IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 15, 2019, accepted February 6, 2019, date of current version April 2, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2903291

NeuFuzz: Efficient Fuzzing With Deep Neural Network

YUNCHAO WANG, ZEHUI WU, QIANG WEI, AND QINGXIAN WANG
China National Digital Switching System Engineering and Technological Research Center, Zhengzhou 450000, China
Corresponding author: Qiang Wei (prof_weiqiang@163.com)

This work was supported by National Key R&D Program of China under Grant 2017YFB0802901.

ABSTRACT Coverage-guided graybox fuzzing is one of the most popular and effective techniques for discovering vulnerabilities due to its nature of high speed and scalability. However, the existing techniques generally focus on code coverage but not on vulnerable code. These techniques aim to cover as many paths as possible rather than to explore paths that are more likely to be vulnerable. When selecting the seeds to test, the existing fuzzers usually treat all seed inputs equally, ignoring the fact that paths exercised by different seed inputs are not equally vulnerable. This results in wasting time testing uninteresting paths rather than vulnerable paths, thus reducing the efficiency of vulnerability detection. In this paper, we present a solution, NeuFuzz, using the deep neural network to guide intelligent seed selection during graybox fuzzing to alleviate the aforementioned limitation. In particular, the deep neural network is used to learn the hidden vulnerability pattern from a large number of vulnerable and clean program paths to train a prediction model to classify whether paths are vulnerable. The fuzzer then prioritizes seed inputs that are capable of covering the likely to be vulnerable paths and assigns more mutation energy (i.e., the number of inputs to be generated) to these seeds. We implemented a prototype of NeuFuzz based on an existing fuzzer PTfuzz and evaluated it on two different test suites: LAVA-M and nine real-world applications. The experimental results showed that NeuFuzz can find more vulnerabilities than the existing fuzzers in less time. We have found 28 new security bugs in these applications, 21 of which have been assigned as CVE IDs.

INDEX TERMS Fuzzing, vulnerability detection, deep neural network, seed selection, software security.

I. INTRODUCTION

Security vulnerabilities hidden in programs can lead to system compromise, information leakage or denial of service. As we have seen from many recent high-profile exploits, such as Heartbleed attacks [1] and WannaCry ransomware [2], these vulnerabilities can often cause disastrous effects, both financially and societally. Therefore, security vulnerabilities in programs must be identified before being found by attackers and eliminated to avoid potential attacks. Since its introduction in the early 1990s [3], fuzzing has become one of the most effective and scalable testing techniques to find vulnerabilities, bugs or crashes in commercial software. It has also been widely used by mainstream software companies, such as Google [4] and Microsoft [5], to ensure the quality of their software products.

The key idea of fuzzing is to feed the program under test (PUT) with a large amount of crafted inputs to triglger unintended program behavior, such as crashes or hangs. Based on the understanding of the structural knowledge inside the PUT, fuzzers can be classified as whitebox, blackbox and graybox. A whitebox fuzzer (e.g., [6], [7]) usually has access to the source code or intermediate representation of the PUT. They usually use heavy weight program analysis methods, such as symbolic execution together with path traversal, to guide fuzzing and thus have scalability issues. A blackbox fuzzer (e.g., [8], [9]) is completely unaware of the internal structure of the program and usually performs random testing blindly and is thus extremely inefficient. A graybox fuzzer (e.g., [10]–[12]) aims for a compromise, employing lightweight program analysis methods (e.g., instrumentation) to obtain feedback from the program to guide fuzzing, which is generally more effective than a blackbox fuzzer and more scalable than a whitebox fuzzer.

In recent years, coverage-guided graybox fuzzing (CGF), which uses code coverage as feedback to guide fuzzing, has become one of the most successful vulnerability discovery solutions. This type of fuzzer gradually increases the code

The associate editor coordinating the review of this manuscript and approving it for publication was Hongbin Chen.

coverage and amplifies the probability of finding vulnerabilities. AFL [10], libFuzzer [11] and honggfuzz [12], state-of-the-art graybox fuzzers, have drawn attention from both industry and academia and have discovered hundreds of high-profile vulnerabilities [13]. In general, these methods instrument the PUT with lightweight analysis to extract coverage information for each executed test case. If a test case exercises a new branch, the case is marked as interesting; otherwise, the case is marked useless. In the following fuzzing iterations, interesting test cases are reused as seeds to mutate and generate new test cases for fuzz testing.

Despite its considerable success in vulnerability discovery, we figured out that CGF fuzzers (e.g., AFL) have a critical limitation. The current fuzzers aim to cover as many paths as possible rather than to explore paths that are more likely to be vulnerable. They select seeds from the seed queue in the order they are added, and the testing energy is spent equally on all program paths. However, different paths have different probabilities of being vulnerable. As reported in [14], the bug distribution in programs is often unbalanced, i.e., approximately 80% of bugs are located in approximately 20% of program code. As a result, existing CGF fuzzers waste considerable time testing uninteresting (i.e., not vulnerable) paths, thereby reducing the efficiency of fuzzing. Intuitively, test cases that exercise more bug-prone program paths are more likely to trigger bugs, so we should spend most of the fuzzing effort on these paths to increase the probability of triggering vulnerability during testing.

In this paper, we propose a novel path-sensitive graybox fuzzing solution, NeuFuzz, to alleviate the aforementioned limitation. It applies a new seed selection strategy that prioritizes seeds that are more likely to exercise vulnerable paths and accordingly assigns higher testing energy (e.g., the number of inputs to be generated) to these seeds to improve the efficiency of vulnerability detection. The core challenge is how to determine whether a path is likely to be vulnerable. Inspired by the substantial success in image and speech recognition, we use a deep neural network to learn the hidden pattern of vulnerable program paths and to identify vulnerable paths to guide seed selection. Specifically, a prediction model is learned from a large number of vulnerable and clean program paths and is then used to classify whether a vulnerability exists in the exercised path during fuzzing. The fuzzer therefore prioritizes seeds that exercise vulnerable paths, which are determined by the neural network, and assigns more mutation energy to these seeds to maximize the bug discovery efficiency.

NeuFuzz's intelligent seed selection based on deep neural networks is more efficient than that of state-of-the-art fuzzers. We implemented a prototype of NeuFuzz based on a specific extension of AFL, i.e., PTfuzz [15]. We evaluated NeuFuzz on two different test suites. The first is the LAVA-M [16] benchmark, which contains four Linux programs with manually injected vulnerabilities; the other is a set of real-world applications (i.e., libtiff, binutils, libav, podofo, bento4, libsndfile, audilfile, nasm) with different complexity and functionalities, which are often used as third-party libraries by some widely used programs. The experimental results indicate that NeuFuzz can find more vulnerabilities than PTfuzz and QAFL [17] in less time.

In summary, we make the following contributions:

- We propose a new seed selection strategy that prioritizes seeds exercising vulnerable paths to improve the efficiency of vulnerability discovery.
- We propose a novel deep neural network solution to predict which program paths are vulnerable.
- We implement the NeuFuzz prototype to fuzz binary programs and evaluate it with crafted benchmark and real-world applications, showing that this solution is effective.
- We find 21 CVEs and 7 unknown security bugs in some widely used real-world applications and help to improve the security of the vendors' products.

The rest part of this paper is organized as follows. Section 2 presents the background and related work. Section 3 describes the overview of NeuFuzz. Section 4 details the technical details of NeuFuzz. Section 5 describes the implementation and experimental evaluation results of NeuFuzz. Section 6 concludes the paper.

II. BACKGROUND AND RELATED WORK

Recent studies have implemented many boosting techniques, such as program analysis and machine learning, to improve the efficiency and effectiveness of CGF from different aspects, including coverage tracking, seed selection, and seed mutation.

A. FUZZING WITH HARDWARE FEEDBACK

PTfuzz is a graybox fuzzer with hardware feedback and is also based on AFL. It uses the CPU hardware mechanism Intel PT (Intel Process Tracer) [18] to collect branch information instead of compile-time instrumentation (e.g., the one used by AFL's gcc mode) or runtime instrumentation (e.g., the one used by AFL's QEMU mode, denoted as QAFL). Intel PT is an extension of Intel Architecture that is capable of accurately tracing program control flow information with minimal performance overhead. Therefore, the fuzzer does not rely on the source code and achieves higher performance compared to QAFL without loss of coverage tracking accuracy. Another recent study KAFL [19] is also implemented based on AFL and Intel PT, which can support kernel fuzzing in ring0. Our prototype is implemented based on PTfuzz, because our test target is binary programs in ring3.

