V-Fuzz: Vulnerability Prediction-Assisted Evolutionary Fuzzing for Binary Programs

Yuwei Li, Shouling Ji, *Member, IEEE*, Chenyang Lyu, Yuan Chen, Jianhai Chen, *Member, IEEE*, Qinchen Gu, *Member, IEEE*, Chunming Wu, and Raheem Beyah, *Senior Member, IEEE*

***Abstract*—Fuzzing is a technique of finding bugs by executing a target program recurrently with a large number of abnormal inputs. Most of the coverage-based fuzzers consider all parts of a program equally and pay too much attention to how to improve the code coverage. It is inefficient as the vulnerable code only takes a tiny fraction of the entire code. In this article, we design and implement an evolutionary fuzzing framework called V-Fuzz, which aims to find bugs efficiently and quickly in limited time for binary programs. V-Fuzz consists of two main components: 1) a *vulnerability prediction model* and 2) a *vulnerability-oriented evolutionary fuzzer*. Given a binary program to V-Fuzz, the vulnerability prediction model will give a prior estimation on which parts of a program are more likely to be vulnerable. Then, the fuzzer leverages an evolutionary algorithm to generate inputs which are more likely to arrive at the vulnerable locations, guided by the vulnerability prediction result. The experimental results demonstrate that V-Fuzz can find bugs efficiently with the assistance of vulnerability prediction. Moreover, V-Fuzz has discovered ten common vulnerabilities and exposures (CVEs), and three of them are newly discovered.**

***Index Terms*—Fuzz testing, graph embedding, vulnerability prediction.**

I. Introduction

FUZZING is an automated vulnerability discovery technique by feeding manipulated random or abnormal inputs to a target program [1]. With the rapid improvement of computer performance, fuzzing has been widely used by industrial software vendors, such as Google [2] and Microsoft [3] for detecting bugs in software. However, the efficiency of fuzzing is still badly in need of being improved, especially when detecting bugs in relatively limited time.

How to improve the effectiveness of fuzzing is a popular research field. One of the most important research directions is coverage-based fuzzing which is focused on how to improve the code coverage of the target application. The intuitions of coverage-based fuzzers can be summarized as, if a fuzzer can cover more paths/branches of the target program, it means the program is tested more thoroughly and has a higher probability of finding vulnerabilities. Nevertheless, it is not appropriate to treat all components of the target program equally when aiming at finding bugs efficiently in limited time. The reasons are as follows. First, the vulnerable code usually takes a tiny fraction of the entire code. For instance, Shin and Williams [4] found that only 3% of the source code files in Mozilla Firefox are vulnerable. Although a coverage-based fuzzer can improve the coverage, most of the improved coverage may not be helpful in finding bugs. Second, there are still many difficulties in achieving relatively high coverage. For most of the mutation-based fuzzers, it is difficult for them to generate valid inputs which can reach paths that are relatively complicated, for example, a path contains magic bytes. For hybrid fuzzers [5], which utilize symbolic execution to generate test cases that can satisfy path constraints, they usually face the path explosion problems when fuzzing the real-world programs. Third, in practice, especially in industrial scenarios, the time for fuzzing is usually limited. Therefore, it is crucial to seek a balance between efficiently detecting bugs and exploring more paths. When fuzzing time is limited, fuzzers should prioritize the suspiciously vulnerable components of the target program instead of blindly persuing high coverage.

Based on the above motivation, we propose V-Fuzz, which aims to find bugs quickly in limited time. In particular, V-Fuzz is a binary-oriented fuzzing framework that can test a target program without its source code. Note that binary-oriented fuzzing is significant in practice, as the source code of a program (e.g., commercial software) is not always available. Nevertheless, most existing fuzzers [6]–[13] need the source code and the binary-oriented fuzzers are few [14]–[16]. One type of binary-oriented fuzzing is black-box fuzzing [14], which does not need knowledge of the target program. Nevertheless, it is usually inefficient. The other type is

Manuscript received November 26, 2019; revised May 21, 2020; accepted July 25, 2020. This work was supported in part by NSFC under Grant U1936215, Grant U1836202, and Grant 61772466; in part by the National Key Research and Development Program of China under Grant 2018YFB0804102 and Grant 2020YFB1804705; in part by the Zhejiang Provincial Natural Science Foundation for Distinguished Young Scholars under Grant LR19F020003; in part by the Zhejiang Provincial Key Research and Development Program under Grant 2019C01055 and Grant 2020C01021; in part by the Industrial Internet Innovation and Development Project under Grant TC190A449; and in part by the Major Scientific Project of Zhejiang Lab under Grant 2018FD0ZX01. This article was recommended by Associate Editor P. P. Angelov. *(Yuwei Li and Shouling Ji are co-first authors.) (Corresponding authors: Shouling Ji; Chunming Wu.)*

Yuwei Li, Chenyang Lyu, Yuan Chen, Jianhai Chen, and Chunming Wu are with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China (e-mail: liyuwei@zju.edu.cn; puppet@zju.edu.cn; chenyuan@zju.edu.cn; chenjh919@zju.edu.cn; wuchunming@zju.edu.cn).

Shouling Ji is with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China, and also with the School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332 USA (e-mail: sji@zju.edu.cn).

Qinchen Gu and Raheem Beyah are with the School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332 USA (e-mail: qgu7@gatech.edu; raheem.beyah@ece.gatech.edu).

Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCYB.2020.3013675

gray-box fuzzing [15], [16], which generally conducts analysis on the transformed binaries (e.g., disassembled binaries) to guide the fuzzing process and thus is more efficient. However, most existing gray-box binary-oriented fuzzers [15], [16] are coverage-based, which do not pay enough attention to the vulnerable code. To improve the efficiency in discovering bugs for binary programs assisted by vulnerability prediction, V-Fuzz leverages an evolutionary algorithm to generate inputs that tend to arrive at the vulnerable code of a target program. It needs to be emphasized that V-Fuzz is neither coverage-based nor directed. V-Fuzz is different from most coverage-based fuzzers which regard all codes equally since it pays more attention to the code which has a higher probability to be vulnerable. From this perspective, V-Fuzz is designed more like a "weighted coverage-based" fuzzer. In addition, unlike directed fuzzers [10], [12], which generate inputs with the objective of reaching a given set of target program locations, V-Fuzz gives relatively small weights to other code which are unlikely to be vulnerable. This is because the vulnerability prediction model may not always be accurate and the components that are predicted to be safe may still be vulnerable. Therefore, V-Fuzz leverages the advantages of vulnerability prediction and fuzzing, and meanwhile, reduces the disadvantages of them.

In summary, our contributions are the following.

1) We propose V-Fuzz, a fuzzing framework that combines vulnerability prediction with evolutionary fuzzing.
2) We design and implement a vulnerability prediction model based on graph neural networks. The model is able to predict the vulnerable probability (VP) of each function of a target binary program.
3) To examine the performance of V-Fuzz, we conduct extensive evaluations leveraging ten popular Linux applications and three programs of the popular fuzzing benchmark LAVA-M [17]. The results demonstrate that V-Fuzz is efficient in discovering bugs for binary programs. Moreover, we discovered ten common vulnerabilities and exposures (CVEs) by V-Fuzz, among which, three were newly discovered. We reported the new CVEs, and they have been confirmed and fixed.

II. V-Fuzz: System Overview

In this section, we introduce the main components and workflow of V-Fuzz. Fig. 1 shows the architecture of V-Fuzz, which consists of two main components: 1) a *neural network-based vulnerability prediction model* and 2) a *vulnerability-oriented evolutionary fuzzer*.

