the effectiveness of our heat map-based mutation guidance, we perform a ratio analysis of inputs that trigger critical blocks using different fuzzers, as shown in Fig. 8. To determine the average critical block coverage ratio, we captured the inputs produced by ATTUZZ at the onset of guiding mutations following each bottleneck encounter, calculating the percentage of these inputs that could cover the critical block. Similarly, for other fuzzers, we logged the inputs generated upon encountering bottlenecks and computed the percentage of critical blocks covered. Given the potential occurrence of bottlenecks multiple times during the 24-hour testing period, we averaged all the ratios obtained. In our analysis, we focus on the AFL-based fuzzer and observe that the inputs generated by AFL and AFLfast cover only a sparse ratio of the critical blocks, with only about 10% of the generated inputs able to cover them. This indicates that most of the fuzzing effort is spent on program branches that are less useful in inducing new coverage. It is worth noting that, while ATTUZZ uses heat maps to guide the mutation process, it still maintains a certain level of randomness in the exploration, akin to the spirit of MCMC [62]. Specifically, ATTUZZ mutates the hot bytes with a small probability (5% in our experiments) to improve the diversity of the generated inputs. Overall, we find
that guided by the heat maps, ATTUzz pays significant attention to the critical blocks, with approximately 75% of the generated inputs able to reach these targets. This significantly increases the likelihood of breaking the bottlenecks and improving the overall performance of the fuzzer.

2) Case Study of Heat Map

As an illustrative example, let us consider libjpeg. The top part of Fig. 9 displays a segment of a jpeg file, with different colors representing distinct areas and structures. The red circles highlight a crucial field identifier. Altering any of these bytes would disrupt the normal recognition of the jpeg file while mutating the remaining bytes would not necessarily have the same effect. The heat map (the lower part of Fig. 9) effectively captures the correlation between program coverage and byte mutation, revealing that the bytes in the red circles are mostly associated with high heat values (hot bytes). To ensure the validity of the generated inputs, the ATTUzz avoids mutating these bytes. A similar observation can be made for XML syntax, where hot bytes (as indicated in Fig. 11) correspond to words closely tied to the xml syntax. Mutating these bytes would lead to invalid inputs, making them critical for preserving the syntax correctness of the generated inputs.

To compare heat map and dynamic taint analysis, we still use the libjpeg example to highlight the differences between
the two, Angora, as a representative of dynamic taint analysis fuzzer, records its taint analysis results in the cond_queue file, and by analyzing it, we compare the difference between taint analysis and ATTuzz. We visualize the input offset identified by Angora as well at the bottom of Fig. 9. We conducted a thorough analysis of the key bytes identified by Angora in all 13 programs that we tested and compared them with the high-heat bytes in all our merged heat maps. We found that only around 9% of the key bytes identified by Angora overlapped with our high-heat bytes in ATTuzz. This finding demonstrates the difference of our heat maps over Angora in identifying the most critical and relevant bytes in a program.

Not surprisingly, both from the case study and statistics, we find that the “critical” bytes identified by the two techniques are quite different. Since ATTuzz’s high heat refers to bytes that are particularly relevant to achieving a particular coverage, rather than a specific sequence of bytes designed to trigger a particular behavior (e.g., condition comparison). They are complementary in terms of finding critical bytes for different purposes.

**Answer to RQ2:** ATTuzz is able to identify critical blocks when program coverage growth is slow and is able to guide input generation and increases the likelihood of breaking the bottleneck and improving the overall performance of the fuzzer.

**C. RQ3. Are Reward Computation and Attention Guidance Complementary to Each Other?**

In the following, we show that either technique by itself is not impressive using two sets of experiments. First, we keep using reward calculation to guide the 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 the 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 Fig. 12. The figure shows that introducing a reward mechanism to select seeds alone can slightly improve the coverage (triangularly marked curves), as our reward calculation identifies these seeds as more promising for coverage. However, this improvement is limited since the validity of the seed file can be easily compromised due to the random mutation strategy.