B. SEED SELECTION STRATEGY

The seed selection strategy is critical for graybox fuzzing. A good seed selection strategy can improve the ability of path traversal and vulnerability detection. AFL takes a simple seed selection strategy, i.e., preferring smaller and faster seeds, to generate and test more test cases in a given amount of time. Rawat *et al.* [20] prioritize seeds that exercise deeper paths, deprioritizes seeds exercising error-handling blocks and

high-frequency paths, and thus it is likely that hard-to-reach paths could be tested and useless error-handling paths will be avoided. AFLFast [21] prioritizes seeds exercising low-frequency paths and being selected fewer, and thus it is likely that cold paths could be tested thoroughly and fewer energy will be wasted on hot paths. AFLGo [22] prioritizes seeds that are closer to the predetermined target location, thus achieving the purpose of directed fuzzing. CollAFL [23] uses three new seed selection policies to drive the fuzzer directly towards non-explored paths and improve code coverage more quickly. Angora [24] prioritizes seeds whose paths contain conditional statements with unexplored branches, which enables focus on low-frequency paths after exploring high-frequency paths.

The existing seed selection strategies focus mainly on execution speed, path frequency, path depth and path branches that are not traversed. In essence, they focus on code coverage rather than guiding the fuzzer to test bug-prone paths or code. As a result, the current seed selection strategies waste considerable time testing uninteresting paths, which reduces the efficiency of fuzzing. We focus on solving this problem in this paper.

C. SEED MUTATION STRATEGY

AFL sees the input as a sequence of consecutive binary bytes because the structure of the input cannot be perceived. This random mutation strategy makes it possible to find new paths or branches entirely by luck. To this end, researchers have proposed many improvements based on a variety of program analysis methods. Stephens *et al.* [25] combines the strengths of fuzzing and symbolic execution to detect vulnerabilities. Specifically, when the fuzzer is stuck in some complex comparison branches (e.g., magic bytes), concolic execution is used to generate test input that can drive the execution past the check, but it is limited to the poor scalability of symbol execution, thus, the fuzzer is difficult to apply to real-world applications. Li *et al.* [26] uses a lightweight runtime instrumentation method to monitor comparison instructions or functions and guide the mutation according to the progress feedback information to generate input bytes that can pass the branch. T-fuzz [27] removes some complex branch checks (e.g., magic bytes, checksum, and hash) by directly modifying the binary program. When a crash is found, the symbol execution technique is used to remove the false positives and reproduce the bug in the original program. Rawat *et al.* [20] uses static analysis to infer interesting values and then uses taint analysis to determine the input offset for compare instructions so that these offsets are mutated by inferred values to pass some complex conditions. Chen and Chen [24] uses the gradient descent algorithm instead of symbolic execution to guide the mutation to achieve coverage improvement.

We can see that all these methods are designed to improve the code coverage by breaking through some of the complex condition checks in the program. These methods are orthogonal to the methods presented in our paper, and we can integrate them into our tool.

D. LEARNING BASED VULNERABILITY DETECTION

Recently, researchers have begun to apply artificial intelligence technology to aid vulnerability detection. Learn&Fuzz [28] uses neural networks to learn the generation model of the file input format for grammar-based fuzzing. Augmented-AFL [29] uses neural networks to learn a function from past fuzzing to predict a good position in a seed file for performing mutations. Böttinger *et al.* [30] used state-of-the-art Q-learning algorithms to optimize rewards and learn to select high-reward mutation operations for specific program inputs. Li *et al.* [31] proposed VulDeePecker, a vulnerability detection method based on deep learning, which eliminates the need to manually define vulnerability features, but the method requires source code and can detect vulnerabilities related only to library functions. VDiscover [32] extracts the dynamic and static API sequence of a program as a feature and uses a traditional machine learning algorithm to train the model to predict vulnerabilities in unknown programs. However, because the API sequence does not represent the vulnerability feature well, the accuracy is not high.

In summary, on the one hand, the current machine learning methods for fuzzing are used mainly to guide seed generation and seed mutation to achieve high code coverage. On the other hand, machine learning is directly applied to vulnerability prediction, which is essentially a static analysis method, and manual verification is required. Our work uses the vulnerability prediction results to guide the seed selection of the fuzzing process. To the best of our knowledge, we are the first to use a deep neural network to learn interesting paths covered by seed inputs during the fuzzing process.

III. OVERVIEW

In this section, we further explain the problems we need to solve through a motivating example and describe an overview of our approach.

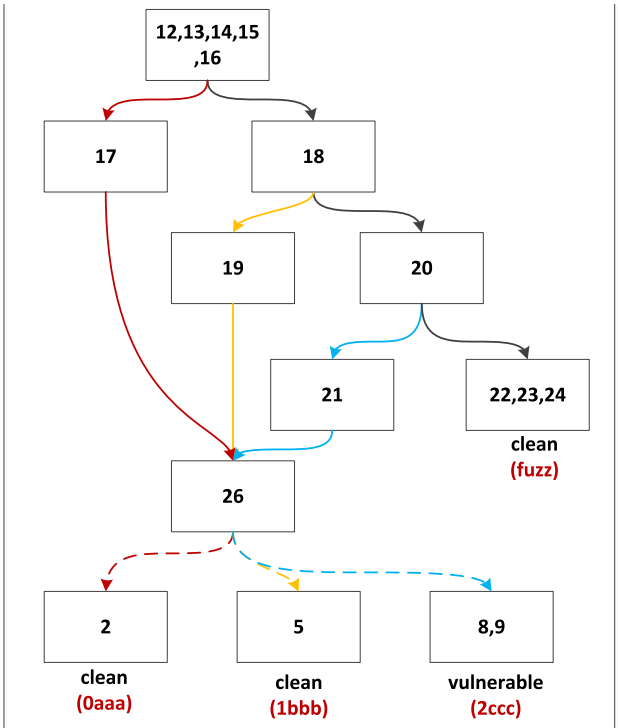
A. MOTIVATING EXAMPLE

As mentioned above, the seed selection strategy of current fuzzers greatly reduces the efficiency of vulnerability detection. To clearly illustrate how the seed selection strategy makes it difficult for AFL to find vulnerabilities in the PUT, consider the code in Fig. 1. Fig. 1 (a) is the source code, and Fig. 1 (b) is the corresponding inter-process control flow graph. An input string longer than 32 characters starting with character '2' will cause a stack overflow at the 9th line of code. Note that this example is relatively simple, and for the sake of convenience, we show it in the source code. In fact, our method can be applied directly to binary programs.

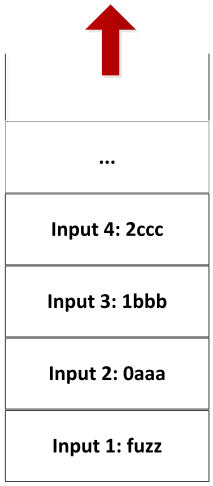
Given the initial seed input "fuzz", AFL can quickly find the other three paths as shown in the control flow graph of Fig. 1(b), assuming that test cases that exercise these paths are "0aaa", "1bbb" and "2ccc". Then AFL will perform mutation on each test case in the order they are added as shown in Fig. 1(c). However, the three paths (denoted by the red, yellow and black lines, respectively) that are exercised by the

```
1  void foo(char *str){
2      printf("foo:%s\n", str);
3  }
4  void bar(char *str){
5      printf("bar:%s\n", str);
6  }
7  void overflow(char *str){
8      char buf[32];
9      strcpy(buf, str);  /*crash*/
10 }
11 void main(int argc, char **argv){
12     void (*fptr) (char *);
13     char buf[100];
14     char choice = buf[0];
15     switch (choice) {
16     case '0':
17         fptr = foo; break;
18     case '1':
19         fptr = bar; break;
20     case '2':
21         fptr = overflow; break;
22     default:
23         printf("wrong choice\n");
24         return;
25     }
26     fptr(&buf[1]);
27 }
```

(a)



(b)



(c)

FIGURE 1. A motivating example. (a) Source code. (b) Control flow graph. (c) fuzzer seed queue.

seed inputs 0aaa, 1bbb and 3ddd are clearly unlikely to have vulnerabilities; thus, the importance of fuzzing them is very low, and the path (denoted by the blue line) exercised by seed input 2ccc is vulnerable and more meaningful to fuzz. And also in order to trigger the potential vulnerability in the path, the seed should be fuzzed more times than others. In fact, real-world programs are more complicated than this example and generally contain more sanity checks and path branches. Assuming that a program executes n ($n > 1000$) paths and that only one path has a vulnerability, the advantage of our method is clear: we can test the vulnerable path first to expose the vulnerability early on rather than wasting considerable time testing the other n-1 paths.

