FuzzGuard: Filtering out Unreachable Inputs in Directed Grey-box Fuzzing through Deep Learning

Peiyuan Zong[1,2], Tao Lv[1,2], Dawei Wang[1,2], Zizhuang Deng[1,2], Ruigang Liang[1,2], Kai Chen[1,2*]
[1] *SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China*
[2] *School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China*
{zongpeiyuan, lvtao, wangdawei, dengzizhuang, liangruigang, chenkai}@iie.ac.cn

Abstract

Recently, directed grey-box fuzzing (DGF) becomes popular in the field of software testing. Different from coverage-based fuzzing whose goal is to increase code coverage for triggering more bugs, DGF is designed to check whether a piece of potentially buggy code (e.g., string operations) really contains a bug. Ideally, all the inputs generated by DGF should reach the target buggy code until triggering the bug. It is a waste of time when executing with unreachable inputs. Unfortunately, in real situations, large numbers of the generated inputs cannot let a program execute to the target, greatly impacting the efficiency of fuzzing, especially when the buggy code is embedded in the code guarded by various constraints.

In this paper, we propose a deep-learning-based approach to predict the reachability of inputs (i.e., miss the target or not) before executing the target program, helping DGF filtering out the unreachable ones to boost the performance of fuzzing. To apply deep learning with DGF, we design a suite of new techniques (e.g., *step-forwarding approach*, *representative data selection*) to solve the problems of unbalanced labeled data and insufficient time in the training process. Further, we implement the proposed approach called FuzzGuard and equip it with the state-of-the-art DGF (e.g., AFLGo). Evaluations on 45 real vulnerabilities show that FuzzGuard boosts the fuzzing efficiency of the vanilla AFLGo up to 17.1×. Finally, to understand the key features learned by FuzzGuard, we illustrate their connection with the constraints in the programs.

1 Introduction

Fuzzing is an automated program testing technique, which is usually divided into two categories: coverage-based fuzzing and directed fuzzing. The goal of the former one is to achieve high code coverage, hoping to trigger more crashes; while directed fuzzing aims to check whether a given potential buggy code really contains a bug. In real analysis, directed fuzzing is very popularly used since the buggy code is often specified. For example, security analysts usually pay more attention to the buffer-operating code or want to generate a proof-of-concept (PoC) exploit for a given CVE [3] whose buggy code is known. There are some directed fuzzing tools such as AFLGo [9], SemFuzz [35] and Hawkeye [12]. As we know, a random input is less likely to reach the buggy code, not to mention triggering the bug. Thus, most of the tools instrument the target program for observing the run-time information and leveraging the information to generate the inputs that could reach the buggy code. Such fuzzing method is also referred to as Directed Grey-box Fuzzing (*DGF* for short).

An ideal DGF should generate the inputs which can all reach the buggy code. Unfortunately, in real situations, a large number of the generated inputs could miss the target, especially when the buggy code is embedded in the code guarded by many (complicated) constraints (e.g., thousands). Facing this situation, various techniques (e.g., Annealing-based Power Schedules [9]) are designed to generate reachable inputs. However, even for the state-of-the-art DGF tools (e.g., AFLGo [9]), the ratio of unreachable inputs is still high. Based on our evaluation using AFLGo, on average, over 91.7% of the inputs cannot reach the buggy code (Section 6).

Such a large amount of unreachable inputs waste lots of time in the fuzzing process. Traditional program analysis approaches such as symbolic execution [20], theoretically, could use the constraints of all branches in the target program to infer the execution result of the input. However, the time spent on solving constraints will dramatically increase together with the increase of the constraints' complexity. In other words, the constraints in the path from the program's start point to the buggy code could be very complex, which makes them difficult or even not possible to be solved in limited time.

Inspired by the success of pattern recognition [11,19,34,36] which could accurately classify millions of images even if they are previously unseen, our idea is to view program inputs as a kind of pattern and identify those which can reach the buggy code. Basically, by training a model using a large number of inputs labeled with the reachability to the target code from previous executions, we could utilize the model to predict the reachability of the newly generated inputs without running

*Corresponding author

the target program. However, it is challenging to build such an accurate model for DGF due to the following reasons.

Challenges. **C1:** Lack of balanced labeled data. It is necessary to acquire enough and balanced labeled data to train a deep learning model (e.g, classification for cats and dogs). In other words, the number of one object's images should be close to the number of the other object's. However, in the process of fuzzing, the (un)reachable inputs are usually unbalanced. Especially, in the early stage of fuzzing, there is even no reachable input (e.g., for the bug #7 in GraphicsMagick, the first reachable input is generated after more than 22.6 million executions). Without the balanced labeled data, the trained model will be prone to over-fitting. One may think of extending the labeled data just like the way of image transformation (e.g., resizing, distortion, perspective transformation), which could increase the number of the object's images to balance the training data without changing the identified object. However, such transformation cannot be applied to program inputs since even one bit flip may change the execution paths of inputs and further impact the labels (i.e., let a reachable input become unreachable).

C2: Newly generated reachable inputs could look quite different from the reachable ones in the training set, making the trained model fail to predict the reachability of the new inputs. This is mainly because the new inputs may arrive at the buggy code through a different execution path never seen before. So simply using the inputs along one execution path to train a model may not correctly predict the reachability of a new input. One may think of generating various inputs along different execution paths to the buggy code before training. Unfortunately, such generation process is out of our control. He may also wait for a long time before training, hoping to collect enough inputs along different paths. However, this may waste lots of time since many unreachable inputs have been executed with.

C3: Efficiency. In the task of training a model for traditional pattern recognition, the time spent on training is not strictly limited. However, in the fuzzing process, if the time spent on training a model and predicting an input's reachability is more than the time spent on executing the program with the input, the prediction is of no use. So the time cost of training and prediction should be strictly limited.

Our approach. In this paper, we overcome the challenges mentioned above and design an approach to build a model for DGF to filter out unreachable inputs, called FuzzGuard. The basic idea of FuzzGuard is to predict whether a program can execute to the target buggy code with a newly generated input by learning from previous executions. If the result of prediction is unreachable, the directed grey-box fuzzer (we use *"the fuzzer"* for short in the rest of the paper) shouldn't execute this input anymore, which saves the time spent on real execution. Note that FuzzGuard is not meant to replace the fuzzer (e.g., AFLGo), but to work together with the fuzzer to help it filter out unreachable inputs.

FuzzGuard works in three phases: model initialization, model prediction, and model updating. (1) In the first phase, the fuzzer generates various inputs and runs the target program with them to check whether a bug is triggered. At the same time, FuzzGuard saves the inputs and their reachability, and trains the initial model using the labeled data, which may be unbalanced (C1). To solve this problem, we design a *step-forwarding* approach: choosing the dominators (referred to as "*pre-dominating nodes*" [5]) of the buggy code as the middle-stage targets, and letting the execution reach the pre-dominating nodes first. In this way, the balanced data could be gained earlier for training some models only targeting the pre-dominating nodes, which minimizes the time of execution. (2) In the second phase, after the fuzzer generates a number of new inputs, FuzzGuard utilizes the model to predict the reachability of each input. As mentioned in C2, the trained model may not work for the newly generated inputs. To solve this problem, we design a representative data selection approach to sample training data from each round of mutation, which minimizes the number of sampled data to increase efficiency. (3) In the third phase, FuzzGuard updates the model using the labeled data collected in the second phase to increase its accuracy. Note that the time spent on the model updating should be strictly limited (C3). We tackle this challenge by carefully choosing the time to update. To the best of our knowledge, previous studies of fuzzing focus on generating various inputs, to cover more lines of code (CGF) or to reach buggy code (DGF). Various mutation strategies on inputs are designed. In contrast, our study does not directly mutate inputs (we rely on current mutation strategies, e.g., AFLGo). Instead, we filter out unreachable inputs. In this way, a DGF does not need to run the target program with unreachable inputs (which definitely cannot trigger the target bug), which increases the overall efficiency.

We implement FuzzGuard on the base of AFLGo [9] (an open-source state-of-the-art DGF tool), and evaluate the performance using 45 real vulnerabilities on 10 popular programs. The results show that FuzzGuard boosts the fuzzing performance by 1.3× to 17.1×. Interestingly, we find that the more the unreachable inputs the fuzzer generates, the better FuzzGuard could perform. Also, more time could be saved if the target node reach a balanced state earlier. At last, we design an approach to understand the extracted features of FuzzGuard, and find that the features are correlated with the constraints in the if-statements in target programs, which indeed impacts the execution on code level.

Contribution. The contributions of this paper are as follows:

• *New technique*. We design and implement FuzzGuard which helps DGF to filter out unreachable inputs and save the time of unnecessary executions. To the best of our knowledge, this is the first deep-learning-based solution to identify and remove unreachable inputs. The core of FuzzGuard is the *step-forwarding* approach, and *representative data selection*. Evaluation results show that up to 88% of fuzzing time can be saved for state-of-the-art tools (e.g., AFLGo). We also release our FuzzGuard for helping researchers in the community[1].

• *New understanding*. We design an approach to study the features utilized by the model in FuzzGuard for prediction, and find them correlated with the branches in target programs. The understanding of such relationship helps to explain the deep learning model and further helps to improve FuzzGuard.

