Suzzer: A Vulnerability-Guided Fuzzer Based on Deep Learning

Yuyue Zhao[1(✉)], Yangyang Li[2], Tengfei Yang[2], and Haiyong Xie[1,2,3]

[1] University of Science and Technology of China, Hefei 230027, Anhui, China
yyzha0@mail.ustc.edu.cn
[2] National Engineering Laboratory for Public Safety Risk Perception and Control by Big Data (NEL-PSRPC), Beijing 100041, China
{liyangyang,yangtengfei}@cetc.com.cn
[3] Advanced Innovation Center for Human Brain Protection, Capital Medical University, Beijing 100054, China

Abstract. Fuzzing is a simple and effective way to find software bugs. Most state-of-the-art fuzzers focus on improving code coverage to enhance the possibility of causing crashes. However, a software program oftentimes has only a fairly small portion that contains vulnerabilities, leading coverage-based fuzzers to work poorly most of the time. To address this challenge, we propose *Suzzer*, a vulnerability-guided fuzzer, to concentrate on testing code blocks that are more likely to contain bugs. Suzzer has a light-weight static analyzer to extract ACFG vector from target programs. In order to determine which code blocks are more vulnerable, Suzzer is equipped with prediction models which get the prior probability of each ACFG vector. The prediction models will guide Suzzer to generate test inputs with higher vulnerability scores, thus improving the efficiency of finding bugs. We evaluate Suzzer using two different datasets: artificial LAVA-M dataset and a set of real-world programs. The results demonstrate that in the best case of short-term fuzzing, Suzzer saved 64.5% of the time consumed to discover vulnerabilities compared to VUzzer.

Keywords: Prediction model · Deep learning · Vulnerability-guided fuzzing

1 Introduction

Vulnerability is the most intuitive manifestation of information security. Currently the main threat to computer systems is software vulnerabilities. Fuzzing [1] is an automated penetration technique that discovers vulnerabilities in software by sending randomly generated data to the program and monitoring for anomalies in the output.

In view of the current research status, fuzzing can be divided into two categories, namely, generation-based fuzzing (see, *e.g.*, [2–5]) and mutation-based fuzzing (see, *e.g.*, [6–13]). More specifically, generation-based fuzzing is based

Z. Liu and M. Yung (Eds.): Inscrypt 2019, LNCS 12020, pp. 134–153, 2020.
https://doi.org/10.1007/978-3-030-42921-8_8

primarily on model-based or grammar-based approaches to satisfy the software program syntax and semantic test input generation. In particular, model-based generations construct input through some of the given format specifications (see, *e.g.*, [2–4]), while grammar-based fuzzing test uses the known input syntax to construct the input test set (see, *e.g.*, [5]). Mutation-based fuzzing are guided by the program execution environment information and program analysis techniques to mutate the inputs in fuzzing (see, *e.g.*, [6–13]).

Generation-based fuzzing can generate valid inputs using known information to increase code-coverage. However, it needs significant amounts of manual input to satisfy the format specification, especially when testing large-scale software; if the input format is wrong, it reports a significant number of errors. Thus, generation-based fuzzing is not flexible enough, and cannot process unknown programs. On the contrary, mutation-based fuzzing can use existing input mutated to generate new inputs without relying on input syntax, with better scalability and applicability. Therefore, most state-of-the-art fuzzers are mutation-based fuzzers.

Most mutation-based fuzzers focus on improving code coverage to increase the likelihood of triggering a crash. Improving code coverage can indeed explore more program location areas and discover more vulnerabilities. However, it is not practicable to completely explore all the code branches of a given program for many reasons. First, if the software program grows larger, it's more difficult to solve all path constraints. Second, it is time consuming to improve code coverage. Third, the vulnerable code usually contributes to only a small portion of the entire code. For example, Liu *et al.* [15] found 52.31% bugs are located with no more than 10% of the code. Shin *et al.* [14] revealed that 21% of source code files in Firefox browsers are faulty, and only 3% of them have vulnerabilities.

To address the above challenge, we propose a vulnerability-guided fuzzing framework, *Suzzer*, to achieve efficient fuzzing. By vulnerability-guided, we mean that fuzzers could focus on testing inputs for the basic blocks with higher probability of vulnerability rather than blindly treating them as equal.

We summarize our contributions as follows. Firstly, we studied the negative impacts of inefficiently fuzzing in coverage-based fuzzers. Especially, we noted that by focusing on vulnerabilities, we could solve such problem and improve its capability of vulnerability discovery. Secondly, we built a vulnerability prediction model based on basic blocks to achieve vulnerability-guided strategy in Suzzer. Since there are no readily available datasets for vulnerability detection in basic block granularity, we present the first dataset for setting up the prediction model. The dataset is derived from NIST. Lastly, We implemented a prototype based on VUzzer, and evaluated it on three artificial applications and six real-world programs. The effectiveness of Suzzer has been partially validated by its ability to save up to 64.5% time consumed to discover vulnerabilities compared to VUzzer.

2 Motivation

2.1 Vulnerability-Guided Fuzzing

Since most of the existing state-of-the-art fuzzers [6,10–13] are mutation-based, how to improve code coverage has become a top priority. Figure 1 breaks a typical mutation-based workflow into stages. Fuzzers take in the program and inputs, analysis program and use those information to mutate inputs, select inputs to feed the target program, monitor program to filter inputs and export vulnerabilities. Suzzer is a mutation-based fuzzer.

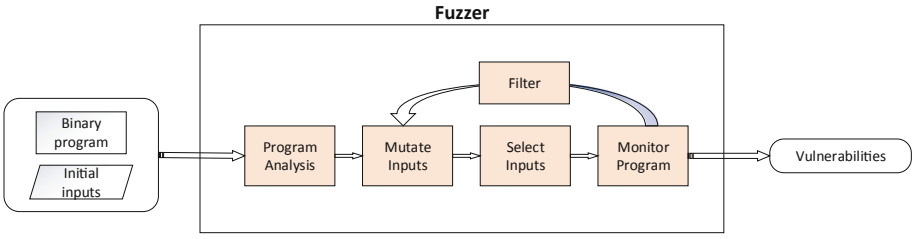


Fig. 1. Mutation-based fuzzing workflow

Corresponding to input mutation, fuzzers can be divided into three categories. Blackbox fuzzers [3,16,17] mutate inputs blindly until causing application crash. Whitebox fuzzers [18,19] usually test with the source code in mind, it performs better because of enough knowledge. Greybox fuzzers [6,10–13,20] still tests black box software without source code, but it can analyze binary program to get some useful information before fuzzing. Since the current commercial software is closed source, Suzzer is a greybox fuzzer.