Our idea is to use a seed priority strategy to guarantee that better seeds will be tested first and mutated with more mutation energy, thus improving the efficiency and effectiveness of the fuzzer. Therefore, we need to consider and address the following three heuristic problems.

1: How to infer whether a seed input is better than others?

The importance of a seed input depends on the degree of interest in the path it exercises. For example, AFLFast believes that seeds that trigger low frequency path are better from the perspective of code coverage. However, in this paper if its execution path is more interesting (i.e., the path is more likely to be vulnerable), then we think this input is better and more meaningful to be tested first. Clearly, focusing on testing seeds that execute likely to be vulnerable paths will increase the probability of triggering vulnerabilities. For example, in the example shown in Fig. 1, the path covered by seed 2ccc is vulnerable, so this seed should be fuzzed first and tested more times. Notably, given one seed exercising a vulnerable path, there is no guarantee that the hidden vulnerability in this path will be definitely triggered by mutations from this seed. However, mutating from this seed will certainly increase the probability of triggering the hidden vulnerability.

2: How to determine a path is of interest, that is, more likely to be vulnerable?

We know that a buffer overflow vulnerability exists in the blue path (2ccc) in the example, which calls the dangerous strcpy function without any security check. The traditional method of automatically discovering this vulnerability needs to rely on feature database. However, in practice, the vulnerability condition is more complicated than we can model directly. Extracting features to characterize vulnerabilities is challenging; therefore, we do not rely on traditional vulnerability features based on expert experience to determine whether a path has a vulnerability. Instead, we rely on a deep neural network to automatically extract the features of vulnerabilities, learn vulnerability patterns from a large amount of training data, and identify other vulnerable paths that are of interest.

3: How can we apply our method to the existing binary fuzzer without losing the performance advantage?

We can guide the seed selection of graybox fuzzing based on the learned neural network model. Because the appeal

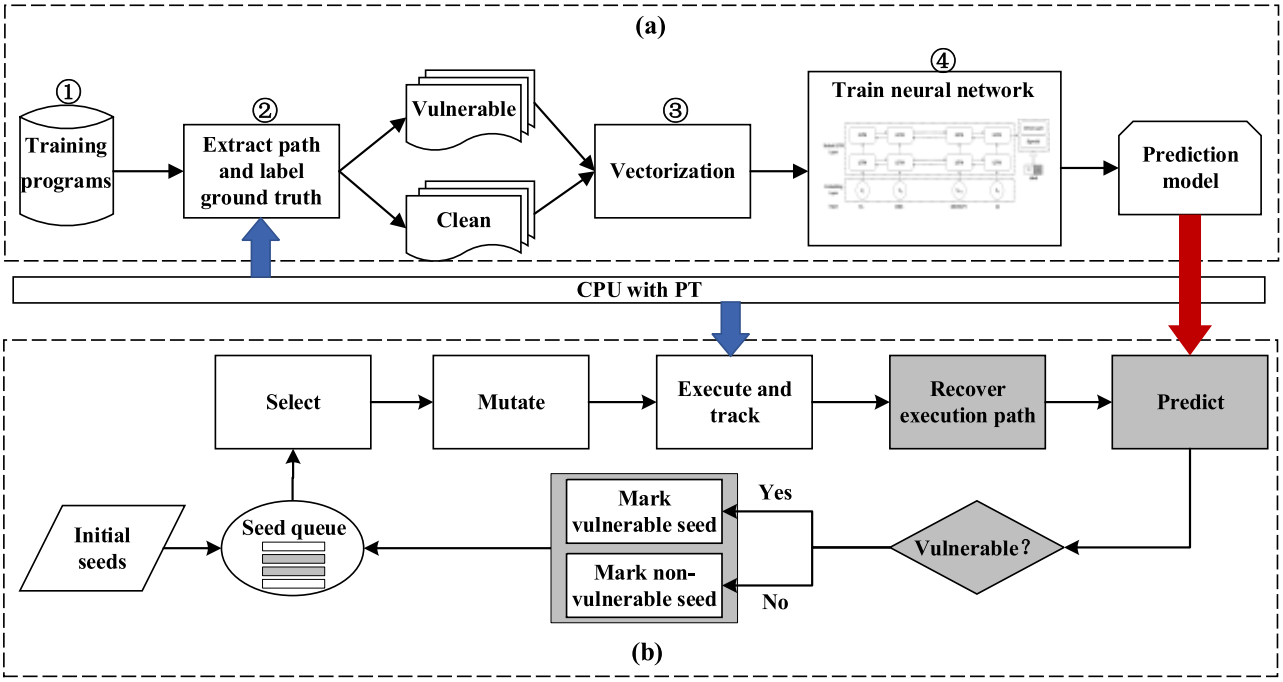


FIGURE 2. Overview of the proposed NeuFuzz approach. (a) offline model training. (b) online guided fuzzing.

of CGF is its speed, the introduction of performance overhead to the fuzzer should be limited when implementing this strategy. Therefore, we have considered several existing fuzzers that support binary fuzzing, including AFL-PIN [33], Vuzzer and PTfuzz, but the implementation of the former two tools relies on the dynamic instrumentation Pin [34] which could incur substantial cost, thus violating our principle. As mentioned above, PTfuzz has been proven to have a large performance improvement compared to QAFL, and it relies on the Intel PT to support execution control flow tracing, which helps to recover the program execution path as input to the model without relying on expensive methods, such as dynamic instrumentation. Therefore, we choose to implement our approach based on PTfuzz.

B. APPROACH OVERVIEW

To address the limitation of the existing graybox fuzzers mentioned above, we propose NeuFuzz, as shown in Fig. 2, which consists of two phases, namely, offline model training and online guided fuzzing, corresponding to Fig. 2(a) and Fig. 2(b), respectively. Both phases require hardware support from a CPU with Intel PT. In the offline model training phase, the input is a large number of binary programs, including vulnerable and non-vulnerable programs. A vulnerable program means that it contains one or more known vulnerabilities, and a non-vulnerable program means that a patch has been applied to fix the vulnerability. The output is a learned prediction model used to classify whether vulnerabilities exist in unseen program paths. In the online guided fuzzing phase, we integrate the prediction model into a graybox fuzzer to determine the priority and mutation energy of the seed by vulnerability prediction of its exercised path before adding the seed to the seed queue. Then, the fuzzer selects seed inputs from the seed queue according to their priority. This process makes seed selection smarter.

Two points should be noted. First, we do not discard seeds that do not cover the vulnerable paths identified by our model. We still add these seeds to the seed queue for two reasons: on one hand, these seeds are still valuable from the perspective of the code coverage of the fuzzer because fuzzing them can discover new paths and on the hand our model prediction accuracy is not guaranteed to be 100%, so we cannot miss the opportunity to find these vulnerabilities, even if the probability is much lower. Second, although training the neural network requires time, we need to train the network only once before applying it to all applications. In addition, we add path recovery and vulnerability prediction in the main fuzzing loop, which of course can lead to extra overhead, but our experiment indicated that the overhead of these two modules is very low and acceptable.

IV. APPROACH

In this section, we describe the key technical details of NeuFuzz, which is based on PTfuzz; our approach is orthogonal to the method of improving code coverage proposed by previous researchers.

A. OFFLINE MODEL TRAINING

As highlighted in Fig. 2(a), this phase has four steps. First, we collect training programs, including the vulnerable

version and the non-vulnerable version, for neural model training. Second, proofs of concept (POCs) are used as the inputs of these programs. With the ability of control flow tracing of Intel PT, we can obtain two types of program paths: vulnerable and clean. Third, these paths are transformed into vector representations as input to the neural network before training. Fourth, a prediction model for guiding seed selection in the online guided fuzzing stage is produced based on the feature learning ability of the neural network from a large amount of training data.