On the other hand, if we randomly select some basic blocks as critical blocks, sometimes their effect decreases rather than increases compared to AFL++ (diamond-shaped marked curve). This is because if a randomly selected critical block has a very low reward, generating many inputs against it will provide very little gain, resulting in wasted time in unimportant places. 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 few child nodes).

**Answer to RQ3:** Reward computation and attention guidance are both important for ATTuzz.

**D. RQ4. How the Use of a CNN vs. RNN Affect the Performance of ATTuzz Over Different Input Formats?**

Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) are two fundamental types of neural networks that are commonly used in deep learning for various applications. While both networks can be used for classification tasks, they are designed to handle different types of data. CNNs are commonly used for image and video processing tasks, such as image classification. They are well-suited for these tasks because they can capture spatial and temporal dependencies in the data. RNNs, on the other hand, are commonly used for sequential data processing tasks, such as language translation. They are well-suited for these tasks because they can capture the temporal dependencies in the data. Both CNNs and RNNs have been shown to be effective in various machine learning tasks and have unique advantages in their respective areas of application.

Our approach to utilizing CNN and RNNs in our study followed the conventional use of these models in neural networks. We employed CNN for image-type inputs such as jpeg, png, and tiff files, while RNN was used for text-type inputs like xml and sql files. For file types that were not explicitly categorized, such as zip and bin files, we selected CNNs for feature extraction since these file types often have structured features similar to image files. To validate the soundness of our approach, we conducted an ablation study on a set of the most typical programs, libjpeg, libxml, and mupdf. We compared the results of using CNNs and RNNs for feature extraction in three aspects: (1) the training speed of the neural network, (2) the accuracy of the training results, measured by the proportion of critical blocks covered by the new input generated by the heat map guide, and (3) the overall program coverage results. By carefully analyzing the results of the study, we were able to confirm that our selection of CNNs and RNNs for feature extraction was reasonable and effective. The experimental results are presented in Fig. 13, highlighting the differences between CNNs and RNNs in three aspects. Fig. 13(a) depicts the average...
training time required for three typical software programs using either CNNs or RNNs. Fig. 13(b) displays the proportion of heat maps generated by CNNs or RNNs, respectively, that can guide the newly generated input to cover the critical block. Finally, Fig. 13(c) demonstrates the coverage growth using CNNs or RNNs, respectively. For the typical program libxml with text format as program input, RNNs tend to be slower to train than CNNs due to their sequential nature. However, for a small file like an XML file, the difference in training speed between RNNs and CNNs is negligible. Moreover, as RNNs are naturally suited to learning sequential properties, they are slightly better at guiding newly generated inputs, as reflected in the final coverage growth. Thus, using RNNs for such tasks is preferable. However, for libjpeg, a typical program that uses images as input, and heatmap, the generation quality is slightly inferior, and it takes longer to train, leading to an overall lag in coverage growth. Finally, for mupdf, similar to libjpeg, although the training time of RNN is slightly longer than CNN, the generation quality of the heat map is the same. Thus, using CNNs for such tasks is reasonable.

**Answer to RQ4:** The choice between CNNs and RNNs does not have a significant impact on ATTuzz; overall CNNs are faster to train and have better critical block coverage, but RNNs have a slight advantage in accuracy for programs that use text as input.

**E. RQ5. Does ATTuzz Discover More Bugs?**

The main purpose of fuzzing is to find as many bugs as possible. In this evaluation, we evaluate each fuzzer’s capability of detecting bugs in the software binaries in 24 hours and select the one with the most bug findings among the ten repetitions for recording. We summarize the results in Table IV. Note that we
omit that software that all fuzzers fail to find any bug. For the remaining programs, ATTUZZ is able to find 1.1X, 1.8X, 3.2X, 1.6X, 2.0X, and 1.7X times of bugs than NEUZZ, AFL++, AFLfast, MOpt, QSYM, and Angora respectively, which shows the usefulness of improved coverage by ATTUZZ in discovering more bugs. Besides, ATTUZZ not only finds the maximum number of bugs but also triggers more types of them.