2 Background

In this section, we give a brief background of directed grey-box fuzzing and recent studies utilizing deep learning to improve the fuzzing performance.

2.1 Fuzzing

Fuzzing [27] is one of the classical software testing techniques to expose exceptions of a computer program [32]. The main idea of fuzzing is to feed a massive number of inputs (i.e., test cases) to the target program, exposing bugs through observed exceptions. Among all techniques of fuzzing, grey-box fuzzing [12] recently becomes quite popular due to its high efficiency and reasonable performance overhead. With different goals, grey box fuzzing can usually be divided into two types as follows.

Coverage-based Grey-box Fuzzing. One main goal of this type of fuzzing technique is to achieve the high coverage of code in the target program. Therefore, some fuzzers [2, 10, 15, 16, 24, 25] aim to achieve high code coverage of the target program, expecting to accidentally trigger the bug, namely *Coverage-based Grey-box Fuzzing* (CGF). Typically, CGF generates the inputs by mutating the seed inputs which could traverse previous undiscovered program statements in order to increase the coverage rate of the code. AFL [2], as a representative of CGF, employs light-weight compile-time instrumentation technique and genetic algorithms to automatically discover interesting test cases, selects seed inputs that trigger new internal states in the fuzzing process, and mutates seed inputs in various ways (e.g., bit and byte flips, simple arithmetics, stacked tweaks and splicing [22]).

Directed Grey-box Fuzzing. Sometimes, the potential buggy code is known. So there is no need to increase the code coverage. In this situation, fuzzers [9, 12, 35] are designed to generate inputs that reach the buggy code for triggering a specified bug, which is referred to as *Directed Grey-box Fuzzing* (DGF). DGF is commonly used since some kinds of code may be highly possible to contain a bug (e.g., string copy operations) which should be emphasized more in the fuzzing. Also, sometimes the buggy code is known (e.g., from CVEs). So those fuzzers are utilized to generate a proof-of-concept exploit toward the buggy code [35]. With the different goals from CGF, current DGF aims to generate the inputs which could reach the specific potential buggy code, further expecting to trigger the bug. For example, AFLGo [9] calculates the distance between each basic blocks and the path from the entry point to the buggy code in the control flow graph; then utilizes the distance to choose suitable inputs for mutation.

However, even for state-of-the-art fuzzers, still lots of time is spent on unnecessary executions. In our experiments, we find that for a typical vulnerability whose location is known, more than 91.7% of the generated inputs cannot reach the buggy code (unreachable inputs) on average. Running the target program with the unreachable inputs is highly time-consuming. If there is a fuzzer that could judge the reachability of an input without executing the program, a huge amount of time could be saved. In this paper, we design such a filter called FuzzGuard, which leverages a deep learning model to achieve this goal without real execution. Also, it could be adapted to existing fuzzers (e.g., AFLGo) and work together with them, without replacing them. To the best of knowledge, this is the first deep-learning-based solution to filter out unreachable inputs for DGF.

2.2 Deep Learning

Security researchers apply deep learning to fuzzing, which provides new insights for solving difficult problems in previous research. For example, Godefroid et al. utilize RNNs to generate program inputs that have higher code coverage [17]. Rajpal et al. [29] utilize RNN-guided mutation filter to locate which part of an input impacts more on code coverage. In this way, they could achieve higher code coverage by mutating the located part. Nichols et al. [28] show that GANs could be used to predict the executed path of an input to improve the performance of the AFL [2]. Angora [15] and NEUZZ [31] adapt the gradient descent algorithm to solve path constraint and learn a model to improve code coverage respectively. All these studies concentrate on leveraging the ability of deep learning to cover more code. Different from them, our goal is to help directed grey-box fuzzers to filter out the inputs that cannot hit the buggy code before real execution. In this way, the time spent on running the program with unreachable inputs could be saved, which greatly increases the efficiency of fuzzing. Note that our tool can be adapted to existing DGF tools (e.g., AFLGo), which means that we could further increase the fuzzing efficiency together with the performance boosted by other fuzzers.

3 Motivation

As mentioned above, current DGF aims to generate the inputs which could reach the specific buggy code, further expecting to trigger the bug. In the fuzzing process, lots of inputs cannot reach the buggy code in the end (impossible to trigger the bug). Based on our evaluation, more than 91.7% of the inputs can't hit the buggy code on average (see Table 1). Executing millions of unreachable inputs could cost very long time (e.g., 76 hours for a million inputs when fuzzing Podofo, a library to work with the PDF file format with a few tools [1]). Especially,

[1]The release is available at https://github.com/zongpy/FuzzGuard.

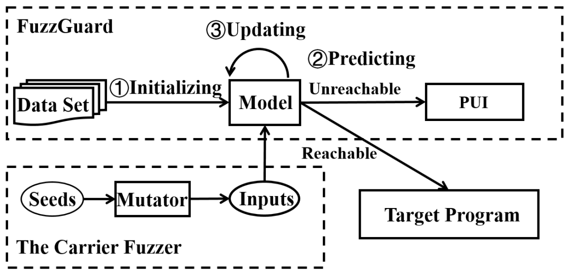

Figure 1: An overview of FuzzGuard.

when the execution time of the target program takes part the most in the whole fuzzing process, the wasted time is even more. If there exists an approach that is quick enough to predict the reachability of an input, the fuzzing process does not need to execute the target program with the unreachable ones. In this way, the overall performance of fuzzing could be increased.

Inspired by the recent success of deep learning in pattern recognition [11, 19, 34, 36], we are wondering whether deep learning could be applied to identify (un)reachable inputs. Carefully comparing the processes of pattern recognition and identification of (un)reachable inputs, we found similarities between them: they both classify data (a certain objects v.s. (un)reachable inputs) based on either prior knowledge or statistical information extracted from the patterns (many labeled images of the object v.s. many labeled inputs from previous executions). However, they do have essential differences (e.g., the distribution of labeled data, requirements on efficiency, etc.) which makes the process of unreachable input identification very challenging (see Section 1).

Example. List 1 gives an example. The vulnerable code is at Line 6 (see Section 7). So the goal of DGF (e.g., AFLGo) is to generate as many as inputs that could reach there and hope to trigger the bug. The seed input is chosen from AFLGo's seed corpus (e.g., `not_kitty.png`). It takes 13 hours to generate 16 million inputs and needs to test the program with them before the bug is triggered. Among these inputs, only 3.5 thousand (0.02%) can reach the buggy code. One may think of leveraging symbolic execution to generate constraints from the execution path to the destination. However, the full constraints are very hard to generate since several paths could reach the buggy code. Even if the constraints could be generated, the calculation of reachability using the constraints is still very time-consuming, which is even similar to the time spent on running the target program. Our idea is to generate a deep learning model to automatically extract features of reachable inputs and to identify future reachable ones. Based on our evaluation, nearly 14 million inputs (84.1%) are identified which saves 9 hours of unnecessary executions. Also note that the false positive rate and false negative rate are only 2.2% and 0.3% for this example, respectively.

Scope and Assumption. Different from previous research on CGF using deep learning [15, 17, 28, 29], our approach focuses on filtering out unreachable inputs in DGF. In this way, lots of necessary time on executing the program with unreachable inputs could be saved. Note that our approach is complementary to other DGF tools and can work together with them, instead of replacing them. Also note that we do not assume that small mutations in the input will produce similar or identical behavior. The trained model should characterize different behaviors of similar-looking inputs.

4 Methodology

We propose the design of FuzzGuard, a deep-learning-based approach to facilitate DGF to filter out unreachable inputs without really executing the target program with them. Such a data-driven approach avoids using traditional time-consuming approaches such as symbolic execution for better performance. Below we elaborate the details of FuzzGuard.

4.1 Overview

The overview of FuzzGuard is illustrated in Figure 1, including three main phrases: model initialization (MI), model prediction (MP) and model updating (MU). It works together with a DGF (referred to as *"the carrier fuzzer"*). As shown in Figure 1, in the MI phrase, the carrier fuzzer generates a great number of inputs and observes any exceptions. FuzzGuard records whether the program can execute the target buggy code for each input. Then FuzzGuard trains a model using the inputs and their reachability. In the MP phrase, FuzzGuard utilizes the model to predict the reachability of a newly generated input. If the input is reachable, it is fed into the program for real execution. In this process, FuzzGuard observes whether the input can really reach the target code. In the MU phrase, FuzzGuard further updates the model with incremental learning to maintain its efficiency and increase its performance. The unreachable inputs will be temporarily saved in a data pool (referred to as *"the pool of unreachable inputs (PUI)"*) for further checking by a more accurate model after model updating. As combining deep learning with fuzzing is not trivial, we face the new challenges (as mentioned in Section 1).