1) COLLECTING TRAINING PROGRAMS

Given the complexity and variety of programs, a large number of training examples are required to train machine learning models to effectively learn the patterns of security vulnerabilities directly from code. In addition, because we need to dynamically run the program to obtain the execution path, we require a binary program that can be executed and the corresponding test case. However, to the best of our knowledge, no publicly available complete binary program dataset exists. Although VulDeePecker provides a dataset [35], only source code is included, and no test cases are provided for program execution.

We construct the first binary dataset derived from three data sources: the NIST SARD project [36], GitHub [37] and Exploit-DB [38]. The NIST SARD project contains a number of synthetic programs, each of which has one good (post-patch) and bad (pre-patch) program and covers various type of CWEs (Common Weakness Enumeration), we choose memory corruptions vulnerabilities such as stack overflow (CWE121), heap overflow (CWE122), integer overflow (CWE190), UAF (CWE416) and so on. We eventually collect 26,080 binary programs from SARD, each with a pre-patch and post-patch version. However, the programs in SARD are synthetic, so their vulnerabilities may differ from the vulnerabilities in real-world applications, which can limit the scalability of our model. Therefore, we also collect real-world applications (such as ImageMagick, libtiff, and bintuils) from GitHub and Exploit-DB. GitHub can track changes to source files, and we can identify the pre-patch and post-patch version for each bug by retrieving the commit history of the master branch of the application and crawl POCs from the public repositories, such as bugtracker and GitHub issues, to verify and retain the test case that can trigger the bug. We ended up collecting 560 applications from Github. In addition, we collect 1039 vulnerable programs and corresponding POCs from Exploit-DB.

As shown in Table 1, we collect a total of 28,475 vulnerable binary programs and 27,436 non-vulnerable binary programs in our dataset for neural network training and testing, as well as test cases that can trigger vulnerable paths and clean paths.

TABLE 1. Dataset for training and testing.

Data source	Vulnerable	Non-vulnerable
SARD	26080	26080
GitHub	560	560
Exploit-DB	1039	0
Total	**28475**	**27436**

2) EXTRACTING EXECUTION PATH AND LABELING GROUND TRUTH

With the constructed dataset and POCs, we can extract the program execution paths by executing the programs in both phases of our approach for training and predicting. The most common method is using Pin dynamic instrumentation to extract the execution path, but this approach can result in high overhead and reduce the execution speed of the fuzzer, especially during the online guided fuzzing phase. Therefore, we use Intel PT technology to extract paths, which will be explained in detail in the first part of Section B. Each instruction of the program execution path is recorded in bytecode form, and we do not trace the instructions in the system library function and retain the library function name (e.g., strcpy and memcpy) related to the vulnerability feature in the process of path tracking, which reduces the number of recorded instructions in the path and at the same time keeps the vulnerability semantics as much as possible.

Next, we label the ground truth. VulDeePecker conservatively believes that code is non-vulnerable as long as no vulnerability has been found in them; in reality, this assumption does not hold. We assume that the code is non-vulnerable after patching; thus, in comparison the ground truth is more accurate. Vulnerable and non-vulnerable programs can be executed with POCs as input to obtain vulnerable paths (labeled as "1") and clean paths (labeled as "0"). Finally, a total of 27,820 vulnerable paths and 26,871 clean paths are extracted from our constructed dataset, and 4/5 of them are used for training, the remaining 1/5 are used for testing.

3) TRANSFORMING EXECUTION PATHS INTO VECTOR REPRESENTATIONS

Before taking the program execution path consisting of the instruction bytecode as input to the deep neural network, the path needs to be converted into a vector representation while retaining as much of the original semantic information of the execution path as possible. We can learn via a text processing method: one path can be seen as one sentence, and each instruction in the path can be regarded as a word in the sentence. Therefore, we must perform word embedding, for which one-hot [39] and word2vec [40] are common methods. One-hot represents the ith instruction as a vector with its ith element set to 1 and the other elements set to 0. For example, if there are 5 instructions, the vector of the second instruction is represented as [0, 1, 0, 0, 0]. However, this method does not contain any corpus information, and the distance between all words is the same. Word2vec represents the vector according to the context, and words with high relevance have closer distances; therefore, word2vec is more expressive in terms of the intrinsic characteristics of the data. Therefore, we choose word2vec for word embedding.

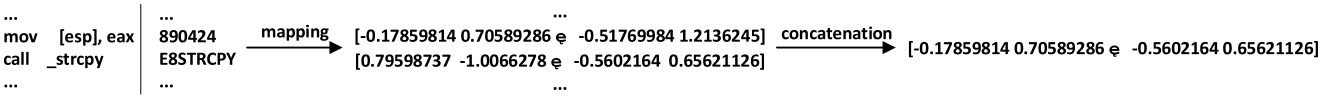


FIGURE 3. An example of vector representation.

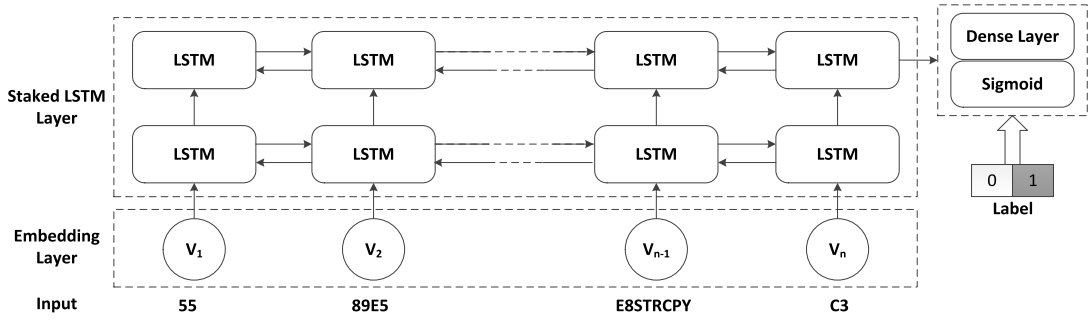


FIGURE 4. Neural network architecture.

The hexadecimal bytecode of each instruction is treated as a token, for example, 0x890424 represents mov [ebp], eax; and then, the bytecode sequence is trained with word2vec. The output is a vector representation of each instruction in 256-dimensional space. We can then transform the execution path into a vector representation of each instruction, as shown in Fig. 3.

It is obvious that the lengths of different paths are unequal and highly variant. Such paths are difficult to feed into deep neural network architectures. Therefore, we set the maximum length of the path to be n. Paths shorter than n are padded with 0 from the back end of them. Whereas paths longer than n are truncated from the front end of them, because new triggered path branches are usually located at the end of the paths. When padding to a fixed length of elements $X1, X2, \ldots, Xn$ (Xi is the vector representation of each instruction), the input sequence of one path can be represented as $X1 : n = X1 \oplus X2 \oplus \ldots \oplus Xn$, where $\oplus$ is the concatenator.

4) TRAINING THE NEURAL NETWORK MODEL

In this step, the hidden vulnerability pattern is learned from training data containing a large number of vulnerable and clean paths. We choose the long short-term memory (LSTM) neural network for training to output a model that can be used to classify unseen paths triggered during the fuzzing process. On the one hand, LSTM is good at processing sequential data: the program path is very similar to the statement in natural language, and whether a piece of code is vulnerable depends on the context. On the other hand, LSTM has a memory function that is suitable for handling long dependencies because the code associated with the vulnerability may be located at a relatively long distance in the path.

Our LSTM-based neural network consists of a total of 4 layers, as shown in Fig. 4. The first layer is the embedding layer, which maps all the elements in the sequence to a fixed-dimensional vector. The second and third layers are stacked LSTM layers, each of which contains 64 LSTM units in a bidirectional form, and the stacked LSTM model can learn a higher-level time-domain feature representation. Finally, we use a dense output layer with a single neuron and a sigmoid activation function to make predictions for the two classes in our task. Our loss function is binary_crossentropy, the optimizer is adam, and because LSTM often suffers from overfitting, we use dropout to overcome this problem. Our model achieves the best results when the dropout rate is set to 0.6.