Based on the application exploration strategy, there are two different fuzzers. Directed fuzzers [10,13,20] tend to select inputs that cover a specific set of paths, whereas coverage-based fuzzers [6,11,12] are more inclined to choose those inputs that can cover a wider range of paths in order to trigger more crash. Suzzer is a directed fuzzer.

In order to understand why we proposed suzzer, Listing 1.1 describes a simple situation that usually happens in program. In the main function, there is a normal magic byte check from line 6 to line 11. Then it is from the second check for buffer content at line 12. Within the subsequent check, there are several for loops and nested conditions to match the fields in data and buffer from line 15 to line 24. In the most state-of-the-art fuzzers, they will spend a lot of computing resources to generate inputs to bypass conditional checks. However, those checks code don't contain any vulnerabilities. In fact, there is a function code that contains a stack buffer overflow just in line 34.

Based on the situation we have just discussed, we developed Suzzer, that automatically tests for code which has a higher probability of containing vulnerabilities. Unlike existing state-of-the-art fuzzers, Suzzer doesn't judge inputs

by triggering depth, breadth, size of unknown area, etc. Suzzer is more focus on bugs. Before starting the fuzzing test, Suzzer first perform static analysis on the software program to obtain the code path, branch, basic-block distribution, the internal assembly code of the code basic-block, etc. We input the information obtained into the vulnerability prediction model, predict the prior probability that each basic-block may contain bugs, and then convert the obtained prior probability into a score and enter the fuzzing loop. In the fuzzing loop session, those test inputs that are more likely to trigger a program crash are selected for the next loop according to the path scores.

Listing 1.1. A motivating example that usually happends

```
#define SIZE 1024
int main(int argc, char **argv){
    unsigned char data[SIZE];
    unsigned char buffer[SIZE];
    char *fd;
    //======= Magic bytes check ==========
    if(buffer[0]==0xAB && buffer[1]==0xCD)
        printf("Magic bytes!");
    else{
        printf("InCorrect!");
        return 0;}
    //========= Another check ==========
    if(buffer[5]=='yes' && buffer[6]=='hello'){
        printf("One step check");
        for(int i=0; i < sizeof(data); i++){
            //===== Other nested checks ======
            if (strcmp(&buffer[10], "World!", 4) == 0){
                printf("Two step check passed");
                /* some nested condition*/
                printf("some harder condition passed");
                ...}
            else
              /* some other nested condition*/
              ...}}
        else{
            printf("Invalid ERROR!");
    '''
    Start any other Task!
    '''
    return 0;}
    /*
    *Some code that hide vulnerabilities.
    */
    stack_buffer_overflow(fd, buffer, data);
    return 0;
}
```

2.2 Vulnerability Prediction

In order to bulid the prediction model, we propose some guiding principles in this section. Those principles are concentrated on answering three questions: A. How to represent software to feed the prediction model? B. What is the appropriate granularity for both vulnerability detection and fuzzing? C. Which model is suitable for vulnerability prediction?

A. Software Representation. Since machine learning model takes vectors as input, we need to represent software as vectors. In bug search area, the CFG (Control Flow Geaph) [26] is widely used as a common feature. CFG can be transformed into different basic-block level attributes named ACFG (Attributed Control Flow Graph) [25], that is the actual input vector of our prediction model.

B. Appropriate Granularity. As a vulnerability detection tool, Suzzer not only needs to predict the possibility of vulnerability in the program, but also needs to detect the location. If the program is represented by too large a granularity [20] (function mode, etc.), although the display of the vulnerability rate can be obtained, it is not conducive to locating the location of the program vulnerability. By contrary, too small granularity (assembly statements [21]) covers too little information, which is not conducive to vulnerability prediction. Therefore, we adopt the basic block as the granularity unit, which is more suitable for fuzzing.

C. Model Selection. Traditional machine learning algorithms rely on human experts to define learning rules and may miss many information, which is subjective and often incur high false negative rates. Therefore, at least in vulnerability detection area, traditional ML methods are not appropriate. However, neural networks can automatically learn data features by adjusting the size and structure of the network to control learning, which makes it more flexible and more robust.

The vulnerability probability of a basic-block depends not only on its own information and architecture, but also on the location of basic-block, and the neighboring blocks of the basic block. Therefore, the basic M-P neuron network doesn't work well.

While Long Short-Term memory [30] (LSTM) neurons differ from standard M-P neuron networks. LSTM has a feedback connection that not only processes a single data point (such as an image), but also processes sequence data (such as statements and audio). Each of our programs can be considered as a sequence of basic blocks, so it is suitable for our prediction model. We will discuss the details in subsequent sections.

3 Design Overview

The main components and workflow of Suzzer are described in Fig. 2. Here we introduce the main components. The details will be shown in the following sections.