Vulnerability Prediction Model: This component is to give a prior static analysis on the target program to find which components are more likely to be vulnerable. There are two main approaches for vulnerability prediction. One is using traditional static vulnerability detection methods. The other is leveraging machine-learning or deep-learning techniques. Among the two approaches, we choose to use deep learning to build the vulnerability prediction model due to the following reasons. First, most of the traditional static analysis methods

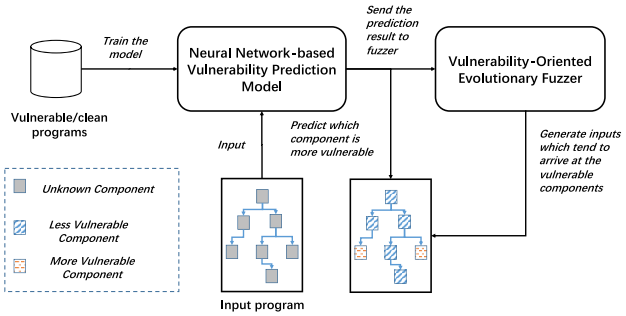


Fig. 1. Architecture of V-Fuzz.

use pattern-based approaches to detect vulnerabilities. The patterns are manually defined by security experts. This procedure is difficult, tedious, and time consuming. Although there exist many static analysis tools, such as flawfinder [18] and RATS [19], these tools are used to detect vulnerabilities on source code. As in this article, we mainly focus on detecting vulnerabilities of binary programs. Therefore, these tools are not suitable for us. Second, deep learning has been successfully applied not only in the traditional fields, such as image classification [20], object detection [21], natural language processing [22], and recommendation system [23] but also in the fields of software security [24]–[28]. In these applications, deep learning has several advantages when compared with pattern-based methods: 1) deep-learning methods do not need experts to define the features or patterns, which can reduce lots of overhead; 2) even for the same type of vulnerabilities, it is hard to define an accurate pattern that can describe all forms of it; and 3) pattern-based methods usually can only detect one specific type of vulnerability while deep learning methods have been proven to be able to detect several types of vulnerabilities simultaneously [28]. Therefore, we choose to leverage deep-learning methods to build our vulnerability prediction model. Moreover, it is worth noting that there has been no such approach that leverages deep learning to detect or predict vulnerabilities for binary programs to the best of our knowledge.

Specifically, we design the vulnerability prediction model based on a graph embedding network. In this model, we regard each function of a binary program as a graph and the model will output the VP of it. The prediction results will be used to assist a fuzzer in discovering bugs. The details of this model are presented in Section III.

Vulnerability-Oriented Evolutionary Fuzzer: Based on the previous vulnerability prediction result, the fuzzer will assign more weight to the functions that have higher vulnerable probabilities. The process is as follows: for each function of the binary program with a VP, V-Fuzz will give each basic block in the function of a static vulnerable score (SVS), which represents the importance of a basic block. The detailed scoring method is described in Section IV. Then, V-Fuzz starts to test the program with some initial inputs provided by users. It implements an evolutionary algorithm to generate proper inputs. For each executed input, V-Fuzz gives a *fitness score* for it, which is the sum of the SVS of all basic blocks that are on its execution path. Then, the inputs that have higher

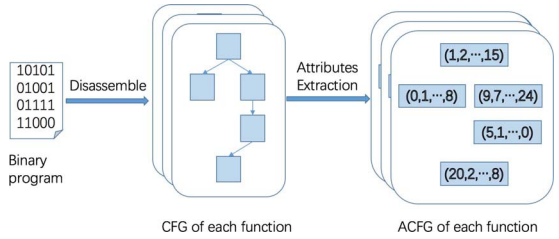


Fig. 2. Workflow of data preprocessing.

fitness scores or cause crashes will be selected as new seed inputs. Afterward, new inputs will be continuously generated by mutating the selected seed inputs. In this way, V-Fuzz tends to generate inputs that are more likely to arrive at the vulnerable regions.

III. Vulnerability Prediction

A. Problem Formalization

In this section, we formalize the vulnerability prediction problem. We denote the vulnerability prediction model as M. Given a binary program $\hat{p}$, suppose it has τ functions $F = \{f_1, f_2, \ldots, f_\tau\}$. For any function $f_i \in F$, it is an input of M, and the corresponding output VP_{f_i} denotes the VP of f_i, that is

$$VP_{f_i} = M(f_i). \tag{1}$$

In the following, we will explain the details of the model M from three aspects: 1) *the representation of input data*; 2) *the model structure*; and 3) *how to train and use the model*.

B. Data Preprocessing

As discussed in Section II, to build and train M, we should seek a method to transform binary program functions into numerical vectors. Moreover, the vectors should be able to carry enough information for future training. Toward this, we choose to use the attributed control flow graph (ACFG) [29] to represent a binary function.

ACFG is a directed graph $g = <V, E, \phi>$, where V is the set of vertices, E is the set of edges, and ϕ is a mapping function. In ACFG, a vertex represents a basic block, an edge represents the connection between two basic blocks, and $\phi : V \to \sum$ maps a basic block in g to a set of attributes $\sum$.

As we know, it is common to use control flow graph (CFG) to find bugs [30], [31]. However, CFG is not a numerical vector, which means it cannot be used to train a deep learning model directly. Fortunately, ACFG is another form of CFG by describing CFG with a number of basic block-level attributes. In ACFG, each basic block is represented by a numerical vector, where each dimension of the vector denotes the value of a specific attribute. In this way, the entire binary function can be represented as a set of vectors. Therefore, ACFG is suitable for our requirements to represent a binary function.

Now, we show how to vectorize a binary program. Fig. 2 shows the workflow of data preprocessing. First, we disassemble the binary program to obtain the CFGs of its functions. Then, we extract attributes for basic blocks and transform each basic block into a numerical vector. The attributes are used to characterize a basic block, and they can be statistical, semantic, and structural. Here, we only extract the statistical attributes for the following reasons. The first reason is for efficiency. As indicated in [29], the cost of extracting semantic features such as I/O pairs of basic blocks is too expensive. Second, the graph embedding network can learn the structural attributes automatically. We extract 255 attributes in total. Table I shows some examples of all the 255 attributes, and all the instruction type-related attributes can be found in [32, Sec. 5.1]. There are mainly three kinds of attributes: 1) instruction-related attributes; 2) operand-related attributes; and 3) string-related attributes. Then, each basic block can be represented by a 255-D vector, and the binary program now is represented by a set of 255-D vectors.

TABLE I
Used Attributes of a Basic Block

Type	Attributes (Examples)	Total Num
Instructions	# *call* instruction # *cmp* instruction # *mov* instruction	244
Operand	# Void operand # General register operand # Direct memory reference operand	8
Other	# string "malloc" # string "calloc" # string "free"	3
	All attributes num	255

C. Model Structure

Based on the discussion in Section II, we choose to adapt a *graph embedding network* [27], [33] as the core of our vulnerability prediction model. First, we give a brief introduction to the graph embedding network. Then, we detail the design of our model.

Graph embedding is an efficient approach to solve graph-related problems [34], [35] such as node classification. It transforms a graph into an embedding vector that contains sufficient information of the graph for solving corresponding problems. In our scenario, the embedding vectors of a binary function should be able to contain sufficient features for vulnerability prediction. In addition, graph embedding can be considered as a mapping λ, which maps a function's *ACFG* g into a vector $\lambda(g)$.

We use a neural network to approximate the mapping λ, and train the model with vulnerable and secure binary functions to enable the graph embedding network to learn the features related to vulnerabilities. As the vulnerability prediction model is required to output the VP of a binary function, we combine the graph embedding network with a pooling layer and a softmax layer. The pooling layer transforms the embedding vector into a 2-D vector Z, and the softmax-layer maps the 2-D vector Z of arbitrary real values into another 2-D vector Q, where the value of each dimension is in the range [0, 1]. The first dimension represents the VP, which is represented by p. The second dimension represents the secure probability, and naturally, the value is $1 - p$. The entire model is trained by labeled data end to end, and the parameters of the model can be learned by minimizing a loss function.