Figure 2 shows a concrete example of FuzzGuard. Firstly, the carrier fuzzer generates a number of inputs (referred to as *"data"*) and runs the target program with them to get the reachability (referred to as *"label"*) in the MI phrase (the step ① and step ② in the figure). During this process, an ideal situation is to train a deep learning model using balanced data. That is, about half of the inputs could reach the buggy code while the other cannot. Unfortunately, in real situation, the carrier fuzzer hardly generates the inputs that reach the buggy code in the initial phrase of fuzzing. As a result, the labeled data are usually extremely unbalanced at this stage. For example, only one input can actually reach the buggy code after over 22 million inputs generated (#7 in Table 1). To solve this problem, we design *the step-forwarding approach* for MI (step ②), which lets the inputs reach the predominating nodes (we use *"node"* to refer to *"basic blocks"*

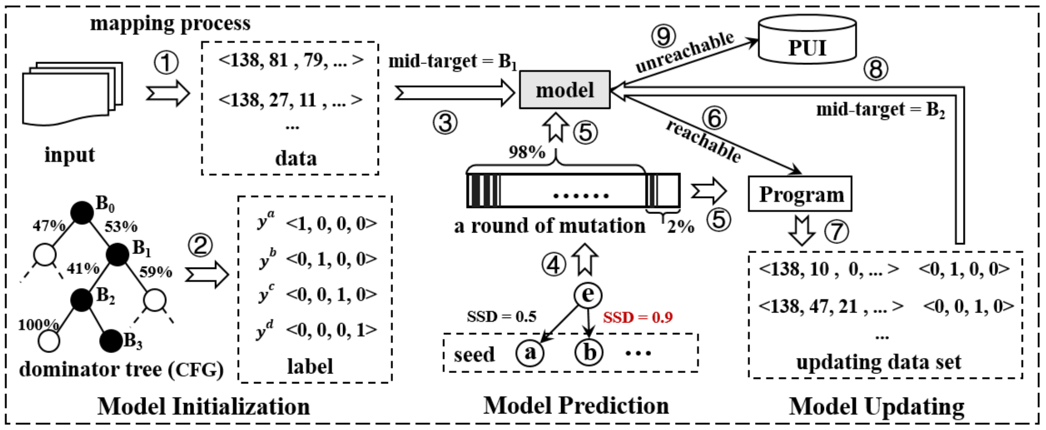

Figure 2: An example of filtering unreachable inputs by FuzzGuard.

in the rest of the paper) of the buggy code (i.e., B_0, B_1 and B_2 in Figure 2) step-by-step to the destination (i.e., B_3 in Figure 2). Particularly, FuzzGuard chooses a pre-dominating node (e.g., B_1) as a middle-stage destination (i.e., referred to as *"mid-target"*) and generates a model to filter out the inputs that cannot reach B_1 (step ③). Usually, compared with B_3, more balanced labeled data could be gained when the program runs to B_1. So the model can be trained earlier and also starts to work earlier. Then MP judges whether a newly generated input (in step ④) could reach B_1 using the model (step ⑤). For reachable inputs (e.g., with label $< 0,1,0,0 >$ in step ⑥), FuzzGuard runs the program with it and records whether it can really reach the buggy code (step ⑦). Such information is further leveraged to continuously update the model by MU (step ⑧). The unreachable inputs are put into PUI (step ⑨). After more inputs are tested, a closer pre-dominating node (to the buggy code) having the balanced labeled data will appear (e.g., B_2 in this case). Such a process will continue until the buggy code is arrived at, and finally triggered. Below we provide the details of the three modules.

4.2 Model Initialization

As mentioned previously, one main challenge of applying deep learning on fuzzing is the unbalanced data for training. Usually, the number of reachable inputs is far less than that of unreachable ones. In order to tackle this challenge, we present a *step-forwarding* approach. The basic idea is based on the observation: the pre-dominating nodes of the buggy code are earlier to be reached, which should gain balanced data earlier. Note that the *pre-dominating nodes* of the buggy code are the nodes that dominate the buggy code: every execution path towards the buggy code will pass through the pre-dominating nodes [5]. So the reachability of the marked pre-dominating nodes is guaranteed. Therefore, we could train a model to filter out those inputs that cannot reach the pre-dominating node (neither can they reach the buggy code). In this way, we gradually get balanced data of the pre-dominating nodes, toward the buggy code in the end. For example, as to the control flow graph shown in Figure 2, the nodes represent basic blocks in the program in List 1. B_0 is the entry point and the buggy code is in B_3. B_1 and B_2 are the pre-dominating nodes of B_3. At the beginning of fuzzing, no input reaches B_3, while half of the inputs could reach B_1. Now B_1 is the closest balanced pre-dominating node to the buggy code. So we view B_1 as the target, and train the model using these inputs. In this way, the unreachable inputs to B_1 are filtered out, saving the time spent on executing the target program with them. When the fuzzing process goes further, B_2 or B_3 will get balanced data for training. Note that, different from CGF whose goal is to achieve high coverage, DGF aims to generate inputs to trigger a given (potential) bug at a certain place. So it does not care about whether new bugs are found in other paths. Interestingly, we did see that FuzzGuard+AFLGo still discovers undisclosed bugs (see Section 6) which are located deeply in a program (also near the target buggy code). An ordinary CGF is hard to trigger them in a limited time.

However, it takes too long to train a single model for each pre-dominating node. This is mainly because the model needs to be retrained when FuzzGuard steps forward to the next pre-dominating node. Our idea is to only train one model for all the pre-dominating nodes including the buggy code itself. To achieve this goal, formally, we label the reachability of the nodes (i.e., $B = \{B_1, B_2, ..., B_n\}$) in the vector y. For each label, it is represented as a unit vector $\hat{y}$, i.e., $\hat{y} =< y_1, y_2, \ldots, y_m >$, where m is the number of the pre-dominating nodes of the buggy code, y_i represents whether the i-th node is the last one could be reached by the program fed with x, $y_i \in \{0,1\}, i \in \{1,2,\ldots,m\}$. As shown in Figure 2, for the input a, the label is represented as $y^a =< 1,0,0,0 >$, which means that B_0 is reached but others are not. Similarly, $y^b =< 0,1,0,0 >$ means that the input b can let the execution reach B_0 and B_1, but neither B_2 nor B_3. $y^d =< 0,0,0,1 >$ means that the buggy code is finally reached. For simplicity, we directly map each byte of the input to an element in the feature vector. This approach makes FuzzGuard handle different programs with various formats of inputs in a unified way. For each data, it can be represented as a vector $x =< x_1, x_2, \ldots, x_n >$, where n is the max length of the input. And $x_i = byte_i + 1$ ($x_i \in$

$\{0,1,\ldots,256\}$), where $x_i = 0$ means that i-th byte of the input does not exist (i.e., the length of the input is less than n).

After designing the representation of data and label, we carefully choose a deep learning model. Such a model should be good at extracting features from inputs and making the correct classification. Recall the problem of image recognition: features of an object in an image are expressed by combinations of several pixels (i.e., elements in the input vector, as shown in Figure 2), which could be well extracted by the CNN models [11, 19, 34, 36]. Similarly, the features of inputs that impact their reachability could be expressed by combinations of several bytes in program inputs. Actually, the constraints in if-statements in target programs use these bytes for deciding execution directions. Thus, our idea is to make use of CNN to accomplish the classification task. On one hand, compared with RNN which is more suitable for training with the byte sequence, CNN is good at dealing with long data. The longer the inputs, the faster the RNN model forgets the former features. On the other hand, the time for training a CNN model is much less than the time for training an RNN model, which is suitable for our problem (the time spent on training and prediction should be less than the time on real execution).

Thus, we choose to use a 3-layer CNN model (detailed implementation is shown in Section 5). In this way, the first layer could learn the relationship between each byte, and the other two layers could learn high dimensional features (e.g., combining several bytes to form a field in an input, and combining several fields to impact program execution). Interestingly, we find that such extracted high dimensional features are correlated with the constraints in the if-statements in target programs (see Section 7). We also discuss other machine learning models in Section 8. Note that, the model needs to be trained for each program from scratch due to different implementations (which parse inputs in different ways). It is also an interesting topic to explore the similarity between different programs and leverage such similarity to increase the efficiency of training.

In this way, we can let the carrier fuzzer run for a while to collect an initial training data set. After the initial training data set reaches balanced, the model can learn the reachability to all nodes of the inputs. The goal of the model is to learn a target function f (i.e., $y = f(x)$), which consists of a number of convolution operations. The convolution operation uses a number of filters to extract the features from the data:

$$y_i = w^T x_i = \sum_{i-k<j<i+k} w_j \cdot x_j, i \in \{1,2,\ldots,n-k\}$$

where k is the width of the convolution kernel of the filter w. Gradient descent algorithm will update weights of each filter w to decrease the loss value to achieve a more accurate prediction. For classification tasks, compared to Cross Entropy [18] loss, the Mean Square Error (MSE) [23] loss could balance the error rate for each category, avoiding a particularly high error rate for a single category. Considering the step-forwarding approach needs the trained model to predict the reachability of each pre-dominating node as accurate as possible, we choose to use MSE. So when the value of the $loss = \frac{1}{m}\sum_{i=1}^{m}(y_i - y_i^p)^2$ is close to 0, we believe that the target function in the classification model has been converged and the model is ready to predict the newly generated inputs.