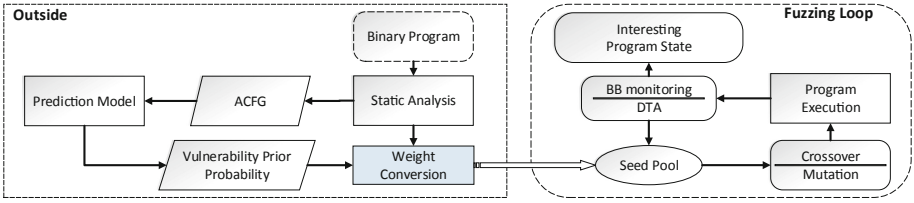


Fig. 2. Overview of suzzer

- Prediction Model: Suzzer uses a predictive model built by a neural network (see Sect. 4.3) to predict a priori vulnerability probability for the target software basic block. The input of the prediction model is the extracted software basic block feature data ACFG (see Sect. 4.2), and the output is a prior probability that the basic block may cause a bug. The fuzzer uses this as a standard for vulnerability-oriented testing.
- Fuzzer: We use an existing coverage guided fuzzer—VUzzer [10] to complete the fuzzing loop (see Sect. 6). Suzzer needs to monitor the probability of a basic block vulnerability, which could generate test inputs with higher vulnerability score and monitor the test state of the program, and recording whether the program crashes at the same time. All of the above provide interested states for the next fuzzing.
- Link module: We use the weight conversion module (see Sect. 5) to connect the Prediction Model and the Fuzzer to convert the vulnerability probability into a scalar score that the Fuzzer can recognize.

4 Prediction Model

4.1 Problem Description

In this section, we abstract the basic block sequence triggered by fuzzing input into a path. The path triggered by various inputs are obviously different. Assuming that the software contains a set of basic-blocks $SW = \{b_1, b_2, b_3, ..., b_n\}$, then considered the triggered basic blocks sequence is $[b_1, b_3, b_7, b_{12}, b_{14}, ...]$. Each basic block in sequence includes the probability of its vulnerability $prob(b_i)$. This prior probability then enters the fuzzing loop, guiding the fuzzer to spend more resources on those block sequences that have a higher probability of containing a bug, making the trigger vulnerability more sensitive.

The input of the prediction model is the basic block vector ACFG [25], and the output is the vulnerability probability.

4.2 Software Feature Extraction

The input for deep learning or neural networks is vectors, so the selection of vector granularity is the first step in predicting. In this paper, the basic block is used as the granularity unit, and the ACFG is used as the carrier to predict the program vulnerabilities, which is more suitable for fuzzing.

ACFG, means the attributed control flow graph, is a directed graph $G =< V, W, \phi >$, and V is a set of blocks; $E \subseteq V \times V$ is a set of edges, which means the connections between these basic blocks V, and $\varphi : V \rightarrow \sum$ is the mapping function, which extracts a set of attributes $\sum$ from a basic block in V.

In the usual software analysis, CFG(control flow graph) is used to find the vulnerability, but CFG is not a digital vector, which means that CFG cannot be directly used as input to the deep learning model. ACFG is a suitable way to represent CFG with a large number of basic block level feature values. Each basic block in ACFG is represented by a set of feature values, where each dimension of the feature number set is a specified attribute value. In this approach, the entire binary program can be converted into a vector of values.

Figure 3 shows the workflow of extracting software data. First, we disassembled the binary software datasets to obtain the control flow graph(CFG) of the program. The CFG is the common feature used in bug search and can be extracted by popular reverse tools like IDA pro [32].

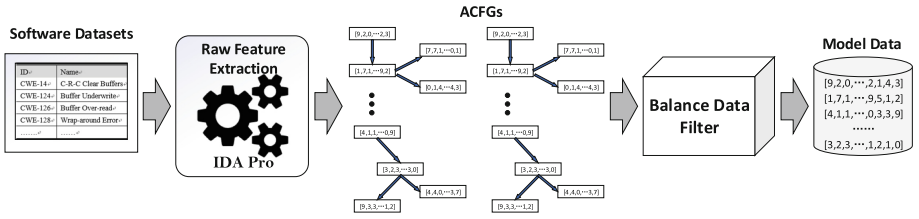


Fig. 3. The data extracting overview

Then, the raw feature is extracted from each basic block and converted into a numerical vector, the vector of the feature can be utilized to distinguish each basic block. Since a software is complex and consists of a large number of network relations, the raw feature can be a statistical attribute, a structural attribute, or a semantic attribute, etc. We choose statistical and structural two properties to represent the basic block vector, because a vector not only needs to count in the operation memory and the instruction characteristics of the data, but also even if it is the same basic block, the location of the basic block will have a great impact on the probability.

According to the extraction example of the firmware program ACFG in references [20,25,27], our statistical attribute contains 8-dimensional vectors including no. of call instruction, etc. For the specific application scenarios such as fuzzing, we refer to Intel's assembly development manual [31] and expand the number of statistical attribute to 19-dimensional vectors. Inspired by related work and the work on complex network analysis, we divide structural features into two types: num and location. The num contains $no.of offspring$ and $no.of betweenness$. $No.of offspring$ means the number of children nodes for a basic block, and, $no.of betweenness$ counts how many neighbor blocks between a basic block. These two information helps identify the relationships

Table 1. Detailed descriptions of ACFG

Type		Feature name	Num
Statistical features	Instruction	No. of push instruction	10
		No. of pop instruction	
		No. of lea instruction	
		No. of neg instruction	
		No. of cmp instruction	
		No. of test instruction	
		No. of call instruction	
		No. of retn instruction	
		No. of proc instruction	
		No. of endp instruction	
	Instruction set	Data transfer instructions	7
		Logical instructions	
		Branch instruction	
		Boolean instructions	
		Arithmetic instructions	
		Shift instructions	
		All assembly directives	
	Operand	No. of operand	1
	Register	No. of register	1
Structural features	Num	No. of offspring	2
		No. of betweeness	
	Location	Whether head or tail	3
		Distance of entrance (%)	
		Distance of export (%)	
All		Attribute dimensions	24

of basic blocks in software. The location contains three elements. First is *whether head or tail*. Judging *head node* or *tail node* in a program is critical to determining, because it could affect most areas if contains vulnerabilities. The other two are *distance of entrance* and *distance of export*. We determine the distance by calculating the proportion of their current position from the entrance or export. Detailed descriptions of ACFG are listed in Table 1.

Finally, we send those raw features into the balance data filter in order to get suitable data distribution for the model training.

In the feature data extraction part, in order to get the basic block features of structural attributes, such as the number of child nodes and neighbor nodes, we simulate the data structure of the control Flow Graph, named Suree. Suree mainly consists of five parts: head node, head area, data area, tail area and tail node. The head node represents the start address of the basic block, and the tail node represents the end address of the basic block, these two are constituted by a bidirectional pointer. The data area stores are specific assembly instructions.