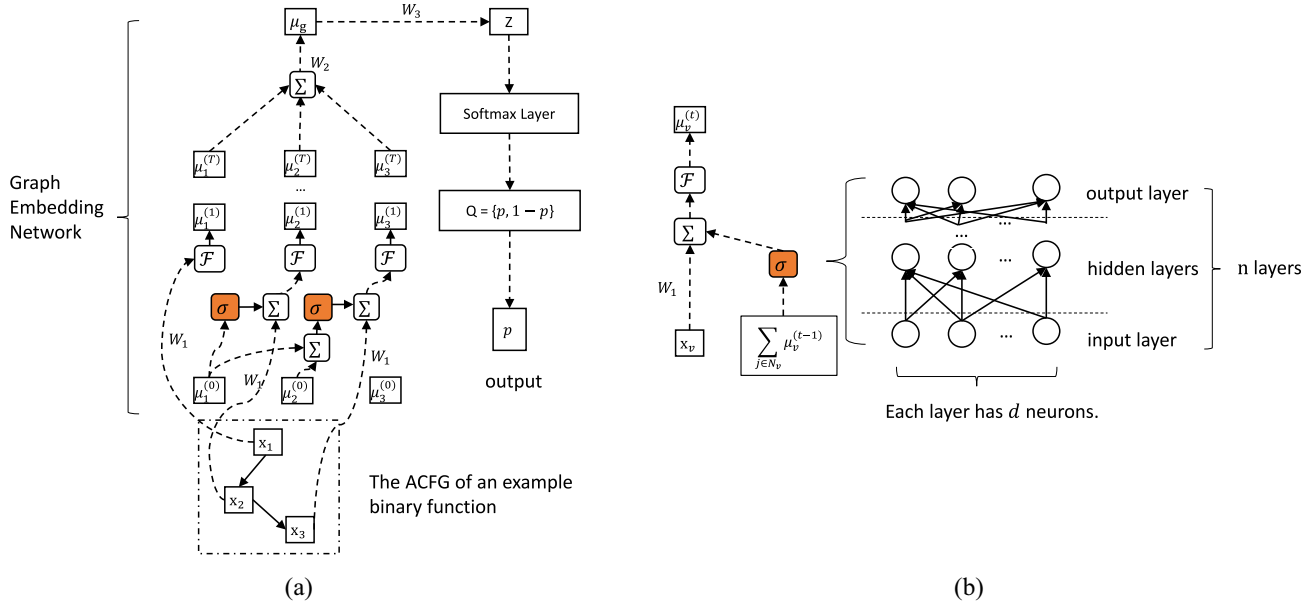


Fig. 3. Structure of the vulnerability prediction model. Fig. 3(a) shows the overview of the graph embedding network. Fig. 3(b) shows the structure of the neural network σ in one iteration t. For the ACFG of an example binary function in Fig. 3(a), in the first iteration, the embedding vector of the basic block is calculated by the following equations: $\mu_1^{(1)} = F(W_1 x_1)$, $\mu_2^{(1)} = F(W_1 x_2 + \sigma((\mu_1)^{(0)}))$, and $\mu_3^{(1)} = F(W_1 x_3 + \sigma((\mu_1)^{(0)} + (\mu_2)^{(0)}))$, which explains the link relationship in this figure. (a) Overview of the graph embedding network. (b) Structure of the neural network σ.

TABLE II
NOTATIONS

Notation	Meaning
a	The number of attributes of a basic block
g	The ACFG of a binary function
V	The set of vertices
E	The set of edges
v	a vertex in V
x_v	The attributed vector of vertex v
d	The dimension of an embedding vector
μ_v	The embedding vector of vertex v
μ_g	The graph embedding vector of ACFG g
n	The number of layers of the fully-connected network σ

Below is the formalization of the model. Table II shows all the notations related to the model, and Fig. 3 presents the structure of the model. The input of the model is the *ACFG* g of a binary program function, $g = <V, E, \phi>$. Each basic block $v \in V$ in ACFG has an attribute vector x_v which can be constructed according to all selected attributes. The number of attributes for each basic block is a. Thus, x_v is an a-D vector. For each basic block v, the graph embedding network computes an embedding vector μ_v, which combines the topology information of the graph. The dimension of the embedding vector μ_v is d. Let N_v be the set of neighboring vertices of v. Since ACFG is a directed graph, N_v can be considered as the set of precursor vertices of v. Then, the embedding vector μ_v can be computed by $\mu_v = F(x_v, \sum_{j \in N_v}(\mu_j))$, where F is a nonlinear function that can be *tanh*, *sigmoid*, etc. The embedding vector μ_v is computed for T iterations. For each iteration t, the temp embedding vector can be obtain by equation $\mu_v^{(t)} = F(W_1 x_v + \sigma(\sum_{j \in N_v}(\mu_j)^{(t-1)}))$, where x_v is an $a \times 1$ vector and W_1 is a $d \times a$ matrix. The initial embedding vector $\mu_v^{(0)}$ is set to zero. After T iterations, $\forall v \in V$, we obtain the final graph embedding vector $\mu_v^{(T)}$. As shown in Fig. 3(b), σ is an n-layer fully connected neural network with parameters $P = \{P_1, P_2, \ldots, P_n\}$. Each layer of σ has d neurons. Let $ReLU(\cdot) = \max\{0, \cdot\}$ be a rectified linear unit. We have $\sigma(x) = P_1 \times ReLU(P_2 \times \cdots ReLU(P_n x))$. After T iterations, we can obtain the final graph embedding vector $\mu_v^{(T)}$ for each vertex $v \in V$. Then, the graph embedding vector μ_g of the ACFG g can be represented by the summation of the embedding vector of each basic block, that is, $\mu_g = W_2(\sum_{v \in V}(\mu_v^{(T)}))$, where W_2 is a $d \times d$ matrix. To compute the VP of the function, we map the graph embedding vector into a 2-D vector $Z = \{z_0, z_1\}$, that is

$$Z = W_3 \mu_g \tag{2}$$

where W_3 is a $2 \times d$ matrix. Then, we use a softmax function to map the values of Z into the vector $Q = \{p, 1 - p\}$, $p \in [0, 1]$, that is

$$Q = \text{Softmax}(Z). \tag{3}$$

The value of p is the output of the model, which represents the VP of the binary program function g.

D. Train and Use the Model

In order to predict the VP, the model needs to be trained with labeled data, where the label is either "vulnerable" or "secure." For the ACFG g of a function, the label l of g is 0 or 1, where $l = 1$ means the function has at least one vulnerability, and $l = 0$ means the function is secure.

The model's training process is similar to a classification model. Then, the parameters of M can be learned by

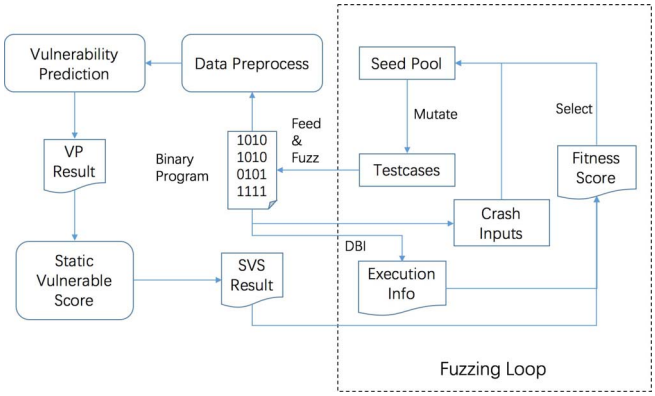


Fig. 4. Overview of vulnerability-oriented fuzzing module of V-Fuzz. DBI: dynamic binary instrumentation; SVS: static vulnerable score; and VP: vulnerable probability.

optimizing the following equation:

$$\min_{W_1,W_2,W_3,P_1,P_2,...,P_n} \sum_{i=1}^{m} (H(Q, l)) \tag{4}$$

where m is the number of training data and H is a cross-entropy loss function. We optimize (4) with a stochastic gradient descent (SGD) method. Although the model's training process is similar to training a classification model, when using the model, the classification result that whether a binary function is vulnerable or not is too coarse grained and not suitable for the subsequent fuzzing. Therefore, we choose to use VP p as the output of the model.

IV. Vulnerability-Oriented Fuzzing

Based on the result from the prediction model, the vulnerability-oriented fuzzer will pay more attention to the functions with higher vulnerable probabilities. Fig. 4 shows the workflow of vulnerability-oriented fuzzing, where V-Fuzz leverages an evolutionary algorithm to generate inputs that tend to arrive at the vulnerable components. Specifically, for a binary program, V-Fuzz uses the data preprocess module to disassemble the binary to obtain the ACFG of each function, which is the input of the vulnerability prediction model. Then, the prediction model will give each binary program function a VP. Based on the *VP* result, each basic block in the program is given an SVS, which will be used later to evaluate the executed test cases.