4.3 Prediction

After the model is initialized, FuzzGuard utilizes the model to predict the label of each input and filters out those unreachable ones. For the reachable ones, they will be executed by the target program and further be collected as new labeled data for model updating. In particular, for an input x, we assume that the model can only predict the pre-dominating nodes before B_t (i.e., *the mid-target*), and the prediction result is y^p. The following function f' is used to check whether the input x is reachable to the target node.

$$f'(y^p,t) = \begin{cases} reachable & y_i^p = 1 \land i \geq t \\ unreachable & y_i^p = 1 \land i < t \end{cases}$$

However, in real situation, we find the prediction results are not accurate enough, even after many labeled data are produced. The main reason is that even if the newly generated inputs could reach the target, they may look quite different from the reachable ones in the training set. This is understandable: these inputs could be generated from different seeds. Most of the inputs mutated from the same seed are slightly different with each other, while many differences could be found between the inputs mutated from different seeds. Thus, using the inputs totally from previous executions may not be able to train a very accurate model to predict the reachability of newly generated inputs. For example, a model trained with the data in set S_1 mutated from the seed s^1 may fail to predict the labels of the data in S_2 mutated from the seed s^2.

To solve this problem, we propose a *representative data selection* approach, which selects a number of representative inputs from each round of mutation for executing and training. We consider a fixed number of inputs (e.g., 5%) that randomly sampled from a round of mutation as the representative data for this mutation. In this way, within a limited time, inputs generated from more seeds can be utilized for training, which increases the model's accuracy. However, in real execution, even 5% of the inputs constitute a big number (e.g., over 20 thousand), and execution using these inputs cost lots of time. Our idea is to sample even fewer inputs. Suppose in two different mutations, two sets of inputs S_1 and S_2 are generated from the two seeds s^1 and s^2, respectively. If the distribution of S_1 is similar to that of S_2, we can select even fewer inputs. However, we cannot directly assume that the distributions of the two sets are similar only through the similarity of the two seeds. This is mainly because different strategies of mutation (e.g., bit and byte flips, simple arithmetics, stacked tweaks and splicing) could greatly change the seeds and make the descendants look quite different. So our idea is to compare the seeds together with the corresponding strategies of mutations. If the two seeds are similar and the strategies are identical, we consider to select fewer inputs from the combined set. We define the *seed similarity degree* (SSD) between the two seeds s^1 and s^2 as follows:

$$d_{s^1,s^2} = 1 - \sum_{i=1}^{8n} s_i^1 \oplus s_i^2 / 8n$$

where n is the max byte length of the inputs, and s_i means the i-th bit of the seed s. Note that different choices of embedding do not affect the definition of SSD, since SSD is defined using the seeds, not the vectors after embedding. In this way, we could measure the similarity between two sets of inputs through their predecessor seeds. When SSD is over a threshold (θ_s), we consider that the seed s_2 is similar to the seed s_1, and less data from the inputs mutated from s_2 should be selected. For example, in Figure 2, we select less data (e.g., 2%) from the inputs set that generated by seed the e, because e is similar to the seed b (e.g., SSD=90%). In this way, we could select fewer inputs for real execution and training without impacting the model's accuracy. Based on our evaluation, on average, half of the time spent on fuzzing could be saved when applying this technique (Section 6).

4.4 Model Updating

To realize online model updating, we utilize *incremental learning* [26] to train a dynamic model by feeding a set of data each time rather than feeding all data at once. In this case, new incoming data are continuously used to extend the existing model's knowledge. Incremental learning aims to adapt to new data without forgetting its existing knowledge for the learning model, and it does not require retraining the model. It can be applied when the training data set becomes available gradually over time as the carrier fuzzer generates and exercises new inputs continuously. Also incremental learning decreases the time of waiting for data collecting, and filters out more unreachable test cases.

The online deep learning model should be updated to keep its accuracy. Whenever a new set of labeled data is collected, there could be an opportunity for model updating. However, if the model is updated too frequently, the time spent on training will be long, which will impact the efficiency of fuzzing. In contrast, if less frequent updating is performed, the model may not be accurate. So in this process, we should carefully choose when to perform model updating. Also we should let the updating be quick enough. Below we elaborate the details.

We perform model updating when the model is getting "outdated". The outdated model is not accurate enough when a new pre-dominating node is reached. In the first situation, we update the model when the false positive rate γ of the model exceeds a threshold θ_f. To achieve this, we continuously record false positive rates of the model whenever the execution results are different from the predictions, and keep watching γ. After the model is updated, we reset the false positive rate to zero and record it again. Another situation is that when there is a new pre-dominating node B_i ($i > t$) containing the balanced labeled data, it is the time to update the model with the new data (see Section 4.3). In this way, the model could learn new features from the inputs which let the program execute to new code that has never been touched. Using this approach, we could ensure the accuracy of the model while keeping the model updating at a reasonable frequency.

To avoid missing a PoC (i.e., to avoid filtering out any PoC), we temporarily store unreachable inputs in the PUI. When the model is updated, we use the new model to check the inputs in the PUI again, and pick out the reachable ones for execution. Based on our evaluation, the model is accurate enough that no PoC is missed.

5 Implementation

In this section, we describe the implementation of FuzzGuard, including model initialization, model prediction, model updating and the details of the deployment of FuzzGuard.

Model Initialization. At the initial stage, FuzzGuard starts to train the model only after enough data are collected; and continues to update the model after another set of data (not a single input) are collected. Such data should be balanced (i.e., the number of reachable inputs is similar to the number of unreachable ones). Particularly, before the model is trained, all the inputs should be fed into the target program for real execution. FuzzGuard records the reachability of the inputs. Once enough[2] balanced data are gained, FuzzGuard starts to train the model. Then it utilizes the trained model to predict the reachability of a newly generated input, executes the target program if it is reachable, and records the reachability in real execution. Such data are collected to update the model for better performance. As mentioned before, DGF requires a target (potential) buggy code whose location is known for fuzzing. To set the pre-dominating nodes of the buggy node, we generate Call Graph (CG) and Control Flow Graph (CFG) of the target program and set the pre-dominating node according to the definition as mentioned in Section 4. In our implementation, we use NetworkX [6] to automatically find the pre-dominating nodes from the CG and CFG generated by LLVM.

Model Prediction and Updating. To further collect data for updating the model, we set the θ_s for SSD to 0.85 and the default sampling rate is 5% in each round of mutation. When SSD exceeds the threshold, the sampling rate will decrease to $(1-\theta_s)/5$ (i.e., less than 3%). Based on our evaluation, setting the threshold using this value has the best performance. Considering that models' accuracy varies a lot for different

[2]In our implementation, FuzzGuard starts to train the model after 50 thousand balanced inputs are gained. The number is far less than the number of the inputs generated by AFLGo in a testing task (usually 30 million).

Algorithm 1 Function Checker()

```
Input: $argv$, $timeout$ and $input$
1: $fault \leftarrow 0$
2: if $check(input)$ is $reachable$ then
3:     $fault \leftarrow run\_target(argv, timeout)$
4:     $label \leftarrow check\_trace(input)$
5:     $send(label)$
6: end if
7: return $fault$
```

programs, we dynamically change θ_f according to the previous executions: $\theta_f = 1 - acc_{avg}$, where acc_{avg} represents the average accuracy of the models updated previously.

Model Implementation. For the training model, we implement a CNN model using PyTorch [7]. It contains three 1-dimensional convolution layers (k = 3, stride = 1). Note that the 1-dimensional convolution layer takes each input as a row sequence; and each row has 1024 bytes. Each convolution layer is followed by a pooling layer and a ReLU [4] as the activation function. We also have Dropout layers (disabling rate = 20%) to avoid over-fitting of the neural networks. There is a fully-connected layer at the end of the neural networks, which is used to score the reachabilities of each node in the target path to the buggy code. Also we use the Adam optimizer [21] to help the learning function converge to the optimal solution rapidly and stably. The training process ends when the loss value of the learning function becomes stable.

Deployment of FuzzGuard. To achieve the data sharing, we add a function *Checker()* to afl-fuzz.c in AFLGo. Algorithm 1 shows the details of *Checker()*. The function *Checker()* handles all parameters in *run_target()* (i.e., *argv, timeout* in Algorithm 1) and receive an input which is saved in a piece of memory. Before the input is fed into the target program, it is sent to FuzzGuard (i.e., *check(input)* at line 2 in Algorithm 1). Only when FuzzGuard returns with the result showing that the execution path is reachable, the target program is executed with the input (line 3 in Algorithm 1). After executing the target program, *Checker()* reads the reachability of the input from the function check_trace() (Line 4 in Algorithm 1) and sends it to FuzzGuard for further learning (line 5 in Algorithm 1). We plan to release our FuzzGuard for helping researchers in the community.

6 Evaluation

In this section, we evaluate the effectiveness of FuzzGuard with 45 vulnerabilities. The results are compared with a vanilla AFLGo. According to the experiment results, FuzzGuard boosts the performance of fuzzing up to 17.1 times faster than that of AFLGo. Then we provide an understanding of the performance boost and break down the performance overhead of FuzzGuard. We also analyze the accuracy of FuzzGuard and show our findings.

6.1 Settings

We first selected 15 real-world programs handling 10 common file formats, including network packages (e.g., PCAP), videos (e.g., MP4, SWF), texts (e.g., PDF, XML), images (e.g., PNG, WEBP, JP2, TIFF) and compressed files (e.g., ZIP). Unfortunately, three programs (i.e., mupdf, rzip, zziplib) cannot be compiled[3], and two programs (i.e., apache, nginx) do not give the details of vulnerabilities. So we chose the rest 10 as the target programs and the corresponding bugs in the past 3 years[4]. Table 1 shows the details of each vulnerability, including program names and line numbers of the vulnerable code (the column Vuln. Code). For different input formats, we use the test cases provided by AFLGo as the initial seed files to start fuzzing (we believe that AFLGo will perform well using the initial seed files chosen by itself). All the experiments and measurements are performed on two 64-bit servers running Ubuntu 16.04 with 16 cores (Intel(R) Xeon(R) CPU E5-2609 v4 @ 1.70GHz), 64GB memory and 3TB hard drive and 2 GPUs (12GB Nvidia GPU TiTan X) with CUDA 8.0.