The head area records the unique identification number side of Suree, and the tail area records the unique identification number of the child nodes. When we analyse the entire binary program, it needs to read Suree's header area to know the location of the basic block from the root node. Similarly, we can read the parent block of the basic block according to the inverse node of the head node, and then count the number of child nodes of the parent block to determine the number of neighbor blocks. The structure is illustrated in Fig. 4.

For the vulnerability labeling, we use IDA pro [32] to analyze the disassembled binary program. Since the source code has been annotated in the specific vulnerable program statement, we can use the regular module to analyze the sentence in a single basic block. If it matches the 'bad' string, the basic block is marked as vulnerable (1), otherwise it is marked as 0. The basic block is vulnerable means it has at least one bug. The marking algorithm is showed in Algorithm 1.

4.3 Model Building

Based on the research in Sect. 2.2, we choose bidirectional LSTM [30] as the core unit of our vulnerability prediction model. The prediction problem can be simplified to a classification that whether a basic block exists a bug, but the key point is that the model predicts the probability of the vulnerability rather than the existence of the vulnerability. Network architecture is illustrated in Fig. 5.

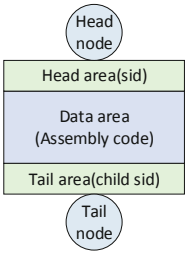


Fig. 4. Suree structure

Algorithm 1. Marking Basic Block as Security/Vulnerable

```
for each ea ∈ readLines(HeadNode, TailNode) do
    String stringToJudge = getCommentString(ea)
    Bool state = regular.match(r'bad', stringToJudge)
    if state then
        label = vulnerable
        return label
    end if
end for
label = security
return label
```

Since the model is used to predict the vulnerability of each basic block, we add the dense layer at the model exit. The dense layer contains 24 input units and 2 output nodes, which could map 24-dimensional input into 2-dimensional vector of output. The activation function is softmax, and the output P is the possibility we want:

$$P = [p_{safe}, p_{vul}] \tag{1}$$

When the batch size is 128, and the num of hidden layers is 2, the model structure is as Fig. 5. In order to use our prediction model, we need to train those parameters into suitable values. We use *cross entropy loss function* for parameter estimation, the equation is as follow:

$$loss_i = -[y_i * \log p_i + (1 - y_i) * \log(1 - p_i)] \tag{2}$$

where y_i is actual label of i-th data, p_i is the possibility of containing vulnerabilities.

Our parameters can be learned by optimizing Eq. 3 with Stochastic Gradient Descent methods:

$$\min_{W_1, W_2, W_3, \ldots, b_1, b_2, b_3, \ldots} \sum_{i=1}^{n} loss_i \tag{3}$$

where W, b is the parameters need to be updated, n is the number of training data. And finally, the vulnerability prediction model is completed.

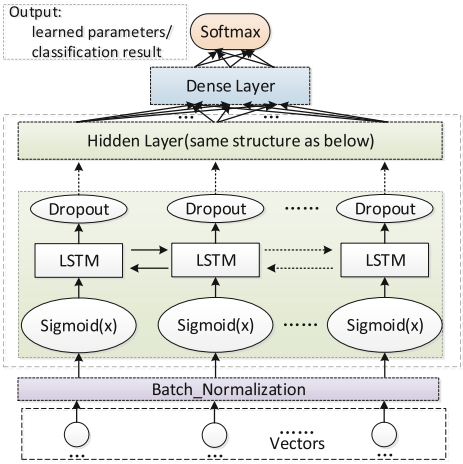


Fig. 5. Prediction model structure

5 Weight-Conversion Module

Weight-conversion module could convert the vulnerability prior probability of the prediction model output to a specific vulnerability score. It is the connection

between the prediction model and the fuzzing loop. Our conversion rules are indicated below.

$$Score(b_i) = \begin{cases} \mu * Pred(b_i) & \text{If } \mu * \text{Pred}(\text{b}_\text{i}) > \xi; \\ \xi & \text{Otherwise;} \end{cases} \tag{4}$$

When a basic block is judged to be vulnerable (the predicted value is 1), μ means the boundary of basic block score. We compared different threshold in 64/128/256/512 to find a better choice. And find when μ=256, Suzzer performs better. Since the prediction model is limited by data size or compute rule, when a basic block is judged to be safe ($Score(b_i) \leq \xi$), it still could cause bugs. So we set $\xi = 1$ to ensure those score bigger than 0 to reduce the shortcomings of the prediction model.

Since the importance of input i depends on the executed path's fitness score f_i, and the path is composed by basic blocks with $Score(b_i)$. We use the method in VUzzer [10] to calculate the fitness score and make it easier to evaluate performance.

$$f_i = \begin{cases} \dfrac{\sum_{b_i \in BB} \log(Freq(b_i)) Score(b_i)}{\log(l_i)} Num(BB) & \text{If } \text{l}_\text{i} > \text{LMAX}; \\ \displaystyle\sum_{b_i \in BB} \log(Freq(b_i)) Score(b_i) & \text{Otherwise;} \end{cases} \tag{5}$$

where BB means a set of basic blocks in the path executed by input i. l_i is the length of input i. $Freq(b_i)$ is the frequency of basic block b executed by i. $LMAX$ is a preconfigured limit on input length to balance effects and loss. The fitness calculation is an important part of our vulnerability-guided fuzzing.

6 Fuzzing Loop

Fuzzing loop is the official start part of bug detection. The design and details of evolutionary fuzzing are orthogonal to this work and covered in [10] for VUzzer. The fuzzing step is as follows (Fig. 6):

- Step 1: The fuzzer receive magic bytes and basic block weight from static analysis and prediction model to initialize.
- Step 2: Then fuzzer send initial input seeds to target program. Similar to most mutation-based fuzzers, it contains a seed pool to hold high-quality inputs. The quality is judged by Eq. 5, fuzzer select top K inputs into the seed pool. K is decided from fuzzing process number.
- Step 3: The mutation module uses crossover/mutate inputs from seed pool, and feeds those testcases to the target program for fuzzing.
- Step 4: The BB monitoring module would monitor the target program. It uses Pintool [34] to trace executed basic blocks and implements dynamic taint analysis based on DataTracker [33] to get cmp-instruction, hooks, etc., and send those information to seed pool.
- Step 5: Loop it until the program crashes.