The fuzzing is a cyclic process. Like most mutation-based evolutionary fuzzers, V-Fuzz maintains a seed pool, which is used to save high-quality inputs as seeds. V-Fuzz starts to execute the binary program with some initial inputs that are provided by users. Meanwhile, it uses dynamic binary instrumentation (DBI) to track the execution information of the program such as basic block coverage. Based on SVS and the execution information, V-Fuzz will calculate a *fitness score* for each executed testcase (i.e., inputs). The testcases with high *fitness scores* are considered as high-quality inputs and will be sent to the seed pool. In addition, the executed testcases which trigger crashes will also be sent to the seed pool, regardless of their fitness scores. The detailed method for calculating *fitness score* is presented in Section IV-B. Next, V-Fuzz generates the next-generation testcases by mutating the seeds in the seed pool. In this way, V-Fuzz continues to execute the program with newly generated inputs until the end conditions are met. Below, we elaborate on the workflow of vulnerability-oriented fuzzing.

A. Static Vulnerable Score

Based on the *VP* result, V-Fuzz gives each basic block an SVS. For a function f, we assume its VP is p_v, and it has ι basic blocks $f = b_1, b_2, \ldots, b_\iota$. For $b_i \in f$, b_i's *SVS*, denoted by $SVS(b_i)$, can be calculated by the following equation:

$$SVS(b_i) = \kappa * p_v + \omega \tag{5}$$

where κ and ω are constant parameters that should be obtained from fuzzing experiments. Hence, the basic blocks that belong to the same function have the same *SVS* values. For parameter κ, we conduct the fuzzing experiments on 64 Linux programs (e.g., some programs in binutils), and 20 of them have crashes. Then, we test these 20 programs individually with the value of $\kappa \in [10, 100]$, and we observe that when $\kappa \in [15, 25]$, the fuzzer performs better on most of the target programs. Therefore, we set $\kappa = 20$ as the default value. For parameter ω, it is used to avoid $SVS = 0$ when functions have very low vulnerable probabilities. As if $SVS(b_i) = 0$, it represents that b_i has no meaning for fuzzing and becomes trivial, which is against our design principle of V-Fuzz. Therefore, we set the value of $\omega = 0.1$, which is small and can make $SVS > 0$ all the time.

Based on the approach of calculating *SVS*, V-Fuzz assigns more weight to the basic blocks that are more likely to be vulnerable, which will further assist the fuzzer in generating inputs that are more likely to cover these basic blocks.

B. Seed Selection Strategy

Algorithm 1 shows the seed selection strategy of V-Fuzz. Specifically, V-Fuzz leverages an evolutionary algorithm to select seeds that are more likely to arrive at the vulnerable components.

After giving every basic block an *SVS*, V-Fuzz enters the fuzzing loop. During each loop, V-Fuzz monitors the program to check if it has exceptions such as crashes. If the input causes a crash, then the input is added to the seed pool. Once an execution has completed, V-Fuzz records the execution path for the input. The *fitness score* of the input is the sum of the *SVS* values of the basic blocks that are on the execution path. Fig. 5 shows an example for *fitness score* calculation. We assume there are two inputs i_1 and i_2 in this generation. The execution paths of the two inputs are path_1 and path_2, respectively. Suppose path_1 is $b_1 \rightarrow b_2 \rightarrow b_4$, and path_2 is $b_1 \rightarrow b_3 \rightarrow b_6 \rightarrow b_8$. The *fitness score* of inputs i_1 and i_2 are f_1 and f_2, respectively. Then, $f_1 = 2 + 5 + 8 = 15$ and $f_2 = 2 + 1 + 1 + 2 = 6$. As f_1 is larger than f_2, the input i_1 will be selected as a seed. It should be noted that in our implementation, if any input causes a crash, no matter how low the *fitness score* it has, it will be sent to the seed pool.

Algorithm 1 Seed Selection Algorithm

Require: Binary Program: p; The set of all basic blocks of p: B; The set of initial inputs: I; The set of seed pool: S; The set of testcases: T;
$T = I$
while In the fuzzing loop **do**
  for t in T **do**
    $Path, Ret = EXE(p, t)$
    $fitness(t) = \sum_{b \in Path}(SVS(b))$
    if $Ret(t) \in CRASH$ **then**
      $S.add(t)$
    end if
  end for
  $Q = SelectKfitness(T)$
  $S.add(Q)$
  $T = Mutate(S)$
end while

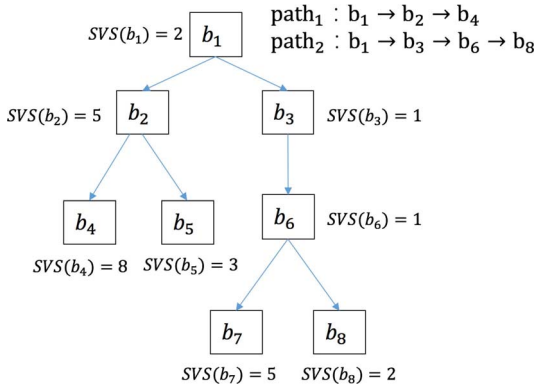


Fig. 5. Example for *fitness score* calculation.

In this way, V-Fuzz not only utilizes the information of the vulnerability prediction model but also considers the actual situation. Therefore, V-Fuzz can mitigate the potential weakness of the vulnerability prediction model.

C. Mutation Strategy

V-Fuzz is a mutation-based evolutionary fuzzing system, which generates new inputs by mutating the seeds. Like most of the mutation-based fuzzers, the mutation operations are byte flips, inserting "interesting" bytes, changing some bytes, selecting some bytes from several seeds, splicing them together, and so on.

The design of the mutation strategy is very important, as an appropriate strategy can help the fuzzer generate good inputs which can find new paths or crashes. For example, Fig. 6 gives the CFG of a simple program. Assume that there is a seed string s_1 = "abx," which can cover the basic blocks b_1 and b_2. Another seed string s_2 = "qwerty" covers the basic blocks b_1 and b_3 . It is obvious that the new inputs mutated from s_1 are more likely to cover other new basic blocks than those mutated from s_2.

It is worth noting that if we want to obtain "ab*" by mutating "abx," the mutation must be slight, which only changes a small part of the original seed. However, if the fuzzer has spent too much time doing the slight mutation operations, while making no progress, the fuzzer should change its mutation strategy and pay more attention to other paths. In this example, if the fuzzer obtains "stuck" for a long time by performing the

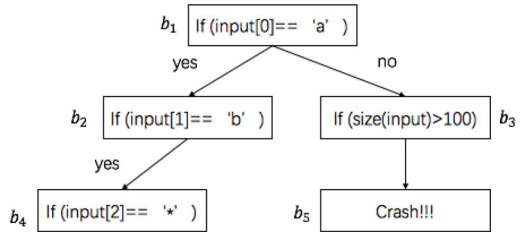


Fig. 6. Simple CFG.

slight mutation on "abx," it would be better to choose heavy mutation that changes more about the original seed, which may help the fuzzer find the basic block b_3. Therefore, the fuzzer should dynamically adjust its mutation strategy according to the actual fuzzing states.

We classify the mutation strategies into slight mutation and heavy mutation. In order to help the fuzzer determine the selection of the mutation strategy, we define the crash window (CW), which is a threshold to determine the selection of mutation strategy. Consequently, we assume that the number of generations whose inputs have not found any new path or crash is denoted by ζ. If $\zeta > CW$, the fuzzer should select a heavy mutation strategy. Furthermore, we propose the *CW jump algorithm* to adjust the value of CW optimally.

The main idea of the *CW jump algorithm* is as follows. First, we assume that the initial value of CW is *ini_cw*, its maximum value is *max_cw*, and its minimum value is *min_cw*. The value of CW starts from *ini_cw*, and the fuzzer selects slight mutation as its initial mutation strategy. During the fuzzing process, if $\zeta > CW$, the fuzzer will change its mutation strategy to heavy mutation, and will double the value of CW. Once an input finds a new path or a crash, then we set $\zeta = 0$ and the new value of CW as twice of its former value. Algorithm 2 shows the pseudocode of the *CW jump algorithm*. In our implementation, we set *ini_cw* as 4, *max_cw* as 16, and *min_cw* as 2. Note that these parameters can be changed according to different target programs.