B. ONLINE GUIDED FUZZING

Online guided fuzzing is implemented on the basis of PTfuzz's main fuzzing loop. When the fuzzer generates one new input and achieves new branch coverage, it will be added to the seed queue. To achieve our seed priority strategy, we add an execution path recovery and vulnerability prediction module before adding seeds to the seed queue, as highlighted in the gray boxes in Fig. 2(b). We can recover the complete execution path of the program based on the control flow information captured by Intel PT and the target binary file. Then, we use our prediction model to determine whether an unseen path is vulnerable. Seeds are marked according to the prediction result (vulnerable seeds are marked as "1", non-vulnerable seeds are marked as "0") and then added to the seed queue. At last, vulnerable seeds will be prioritized and assigned more mutation energy in the next seed selection process and seed mutation process.

1) RECOVERING THE EXECUTION PATH

When fuzzing with PTfuzz, the execution control flow information is collected in real time in the form of data packets as shown in Table 2 (Intel PT specifies instructions that can change program flow as Change of Flow Instructions (COFI).) and stored in specified memory space. Bitmap which represents the code branch coverage information is

TABLE 2. PT packet type description.

Packet Type	COFI	Description
Taken Not Taken (TNT)	Conditional jump instructions such as "jnz 0x804855D" etc.	A specific bit in a TNT packet can indicate whether a branch is taken in conditional jumps.
Target IP (TIP)	Indirect jump or transfer instructions such as "jmp/call [eax]" and "ret" etc.	TIP records the target instruction pointer of indirect jump or transfer instructions.
Flow Update Packet (FUP)	Asynchronous events such as exceptions and interrupts.	FUP provides source IP addresses because TIP is out of function in these events.

updated by decoding these data packets. However, there is no complete execution path information required by our model input throughout this process. So before using our prediction model, we must recover the execution path based on the control flow packets and the target program binary file by decoding the trace stored in the specific memory.

Algorithm 1 describes the process in detail. Specifically, first, the binary program is loaded, and the instruction address is obtained from the main entry function (lines 1,2). Then, the next instruction is obtained according to the current instruction type, and its byte code is added to the path until the last instruction of the trace (lines 4-8). If the instruction is a conditional jump, the jump direction is determined according to the TNT package (lines 11-18). If the instruction is an indirect jump or transfer, the jump target address is obtained according to the TIP packet (lines 19-21). If the instruction is an unconditional direct jump, the jump target address is obtained from the instruction (lines 22,23). If the instruction is not a jump, the next instruction is obtained based on the size of the current instruction (lines 24,25). We decode the PT trace information with the decoding library libipt [41] provided by Intel.

Algorithm 1 Program Execution Path Recovery

```
function RecoverPath(program, trace)
1: image = LoadELF(program)
2: insaddr = GetEntryOffset(image)
3: path = []
4: while true do
5:   insbyte = GetInsByte(insaddr)
6:   add insbyte to path
7:   if insaddr is last instruction then
8:     break
9:   instype = GetInsType(image, insaddr)
10:  switch instype
11:    case conditional jump instruction:
12:      istaken = decodeTNT(trace, insaddr)
13:      if istaken is true then
14:        insaddr = GetTargetAddr(image, insaddr)
15:      end if
16:      else
17:        insaddr = insaddr + size(image, insaddr)
18:      end else
19:    case indirect jump or transition instruction:
20:      targetaddr = decodeTIP(trace, insaddr)
21:      insaddr = targetaddr
22:    case unconditional direct jump instruction:
23:      insaddr = GetTargetAddr(image, insaddr)
24:    default:
25:      insaddr = insaddr + size(image, insaddr)
26:end while
end function
```

2) GUIDING FUZZING WITH THE NEURAL NETWORK MODEL

Our final goal is to leverage the learned model to guide the fuzzer to prioritize seeds that exercise vulnerable paths to expose the vulnerability early on and assign them more mutation energy but less energy to seeds exercise clean paths to increase the probability of triggering vulnerabilities. Therefore, we need to modify the original algorithm of PTfuzz to integrate our trained model to the fuzzer.

$$\text{score} = f_{\text{depth}}\left(f_{\text{time}}\left(f_{\text{cov}}\left(f_{\text{speed}}(s)\right)\right)\right) \tag{1}$$

$$f_{origEnergy}(s) = \begin{cases} 1600 & if\ score \geq 1600 \\ score & otherwise. \end{cases} \tag{2}$$

$$f_{newEnergy}(s) = \begin{cases} 1600 & if\ vul(s) = true \\ 0.5 \cdot f_{origEnergy}(s) & otherwise \end{cases} \tag{3}$$

Our NeuFuzz implementation is shown in Algorithm 2, where the gray boxes indicate the differences between our approach and the PTfuzz's algorithm. Specifically, when a mutated input triggers a new branch (line 14), the execution path is recovered first according to the control flow information captured by the PT and the binary program file (line 15). Then, the model is used to classify whether the path is vulnerable, and the seed is marked according to the prediction results and added to the queue (line 16-20). When seeds are taken from the queue, the first step is to checkwhether there is a seed marked as vulnerable. If vulnerable seeds exist, they are prioritized (line 5-9). Our implementation of AssignEnergy (line 10) assigns the most mutation energy (the default maximum value of PTfuzz is 1600) to vulnerable seeds and less energy (50% of the original allocated energy) to non-vulnerable seeds, as shown in equation (3), vul(s) returns true if the seed is marked as vulnerable. Equation (2) denotes the original energy computation method of PTfuzz, which uses the execution time f_{speed}, block transition coverage f_{cov}, creation time f_{time}, and path depth of the seed f_{depth},

Algorithm 2 NeuFuzz Fuzzing Loop With Learning

```
function Fuzz(program, seeds)
1: add seeds to Queue
2: pending_vul = 0
3: repeat
4:   for seed in Queue do
5:     if pending_vul is not 0 then
6:       if seed is not vulnerable then
7:         continue
8:       end if
9:     end if
10:    energy = AssignEnergy(seed)
11:    for i from 0 to energy do
12:      newinput = MutateInput(seed)
13:      trace = Execute(program, seed)
14:      if HasNewCov(trace) then
15:        path = RecoverPath(program, trace) //
         Algorithm 1
16:        if QueryModel(path) is True then
17:          mark newinput as vulnerable
18:          pending_vul++
19:        end if
20:      add newinput to Queue
21:      end if
22:    end for
23:  end for
24: until timeout reached or abort signal
end function
```

as shown in equation (1). Notably, in an extreme situation, no vulnerable paths may be discovered for a substantial period of time. In this case, NeuFuzz follows the original seed selection strategy of PTfuzz until one vulnerable path is identified.

V. IMPLEMENTATION AND EVALUATION

Our model training, containing approximately 180 lines of python code, is developed based on keras with TensorFlow as the backend. The fuzzer module is based on the PTfuzz implementation, which adds approximately 600 lines of C/C++ code, including module implementation related to path recovery, model prediction, seed selection, and energy allocation and so on.

A. EVALUATION SETUP

To evaluate the effectiveness of our proposed method, we conduct multiple experiments on different test suites for comparison with current advanced tools.

We choose a common benchmark dataset, LAVA-M, which contains four vulnerable Linux programs, namely, base64, md5sum, uniq and who, and each application has been manually injected with multiple bugs. In addition, to verify the scalability and effectiveness of NeuFuzz for real software, we choose nine widely used real-world applications for testing, namely, libtiff, binutils, libav, podofo, bento4, libsndfile, audiofile and nasm which process multiple file formats, including image, elf, document, video, audio and asm file.

We compare NeuFuzz with PTfuzz and QAFL, which both support binary fuzzing without relying on source code. We run all the experiments on a computer equipped with an ubuntu16.04 system, 32 GB RAM, an Intel core i7 8700k processor, and one Nvidia 1080Ti GPU. In all the comparison experiments, we run only one fuzz instance with one CPU core, and each instance runs for 24 hours. The experiments are designed to answer the following three research questions.

- RQ 1. How effective is our learned neural network model?
- RQ 2. How good is the vulnerability detection capability of NeuFuzz?
- RQ 3. How is the overhead of the NeuFuzz fuzzing loop?

B. RESULTS

1) EFFECTIVENESS OF NEURAL NETWORK MODEL (RQ 1)

The effectiveness of the model is critical to NeuFuzz because seed selection is performed based on the model prediction results. Therefore, in this section, we evaluate the results of the trained model. As shown in Fig. 5, after 5 epochs of training, the training accuracy of our model reached 91%, and the training loss decreased to 22%. Currently, no tool is publicly available for vulnerability prediction for binary programs, and the most closely related work to our approach is VDiscover, which extracts API sequences as features and then trains them using traditional machine learning algorithms. By contrast, we take the execution path as input and use LSTM for training. To illustrate the advantages of our method (referred to as PATH+LSTM), we implement VDiscover's method that extracts dynamic API sequences and then trains with a random forest algorithm (referred to as API+RF).