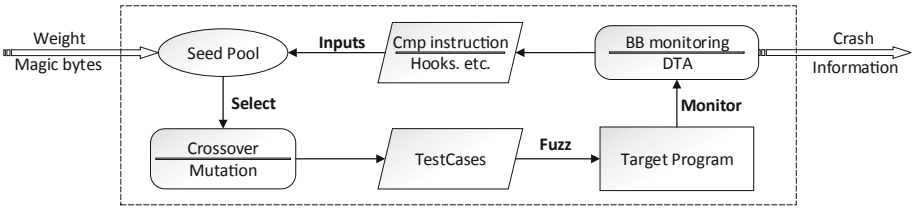


Fig. 6. Fuzzing loop workflow

7 Experimental Evaluation

In this section, we evaluate Suzzer with respect to the accuracy of the prediction model and the effectiveness of fuzzing. First, we briefly describe the data sets used as our train set and test set. Second, we conducted a comparative experiment on the determination of hyperparameters in the existing prediction model, and determined the conclusion that the prediction model can achieve a better result on the test set when the super-parameters have different values. Finally, we put the Suzzer into the LAVA-M [36] dataset and under realistic conditions for testing, and compare the test results with other state-of-art fuzzers.

7.1 Experiment Setup

Prediction Experiment. The prediction model is divided into two parts: ACFG data extraction and model building. For the ACFG extraction, we write the plugin to the disassembler tool IDA Pro in python, and open 30 threads for batch extraction. For the model building, we implement the neural network in Tensorflow in Python.

Our experiments were conducted on a server with 64 GB memory, 40 cores at Intel®Xeon®CPU E5-2640 2.40 GHz, 500 GB hard drives and two GeForce GTX 1080 TI GPU cards.

Fuzzing Experiment. Our fuzzer is improved on the existing fuzzer, VUzzer, and transform coverage-based fuzzing into vulnerability-guided fuzzing.

We conduct the fuzzing test on a virtual machine with Ubuntu 14.04 LTS. The virtual machine is configured with 8 GB memory, 4 cores at 2.40 GHz Intel CPU, 40 GB Storage.

7.2 Data Preparation

Collecting Programs. Since there are not any available data in software vulnerability prediction area, we propose the dataset for evaluating programs and detecting bugs. The dataset is derived from the National Institute of Standards and Technology (NIST) [35], including 83 CWE-ID from category CWE-14 to CWE-911. Each ID class contains several programs written by C/C++, we delete

CWE-364 (signal-handler-race-condition), CWE-365 (race-condition-in-switch), etc., which cannot trigger by fuzzing. Our data set contains a total of 100,883 programs.

7.3 Data Preprocessing

The ACFG data extracted from the binary software cannot be directly input into the model for prediction, because in a software, the basic blocks containing the vulnerability only accounts for a small proportion of all the basic blocks, which leads to data unbalance. For example, assuming that the basic blocks of the vulnerability in the dataset accounts for 10% (which is already a fairly high percentage in normal software), then even if the model predicts all basic blocks as safe (means the model has no classification ability), the model's accuracy remains Up to 90%, that is, the learning space is small. Therefore, we should try to balance the number of basic blocks in the data set that contain vulnerabilities or security.

By Sect. 4.2, 100,883 binary programs are extracted by static analysis and the results are as follows:

```
Before Cleanning data, positive = 1373452
Before Cleanning data, acfg's shape = (24, 6264213)
Before Cleanning data, label's shape = (2, 6264213)
```

More than six million basic blocks have been extracted. The data set is adequate. Therefore, the method of randomly removing the

```
After Cleanning data, positive = 1373452
After Cleanning data, acfg's shape = (24, 2746905)
After Cleanning data, label's shape = (2, 2746905)
```

After culling, it still contains more than two million sets of feature data. We select 99% of them as the training set, and 1% of them as the test set.

7.4 Hyperparametric Analysis

In the process of deep learning, there are many hyperparameters and optimization methods to choose. During our vulnerability prediction model, it's a process based on experiment and experience to achieve better choice for hyperparameters. We compared batch size, hidden layer units and hidden layer depth in the model structure Fig. 5, the performance is showed in Fig. 7.

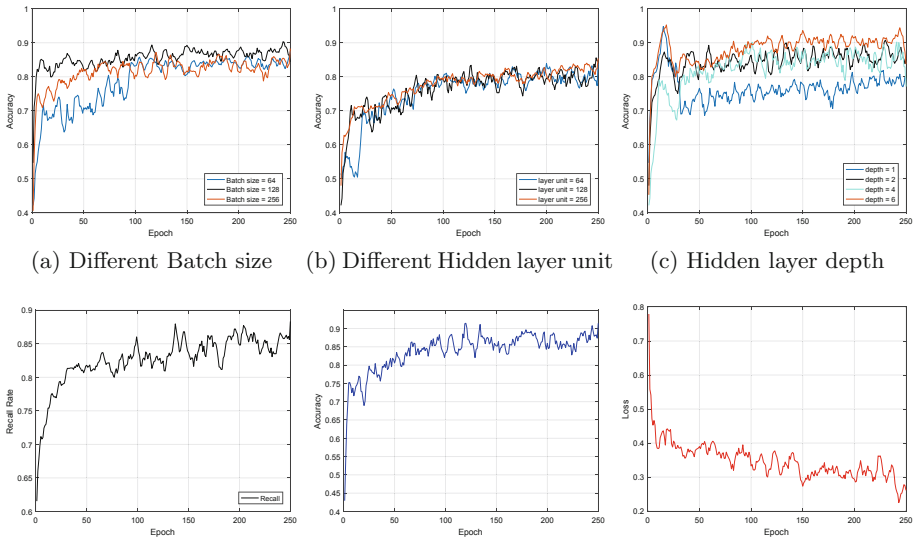
Batch Size: Figure 7a shows the impact of different batch size of 64, 128, 256. Although the accuracy curve of batch size 64,256 reached the same level after epoch 100, the accuracy of batch size 256 is more stable than 64. We set batch size to 128 because it performs best in terms of accuracy level and stability.