V. Implementation and Settings

A. Vulnerability Prediction

The vulnerability prediction module consists of two main components: 1) the ACFG extractor and 2) the vulnerability prediction model. For the ACFG extractor, we implement it by writing a plug-in on the famous disassembly tool IDA Pro [36]. For the vulnerability prediction model, we implement it based on PyTorch [37], which is a popular deep learning framework. We train the vulnerability prediction model on a server which is equipped with two Intel Xeon E5-2640v4 CPUs (40 cores in total) running at 2.40 GHz, 4 TB HDD, 64-GB memory, and one GeForce GTX 1080 TI GPU card.

B. Vulnerability-Oriented Fuzzing

For vulnerability-oriented fuzzing, we implement the fuzzer based on VUzzer [15], a state-of-the-art binary-oriented evolutionary fuzzer. We conduct each fuzzing experiment on a virtual machine with Ubuntu 14.04 LTS. The virtual machine

Algorithm 2 CW Jump Algorithm

Require: Binary Program: p; The initial crash window: ini_cw; Max Crash Window: max_cw; Min Crash Window: min_cw; The current Crash Window: CW; The set of seed pool: S; The set of testcases: T; The mutation strategy: MS; $no\_crash = True$; $no\_new\_bb = True$;

```
CW = ini_cw
T = I
ζ = 0
MS = slight_mutate
while In the fuzzing loop do
    for t in T do
        EXE(p, t)
        if find crash then
            no_crash = False
        end if
        if find new basic block then
            no_new_bb = False
        end if
    end for
    if no_crash == True&&no_new_bb == True then
        ζ++
        if ζ > CW then
            MS = heavy_mutate
            if CW >= min_cw × 2 then
                CW = CW/2
            end if
        end if
    else
        ζ = 0
        MS = slight_mutate
        if CW <= max_cw/2 then
            CW = CW × 2
        end if
    end if
    S = select_good_seed(T)
    T = MS(S)
end while
```

is configured with 32-b single-core 4.2-GHz Intel CPU and 4-GB RAM.

VI. Evaluation

In this section, we evaluate the performance of V-Fuzz. Since V-Fuzz consists of two components, we will present the evaluation results in two parts: 1) the vulnerability prediction and 2) the vulnerability-oriented fuzzing.

A. Vulnerability Prediction Evaluation

1) Data Selection: We use two datasets for training and testing the vulnerability prediction model. The first dataset is from Juliet Test Suite v1.3 [38], which is published by the National Institute of Standards and Technology (NIST). We use this dataset as it has been widely used in many vulnerability related work [39], [28]. This dataset is a collection of C/C++ language programs and each function is labeled with "good" or "bad." The "good" or "bad" represents "vulnerable" or "secure," respectively. Each "bad" example has a common weakness enumeration identifier (CWE ID). Juliet Test Suite v1.3 has examples of 118 different CWEs in total. As fuzzing is suitable for discovering bugs related to memory, we select some CWE samples which are related to memory errors from Juliet Test Suite v1.3, which are shown in Table III. The second dataset is from some real-world programs such as `mpg123`, which contains some CVEs. For the real-world program dataset, we label it manually according to its corresponding CVE information. As our model is trained and tested with binary programs, we implement the labeling process by writing an IDA Pro plugin in Python. Table IV presents the information of the two datasets for training and testing the model.

TABLE III
CWE Types of Juliet Test Suite Dataset

CWE	Type	#Secure	#Vulnerable	Total
121	Stack Based Buffer Overflow	10,187	5,216	15,403
122	Heap Based Buffer Overflow	12,263	6,259	18,522
124	Buffer Under Write	4,183	2,031	6,214
126	Buffer Over Read	3,019	1,376	4,395
127	Buffer Under Read	4,183	2,031	6,214
134	Uncontrolled Format String	6,833	2,357	9,190
190	Integer Overflow	20,187	6,795	26,982
401	Memory Leak	6,756	2,303	9,059
415	Double Free	4,204	1,448	5,652
416	Use After Free	1,760	470	2,230
590	Free Memory not on the Heap	4,049	2,250	6,299
761	Free Pointer not at Start	887	493	1,380
	Total	78,511	33,029	111,540

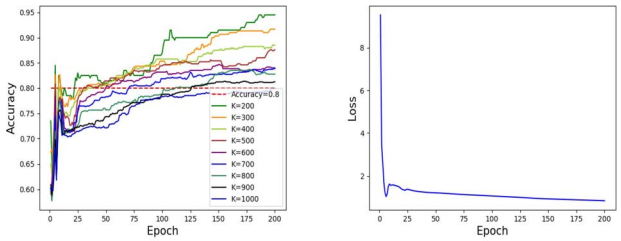
TABLE IV
Datasets

Dataset	Type	#Vulnerable	#Secure	Total
Juliet Test Suite v1.3	Training	20,000	20,000	40,000
	Testing	2,000	2,000	4,000
Real-world Programs	Training	20,000	20,000	40,000
	Testing	2,000	2,000	4,000

2) Pre-Experiments: First, we conduct some pretraining experiments to determine the default parameters of the model. We use dataset of Juliet Test Suite to conduct the pretraining experiments. We use the Adam optimization algorithm [40] and set the learning rate equal to 0.0001. Based on the results of the pretraining experiments, we set the depth of the network as 5, the embedding size as 256, and the number of iterations as 3. We take the above setting as the default of our model. It needs to be emphasized that the parameters of the model may have to be adjusted according to different application scenarios and datasets.

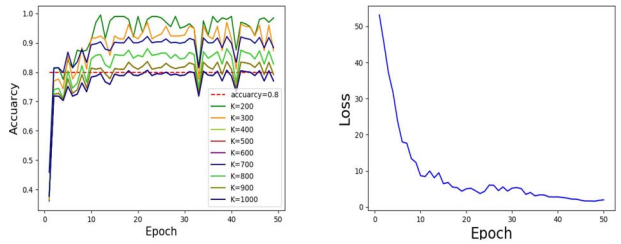
3) Evaluation Metrics: We evaluate the performance of the vulnerability prediction model from the following metrics: *top-K accuracy*, *loss*, *ACFG extraction time*, and *training time*.

Top-K Accuracy: As our model is not a binary classification model, we cannot calculate accuracy in a traditional way. Therefore, we define *top-K accuracy* to evaluate the accuracy of the model. Suppose the number of testing samples is $\hat{L}$. For all testing samples, we sort their vulnerable probabilities in descending order and we select the *top-K* testing samples. Assuming among the *top-K* samples, the number of samples labeled as "vulnerable" is $\hat{v}$. The *top-K accuracy* equals to $\hat{v}/K$. Figs. 7(a) and 8(a) present *top-k accuracy* when setting the threshold K to different values (in range [200, 1000]) for the Juliet Test Suite dataset and the real-world program dataset, from which we can observe that the accuracy of the model is high (greater than 80%).



(a) (b)

Fig. 7. Performance of the model on Juliet Test Suite dataset. (a) Top-K accuracy. (b) Loss.



(a) (b)

Fig. 8. Performance of the model on the real-world program dataset. (a) Top-K accuracy. (b) Loss.

Loss: We calculate *loss* of the model with cross-entropy loss function. Fig. 7(b) and 8(b) present *loss* curves for the two datasets. We can observe that *loss* drops to a low value soon and becomes stable, which means the model converges quickly. For the Juliet Test Suite dataset, the model can converge within ten epochs. For the real-world program dataset, the model can converge within 20 epochs.

ACFG Extraction Time: In order to test the time spent on extracting ACFG, we collect a lot of binary programs and test the extraction time of them. Fig. 9 shows the ACFG extraction time with different number of functions and file sizes. Moreover, we test the debugging binaries and the released binaries, respectively. From these figures, we have the following observations: 1) the extraction time has a positive linear correlation with the number of functions and file size and 2) the extraction time is pretty short. For most of the debugging binaries, the extraction time is within 2.5 s. For most of the released binaries, the extraction time is within 100 s. Thus, we can extract the ACFG of a binary program efficiently.

Training Time: Fig. 10 shows the training time with different parameters. For this model, the parameters which affect the training time more are depth and embedding size. From this figure, we can observe that the training time has a positive correlation with both the depth and the embedding size. As the model can converge within 20 epochs, we only need about 200 min to train a valid vulnerability prediction model, which is significantly efficient. In addition, as the model is trained offline, it does not affect the time overhead of fuzzing.

From the above results, we can see that this model is capable of vulnerable function prediction.