Furthermore, we present a summary of the bug discovery time recorded in the MAGMA data set in Fig. 14, illustrating the mean number of bugs (along with their standard deviation) detected by each fuzzer across ten 24-hour campaigns. Notably, ATTUZZ achieved the highest average number of bugs in five out of the six targeted areas.

Additionally, the inherently stochastic nature of fuzzing often leads to significant variation in the time-to-bug discovery, even across identical trials. To address this issue, we conducted each trial 10 times and to accommodate such variability and missing measurements, we employed survival analysis, specifically adopting Wagner’s method [63], and utilized the Kaplan-Meier estimator [64] to model the bug’s survival function. This function outlines the likelihood of a bug remaining undetected (or “surviving”) within a specified timeframe (here, 24 hours). A smaller survival time indicates superior fuzzer performance, implying that ATTUZZ detected the bug at a faster rate. Notably, out of a total of 25 bugs, ATTUZZ exhibited the shortest survival time for 10 bugs, indicating its efficiency in swiftly identifying these specific bugs as shown in Fig. 15.

In conclusion, the case of openUSD highlights the importance of comprehensive testing archived by ATTUZZ in identifying and mitigating software vulnerabilities.

**Answer to RQ5**: ATTUZZ is effective in real-world bug detection by more useful coverage.

### VI. RELATED WORKS

#### A. Coverage-Guided Greybox Fuzzing

Fuzzing has become one of the most effective methods of vulnerability detection today. Represented by AFL [6], the greybox fuzzing method is widely used in program testing, which quickly generates test cases from seed files and random mutations. Based on this concept, many variants are proposed to better select the seeds and optimize the fuzzing process [4], [5], [12], [13], [14], [15], [16], [67]. ATTUZZ is built on top of the framework and adopts systematic approaches to pay better attention on the bottlenecks during fuzzing.

#### B. Learning Based Fuzzing

Machine learning has been gaining attention in the field of fuzzing, with researchers exploring its potential to improve the process [35], [47], [48], [49], [50], [51], [52], [68], [69]. Various techniques have been proposed, such as RNN fuzzier [38] uses recurrent neural networks (RNNs) to check whether an input can reach the target program block to filter out uninteresting test cases. FuzzGuard [39] adopts a similar idea in the directed fuzzing scenario. Neuzz [40] uses neural networks to smooth the target program and guide the input mutation through gradients. Skyfire’s [48] data-driven seed generation method extracts semantic information through PCFG, Samplefuzz’s [47] neural network-based statistical learning technology for automatic input grammar generation, and GANFuzz’s [70] use of a generative confrontation network to estimate the distribution function of industrial network protocol messages. Essentially, ATTUZZ has a different goal with deep learning from these approaches by using more explainable deep learning models with attention to provide guidance on identifying the right bytes and mutators for mutation. Moreover, ATTUZZ’s learning target block dynamically changes with the program fuzzing state, so that the fuzzing attention can always be focused on the bottleneck of the program.

#### C. Symbolic/Concolic Execution Integrated Fuzzing

One promising approach to improving the efficacy of fuzzing is to integrate fuzzers with concolic/symbolic
execution, known as hybrid fuzzing [17], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33]. This technique leverages the benefits of efficient mutation and precise constraint solving to evaluate programs, potentially representing the future of fuzzing. With the growing importance of fuzzing, many path-exploring demands have been offloaded to fuzzers to avoid path explosion problems in symbolic/concolic execution. At present, state-of-the-art hybrid fuzzing selectively solves path constraints to improve performance. For example, Driller [24] solves uncovered paths for fuzzing, rather than exploring all paths with concolic execution. However, the effective integration of concolic execution and fuzzing remains a matter of ongoing investigation. QSYM [45], for instance, solves part of the path constraint for a basis seed, using mutation for validated inputs satisfying the actual condition. Intriguer [71] replaces symbolic emulation with dynamic taint analysis, reducing the overhead of modeling a large number of instructions. Pangolin [72] abstracts and preserves the constraint, reusing it to guide further input generation. However, Symbolic execution is often limited by the size of the program, i.e., once the execution path is too complex, it is difficult for a solver to generate a valid input that can overcome the program’s bottleneck. ATTUZZ is lightweight compared to these approaches whose main cost is to solve an equation system (in seconds) and model training.