Hidden Layer Units: Figure 7b shows the different hidden layer units in 64,128,256. We can see that the number of hidden layer unit has little influence on the accuracy of the model. As the neural network should be fat enough to fit the parameters, we choose to set hidden layer units as 128.

Hidden Layer Depth: Figure 7c shows the accuracy curve in different hidden layer depth 1,2,4,6. From the figure, we can observe the accuracy performs best when the layer depth equals 6, while single layer networks fared worst, with 2 and 4 equally well. Regardless of the fact that the accuracy of 2 layers is not as good as 6 layers, we choose to set the layer depth of our model as 2 to balance the accuracy, training rate and hardware memory requirements.

When we set hyperparameter to the one discussed above, our model's performance is showed in Fig. 7d, e, f. As our prediction model can be seen as *dualistic classification* in essence, that is, the basic block in the target software is judged to be safe or vulnerable. Therefore, in addition to the accuracy and loss curve, we additionally show recall curve in Fig. 7d. In our model, recall is the fraction of the test sets that are successfully classified.

Obviously, curves together show that as the number of training epoches increases, accuracy and recall are gradually increasing, and loss curve is decreasing. This indicates that the performance of the model is getting better gradually, the prediction error rate of the model on positive and negative samples is reduced, and tends to be stable. And finally, model achieves $accuracy = 90.152\%$, $loss = 0.2137$, $recall = 87.62\%$, and 4.71% of false positive rate, which is a great result in software vulnerability determination that even humans cannot easily determine.



(a) Different Batch size (b) Different Hidden layer unit (c) Hidden layer depth

(d) Recall Curve (e) Accuracy Curve (f) Loss Curve

Fig. 7. Model performance. The accuracy curve when choose different hyperparametric and final model performance.

7.5 Fuzzing Performance

In order to measure the performance of our vulnerability-guided fuzzing, we choose to test Suzzer on two different datasets: A: LAVA-M, B: Real world Binary.

A: LAVA-M Dataset. Dolan-Gavitt *et al.* [36] designed a system to help measuring miss and false alarm rates. LAVA is a technique for manually injecting bugs into real programs, each bug in LAVA-M has a unique ID and the ID would be printed before binary is crashed by that bug. This makes LAVA a great benchmark and we use the dataset to evaluate Suzzer, LAVA-M consists of four linux binaries–*base*64, *who*, *uniq*, *md*5*sum*. Since *md*5*sum* is proved to be not suitable for VUzzer, we remove it.

In order to improve readability, we present the results from the original LAVA paper and 24-h fuzzing in Table 2. The second column in Table 2 shows the total number of injected bugs in LAVA-M, and the third and fourth columns respectively indicate the number of unique bugs triggered by VUzzer and Suzzer for 24-h fuzzing. Each bug in LAVA-M is triggered with a unique ID, and the fault IDs from Suzzer are listed in Table 3.

Obviously, Suzzer's performance is not inferior to VUzzer in 24-h fuzzing, and even better than VUzzer in *uniq* program. Furthermore, as a vulnerability-guided fuzzer, Suzzer is designed to find more bugs in a shorter time and reducing the resource consumption of fuzzing, rather than promoting the code coverage and tries to explore all parts of the target program. Therefore, it's more meaningful to evaluate Suzzer in short-term fuzzing.

Table 2. Number of faults from LAVA paper, suzzer and VUzzer.

Program	Total bugs	VUzzer	Suzzer
uniq	28	3	5
base64	44	18	18
who	2136	44	40

Since four hours are the effective working time(half day) that penetration testers can work continuously, we choose it as the time baseline for our short-term fuzzing. As showed in Fig. 8, Suzzer can indeed find more crashes than VUzzer, especially in terms of timeliness. Table 4 indicates that Suzzer takes 59%, 45.25%, 19% less time to achieve the same crashes which cost VUzzer 4 h in LAVA-M dataset.

Table 3. Fault IDs detected by Suzzer on the LAVA-M dataset.

Program	Fault IDs
uniq	130, 112, 222, 166, 227
base64	1, 582, 843, 841, 222, 386, 831, 284, 784, 806, 805, 278, 584, 276, 583, 235, 790
who	4356, 60, 3798, 3997, 83, 159, 138, 149, 5, 18, 58, 4355, 4364, 10, 9, 6, 7, 22, 1, 14, 79, 75, 3, 26, 89, 8, 81, 4358, 4, 4166, 16, 2, 3968, 20, 4195, 12, 3967, 56, 87

Overall, Suzzer performs well in artificial LAVA datasets. It can detect more vulnerabilities within four hours without decreasing the fuzzing performance of 24-h. We now advance to evaluate Suzzer in real-wrold programs, which is also inspected by other fuzzers.

B: Real-World Dataset. We evaluated Suzzer in a set of real world programs (gif2png, mpg321, tcptrace, pdf2svg, xmlwf, jhead). For each program, we used VUzzer and AFL-QemuMode to fuzz in the same environment (seed selection, etc.) for comparison. We also targeted some dependency libraries and extract some useful information(magic bytes, etc.) to make our real-world evaluation produce progress, such as libpng, libjpeg, libpcap, libpoppler and libexpat.

Figure 8 shows the distribution of crashes within four hours. As showed in this figure, Suzzer and VUzzer can continuously discover crashes in fuzzing, whereas AFL-Qemu performs poorly and can only occasionally trigger 1 or 2 crashes. Unfortunately, those three fuzzers failed to find vaild crashes in xmlwf and jhead. In order to find the reason, we recorded the average fitness input score of programs and find score in jhead&xmlwf(9183, 8468, etc.) is several orders of magnitude lower than others (at least 304457), which may because the program requires a specific set of actions, while our fuzzing strategy does not include it. We intend to fix it in future research.