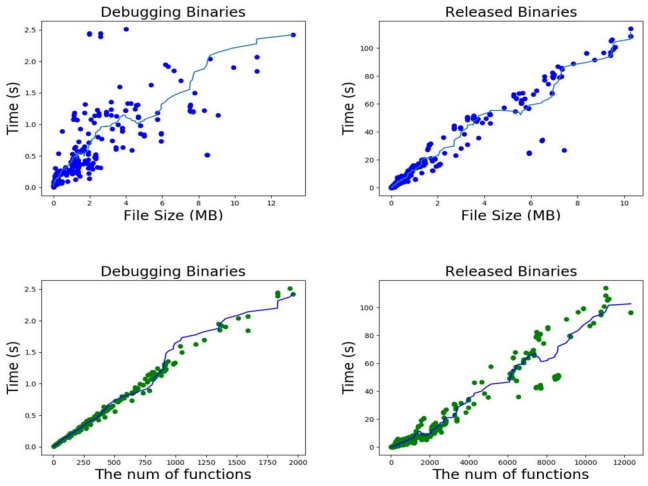


Fig. 9. ACFG extraction time.

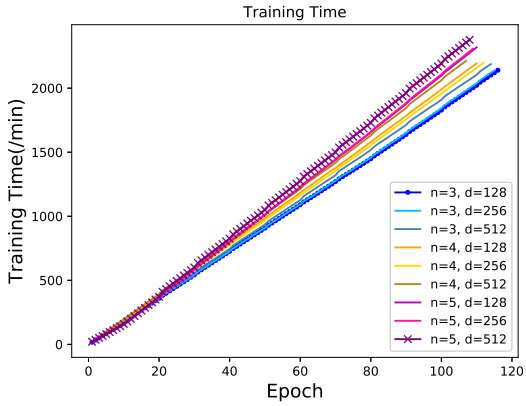


Fig. 10. Training time with different model parameters: depth n and embedding size d.

B. Evaluation of Fuzzing

In this section, we evaluate the performance of V-Fuzz in fuzzing. Toward this, we conduct a number of fuzzing experiments on 13 different applications as shown in Table V. The reasons for selecting these applications are as follows. First, these applications are widely used for evaluating the performance of fuzzers [7], [8], [15]. Second, these applications have various types, such as the audio processing software (e.g., `MP3Gain`), pdf transformation tools (e.g., `pdftotext`), and XPS document library (e.g., `libgxps`). We compare V-Fuzz with several prominent fuzzers: VUzzer [15], AFL [6], and AFLFast [11]. VUzzer is a state-of-the-art binary-oriented fuzzer. AFL and AFLFast are popular gray-box fuzzers that need the source code of a target program.[1] It should be emphasized that all of the fuzzing experiments are conducted based on the following principles: 1) all the running environments are the same, the details are in Section V-B; 2) all the initial inputs for fuzzing are the same; and 3) the running time of all the fuzzing evaluations is the

[1]Although AFL /AFLFast can be combined with QEMU to support fuzzing binary programs, the common usage of AFL/AFLFast utilizes the compile-time instrumentation which needs the source code of a target program. In our evaluation, we followed the common usage of AFL/AFLFast.

TABLE V
NUMBER OF UNIQUE CRASHES FOUND IN 24 HOURS

Application	Version	Fuzzer			
		V-Fuzz	VUzzer	AFL	AFLFast
uniq	LAVA-M	659	321	0	0
base64	LAVA-M	128	100	0	0
who	LAVA-M	117	92	0	0
pdftotext	xpdf-2.00	209	59	12	108
pdffonts	xpdf-2.00	581	367	13	0
pdftopbm	xpdf-2.00	50	25	37	35
pdf2svg + libpoppler	pdf2svg-0.2.3 libpoppler-0.24.5	3	2	0	1
MP3Gain	1.5.2	217	34	103	110
mpg321	0.3.2	321	184	40	17
xpstopng	libgxps-0.2.5	3,222	2,195	2	2
xpstops	libgxps-0.2.5	4,157	3,044	3	3
xpstojpeg	libgxps-0.2.5	4,828	4,243	4	4
cflow	1.5	1	0	0	0
Total		14,493	10,666	214	280
Average		1,114	820	16	21

same (24 h). Therefore, all of the fuzzing experiments are fair and convincing.

There are two main aspects to evaluate a fuzzer's capability of finding bugs: 1) *unique crashes* and 2) *identified vulnerabilities*. We will demonstrate the performance of V-Fuzz from the two aspects.

1) Unique Crashes: The capability of finding unique crashes is an important factor to evaluate a fuzzer's performance. Although a unique crash is not necessarily a vulnerability, in most cases, if a fuzzer can find more crashes, it can find more vulnerabilities. Thus, we will demonstrate V-Fuzz's performance in finding unique crashes by answering the following question.

Whether V-Fuzz Can Find More Unique Crashes in Limited Time? In detail, Table V presents the information of the target programs and the number of unique crashes found in 24 h. From Table V, we have the following observations: 1) for all the 13 programs, V-Fuzz finds the most unique crashes (the average number of unique crashes for one program is 1114) and is much better than the other three fuzzers; 2) compared with VUzzer, the average number of unique crashes found by V-Fuzz is improved by 35.8%. In addition, for the program `cflow`, VUzzer does not find any crash while V-Fuzz finds one crash; 3) compared with AFL, there are five programs (`uniq`, `base64`, `who`, `pdf2svg`, and `cflow`), on which AFL has not found any crash while V-Fuzz had a good performance; and 4) compared with AFLFast, there are also five programs (`uniq`, `base64`, `who`, `pdffonts`, and `cflow`), on which AFLFast does not find any crash while V-Fuzz does find.

Moreover, in order to reduce the impact of randomness, we conduct fuzzing experiments for multiple runs (three times) on two programs (`tiff2pdf` and `tiffsplit`). The results are presented in Fig. 11. From these figures, we can observe that V-Fuzz outperforms VUzzer in the three times runs. For the program `tiff2pdf`, V-Fuzz finds 10, 4, and 2 unique crashes in three times runs while VUzzer finds 6, 1, and 1 unique crashes. For the program `tiffsplit`, V-Fuzz finds at least six unique crashes in three times runs while VUzzer finds no more than two unique crashes on fuzzing program `tiffsplit` within 24 h.

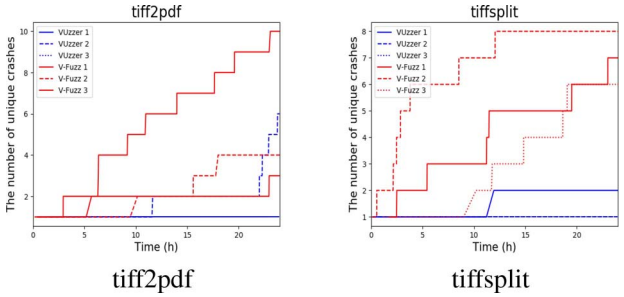


Fig. 11. Number of unique crashes for multiple runs.

TABLE VI
NUMBER OF BUGS FOUND ON LAVA-M

Application	V-Fuzz			VUzzer		
	Listed	Unlisted	Total	Listed	Unlisted	Total
uniq	27	1	28	26	1	27
base64	24	3	27	23	2	25
who	57	9	66	52	7	59

2) Vulnerability Discovery: In this part, we show V-Fuzz's capability of discovering vulnerabilities. During the fuzzing process of V-Fuzz, we collect the inputs which cause crashes. For the three programs of LAVA-M, we run the programs again with the crash inputs and verify the bugs they found. Table VI shows the number of bugs found by V-Fuzz and VUzzer. Each injected bug in LAVA-M has a unique ID, and the corresponding ID is printed when the bug is triggered. There are two kinds of bugs in LAVA-M: 1) listed and 2) unlisted. The listed bugs are those that the LAVA-M authors were able to trigger when creating the LAVA-M programs, and the unlisted bugs are those that the LAVA-M authors were not able to trigger. From Table VI, we can observe that V-Fuzz can trigger more bugs than VUzzer. In addition, V-Fuzz is able to trigger several unlisted bugs and exhibits a better performance than VUzzer in this case too.