D. Taint-Analysis-Based Fuzzing

Numerous evolutionary fuzzers have incorporated taint information to identify high-potential mutating locations [12], [13], [14], [34], [35], [36], [37]. TaintScope [34], for instance,
focuses on input bytes that impact system or library calls and generates mutations that target those specific bytes. Dowser [35] and BORG [36] both utilize taint information to detect buffer boundary violations and buffer over-read vulnerabilities, respectively. Vuzzer [12] utilizes static analysis to capture magic constants and mutates existing values accordingly. Steelix [13] instruments binaries to obtain additional taint information regarding comparison instructions. Last but not least, Angora [14] employs dynamic taint tracking to pinpoint promising mutation locations and perform coordinate descent to direct mutations to those locations. However, these taint-based approaches suffer from fundamental limitations, such as high overhead from dynamic taint analysis and a high rate of false positives from static taint analysis. ATTUZZ benefits from its lightweight data collection approach and parallel training design, allowing us to identify magic bytes in the input while maintaining a low overhead.

VII. CONCLUSION

In this work, we present ATTUZZ, an efficient fuzzer that pays better attention to 1) the valuable blocks identified using a reward calculation mechanism based on the fuzzing data, and 2) the valuable bytes and mutates using a deep learning network with an 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 input generation to help the fuzzer break the bottlenecks met. We extensively evaluate ATTUZZ on 8 real-world programs and MAGMA data set compared to 5 state-of-the-art fuzzers of 3 different categories and show that ATTUZZ achieves up to over 100% more coverage and detects up to 3X bugs. In addition, ATTUZZ also finds 4 new bugs in the latest version of some popular software including p7zip and openusd. ATTUZZ shows the potential of paying attention with the help of deep learning while fuzzing and extends the possibility of detecting more vulnerabilities by fuzzing with attention.

ACKNOWLEDGMENT

We thank the reviewers for their constructive feedback. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore.

REFERENCES

[1] B. P. Miller et al., “Fuzz revisited: A re-examination of the reliability of unix utilities and services,” University of Wisconsin-Madison Department of Computer Sciences, Tech. Rep., 1995. [Online]. Available: https://minds.wisconsin.edu/bitstream/handle/1793/59964/TR1268.pdf

[2] S. Embleton, S. Sparks, and R. Cunningham, “Sidewinder: An evolutionary guidance system for malicious input crafting,” in Proc. Black Hat USA, 2006, pp. 2–48.

[3] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, “SlowFuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities,” in Proc. ACM SIGSAC Conf. Comput. Commun. Secur., 2017, pp. 2155–2168.

[4] M. Böhme, V. Pham, and A. Roychoudhury, “Coverage-based greybox fuzzing as Markov chain,” IEEE Trans. Softw. Eng., vol. 45, no. 5, pp. 489–506, Mar. 2019.

[5] K. Serebryany, “libFuzzer—A library for coverage-guided fuzz testing,” LLVM. Accessed: Apr. 27, 2023. [Online]. Available: http://llvm.org/docs/LibFuzzer.html

[6] M. Zalewski, “American fuzzy lop.” Accessed: Apr. 27, 2023. [Online]. Available: https://lcamtuf.coredump.cx/afl/

[7] OSS-Fuzz Project, “OSS-Fuzz—Google’s continuous fuzzing service for open source software.” 2016. [Online]. Available: https://github.com/google/oss-fuzz

[8] A. Fioraldi, D. Maier, H. EisFéldt, and M. Heuse, “AFL++: Combining incremental steps of fuzzing research,” in Proc. 14th USENIX Workshop Offensive Technol. (WOOT), 2020.

[9] S. Gan et al., “CollAFL: Path sensitive fuzzing,” in Proc. IEEE Symp. Secur. Privacy (SP), Piscataway, NJ, USA: IEEE Press, 2018, pp. 679–696.

[10] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, “Directed greybox fuzzing,” in Proc. ACM SIGSAC Conf. Comput. Commun. Secur., 2017, pp. 2329–2344.