Based on the preceding analysis, we can see that Suzzer is significantly better than VUzzer, AFL-Qemu in both artificial and real-world programs. As showed in Fig. 8 and Table 4: In time consuming, Suzzer can reduce up to 64.5% of the time to discover vulnerabilities.

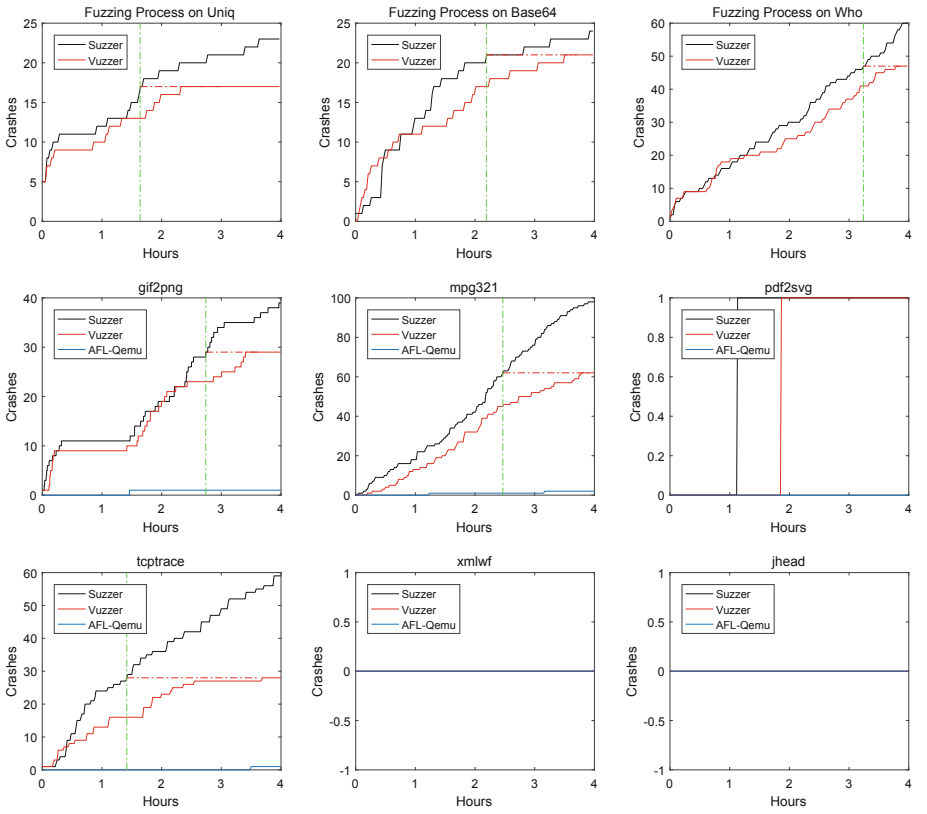


Fig. 8. Distribution of crashes in LAVA-M and real-world programs. (X-axis: The cumulative number of crashes per minute. Y-axis: the specific crashes number. Green line: Time taken by Suzzer to find the same number of crashes as those found by VUzzer during a complete run.) (Color figure online)

8 Related Works

In the preceding sections, we have already stated the vulnerability-guided fuzzing technology. In this section, we investigated additional research work in fuzzing field. This makes the differences and characteristics between our research and the existing work more clearly.

Table 4. Time comparison between Suzzer and VUzzer

TIME (h)	uniq	base64	who	gif2png	mpg321	pdf2svg	tcptrace	xmlwf	jhead
Suzzer	1.64	2.19	3.24	2.74	2.47	N/A	1.42	N/A	N/A
VUzzer	4.0	4.0	4.0	4.0	4.0	N/A	4.0	N/A	N/A
Time saving ratio	59%	45.25%	19%	31.5%	38.25%	N/A	64.5%	N/A	N/A

8.1 Code-Coverage Based Fuzzing Approaches

Existing state-of-the-art fuzzers propose different methods of improving code coverage. For example, AFL [6] generates different inputs to traverse various paths to crash a program, while Angora [12] thinks such method would fail to distinguish the executions of the same branch in different contexts and may overlook new internal states of the program. Angora uses the method of context-sensitive branch coverage to improve code coverage. T-fuzz [11] thinks the main limitation on code coverage is conditional constraint, and removing sanity checks in the target program to increase code coverage. CollAFL [13]'s strategy is providing more accurate coverage information to mitigate path collisions. Those methods are proposed to enhance fuzzing comprehensiveness.

8.2 Deep Learning in Software Area

Deep learning is one of the hottest technologies in the world. It is widely used in computer vision and natural language [24] processing areas but have not yet matured in fields like software. There are some examples of exploration.

For deep learning granularity, Li et al. propose converting the program into a vector representation in the form of SyVC [21]. SyVC is a code snippet consisting of 5 to 10 lines of assembly code, and these codes can be discontinuous, but need to have a semantic relationship, such as operating on the same variable. Xu *et al.* [23] propose to use software functions as the granularity to detect cross-platform binary code similarity. Rajpal *et al.* [29] use input bytes to get higher effect execution seed files. Zuo *et al.* [24] also proposes a method for calculating software similarity, which treats assembly instructions after disassembly of binary programs as plain text.

For model selection, V-Fuzz [20] and VulDeePecker [22] both use neural networks to pre-analyze a large amount of program data to get which parts of the program have higher vulnerability rate, and based on this, the software parts' vulnerability prior probability is predicted. V-fuzz is separated by functions, with graph embedding network [28] as the structure of the vulnerability prediction model. VulDeePecker propose using code gadgets to represent programs, and utilizing long-short-term-memory units to predict vulnerability possibility.

9 Conclusions

In this paper, we argued that coverage-based fuzzers cannot find vulnerabilities efficiently and cost too much computing resources. We proposed a vulnerability-guided fuzzing solution Suzzer, which assigns the priority of input in the state-of-art fuzzer VUzzer, to focus testing on code blocks that are more likely to contain bugs.