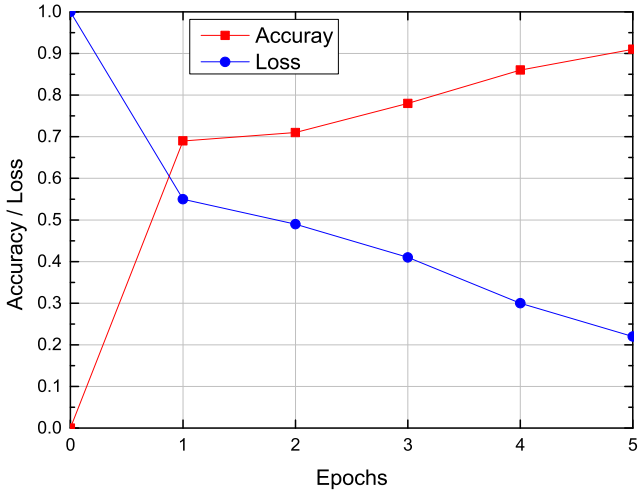


FIGURE 5. Training accuracy and loss of our model.

We consider the following widely used metrics in machine learning to evaluate our method: false positive rate (FPR),

TABLE 3. Performance of VDiscover and NeuFuzz on Test-Data and LAVA-M.

Method	Dataset	TP	FN	FP	TN	FPR (%)	FNR (%)	TPR (%)	P (%)	F1 (%)
VDiscover	Test-Data	3061	2503	1890	3466	35.2	45.0	55.0	61.8	58.2
(API+RF)	LAVA-M	1280	985	-	-	-	43.5	56.5	-	-
NeuFuzz	Test-Data	5119	445	316	5040	5.8	8.0	92.0	94.0	92.9
(PATH+LSTM)	LAVA-M	1869	396	-	-	-	17.5	82.5	-	-

TABLE 4. Detected bugs on the LAVA-M dataset.

Program	Total bugs	NeuFuzz	PTfuzz	QAFL
Base64	44	6	2	0
Md5sum	57	-	-	-
Uniq	28	5	1	0
Who	2136	8	0	0
Total	**2265**	**19**	**3**	**0**

false negative rate (FNR), true positive rate (TPR) or recall, precision (P), and F1-measure (F1). Let TP be the number of samples with correctly detected vulnerabilities, FP be the number of samples with falsely detected vulnerabilities, FN be the number of samples with undetected true vulnerabilities, and TN be the number of samples with no undetected vulnerabilities. The $FPR = FP/(FP+TN)$ is the ratio of false positive vulnerabilities to the entire population of samples that are not vulnerable. The $FNR = FN/(TP+FN)$is the ratio of false negative vulnerabilities to the entire population of samples that are vulnerable. The $TPR = TP/(TP+FN)$ is the ratio of true positive vulnerabilities to the entire population of samples that are vulnerable. Precision $P = TP/(TP + FP)$ measures the correctness of the detected vulnerabilities. $F1 = 2 \cdot P \cdot TPR/(P + TPR)$ accounts for both precision and the TPR.

We evaluate our method on two datasets. The first is part of the dataset that we constructed (referred to as Test-Data), including 5,564 vulnerable paths and 5,356 clean paths, accounting for 1/5 of all our extracted execution paths. The other is the LAVA-M dataset. Although this dataset provides vulnerable programs and inputs that trigger these vulnerabilities, there are no patched versions of the programs, so we consider only the TPR and FNR. The results are shown in Table 3. We can see that the method of NeuFuzz outperforms VDiscover for both datasets, which can be explained by the fact that the vulnerability is reflected not only in the API call sequence but also in the constraints condition. For example, calling strcpy with a complete security check may be safe. In addition, although our model performs worse on LAVA-M than on Test-Data in terms of FPR and TPR, it still achieves a positive rate of 82.5%, i.e., it is very effective at distinguishing vulnerable from non-vulnerable code.

2) VULNERABILITY DETECTION CAPABILITY (RQ 2)

The number of unique crashes is an important factor in measuring the effectiveness of a fuzzer. Some crashes may be caused by the same root cause (i.e., duplicated) or may not be security-related; however, in general, the greater the number of crashes found, the higher the probability that more

TABLE 5. Vulnerabilities found by NeuFuzz, including target programs, number of unique crashes, known and unknown vulnerabilities, unknown vulnerabilities confirmed by CVE, vulnerability type(HO: heap overflow, SO: stack overflow, NP: null pointer deference, IA: invalid memory access, OOM: out of memory) and time to expose the corresponding CVE.

Applications	Unique Crashes			Vulnerability		CVE	CVE-ID	Type	Exposure Time (h)		
	NeuFuzz	PTfuzz	QAFL	Known	Unknown				NeuFuzz	PTfuzz	QAFL
libtiff	22	11	3	0	3	2	CVE-2018-12900	HO	0.95	3.00	18.50
							CVE-2018-17000	NP	2.50	N	N
binutils	18	9	0	0	4	4	CVE-2018-13033	OOM	6.15	N	N
							CVE-2018-17358	IA	1.50	5.20	N
							CVE-2018-17359	IA	4.20	20.58	N
							CVE-2018-17360	HO	5.46	N	N
libav	260	193	99	3	9	7	CVE-2018-18826	HO	0.86	2.60	5.00
							CVE-2018-18827	HO	1.20	4.28	7.50
							CVE-2018-18828	HO	1.38	10.42	9.58
							CVE-2018-18829	NP	2.15	6.83	10.86
							CVE-2018-19128	HO	3.62	11.80	N
							CVE-2018-19129	NP	5.15	N	N
							CVE-2018-19130	IA	5.86	N	N
podofo	393	320	159	2	2	2	CVE-2018-12982	IA	0.50	3.65	4.00
							CVE-2018-12983	SO	2.40	12.30	20.80
bento4	441	341	136	5	6	3	CVE-2018-14543	NP	0.75	1.00	2.56
							CVE-2018-14544	IA	1.25	6.50	15.60
							CVE-2018-14545	IA	2.58	16.30	N
libsndfile	5	0	0	0	1	1	CVE-2018-13139	SO	1.80	N	N
audiofile	126	61	28	2	1	1	CVE-2018-13440	NP	0.25	1.60	2.75
nasm	25	12	5	2	2	1	CVE-2018-16999	IA	2.40	9.80	N
Total	**1290**	**947**	**430**	**14**	**28**	**21**	-	-	-	-	-