[11] H. Peng, Y. Shoshitaishvili, and M. Payer, “T-Fuzz: Fuzzing by program transformation,” in Proc. IEEE Symp. Secur. Privacy (SP), Piscataway, NJ, USA: IEEE Press, 2018, pp. 679–710.

[12] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, “VUZer—Application-aware evolutionary fuzzing,” in Proc. NDSS, 2017, vol. 17, pp. 1–14.

[13] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, “Steelix: Program-state based binary fuzzing,” in Proc. 11th Joint Meeting Found. Softw. Eng., 2017, pp. 627–637.

[14] P. Chen and H. Chen, “Angora: Efficient fuzzing by principled search,” in Proc. IEEE Symp. Secur. Privacy (SP), Piscataway, NJ, USA: IEEE Press, 2018, pp. 711–725.

[15] V. J. Manes, S. Kim, and S. K. Cha, “Ankou: Guiding grey-box fuzzing towards combinatorial difference,” in Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng., 2020, pp. 1024–1036.

[16] V.-T. Pham, M. Böhme, and A. Roychoudhury, “AFLNet: A greybox fuzzer for network protocols,” in Proc. IEEE 13th Int. Conf. Softw. Testing, Validation Verification (ICST), Piscataway, NJ, USA: IEEE Press, 2020, pp. 460–465.

[17] S. K. Cha, M. Woo, and D. Brumley, “Program-adaptive mutational fuzzing,” in Proc. IEEE Symp. Secur. Privacy, Piscataway, NJ, USA: IEEE Press, 2015, pp. 725–741.

[18] C. Lyu et al., “MOTP: Optimized mutation scheduling for fuzzers,” in Proc. 28th USENIX Symp. (USENIX Secur.), 2019, pp. 1949–1966.

[19] M. Böhme, V. J. Manes, and S. K. Cha, “Boosting fuzzer efficiency: An information theoretic perspective,” in Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng., 2020, pp. 678–689.

[20] C. Lemieux and K. Sen, “FairFuzz: A targeted mutation strategy for increasing greybox fuzzing test coverage,” in Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng., 2018, pp. 475–485.

[21] Y. Li et al., “Cerebro: Context-aware adaptive fuzzing for effective vulnerability detection,” in Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng., 2019, pp. 533–544.

[22] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, “REDQUEEN: Fuzzing with input-to-state correspondence,” in Proc. NDSS, 2019, vol. 19, pp. 1–15.

[23] C. Cadar et al., “KLEE: Unassisted and automatic generation of high-quality tests for complex systems programs,” in Proc. USENIX Secur., 2008, vol. 16, no. 2016, pp. 1–16.

[24] N. Stephens et al., “Driller: Augmenting fuzzing through selective symbolic execution,” in Proc. NDSS, 2016, vol. 16, no. 2016, pp. 1–16.

[25] J. C. King, “Symbolic execution and program testing,” Commun. ACM, vol. 19, no. 7, pp. 385–394, 1976.

[26] K. Sen, D. Marinov, and C. Agha, “CUTE: A concordic unit testing engine for C,” ACM SIGSOFT Softw. Eng. Notes, vol. 30, no. 5, pp. 263–272, 2005.

[27] D. Molnar, X. C. Li, and D. A. Wagner, “Dynamic test generation to find integer bugs in x86 binary Linux programs,” in Proc. USENIX Secur. Symp., 2009, vol. 9, pp. 67–82.

[28] P. Godfrey et al., “Automated whitebox fuzz testing,” in Proc. NDSS, 2008, vol. 8, pp. 151–166.

[29] C. Cadar and K. Sen, “Symbolic execution for software testing: Three decades later,” Commun. ACM, vol. 56, no. 2, pp. 82–90, 2013.