6.2 Effectiveness

To show the effectiveness of FuzzGuard, we evaluate AFLGo equipped with FuzzGuard and the original one using 45 vulnerabilities in 10 real programs (as demonstrated in Table 1). The ideal comparison for the AFLGo equipped with FuzzGuard and the vanilla AFLGo is to compare the time of fuzzing using AFLGo (T_{AFLGo}) and the corresponding time when equipping AFLGo with FuzzGuard (T_{+FG}). However, we cannot directly use the same seed input to compare the fuzzing process of AFLGo and that of AFLGo+FuzzGuard. This is because the mutation is random, and the generated sequence of inputs (even if from the same seed input) could be quite different in the two fuzzing processes, which further makes the time spent on execution quite different. So our idea is to make the generated sequence of inputs be the same in the two different fuzzing processes. Particularly, for a vulnerability of a target program, we use a vanilla AFLGo to perform fuzzing and record the sequence of all the mutated inputs I_{AFLGo} in order (the number of the inputs N_{Inputs} is shown in Table 1) until the target vulnerability is triggered (e.g., a crash) or timeout (200 hours in our evaluation). In this process, the fuzzing time T_{AFLGo} (as shown in Table 1) is also recorded. Then we utilize the same sequence of inputs I_{AFLGo} to test AFLGo equipped with FuzzGuard, recording the filtered inputs $I_{filtered}$ (the number of the filtered inputs is $N_{filtered}$, and the ratio of filtered inputs to all the generated inputs $filtered = N_{filtered}/N_{Inputs}$ are shown in Table 1). We also record the time cost of FuzzGuard (T_{FG}) including the time of training and prediction. In this way, we are able to know the time when FuzzGuard is equipped, and compare the

[3]We tried to fix the compile errors (e.g., missing libraries). However, due to too many errors, it is very hard to fix all the errors.

[4]We excluded 5 vulnerabilities (out of 50) triggered in minitues by AFLGo. Obviously, there is no need to utilize FuzzGuard to speedup.

Table 1: Effectiveness of FuzzGuard.

No.	Program	Vuln. Code	$N_{Functions}$	$N_{Constraints}$	N_{Inputs}	$UR.$	$Filtered$	T_{AFLGo}	T_{+FG}	Speedup		
										FG	FG_1	FG_2
1	Bento4 v1.5.1.0	Ap4AvccAtom.cpp:83	676	1.8 K	1.8 M	38.1%	32.3%	44 h	29.9 h	**1.5**	1.3	1.4
2	Ettercap v0.8.2	ec_strings.c:182	420	41.5 K	49.2 M	99.0%	93.9%	80.5 h	6.1 h	**13.3**	1.1	8.3
3	GraphicsMagick v1.3.31	tiff.c:2375	3.3 K	170.3 K	32.1 M	95.9%	90.5%	94.8 h	11.1 h	**8.6**	5.9	7.5
4	GraphicsMagick v1.3.31	png.c:6945	4.9 K	319.8 K	30 M	96.6%	88.8%	200 h	23.4 h	**8.5**	2.0	8.2
5	GraphicsMagick v1.3.31	png.c:7503	1.5 K	21.9 K	16.4 M	99.9%	84.1%	13.2 h	3.9 h	**3.4**	1.0	3.1
6	GraphicsMagick v1.3.31	png.c:5007	4.4 K	317.1 K	16 M	99.9%	34.3%	200 h	132.3 h	**1.5**	1.0	1.5
7	GraphicsMagick v1.3.30	png.c:3810	3.1 K	168.4 K	22.6 M	99.9%	32.8%	31.9 h	22.4 h	**1.4**	1.0	1.4
8	GraphicsMagick v1.3.27	webp.c:716	10.7 K	749.3 K	67.5 M	99.5%	93.4%	200 h	15.2 h	**13.2**	9.7	9.7
9	GraphicsMagick v1.3.26	png.c:7061	4.9 K	320 K	56.9 M	98.4%	93.3%	200 h	16.2 h	**12.3**	8.4	9.4
10	GraphicsMagick v1.3.26	tiff.c:2433	4.4 K	316.4 K	78.4 M	75.3%	69.5%	200 h	66.7 h	**3.0**	2.4	2.7
11	GraphicsMagick v1.3.26	rle.c:753	9.2 K	379.3 K	17.7 M	99.4%	70.5%	30.8 h	10.4 h	**3.0**	1.5	2.6
12	GraphicsMagick v1.3.26	list.c:232	3.6 K	172.1 K	73 M	37.2%	28.4%	200 h	146.4 h	**1.4**	1.6	1.4
13	ImageMagick v7.0.8-13	msl.c:8353	85.5 K	5.4 M	7.3 M	99.2%	92.4%	200 h	15.4 h	**13.0**	3.3	12.9
14	ImageMagick v7.0.8-3	dib.c:1306	117.1 K	7, 883.1 M	3.2 M	99.9%	54.4%	200 h	91.4 h	**2.2**	1.0	1.6
15	ImageMagick v7.0.8-3	bmp.c:2062	117.3 K	6, 306.9 M	3 M	99.9%	52.3%	200 h	95.6 h	**2.1**	1.0	2.0
16	ImageMagick v7.0.7-16	webp.c:769	23.9 K	145.7 K	11.5 M	99.1%	93.9%	200 h	12.6 h	**15.9**	15.5	14.7
17	ImageMagick v7.0.7-16	webp.c:403	14.8 K	116.1 K	19.1 M	96.0%	90.7%	200 h	19.8 h	**10.1**	9.4	10.7
18	ImageMagick v7.0.7-1	tiff.c:1934	149.1 K	1.2 M	9.4 M	98.5%	92.5%	200 h	15.2 h	**13.1**	1.6	8.7
19	ImageMagick v7.0.5-5	bmp.c:894	102.4 K	926.5 K	12.9 M	64.3%	59.9%	200 h	80.5 h	**2.5**	1.7	2.5
20	Jasper v2.0.14	jp2_enc.c:309	13.9 K	17.7 M	28 M	99.4%	50.9%	200 h	99 h	**2.0**	1.7	1.9
21	Jasper v2.0.10	jpc_dec.c:1700	740	9.7 K	11.3 M	99.7%	94.3%	46.9 h	3.7 h	**12.7**	1.4	11.1
22	Jasper v2.0.10	jpc_dec.c:1881	1.7 K	36.8 K	6.1 M	99.9%	94.0%	19.7 h	1.6 h	**12.0**	1.0	10.0
23	Jasper v2.0.10	jas_seq.c:254	1.1 K	11.8 K	22.3 M	62.4%	56.0%	200 h	89 h	**2.2**	2.0	2.2
24	Libming v0.4.8	decompile.c:1930	104	5.3 K	38.6 M	99.9%	70.2%	200 h	63 h	**3.2**	1.0	3.1
25	Libming v0.4.7	parser.c:1645	75	4.7 K	32.3 M	99.8%	94.7%	200 h	11.7 h	**17.1**	8.5	14.1
26	Libming v0.4.7	parser.c:64	170	2.7 K	16.1 M	91.9%	86.6%	200 h	27.2 h	**7.3**	1.7	6.2
27	Libming v0.4.7	parser.c:3381	79	790	38.4 M	99.7%	69.9%	200 h	61.3 h	**3.3**	2.0	3.2
28	Libming v0.4.7	parser.c:3095	25	217	46.8 M	92.9%	65.7%	200 h	70 h	**2.9**	1.9	2.8
29	Libming v0.4.7	parser.c:2993	22	386	45.9 M	97.2%	64.8%	200 h	71.8 h	**2.8**	1.7	2.7
30	Libming v0.4.7	parser.c:3126	24	294	77 M	92.9%	63.6%	200 h	75.3 h	**2.7**	2.0	2.5
31	Libming v0.4.7	parser.c:3232	55	423	12.6 M	99.8%	61.3%	6.1 h	2.8 h	**2.2**	2.0	1.8
32	Libming v0.4.7	parser.c:3221	38	308	13 M	99.9%	43.2%	14 h	8.2 h	**1.7**	1.0	1.6
33	Libming v0.4.7	parser.c:3250	32	340	16.6 M	99.9%	46.0%	7.3 h	4.4 h	**1.7**	1.0	1.4
34	Libming v0.4.7	parser.c:3089	36	396	19.6 M	99.9%	43.3%	5.2 h	3.4 h	**1.5**	1.0	1.1
35	Libming v0.4.7	parser.c:3061	37	637	18.9 M	99.8%	37.2%	3.4 h	2.5 h	**1.4**	1.0	1.1
36	Libming v0.4.7	parser.c:3071	34	1.1 K	17.6 M	99.9%	33.6%	3.8 h	2.9 h	**1.3**	1.0	1.1
37	Libming v0.4.7	parser.c:3209	34	402	30.7 M	99.9%	27.7%	8.9 h	6.9 h	**1.3**	1.0	1.2
38	Libming v0.4.7	outputtxt.c:143	64	2.2 K	27.3 M	65.5%	24.6%	7.7 h	6.1 h	**1.3**	1.1	1.1
39	Libtiff v4.0.9	tif_dirwrite.c:1901	728	14.4 K	8.6 M	99.9%	91.4%	9.6 h	1.3 h	**7.4**	1.0	4.8
40	Libtiff v4.0.7	tif_swab.c:289	631	13.1 K	44.7 M	99.7%	52.8%	29.6 h	15 h	**2.0**	1.1	1.3
41	Libtiff v4.0.7	tiffcp.c:1386	728	13.3 K	15.6 M	99.9%	51.7%	8.9 h	4.6 h	**1.9**	1.0	1.7
42	Libtiff v4.0.7	tif_read.c:346	416	11.6 K	60.6 M	79.5%	36.3%	77.9 h	49.8 h	**1.6**	1.4	1.5
43	Libxml2 v2.9.4	SAX2.c:2035	418	15.7 K	92.6 M	99.9%	94.4%	200 h	17.6 h	**11.3**	1.0	5.2
44	Podofo v0.9.5	PdfPainter.cpp:1945	19.8 K	44.1 K	2.6 M	99.3%	79.7%	200 h	40.7 h	**4.9**	4.8	1.8
45	Tcpreplay v4.3.0-beta1	get.c:174	23	1.1 K	203.3 M	53.2%	49.5%	200 h	105.4 h	**1.9**	1.7	1.9
Avg.			15.5K	315.9 M		91.7%	65.1%			**5.4**	2.6	4.4