In our prototype implementation of Suzzer, we detect which part of the target program has a higher probability of vulnerability by the prediction model. Our prediction results will enter the fuzzing loop through the weight conversion

module. This change allows our fuzzer to produce input that priority trigger the vulnerable part and therefore find vulnerabilities faster.

Experiments showed that this solution performs better in short-term fuzzing than other fuzzers. Compared to other state-of-the-art fuzzers (AFL, VUzzer), Suzzer can reduce the time to discover vulnerabilities in most cases, and, furthermore, can detect more crashes with smaller basic block input. This demonstrates that the combination of deep learning and fuzzing is indeed a viable strategy, but there are still many problems that need to be addressed in future research, such as the limitations of datasets.

Acknowledgements. This research is supported in part by the National Key Research and Development Project (Grant No. 2017YFC0820503) and Beijing Science and Technology Plan (Grant No. Z181100009818020).

References

1. Wikipedia. Fuzzing (2018). https://en.wikipedia.org/wiki/Fuzzing/
2. Roning, J., et al.: Protos-systematic approach to eliminate software vulnerabilities. Invited presentation at Microsoft Research (2002)
3. Eddington, M.: Peach fuzzing platform. Peach Fuzzer **34** (2011)
4. Aitel, D.: An introduction to spike, the Fuzzer creation kit (2002)
5. Yang, X., Chen, Y., Eide, E., et al.: Finding and understanding bugs in C compilers. In: ACM SIGPLAN Notices (2011)
6. Zalewski, M.: American fuzzy lop (2017). http://lcamtuf.coredump.cx/afl/
7. Google. honggfuzz (2017). https://google.github.io/honggfuzz/
8. Caca Labs (2017). http://caca.zoy.org/wiki/zzuf/
9. Chen, Y., et al.: EnFuzz: ensemble fuzzing with seed synchronization among diverse Fuzzers. In: 28th USENIX Security Symposium (USENIX Security 2019) (2019)
10. Rawat, S., et al.: VUzzer: application-aware evolutionary fuzzing. In: NDSS, vol. 17 (2017)
11. Peng, H., Shoshitaishvili, Y., Payer, M.: T-Fuzz: fuzzing by program transformation. In: 2018 IEEE Symposium on Security and Privacy (SP). IEEE (2018)
12. Chen, P., Chen, H.: Angora: efficient fuzzing by principled search. In: 2018 IEEE Symposium on Security and Privacy (SP). IEEE (2018)
13. Gan, S., et al.: Collafl: path sensitive fuzzing. In: 2018 IEEE Symposium on Security and Privacy (SP). IEEE (2018)
14. Shin, Y., Williams, L.: Can traditional fault prediction models be used for vulnerability prediction? Empirical Softw. Eng. **18**(1), 25–59 (2013)
15. Liu, C., et al.: SOBER: statistical model-based bug localization. ACM SIGSOFT Softw. Eng. Notes **30**(5), 286–295 (2005)
16. OpenRCE. Sulley fuzzing framework (2015). https://github.com/OpenRCE/sulley
17. Takanen, A., Demott, J., Miller, C.: Fuzzing for Software Security Testing and Quality Assurance. Artech House (2008)
18. Godefroid, P., Levin, M., Molnar, D.: Automated whitebox fuzz testing. In: Network and Distributed System Security Symposium (2008)
19. Ganesh, V., Leek, T., Rinard, M.: Taint-based directed whitebox fuzzing. In: Proceedings of the 31st International Conference on Software Engineering. IEEE Computer Society (2009)

20. Li, Y., et al.: V-Fuzz: vulnerability-oriented evolutionary fuzzing. arXiv preprint arXiv:1901.01142 (2019)
21. Li, Z., et al.: SySeVR: a framework for using deep learning to detect software vulnerabilities. arXiv preprint arXiv:1807.06756 (2018)
22. Li, Z., et al.: VulDeePecker: a deep learning-based system for vulnerability detection. arXiv preprint arXiv:1801.01681 (2018)
23. Xu, X., et al.: Neural network-based graph embedding for cross-platform binary code similarity detection. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. ACM (2017)
24. Zuo, F., et al.: Neural machine translation inspired binary code similarity comparison beyond function pairs. arXiv preprint arXiv:1808.04706 (2018)
25. Feng, Q., et al.: Scalable graph-based bug search for firmware images. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM (2016)
26. Control-flow graph (2015). https://en.wikipedia.org/wiki/Control-flow_graph
27. Pewny, J., et al.: Cross-architecture bug search in binary executables. In: 2015 IEEE Symposium on Security and Privacy. IEEE (2015)
28. Yan, S., et al.: Graph embedding and extensions: a general framework for dimensionality reduction. IEEE Trans. Pattern Anal. Mach. Intell. **29**(1), 40–51 (2007)
29. Rajpal, M., Blum, W., Singh, R.: Not all bytes are equal: Neural byte sieve for fuzzing. arXiv preprint arXiv:1711.04596 (2017)
30. Gers, F.A., Schmidhuber, J., Cummins, F.: Learning to forget: continual prediction with LSTM (1999)
31. Intel. Intel 64 and IA-32 architectures software developer manuals (2018). https://software.intel.com/en-us/articles/intel-sdm
32. Hex-Rays. The IDA pro disassembler and debugger (2015). https://www.hex-rays.com/products/ida/
33. Stamatogiannakis, M., Groth, P., Bos, H.: Looking inside the black-box: capturing data provenance using dynamic instrumentation. In: Ludäscher, B., Plale, B. (eds.) IPAW 2014. LNCS, vol. 8628, pp. 155–167. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-16462-5_12
34. Luk, C.-K., et al.: Pin: building customized program analysis tools with dynamic instrumentation. ACM SIGPLAN Not. **40**(6), 190–200 (2005)
35. NVD (2017). http://nvd.nist.gov/
36. Dolan-Gavitt, B., et al.: Lava: large-scale automated vulnerability addition. In: 2016 IEEE Symposium on Security and Privacy (SP). IEEE (2016)