[30] C. Cadar et al., “Symbolic execution for software testing in practice: Preliminary assessment,” in Proc. 33rd Int. Conf. Softw. Eng. (ICSE), Piscataway, NJ, USA: IEEE Press, 2011, pp. 1066–1071.
X. Wang, J. Sun, Z. Chen, P. Zhang, J. Wang, and Y. Lin, “Towards N. Nichols, M. Raugas, R. Jasper, and N. Hilliard, “Faster fuzzing: R. H. Byrd, P. Lu, J. Nocedal, and C. Zhu, “A limited memory algorithm O. Bastani, R. Sharma, A. Aiken, and P. Liang, “Synthesizing program input grammars,” ACM SIGPLAN Notices, vol. 52, no. 6, pp. 95–110, 2017.

S. Sivakorn, G. Argyros, K. Pei, A. D. Keromytis, and S. Jana, “HVLearn: Automated black-box analysis of hostname verification in SSL/TLS implementations,” in Proc. IEEE Symp. Secur. Privacy (SP), Piscataway, NJ, USA: IEEE Press, 2017, pp. 521–530.

R. L. How, P. Lu, I. Pedras, and C. Zhu, “A limited memory algorithm for bound constrained optimization,” SIAM J. Sci. Comput., vol. 16, no. 5, pp. 1190–1208, 1995.

N. Nichols, M. Rauagas, R. Jasper, and N. Hilliard, “Faster fuzzing: Reinitialization with deep neural models,” 2017, arXiv:1711.02807.

X. Wang, J. Sun, Z. Chen, P. Zhang, J. Wang, and Y. Lin, “Towards optimal concolic testing,” in Proc. 40th Int. Conf. Softw. Eng., 2018, pp. 291–302.

J. Wang, X. Chen, J. Sun, and S. Qin, “Improving probability estimation through active probabilistic model learning,” in Proc. Int. Conf. Formal Eng. Methods, New York, NY, USA: Springer-Verlag, 2017, pp. 379–395.

J. Yang et al., “Development and validation of a deep learning model for detection of allergic reactions using safety event reports across hospitals,” JAMA Netw. Open, vol. 3, no. 11, pp. e2022836–e2022836, 2020.

K. Da, “A method for stochastic optimization,” 2014, arXiv:1412.6980.

R. Natella and V.-T. Pham, “ProFuzzBench: A benchmark for stateful protocol fuzzing,” in Proc. 30th ACM SIGSOFT Int. Symp. Softw. Testing Toolf., 2021, pp. 662–665.

A. Herrera, H. Gunadi, S. Magrath, M. Norris, M. Payer, and A. L. Hosking, “Seed selection for successful fuzzing,” in Proc. 30th ACM SIGSOFT Int. Symp. Softw. Testing Anal., 2021, pp. 230–243.

J. Metzmann, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, “FuzzBench: An open fuzzier benchmarking platform and service,” in Proc. 29th ACM Joint Meeting Eat. Softw. Eng. Conf. Symp. Found. Softw. Eng., 2021, pp. 1393–1403.

A. Hazimeh, A. Herrera, and M. Payer, “Magma: A ground-truth fuzzing benchmark,” Proc. ACM Meas. Anal. Comput. Syst., vol. 4, no. 3, pp. 1–29, 2020.

G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, “Evaluating fuzz testing,” in Proc. ACM SIGSAC Conf. Comput. Secur. Commun., 2018, pp. 2123–2138.

C. Andrieu, N. De Freitas, A. Doucet, and M. I. Jordan, “An introduction to MCMC for machine learning,” Mach. Learn., vol. 50, no. 1, pp. 4–33, 2003.

J. B. Wagner, “Elastic program transformations: Automatically optimizing the reliability/performance trade-off in systems software,” Ph.D. dissertation, École Polytechnique Fédérale de Lausanne, 2017.

E. L. Kaplan and P. Meier, “Nonparametric estimation from incomplete observations,” J. Am. Stat. Assoc., vol. 53, no. 282, pp. 457–481, 1958.

jinfenian57, “A new pzip fork with additional codec and improvements,” PixarAnimationStudios, “Universal scene description”.

T. Yue et al., “EcoFuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit,” in Proc. 29th USENIX Secur. Symp. (USENIX Secur.), 2020, pp. 2307–2324.