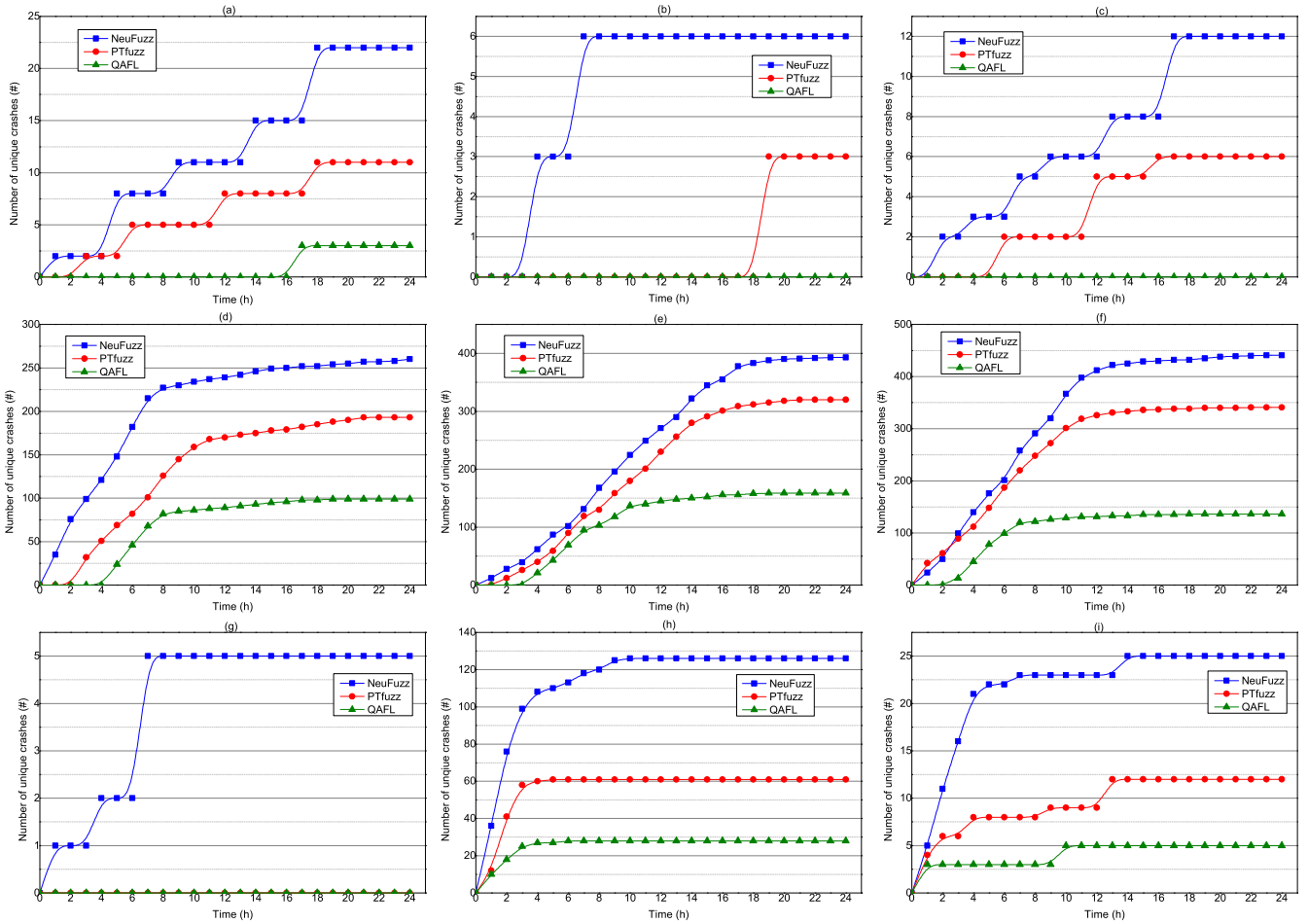


FIGURE 6. Number of unique crashes detected over time. (a) tiffcp + libtiff. (b) objdump + binutils. (c) nm + binutils. (d) avconv + libav. (e) podofocolor + podofo. (f) mp42ts + bento4. (g) sndfile-deinterleave + libsndfile. (h) sfconvert + audiofile. (i) nasm.

vulnerabilities can be identified. We compare the results of NeuFuzz, PTfuzz and QAFL on LAVA-M and real-world applications in this section. Table 4 shows the number of bugs found in LAVA-M with different fuzzers. The first column is the target program name, the second column is the number of known bugs in the target program, and the last three columns are the numbers of bugs found by NeuFuzz, PTfuzz, and QAFL, respectively. NeuFuzz found 19 bugs, exceeding PTfuzz and QAFL. The md5sum experiment could not be completed by NeuFuzz, PTfuzz, and QAFL because the methods crashed on the first input and the same phenomenon occurs in [15].

The results for real-world applications are shown in Table 5. These applications were all the latest versions at the time of testing. NeuFuzz can find more crashes than PTfuzz and QAFL, so NeuFuzz has a higher probability of finding vulnerabilities. A total of 1290 crashes are discovered after testing these programs for 24 hours. Then, we use AddressSanitizer [42] to perform deduplication and further identify 42 vulnerabilities. 14 of these vulnerabilities were previously discovered and disclosed by other researchers, but the vendors have not released patches. The other 28 are unknown, 21 of which are confirmed by CVE. The discovered vulnerabilities include mainly buffer overflow, UAF, null pointer dereference, divide by zero, invalid memory access and other memory corruption problems. In addition, the last column of the table shows that NeuFuzz can find all CVEs within 24 hours, much faster than PTfuzz and QAFL. PTfuzz and QAFL miss some vulnerabilities, and "N" indicates that the tool does not find the corresponding CVE.

Fig. 6 shows the growth of unique crashes of different fuzzers on nine real-world applications within 24 hours. From the growth trend, we can see that the number of unique crashes found by NeuFuzz grows faster than that of PTfuzz and QAFL, and more crashes are found by NeuFuzz. For example, when fuzzing the sndfile-deinterleave program of the libsndfile library, NeuFuzz found 5 crashes, whereas PTfuzz and QAFL found none. Further analysis indicates that this is a stack overflow vulnerability. In addition, QAFL performs the worst of the three fuzzers, because it is based on QEMU and much more inefficient in terms of execution speed than PTfuzz, which is based on hardware implementation. In short, NeuFuzz performs better in terms of both efficiency and effectiveness of vulnerability detection.

3) OVERHEAD OF NeuFuzz (RQ 3)

The effectiveness of a graybox fuzzer is strongly dependent on its execution speed, which, for the most part, is limited only by the time required to execute one test case. Therefore, we need to ensure that our method does not lead to large performance overhead, reducing the fuzzing efficiency. We measure the execution speed of each fuzzer by the number of test cases executed per second (exe/s) to demonstrate the fuzzing overhead. As shown in Fig. 7, the execution speed of NeuFuzz is approximately 2.5 times faster than that of QAFL but approximately 8% slower than that of PTfuzz because NeuFuzz requires additional time for path recovery and vulnerability prediction. In fact, we find that the time overhead originates mainly from path recovery, which is proportional to the length of the execution path. The time complexity of predicting the result in a simple neural network is only $O(n1 \cdot n2 + n2 \cdot n3 + \ldots)$ [43]. Thus, when the number of nodes is within a small range, as in our model, the predicting process requires much less time than actually running the target program once. In short, the overhead is relatively acceptable for fuzzing tasks. Furthermore, although training the neural network model requires some time, the training process is performed offline, and we have to train the model only once for all subsequent tests; thus, we do not consider the time spent on model training.

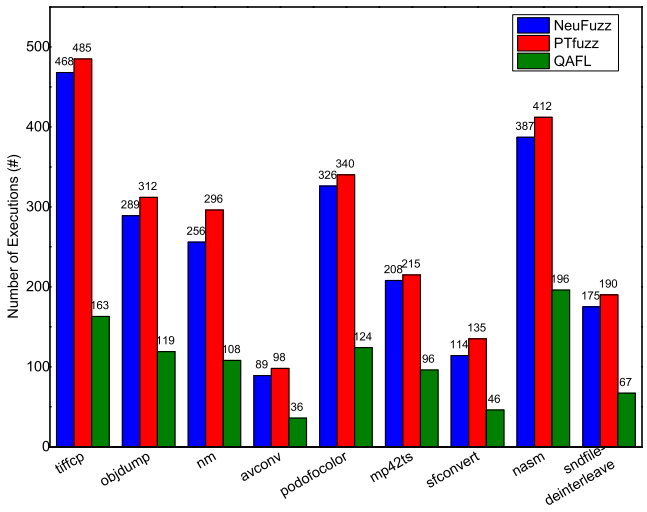


FIGURE 7. The number of executions on real-world programs.

C. DISCUSSION

The evaluation results show that NeuFuzz achieves better vulnerability detection than the current state-of-the-art fuzzers at a reasonable cost, but some remaining limitations deserve further research.

First, NeuFuzz requires the support of a specific hardware and operating system because our method is based on PTfuzz. PTfuzz uses Intel PT, which is a new feature of 5th generation Intel CPUs, and PTfuzz can run only on the Linux platform with a Linux kernel version of at least 4.1.x.

Second, NeuFuzz cannot handle some program paths longer than our threshold when we use the execution paths as input to train the model, which limits the ability of our model. In the future, we will consider other methods to represent the path, such as dynamic program slicing. Furthermore, we should consider applying other neural network models to learn the vulnerability pattern.

Third, NeuFuzz cannot find vulnerabilities hidden behind complex sanity checks, such as magic bytes, because the neural network model can predict only the exercised path to guide seed selection. This problem is a popular topic for improving code coverage, and we can integrate some of the current advanced methods into our tools to enhance the vulnerability detection capability.

Fourth, even if a seed is found to trigger a new path during fuzzing and the learned model accurately identifies this path, NeuFuzz still may not trigger the vulnerability. This is due to the randomness of fuzzing, because new seed generated by mutating may fail to trigger the original vulnerable path. Farifuzz is proposed to solve this problem by dynamic instrumentation. In the future, we can consider applying deep reinforcement learning to guarantee the specified path to be tested as much as possible so that further improve the fuzzing capability.