For the real-world Linux applications, in order to verify the vulnerabilities found by V-Fuzz, we recompile the target programs with AddressSanitizer [42], which is a memory error detector for C/C++ programs. Then, we execute these programs with the collected crash inputs. AddressSanitizer can give detailed information of the vulnerabilities. Based on the information from AddressSanitizer, we search the related information on the official CVE website [43] and validate the vulnerabilities we find. Table VII shows the detailed CVE information that we find. We have found ten CVEs in total, which three of them (CVE-2018-10767, CVE-2018-10733, and CVE-2018-10768) are newly found by us. Moreover, most of the tested applications are shown to have CVEs. The crash inputs which are found when fuzzing the programs of `xpdf-2.0` can also trigger the vulnerability of `xpdf-3.01`. Finally, most of the CVEs are buffer-related errors, which is reasonable as fuzzing is good at finding this type of vulnerabilities.

TABLE VII
CVEs Found by V-Fuzz

Application	Version	CVE	Type
pdftotext pdffonts pdftopbm	xpdf<=3.01	CVE-2007-0104	Buffer errors
mpg321	0.3.2	CVE-2017-11552	Buffer errors
MP3Gain	1.5.2	CVE-2017-14406	NULL pointer dereference
		CVE-2017-14407	Stack-based buffer over-read
		CVE-2017-14409	Buffer overflow
		CVE-2017-14410	Buffer over-read
		CVE-2017-12912	Buffer errors
libgxps	<=0.3.0	CVE-2018-10767 (new)	Buffer errors
		CVE-2018-10733 (new)	Stack-based buffer over-read
libpoppler	0.24.5	CVE-2018-10768 (new)	NULL pointer dereference

TABLE VIII
Top-3 Vulnerable Probabilities

Vulnerable Probability	Function Name
0.99	_start, flushWriteBuff, id3_make_frame, scan, print_buf, etc.
0.96	audio_enclist
0.78	mad_decoder_run

VII. Further Analysis

Assistance of Vulnerability Prediction: Here, we give a case study to show whether vulnerability prediction can help improve fuzzing performance. We demonstrate the effectiveness of vulnerability prediction by answering the following two questions: 1) *does the binary functions with known CVEs receive higher VP?* and 2) *can V-Fuzz generate inputs that can trigger the corresponding CVE?* To answer these questions, we conduct a case study on the software `mpg321`. As for the CVE-2017-11552, the "vulnerable" function is `mad_decoder_run` of the file "mpg321.c". For this function, our model gives it a VP of 0.78, which is the third highest VP presented in Table VIII. In addition, as we show in Table VII, V-Fuzz successfully generates testcases that can trigger this vulnerability. Therefore, the functions with CVEs actually receive higher vulnerability scores and V-Fuzz can successfully generate inputs that can trigger the CVE.

VIII. More Related Work and Discussion

More Related Work: In addition to the related work discussed before, we give more related work in this section. In order to reduce the blindness of fuzzing and improve its efficiency, there are many related works that utilize various techniques to assist fuzzing. One is using symbolic execution to assist fuzzing. Driller [5] combines AFL with concolic execution to generate inputs that can trigger deeper bugs. TaintScope [44] is a fuzzing system that uses dynamic taint analysis and symbolic execution to test $\times 86$ binary programs. However, as symbolic execution still has several problems such as path explosion, it may not perform well on fuzzing the large real-world programs. Second is using program analysis to assist fuzzing. Dowser [45] is a guided fuzzer that combines taint tracking, program analysis, and symbolic execution to detect buffer overflow and underflow vulnerabilities. Compared with Dowser, V-Fuzz is designed to find more types of vulnerabilities. VUzzer [15] utilizes control-flow and data-flow analysis to detect bugs that lie in deeper paths. VUzzer assigns more weights to deeper code components. Different from VUzzer, V-Fuzz assigns more weights to code components that are more likely to be vulnerable.

Limitations of V-Fuzz and Future Work: Here, we give a discussion on the limitations of V-Fuzz and what we can do to improve it in our future work. V-Fuzz is a fuzzing framework that combines fuzzing with vulnerability prediction. For vulnerability prediction, we design and implement a model based on a graph embedding network. The model has the following limitations. First, the model is only trained with limited data in this article. Additionally, we believe the model's performance can be improved when we train it with more high-quality industrial datasets. Second, the model focuses on predicting the VP for binary and the prediction granularity is a function. It is interesting to develop further models that are applicable for other application scenarios, such as for source code and other finer granularity (basic block or lines of code). Third, since V-Fuzz provides a feasible framework for assisting fuzzers with vulnerability prediction, in the future, V-Fuzz can be extended to incorporate with more vulnerability prediction models/tools and other fuzzers.

IX. Conclusion

In this article, we designed and implemented V-Fuzz, a vulnerability-oriented evolutionary fuzzing framework for binary programs. By combining the vulnerability prediction with evolutionary fuzzing, V-Fuzz can generate inputs that tend to arrive at the potential vulnerable regions. We evaluated V-Fuzz on popular benchmark programs (e.g., `uniq`) of LAVA-M [17], and a variety of the real-world Linux applications, including the audio processing software (e.g., `MP3Gain`), pdf transformation tools (e.g., `pdftotext`), and xps documents library (e.g., `libgxps`). Compared with three prominent fuzzers, the experimental results demonstrate that V-Fuzz can find more vulnerabilities quickly. In addition, V-Fuzz has discovered ten CVEs, and three of them are newly discovered. In the future, we will study to take advantage of more advanced program analysis techniques to assist fuzzers in discovering vulnerabilities.

References

[1] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. Upper Saddle River, NJ, USA: Addison-Wesley Professional, 2007.

[2] Google. (2018). *OSS-Fuzz—Continuous Fuzzing for Open Source Software*. [Online]. Available: https://github.com/google/oss-fuzz

[3] Microsoft. (2018). *Microsoft Security Development Lifecycle*. [Online]. Available: https://www.microsoft.com/en-us/sdl/process/verification.aspx

[4] Y. Shin and L. Williams, "Can traditional fault prediction models be used for vulnerability prediction?" *Empirical Softw. Eng.*, vol. 18, no. 1, pp. 25–59, 2013.

[5] N. Stephens *et al.*, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proc. NDSS*, vol. 16, 2016, pp. 1–16.

[6] M. Zalewski. (2017). *American Fuzzy Lop*. [Online]. Available: http://lcamtuf.coredump.cx/afl/

[7] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *Proc. 39th IEEE Symp. Security Privacy*, 2018, pp. 1–15.

[8] S. Gan *et al.*, "Collafl: Path sensitive fuzzing," in *Proc. 39th IEEE Symp. Security Privacy*, 2018, pp. 679–696.

[9] C. Lyu *et al.*, "MOPT: Optimized mutation scheduling for fuzzers," in *Proc. USENIX Security Symp.*, 2019, pp. 1949–1966.

[10] H. Chen *et al.*, "Hawkeye: Towards a desired directed grey-box fuzzer," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2018, pp. 2095–2108.

[11] V. Pham and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2016, pp. 1032–1043.

[12] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2017, pp. 2329–2344.

[13] P. Hui, S. Yan, and P. Mathias, "T-fuzz: fuzzing by program transformation," in *Proc. 39th IEEE Symp. Security Privacy*, 2018, pp. 697–710.

[14] K. Böttinger, "Guiding a colony of black-box fuzzers with chemotaxis," in *Proc. IEEE Security Privacy Workshops (SPW)*, 2017, pp. 11–16.

[15] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *Proc. Netw. Distrib. Syst. Security Symp.*, 2017, pp. 1–12.

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

[17] B. Dolangavitt *et al.*, "Lava: Large-scale automated vulnerability addition," in *Proc. 37th IEEE Symp. Security Privacy*, 2016, pp. 110–121.

[18] (2004). *Flawfinder*. [Online]. Available: https://dwheeler.com/flawfinder/

[19] (2004). *Rats: A Rough Auditing Tool for Security*. [Online]. Available: https://code.google.com/archive/p/rough-auditing-tool-for-security/

[20] Y. Zhou and Y. Wei, "Learning hierarchical spectral-spatial features for hyperspectral image classification," *IEEE Trans. Cybern.*, vol. 46, no. 7, pp. 1667–1678, Jul. 2016.