time with T_{AFLGo}. T_{+FG} can be calculated as follows:

$$T_{+FG} = T_{AFLGo} - \sum_{i \in I_{filtered}} t_i + T_{FG}$$

where $I_{filtered}$ is the inputs filtered out by FuzzGuard and t_i stands for the time spent on executing the target program with the input i.

Note that, the last input in I_{AFLGo} is the first PoC generated by AFLGo (if the target program crashes, e.g., #1 and #2 in Table 1) or the last input generated by AFLGo before timeout (no crash happens, e.g., #8 and #9 in Table 1). We emphasize that FuzzGuard does not know whether a given input is the last one or not. In the fuzzing process, FuzzGuard treats the last input in the same way as the previous inputs. Comparing to FuzzGuard, a method randomly dropping inputs in I_{AFLGo} will randomly decide to drop the last input or not. From Table 1 we can see that FuzzGuard drops 65.1% inputs on average. If the same number of inputs (65.1%) is dropped by the random method, the last input (a possible PoC, e.g., #1 and #2 in Table 1) could also be dropped with the possibility of 65.1%. In contrast, the false negative rate of FuzzGuard is 0.02% (see Section 6.3), which means that even if 65.1% inputs are dropped by FuzzGuard, the possibility of dropping the PoC is only 0.02%.

Landscape. The results are shown in Table 1. Those 45 bugs in Table 1 include 27 CVEs found in the last 3 years and

18 newly undisclosed bugs (see Section 6.5). In our evaluation, the undisclosed bugs (e.g., Line 6 in Table 1) were found when FuzzGuard performing target fuzzing on other vulnerabilities (e.g., CVE-2017-17501, Line 4 in Table 1). Note that the buggy code of this undisclosed bug is actually not our target in this process. Then, we set the newly found buggy code as the target and tried to utilize AFLGo to reproduce it. Unfortunately, in the time limit (200 hours), AFLGo failed to trigger the bug. Neither could AFLGo+FuzzGuard trigger the bug. However, AFLGo+FuzzGuard did save the time from 200 hours to 23.4 hours (8.5 times speedup). From the table, we find that for all the bugs, FuzzGuard can increase the runtime performance of AFLGo from 1.3× to 17.1× (see the "Speedup" column in Table 1, where $Speedup = T_{AFLGo}/T_{+FG}$). The average performance is increased by 5.4×. Note that such performance boost is added to a DGF (i.e., AFLGo) which has already been optimized.

Understanding the performance boost. To understand the performance of FuzzGuard for different programs and bugs, we further study the relationship among the speedup, the time that the model starts to train and the ratio of unreachable inputs, etc.

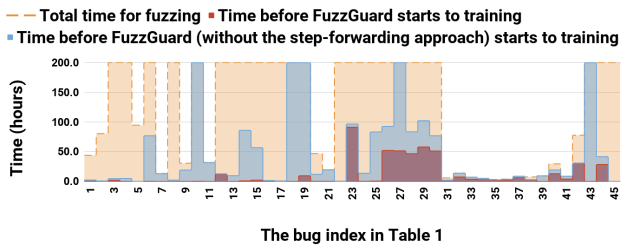

Figure 3: Start time of the first training in FuzzGuard.

• The earlier the model is trained, the more time could be saved. Figure 3 shows the time that each model starts to be trained for the bugs in Table 1 (the red bar). We can see that the model trained later (e.g., #20, #24, #27) achieved no more than 3.3× speedup, while the model trained earlier could achieve over 17× speedup. This is mainly because the earlier the buggy node gains balanced labeled data, the earlier the model can be trained for filtering out unreachable inputs to the buggy code. As a result, more inputs could be filtered out for saving the time on unnecessary executions.

• The more reachable inputs generated by the carrier fuzzer, the less effective FuzzGuard is. For example, as shown in Table 1, when more than 40% of the inputs are reachable (the column "*UR*." is the ratio of unreachable inputs), the speedup gained by FuzzGuard is less than 2 times (e.g., the bug #1, #12 and #45 in Table 1). In a special case, if there are no if-statements or constraints in the path from the entry point to the target buggy code, all the generated inputs are reachable. So there is no need to train a deep learning model.

Complicated Functions. To evaluate FuzzGuard on handling complicated functions with multiple constraints and branches, we measure the number of unique functions and constraints[5] in the path to each bug in Table 1. From the table, we can see that the average number of unique functions and constraints are 15.5 thousand and 315.9 million, respectively. Over 50% of the bugs are guarded by thousands of constraints (e.g., the bugs in GraphicsMagick and ImageMagick). For these bugs, FuzzGuard achieves the speedup from 1.4 to 15.9. For some bugs guarded by millions constraints (e.g., #13 and #18 in Table 1), FuzzGuard achieves over 10× speedup. The results show that FuzzGuard can handle complicated functions well, which could be quite time-consuming for traditional constraint solving.

Cost. In our evaluation of the 45 bugs in Table 1, the time spent on training the online model is 60 minutes on average, which includes 13.5% for data collection, 0.5% for data embedding and 86% for the training process. Note that the time spent on training only takes 6% of the time for input generation by the fuzzer (15 hours on average). The total time spent by FuzzGuard is 1.4 hours on average, which only takes 9.2% of the total time of the fuzzing (T_{+FG} in Table 1) and 2.5% of the total time of the fuzzing process performed by AFLGo (T_{AFLGo} in Table 1). Such a time period is enough for a fuzzer to process 704 thousand inputs, which is far more efficient than directly executing the target program for testing.

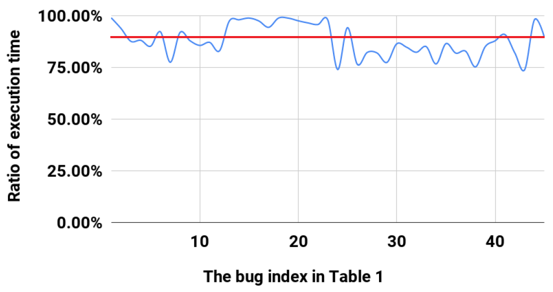

Figure 4: Evaluation on execution time in fuzzing process.

To understand the upper limit of of the fuzzing time that FuzzGuard could save, we perform a 24-hour fuzzing on 45 vulnerabilities (shown in Table 1) using AFLGo. From Figure 4, we can see that the average execution time of the target program is over 88% of the total time of fuzzing, which means that the average upper limit of the fuzzing time that FuzzGuard could save is about 88%. The time cost of FuzzGuard should be less than the limit.

6.3 Accuracy

We measure the accuracy of FuzzGuard. The accuracy is based on whether the reachability is correctly judged. The more accurate it behaves, the more unreachable inputs could be filtered out. Note that no PoC will be missed since the filtered inputs will be saved in the PUI, which will further

[5] As it is very hard to check whether a constraint is dependent on inputs due to inaccuracy of taint analysis, we count the number of all unique constraints. Such problem also happens in symbolic execution.

be checked by an updated model. A more accurate model may find the reachable ones in the pool and let the target program execute with them, which in theory will not have false negatives. However, in real execution, we usually set a timeout for fuzzing. In this case, if a false negative input is left in the pool without being found before the timeout, it will be missed. Fortunately, in our evaluation of the 45 bugs, no PoC is found in the PUI due to the accurate model. We define false positive rate as follows: $fpr = N_{fp}/N_n \times 100\%$, where N_n represents the number of the unreachable inputs generated by AFLGo, and N_{fp} is the number of inputs that cannot reach the buggy code but be viewed as reachable ones by FuzzGuard. The false negative rate is: $fnr = N_{fn}/N_p \times 100\%$, where N_p represents the number of the reachable inputs, and N_{fn} is the number of reachable inputs but be filtered out by FuzzGuard. The higher the fpr, the more time is spent on executions with unreachable inputs. The higher the fnr, the more likely the PoC is executed late in the fuzzing. The accuracy is calculated by $acc = \frac{N_p+N_n-N_{fp}-N_{fn}}{N_p+N_n}$.