VI. CONCLUSION

The current graybox fuzzing technologies mainly focus on how to improve code coverage but ignore the distribution of vulnerable code in the program; that is, they attempt to cover as many program paths as possible rather than to explore likely to be vulnerable paths. In this paper, we propose a new seed selection strategy based on a deep neural network and implement a path-sensitive binary fuzzing tool, NeuFuzz. We construct a large dataset to train our neural network model to learn the hidden vulnerability patterns and then use the produced model to predict the new path triggered during the fuzzing process. Finally, we prioritize the seeds that exercise the vulnerable path and accordingly assign these seeds more mutation energy according to the prediction results. We conducted experiments on both crafted benchmark and real-world applications. Results showed that our proposed method is both effective and efficient in terms of crash finding and vulnerability detection. Compared with PTfuzz and QAFL, NeuFuzz can find more vulnerabilities in less time. We have found 28 new security bugs in nine widely used real-world applications, and 21 of them are confirmed by CVE.

REFERENCES

[1] Z. Durumeric *et al.*, "The matter of heartbleed," in *Proc. Conf. Internet Meas. Conf.* New York, NY, USA: ACM, 2014, pp. 475–488.
[2] S. Mohurle and M. Patil, "A brief study of wannacry threat: Ransomware attack 2017," *Int. J. Adv. Res. Comput. Sci.*, vol. 8, no. 5, pp. 1–4, 2017.
[3] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, 1990.
[4] *Sdl Process: Verifcation*. Accessed: Oct. 1, 2018. [Online]. Available: https://www.microsoft.com/en-us/sdl/process/verifcation.aspx
[5] *Google Online Security Blog-Fuzzing at Scale*. Accessed: Oct. 1, 2018. [Online]. Available: https://security.googleblog.com/2011/08/fuzzing-at-scale.html

[6] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Proc. NDSS*, vol. 8, 2008, pp. 151–166.
[7] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *Proc. 31st Int. Conf. Softw. Eng.*, 2009, pp. 474–484.
[8] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling black-box mutational fuzzing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.* New York, NY, USA: ACM, 2013, pp. 511–522.
[9] *Peach*. Accessed: Oct. 1, 2018. [Online]. Available: https://www.peach.tech/
[10] *American Fuzzy Lop*. Accessed: Oct. 1, 2018. [Online]. Available: http://lcamtuf.coredump.cx/a?/
[11] *LibFuzzer—A Library for Coverage-Guided Fuzz Testing*. Accessed: Oct. 1, 2018. [Online]. Available: https://llvm.org/docs/LibFuzzer.html
[12] *Honggfuzz*. Accessed: Oct. 1, 2018. [Online]. Available: https://github.com/google/honggfuzz
[13] *The Bug-O-Rama Trophy Case of AFL*. Accessed: Oct. 1, 2018. [Online]. Available: http://lcamtuf.coredump.cx/a?/#bugs
[14] T. J. Ostrand and E. J. Weyuker, "The distribution of faults in a large industrial software system," *ACM SIGSOFT Softw. Eng. Notes*, vol. 27, no. 4, pp. 55–64, 2002.
[15] G. Zhang, X. Zhou, Y. Luo, X. Wu, and E. Min, "PTfuzz: Guided fuzzing with processor trace feedback," *IEEE Access*, vol. 6, pp. 37302–37313, 2018.
[16] B. Dolan-Gavitt *et al.*, "Lava: Large-scale automated vulnerability addition," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2016, pp. 110–121.
[17] *High-Performance Binary-Only Instrumentation for Afl-Fuzz*. Accessed: Oct. 1, 2018. [Online]. Available: https://github.com/mirrorer/afl/tree/master/qemu_mode
[18] *Intel Processor Trace*. Accessed: Oct. 1, 2018. [Online]. Available: https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing
[19] S. Schumilo *et al.* (2017). *KAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels*. [Online]. Available: https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-schumilo
[20] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, 2017, pp. 1–14.
[21] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," in *Proc. CCS*, New York, NY, USA, 2016, pp. 1032–1043.
[22] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.* New York, NY, USA: ACM, 2017, pp. 2329–2344.
[23] S. Gan *et al.*, "CollAFL: Path sensitive fuzzing," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2018, pp. 679–696.
[24] P. Chen and H. Chen. (2018). "Angora: Efficient fuzzing by principled search." [Online]. Available: https://arxiv.org/abs/1803.01307
[25] N. Stephens *et al.*, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proc. NDSS*, vol. 16, 2016, pp. 1–16.
[26] Y. Li *et al.*, "Steelix: Program-state based binary fuzzing," in *Proc. 11th Joint Meeting Found. Softw. Eng.* New York, NY, USA: ACM, 2017, pp. 627–637.
[27] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-Fuzz: Fuzzing by program transformation," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2018, pp. 697–710.
[28] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *Proc. 32nd IEEE/ACM Int. Conf. Automated Softw. Eng.* Piscataway, NJ, USA: IEEE Press, 2017, pp. 50–59.
[29] M. Rajpal, W. Blum, and R. Singh. (2017). "Not all bytes are equal: Neural byte sieve for fuzzing." [Online]. Available: https://arxiv.org/abs/1711.04596
[30] K. Böttinger, P. Godefroid, and R. Singh. (2018). "Deep reinforcement fuzzing." [Online]. Available: https://arxiv.org/abs/1801.04589
[31] Z. Li *et al.* (2018). "VulDeePecker: A deep learning-based system for vulnerability detection." [Online]. Available: https://arxiv.org/abs/1801.01681
[32] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier, "Toward large-scale vulnerability discovery using machine learning," in *Proc. 6th ACM Conf. Data Appl. Secur. Privacy*. New York, NY, USA: ACM, 2016, pp. 85–96.
[33] *AFLPIN*. Accessed: Oct. 1, 2018. [Online]. Available: https://github.com/mothran/a?pin
[34] C.-K. Luk *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. PLDI*. New York, NY, USA: ACM, 2005, pp. 190–200.
[35] *Code Gadget Database of VulDeePecker*. Accessed: Oct. 1, 2018. [Online]. Available: https://github.com/CGCL-codes/VulDeePecker
[36] *NIST Test Suites*. Accessed: Oct. 1, 2018. [Online]. Available: https://samate.nist.gov/SRD/testsuite.php
[37] *Github*. Accessed: Oct. 1, 2018. [Online]. Available: https://github.com/
[38] *EXPLOIT DATABASE*. Accessed: Oct. 1, 2018. [Online]. Available: https://www.exploit-db.com/
[39] *Onehot*. Accessed: Oct. 1, 2018. [Online]. Available: https://en.wikipedia.org/wiki/One-hot
[40] *Word2Vec*. Accessed: Oct. 1, 2018. [Online]. Available: http://radimrehurek.com/gensim/models/word2vec.html
[41] *Intel PT Decoder Library*. Accessed: Oct. 1, 2018. [Online]. Available: https://github.com/01org/processor-trace
[42] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Proc. USENIX Annu. Tech. Conf.*, 2012, pp. 309–318.
[43] P. Orponen, "Neural networks and complexity theory," in *Proc. Int. Symp. Math. Found. Comput. Sci.* Berlin, Germany: Springer, 1992, pp. 50–61.

YUNCHAO WANG received the M.S. degree in computer science and technology from the China National Digital Switching System Engineering and Technological Research Center, Zhengzhou, China, in 2016, where he is currently pursuing the Ph.D. degree in cyberspace security. His research interests include reverse engineering and vulnerability discovery.

ZEHUI WU received the Ph.D. degree in software engineering from the China National Digital Switching System Engineering and Technological Research Center, Zhengzhou, China, where he is currently a Lecturer. His research interests include program analysis, reverse engineering, and SDN security.

QIANG WEI received the Ph.D. degree in computer science and technology from the China National Digital Switching System Engineering and Technological Research Center, Zhengzhou, China, where he is currently a Professor. His research interests include network security, the industrial Internet security, and vulnerability discovery.

QINGXIAN WANG received the M.S. degree from the Department of Computer Science and Technology, Peking University. He is currently a Professor with the China National Digital Switching System Engineering and Technological Research Center. His research interests include network security, trusted computing, and vulnerability discovery.