K. Böttinger, P. Godefroid, and R. Singh, “Deep reinforcement fuzzing,” in Proc. IEEE Sysec. Priv. Workshops (SPW), Piscataway, NJ, USA: IEEE Press, 2018, pp. 116–122.

X. Xie et al., “DeepHunter: A coverage-guided fuzz testing framework for deep neural networks,” in Proc. 28th ACM SIGSOFT Int. Symp. Softw. Testing Anal., 2019, pp. 146–157.

Z. Hu, J. Shi, Y. Huang, J. Xiong, and X. Bu, “GANFuzz: A GAN-based industrial network protocol fuzzing framework,” in Proc. 15th ACM Int. Conf. Comput. Frontiers, 2018, pp. 138–145.

M. Cho, S. Kim, and T. Kwon, “Intriguer: Field-level constraint solving for hybrid fuzzing,” in Proc. ACM SIGSAC Conf. Comput. Commun. Secur., 2019, pp. 515–530.

H. Huang, P. Yao, R. Wu, Q. Shi, and C. Zhang, “Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction,” in Proc. IEEE Sysec. Priv. (SP), Piscataway, NJ, USA: IEEE Press, 2020, pp. 1613–1627.

Shunkai Zhu received the B.Eng. degree in cyber security from Xidian University, in 2019. He is currently pursuing the Ph.D. degree with the State Key Laboratory of Industrial Control Technology, Group of Networked Sensing and Control, Zhejiang University. His research interests include software testing and control system security.

Jingyi Wang (Member, IEEE) received the bache- lor’s degree in information engineering from Xi’an Jiaotong University, in 2013, and the Ph.D. degree from Singapore University of Technology and Design, in 2018. He is currently a Tenure-Track Assistant Professor with the College of Control Science and Engineering, Zhejiang University, China. He was a Research Fellow with the School of Computing, National University of Singapore, during 2019–2020 and with Information Systems Technology and Design Pillar, Singapore University of Technology and Design, during 2018–2019. His research interests include formal methods, software engineering, cyber security, and machine learning.
Jun Sun received the bachelor’s and Ph.D. degrees in computing science from the National University of Singapore (NUS), in 2002 and 2006, respectively. He is currently a Tenured Professor with the School of Information Systems, Singapore Management University. From 2010 to 2019, he was an Assistant/Associate Professor with the Singapore University of Technology and Design. He was a Visiting Scholar with MIT, from 2011 to 2012. His research focuses on software engineering, formal methods, program analysis, and cybersecurity. He is the Co-Founder of the PAT model checker.

Jie Yang is an Assistant Professor with Zhejiang University. His research interests include natural language processing, deep learning, and artificial intelligence in healthcare and social science. He won the best paper award at COLING 2018, the best demonstration paper nomination at ACL 2018, and ISTD best Ph.D. dissertation award. He has served as an Area Chair at several NLP conferences including EMNLP, COLING, and CCL.

Xingwei Lin is a Senior Security Engineer with Ant Group, specializing in program analysis, fuzzing techniques, vulnerability hunting, and web3 security. He has contributed several research papers to top-tier academic and industry conferences, including USENIX security, ACM CCS, ASE, and BlackHat Asia. He received the Best Paper Award at ACM CCS in 2021. He was nominated for the “Most Innovative Research” category at the Pwnie Awards in two consecutive years, 2021 and 2022. His research has garnered acknowledgment from leading vendors, such as Apple, Microsoft, Qemu, and Oracle.

Tianyi Wang received the B.S. degree in automation from Zhengzhou University, China, in 2020. He is currently pursuing the Ph.D. degree with the State Key Laboratory of Industrial Control Technology, Group of Networked Sensing and Control, Zhejiang University. His research interests include control system security and fuzzing.

Liyi Zhang received the bachelor’s degree in computer science degree from the University of Waterloo, in 2020. He is currently pursuing the master’s degree in mathematics with David R. Cheriton School of Computer Science, University of Waterloo, Canada. He was an Intern Research Assistant with Zhejiang University, during 2020–2021. His research interests include system security and software engineering.

Peng Cheng, photograph and biography not available at the time of publication.