[21] B. Xue and N. Tong, "DIOD: Fast and efficient weakly semi-supervised deep complex isar object detection," *IEEE Trans. Cybern.*, vol. 49, no. 11, pp. 3991–4003, Nov. 2019.

[22] J. Du, C. Vong, and C. L. P. Chen, "Novel efficient rnn and lstm-like architectures: Recurrent and gated broad learning systems and their applications for text classification," *IEEE Trans. Cybern.*, early access, Feb. 20, 2020, doi: 10.1109/TCYB.2020.2969705.

[23] M. Fu, H. Qu, Z. Yi, L. Lu, and Y. Liu, "A novel deep learning-based collaborative filtering model for recommendation system," *IEEE Trans. Cybern.*, vol. 49, no. 3, pp. 1084–1096, Mar. 2019.

[24] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *Proc. IEEE Int. Conf. Softw. Qual. Rel. Security*, 2015, pp. 17–26.

[25] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *Proc. 24th USENIX Security Symp.*, 2015, pp. 611–626.

[26] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *Proc. 12th Working Conf. Min. Softw. Repositories*, 2015, pp. 334–345.

[27] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2017, pp. 363–376.

[28] Z. Li *et al.*, "Vuldeepecker: A deep learning-based system for vulnerability detection," in *Proc. Netw. Distrib. Syst. Security Symp.*, 2018, pp. 1–15.

[29] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2016, pp. 480–491.

[30] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "Discovre: Efficient cross-architecture identification of bugs in binary code," in *Proc. NDSS*, 2016, pp. 381–396.

[31] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *Proc. IEEE Symp. Security Privacy*, 2015, pp. 709–724.

[32] Intel. (2019). *Intel 64 and IA-32 Architectures Software Developer Manuals*. [Online]. Available: https://software.intel.com/en-us/articles/intel-sdm

[33] H. Dai, B. Dai, and L. Song, "Discriminative embeddings of latent variable models for structured data," in *Proc. Int. Conf. Mach. Learn.*, 2016, pp. 2702–2711.

[34] H. Cai, V. W. Zheng, and K. C.-C. Chang, "A comprehensive survey of graph embedding: Problems, techniques, and applications," *IEEE Trans. Knowl. Data Eng.*, vol. 30, no. 9, pp. 1616–1637, Sep. 2018.

[35] X. Shen and F. Chung, "Deep network embedding for graph representation learning in signed networks," *IEEE Trans. Cybern.*, vol. 50, no. 4, pp. 1556–1568, Apr. 2020.

[36] Hex-Rays. *The IDA Pro Disassembler and Debugger*. Accessed: 2018. [Online]. Available: https://www.hex-rays.com/products/ida/

[37] PyTorch. *Pytorch: Tensors and Ynamic Neural Networks in Python With Strong GPU Acceleration*. [Online]. Available: http://pytorch.org/

[38] (2017). *Juliet Test Suite for C/C++*. [Online]. Available: https://samate.nist.gov/SRD/testsuite.php

[39] W. Han, B. Joe, B. Lee, C. Song, and I. Shin, "Enhancing memory error detection for large-scale applications and fuzz testing," in *Proc. Netw. Distrib. Syst. Security Symp. (NDSS)*, 2018, pp. 1–47.

[40] D. P. Kingma and J. Ba. (2014). *Adam: A Method for Stochastic Optimization*. [Online]. Available: https://arxiv.org/abs/1412.6980

[41] L. Van Der Maaten, "Accelerating T-SNE using tree-based algorithms," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 3221–3245, 2014.

[42] chefmax. (2017). *Addresssanitizer*. [Online]. Available: https://github.com/google/sanitizers/wiki/AddressSanitizer

[43] NVD. (2018). *CVE: Common Vulnerabilities and Exposures*. [Online]. Available: https://cve.mitre.org/

[44] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *Proc. 31st IEEE Symp. Security Privacy*, 2010, pp. 497–512.

[45] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *Proc. USENIX Conf. Security Symp.*, 2013, pp. 49–64.

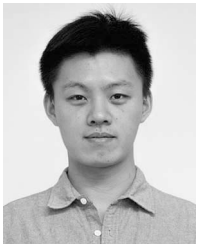
Yuwei Li received the B.E. degree from the Nanjing University of Posts and Telecommunications, Nanjing, China, in 2016. She is currently pursuing the Ph.D. degree with the Computer Science and Technology College, Zhejiang University, Hangzhou, China.

Her current research interests include software security, system security, and AI security.

Shouling Ji (Member, IEEE) received the first Ph.D. degree in electrical and computer engineering from the Georgia Institute of Technology, Atlanta, GA, USA, in 2015, and the second Ph.D. degree in computer science from Georgia State University, Atlanta, in 2013.

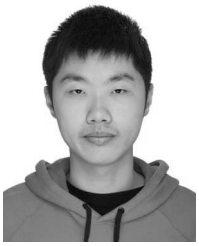
He is currently a ZJU 100-Young Professor with the College of Computer Science and Technology, Zhejiang University, Hangzhou, China, and a Research Faculty with the School of Electrical and Computer Engineering, Georgia Institute of Technology. His current research interests include AI security, data-driven security, privacy, and data analytics.

Dr. Ji was the Membership Chair of the IEEE Student Branch at Georgia State from 2012 to 2013. He is a member of ACM.



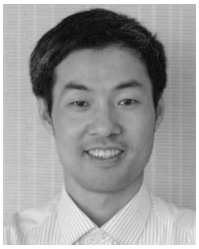
Chenyang Lyu received the B.E. degree in computer science and technology from the Huazhong University of Science and Technology, Wuhan, China, in 2017. He is currently pursuing the Ph.D. degree with the Computer Science and Technology College, Zhejiang University, Hangzhou, China.

His current research interests include deep learning and fuzz testing.



Yuan Chen received the B.S. degree in biology from Zhejiang University, Hangzhou, China, in 2018, where he is currently pursuing the Ph.D. degree with the Computer Science and Technology College.

His current research interests include Web and software security.



Jianhai Chen (Member, IEEE) received the M.S. and Ph.D. degrees in computer science and technology from Zhejiang University (ZJU), Hangzhou, China, in 2006 and 2016, respectively.

He is currently an Associate Professor with the College of Computer Science and Technology, ZJU, where he is also the Director of ZJU SuperComputing Team, and the Director of ZJU Intelligent Computing Innovation and Entre-Preneurship Laboratory. His research interests include blockchain system security, cloud computing scheduling algorithms and game theory, supercomputing application optimization, and AI data mining.

Dr. Chen is a member of the CCF and ACM.

Qinchen Gu (Member, IEEE) received the M.S. degree in electrical and computer engineering from the Georgia Institute of Technology (Georgia Tech), Atlanta, GA, USA, in 2015, where he is currently pursuing the Ph.D. degree with the School of Electrical and Computer Engineering.

He is also a Graduate Research Assistant with the Communications Assurance and Performance Group, Georgia Tech. His research primarily focuses on the security for cyber–physical systems.

Chunming Wu received the Ph.D. degree in computer science from Zhejiang University, Hangzhou, China, in 1995.

He is currently a Professor with the College of Computer Science and Technology, Zhejiang University. His research interests include software-defined networks, proactive network defense, network virtualization, and intelligent networks.

Raheem Beyah (Senior Member, IEEE) received the Bachelor of Science degree in electrical engineering from North Carolina A&T State University, Greensboro, NC, USA, in 1998, and the masters's and Ph.D. degrees in electrical and computer engineering from Georgia Tech, Atlanta, GA, USA, in 1999 and 2003, respectively.

He was an Assistant Professor with the Department of Computer Science, Georgia State University, Atlanta, a Research Faculty Member with the Communications Systems Center (CSC), Georgia Tech, and a Consultant with Andersen Consulting's (currently, Accenture) Network Solutions Group. He is the Motorola Foundation Professor and the Associate Chair with the School of Electrical and Computer Engineering, Georgia Tech, where he leads the Communications Assurance and Performance Group and is a member of CSC. His research interests include network security, wireless networks, network traffic characterization and performance, and critical infrastructure security.

Dr. Beyah received the National Science Foundation CAREER Award in 2009 and was selected for DARPA's Computer Science Study Panel in 2010. He is a member of AAAS and ASEE, is a Lifetime Member of NSBE, and is a Senior Member of ACM.