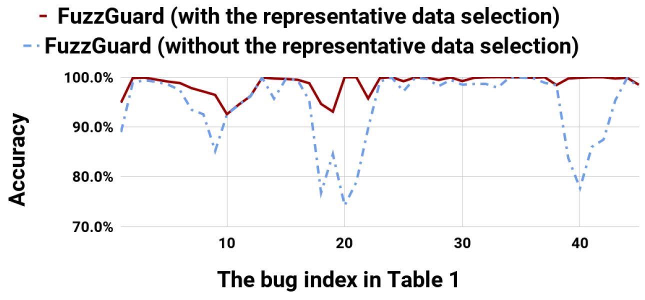

Figure 5: The accuracy of FuzzGuard.

From Figure 5, we can find that FuzzGuard is very accurate (ranging from 92.5% to 99.9%). The average accuracy is 98.7%. The false positive rate for all vulnerabilities is 1.9% on average. Note that false positives do not let a PoC be missed. Neither do they increase the time spent on executing the inputs (such inputs are always executed by the program if there is no FuzzGuard). The false negative rate is negligible, which is 0.02% on average. There are only 4 vulnerabilities that have false negatives, and the highest one is 0.3%. We further check those false negatives manually and confirm that there is no PoC in those inputs. Even if a PoC is included, as mentioned previously, FuzzGuard will save it to the PUI for further testing by updated models (no PoC will be missed). Such an accurate model enables FuzzGuard to have high performance.

The main reason for false positives and false negatives is due to lack of balanced representative data. For example, an unreachable input could be predicted by FuzzGuard as reachable (i.e., a false positive) if it is similar enough to previous reachable inputs. The execution path of the input can also be similar to the path to the buggy code (covering some predominating nodes of the buggy code). But some bytes in the input stop the execution to the buggy code eventually. A false negative may let the program reach the target buggy code through an execution path that is never seen before. If those new execution paths could be learned by the model, the prediction will be more accurate. In our evaluation, the number of unseen paths becomes less after long-time fuzzing, which is probably the reason for the low false positive rate.

6.4 Contribution of Individual Techniques

To investigate the individual contribution of the step-forwarding approach and the representative data selection, we measure the performance boost with and without each technique for all the bugs in Table 1. In particular, to be fair in the comparison, for each bug to test, we use the same sequence of inputs. We first perform the evaluation without the step-forwarding approach, and record the performance increase (column FG_1 in Table 1). Then we do not use representation data selection and record the corresponding performance increase (column FG_2 in Table 1). The results indicate that FuzzGuard (with both the two techniques) can gain 5.4× speedup compared to the vanilla AFLGo implementation, while FuzzGuard without step-forwarding and FuzzGuard without representative data selection can gain only 2.6× and 4.4× speedup, respectively.

We also made further analysis. As we know, the step-forwarding approach is designed to help FuzzGuard to get balanced data earlier in the fuzzing process, further to let the training process start earlier. So we want to measure how much step-forwarding can help. We record the start time of the first training with and without the step-forwarding approach (see Figure 3). The x-axis in the figure shows the bug index in Table 1, and the y-axis gives the start time in hours. From the figure, we find that if step-forwarding is not used, FuzzGuard fails to start the training process for 14 bugs (e.g., #5 , #6 and #7) due to lack of balanced data. For other bugs, even if the training process starts, the time of start will be postponed by 17.4 hours on average compared with the model using step-forwarding. This also postpones the filtering process and finally impacts the overall performance.

Regarding representative data selection, we also measure its impact on the accuracy of the model. For each bug, we record the model's accuracy with and without using representative data selection. The results are shown in Figure 5. The x-axis shows the bug index and y-axis gives the accuracy of the model. From the figure, on average, representative data selection increases the accuracy by 4.4%. For some cases (#14, #21 and #40 in Figure 5), the accuracy of the model decreases dramatically without representative data selection. Based on the individual evaluations above, we find that FuzzGuard needs both step-forwarding and the representative data selection for efficiency and accuracy.

6.5 Findings

Interestingly, in our evaluation, we find 23 undisclosed bugs (4 of them are zero-day vulnerabilities). Note that the buggy code of the undisclosed bugs is actually not our target. The goal of FuzzGuard is to increase the efficiency of fuzzing by removing unreachable inputs, instead of triggering new bugs. All the bugs found by FuzzGuard+AFLGo could eventually be discovered by AFLGo. The undisclosed bugs are patched in the new versions of the corresponding programs. For the four zero-day vulnerabilities, we successfully gain the CVE numbers[6]. The vulnerabilities are triggered when we perform target fuzzing on other vulnerabilities. For example, CVE-2018-20189 is found in the fuzzing process of CVE-2017-17501; and CVE-2019-7663 is found in the fuzzing process of CVE-2016-10266. Also, we discover CVE-2019-7581 and CVE-2019-7582 when verifying CVE-2016-9831. After manually analyzing the undisclosed bugs and zero-day vulnerabilities, we find that their locations are quite near the buggy code (i.e., the destination in targeted fuzzing). For example, List 2 and List 3 show the call stacks of triggering CVE-2017-17501 and CVE-2018-20189 respectively. The first 8 pre-dominating nodes are the same for both the two call stacks, while only the last basic blocks differ. We guess the code near the buggy code could be more likely to contain a new bug[7].

7 Understanding

Our evaluation results show that FuzzGuard is highly effective to filter out unreachable inputs, with an average accuracy of 98.7%. We want to understand from the features why FuzzGuard has such a good performance. If the learned features by FuzzGuard are reasonable, the results of FuzzGuard are also understandable. To achieve this goal, our idea is to extract the features from the model and analyze them manually. However, as we know, the high-dimensional features extracted by the deep neural network are hard to be understood directly. Inspired by saliency maps [8], our idea is to project the features to individual bytes (referred to as the key features), and to check whether the key features could impact the execution of the target program.

In particular, to get the key features, we design a mask-based approach to obtain the corresponding key bytes of an input used by the model. The basic idea is as follows: we use a *mask* (i.e., a vector with the same length as the input) to cover the bytes of the input x (the covered fields are set to 0). If the covered input has the same prediction result as the uncovered one (i.e., $f(mask \cdot x) = f(x)$, where f is the CNN model used by FuzzGuard), the covered fields will not impact the prediction result, which means that they are not the key features. By increasing the number of covered fields in the input step by step, we could acquire all the key features in the end. The mask at this time is referred to as the *maximum mask*. For example, an input is shown in Figure 6. The *mask* sets the value of the shaded part of the input to 0. When $f(x) = f(m \cdot x)$, the shaded part will not impact the reachability of the input x. So we shade more bytes and iterate this process. The problem here is that the covered fields have too many combinations. So our idea is to leverage gradient descent to calculate the maximum mask. In particular, we adjust the *mask* according to the deviation between the predicted label y^p and the real label y of x until $y^p = y$. To utilize this approach, we design a loss function that considers not only the deviation between the predicted and actual values, but also the coverage rate in the mask as follows:

$$loss = \frac{\sum_{i=1}^{n} mask_i}{n} + \frac{\sum_{i=1}^{m} (y_i^x - y_i)^2}{m}$$

where n is the number of bytes of *mask* and m is the length of y mentioned in Section 4.3. When the gap between y^p and y is minimal and the number of covered bytes is maximum, the uncovered bytes in x are the key features, which are the fields in the input affecting the reachability viewed by FuzzGuard. In this way, the key features could be compared with the constraints in the target program to check whether the key features can really impact the execution.

```
             0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
                                           number_colors bits_per_pixel
00000000h: 28 00 00 00 04 00 00 00 01 00 00 00 01 00 20 00
00000010h: 00 26 00 00 17 00 00 00 00 00 00 00 00 00 00 00
00000020h: 01 00 00 00 00 00 00 00 00 00 26 26 26 26 26 26
00000030h: 26 26 26 26 26 26 26 26 26 26 26 26 26 26 26 26
00000040h: 26 26 26 26 26 26 26 26 26 26 26 26 26 26 26 26
00000050h: 26 26 26 28 36
```

Figure 6: A PoC of CVE-2018-20189.

For example, the PoC (a PNG file) of CVE-2018-20189 is shown in Figure 6. The key features in this PoC are unshaded. After manual analysis, we verify that the field from offset `0x0e` to `0x0f` (`bits_per_pixel` in List 1) in the input decides the execution direction of the branch in Line 6; and the fields from offset `0x0c` to `0x0d` (`number_colors` in List 1) in the input impact the execution. For example, when `bits_per_pixel` < 16 or `number_colors` $\neq 0$, the buggy code will be executed. The bug will be triggered when `bits_per_pixel` > 8. Through the above analysis, we can confirm that the key features do affect the reachability of the input, which means that the model successfully captures the fields as features when the number of such inputs is enough for training.

```
1   ThrowReaderException(...);
2 if (dib_info.colors_important > 256)
3   ThrowReaderException(...);
4 if ((dib_info.image_size != 0U) && (dib_info.image_size
      > file_size))
5   ThrowReaderException(...);
6 if ((dib_info.number_colors != 0) ||
      (dib_info.bits_per_pixel < 16)) {
7     image->storage_class=PseudoClass;
```

Listing 1: The vulnerable code of CVE-2018-20189.

[6]CVE-2018-20189, CVE-2019-7581, CVE-2019-7582, CVE-2019-7663.

[7]One reason could be that both the two pieces of code are written by the same developer.

8 Discussion

Benefit to input mutation. Most of the current fuzzers focus on mutating inputs for enhancing the performance of fuzzing (e.g., AFL [2], AFLFast [10] and AFLGo [9]). Different from them, our idea is to help DGF filter out unreachable inputs. Interestingly, we find our approach could also potentially help them to optimize the strategy of input mutation. If a fuzzer knows the fields in inputs impacting the execution, it can mutate them for letting the program execution reach the buggy code. Modification of other fields would not help in this process. Based on the understanding of features extracted by FuzzGuard, we find that FuzzGuard could learn the fields impacting the execution (see Section 7). Thus, FuzzGuard could further help the DGF in the process of input mutation.

Learning models. Intuitively, the convolutional architecture uses local patterns. But CNN can actually handle non-local patterns as long as it has enough neural network layers. RNN is similar: when it has enough layers, it can handle non-local patterns; otherwise, it will forget former features. However, the overhead of RNN to handle long data is very large. So we choose to use a 3-layer CNN. In our evaluation, the results show that CNN achieved a good performance (1.9% false positive rate and 0.02% false negative rate on average), which may indicate that most key features in the inputs are local patterns (e.g., the field `bits_per_pixel` in Figure 6). This is understandable: for a single constraint in an if-statement, it usually relies on the local bytes in inputs to make decisions.

Memory usage. In theory, we could keep the unreachable inputs in memory forever to avoid missing a PoC. However, in real situation, the memory is limited. So our idea is to remove those inputs that are highly impossible to reach the buggy code. In other words, if an input is judged as “unreachable” by the updated models for dozens of times, it is highly possible that it cannot reach the buggy code. In this way, we could save memory while at the same time keeping the accuracy. Based on our evaluation, no PoC is dropped in this way.

9 Related Work

Traditional Fuzzers. A lot of state-of-the-arts are proposed in recent years. AFL [2] is a representative CGF fuzzer among them, which gives other fuzzers a guidance. For example, Böhme et al. [10] use the Markov model to construct the fuzzing process. It chooses the seeds which exercise the low-frequency execution paths, and then mutates them to cover more code to find bugs. FairFuzz [24] is similar to AFLFast [10], but it provides new mutation strategies (i.e., overwritten, deleted and inserted). Gan et al. [16] fix the problem of path collision in AFL by correcting the path coverage calculation in AFL. Another variant of AFL is AFLGo [9], it selects the seeds which have the execution path closer to the targets path, and mutates them to trigger the target bugs. And Chen et al. [12] improve AFLGo by new strategies of seed selection and mutation. Some researchers improve the effectiveness by traditional program analysis. For example, Li et al. [25] use static analysis and instrumentation to acquire the magic number position during execution and apply them to the mutation to improve the execution depth of the test case. Chen et al. [13] use dynamic techniques such as colorful taint analysis to find bugs. Rawat et al. [30] use both static and dynamic analysis techniques to obtain control flow and data flow information to improve the effectiveness of the mutation. Chen et al. [14] discover memory layouts to perform accurate fuzzing. Different from their work, we leverage deep-learning-based approach to filter out unreachable inputs to increase the performance of fuzzing.

Learning-based Fuzzers. There are also some fuzzers using intelligent techniques. For example, You et al. [35] extract vulnerable information from CVE descriptions and trigger the bugs in Linux kernel. Wang et al. [33] learn the grammar and semantics features from a large number of program inputs through probabilistic context sensitive grammar (PCSG), and then generate program inputs from that PCSG. Similarly, there are some previous studies [17, 28, 29] training static models to improve the mutation strategy of the fuzzer by generating inputs that are more likely to trigger bugs. Godefroid et al. [17] apply RNN to learn the grammar of program inputs through a large number of test cases, and further leverage the learned grammar to generate new inputs consequently. Rajpal et al. [29] utilize a LSTM model to predict suitable bytes in inputs and mutates these bytes to maximize edge-coverage based on previous fuzzing experience. Nichols et al. [28] train a GAN model to predict the executed path of an input. Chen et al. [15] apply gradient descent algorithm to solve the path constraint problem and find the key bytes in an input to the buggy code. She et al. [31] also utilize gradient descent to smooth the neural network model and learn branches in the program to improve program coverage. Different from these studies which mainly focus on mutating inputs to achieve high code coverage or to efficiently reach target buggy code, the goal of FuzzGuard is to help DGF filter out unreachable inputs, which is complementary and compatible with other fuzzers, instead of replacing them.

10 Conclusion

Recently, DGF is efficient to find the bugs with potentially known locations. To increase the efficiency of fuzzing, most of the current studies focus on mutating inputs to increase the possibility to reach the target, but little has been done on filtering out unreachable inputs. In this paper, we propose a deep-learning-based approach, named FuzzGuard, which predicts reachability of program inputs without executing the program. We also present a suite of novel techniques to handle the challenge of lacking representative labeled data. The results on 45 real bugs show that up to 17.1$\times$ speedup could be gained by FuzzGuard. We further show the key features learned by FuzzGuard, which indeed impact the execution.

Acknowledgments

The authors would like to thank our shepherd Konrad Rieck and anonymous reviewers for their insightful comments. The authors are supported in part by Beijing Natural Science Foundation (No. JQ18011), NSFC U1836211, 61728209, National Top-notch Youth Talents Program of China, Youth Innovation Promotion Association CAS, Beijing Nova Program, National Frontier Science and Technology Innovation Project (No. YJKYYQ20170070).

References

[1] podofo. http://podofo.sourceforge.net, 2006.

[2] American fuzzy lop. http://lcamtuf.coredump.cx/afl, 2018.

[3] Information of cve-2018-20189. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-20189, 2018.

[4] Rectified linear unit. https://ldapwiki.com/wiki/Rectified%20Linear%20Unit, 2018.

[5] Dominator (graph theory). https://en.wikipedia.org/wiki/Dominator_(graph_theory), 2019.

[6] Networkx. https://networkx.github.io, 2019.

[7] pytorch. https://pytorch.org/, 2019.

[8] Julius Adebayo, Justin Gilmer, Michael Muelly, Ian Goodfellow, Moritz Hardt, and Been Kim. Sanity checks for saliency maps. In Advances in Neural Information Processing Systems, pages 9505–9515, 2018.

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

[10] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS 2016), pages 1032–1043. ACM, 2016.

[11] Zhaowei Cai and Nuno Vasconcelos. Cascade r-cnn: Delving into high quality object detection. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 6154–6162, 2018.

[12] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: towards a desired directed grey-box fuzzer. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pages 2095–2108. ACM, 2018.

[13] Kai Chen, DengGuo Feng, PuRui Su, and YingJun Zhang. Black-box testing based on colorful taint analysis. Scientia Sinica Informationis, 55(1):171–183.

[14] Kai Chen, Yingjun Zhang, and Peng Liu. Dynamically discovering likely memory layout to perform accurate fuzzing. IEEE Transactions on Reliability, 65(3):1180–1194, 2016.

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

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

[17] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, pages 50–59. IEEE Press, 2017.

[18] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. Deep learning, volume 1. MIT press Cambridge, 2016.

[19] Kai Kang, Hongsheng Li, Junjie Yan, Xingyu Zeng, Bin Yang, Tong Xiao, Cong Zhang, Zhe Wang, Ruohui Wang, Xiaogang Wang, et al. T-cnn: Tubelets with convolutional neural networks for object detection from videos. IEEE Transactions on Circuits and Systems for Video Technology, 28(10):2896–2907, 2018.

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

[21] D Kinga and J Ba Adam. A method for stochastic optimization. In International Conference on Learning Representations (ICLR), volume 5, 2015.

[22] lcamtuf. America Fuzz Loop strategies. https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html, 2014.

[23] Erich L Lehmann and George Casella. Theory of point estimation. Springer Science & Business Media, 2006.

[24] Caroline Lemieux and Koushik Sen. Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage. In Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering, 2018.

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

[26] Edwin David Lughofer. Flexfis: A robust incremental learning approach for evolving takagi–sugeno fuzzy models. IEEE Transactions on fuzzy systems, 16(6):1393–1410, 2008.

[27] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. Communications of the ACM, 33(12):32–44, 1990.

[28] Nicole Nichols, Mark Raugas, Robert Jasper, and Nathan Hilliard. Faster fuzzing: Reinitialization with deep neural models. arXiv preprint arXiv:1711.02807, 2017.

[29] Mohit Rajpal, William Blum, and Rishabh Singh. Not all bytes are equal: Neural byte sieve for fuzzing. arXiv preprint arXiv:1711.04596, 2017.

[30] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS 2017). ISOC, 2017.

[31] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program learning. In 2019 IEEE Symposium on Security and Privacy (SP). IEEE, 2019.

[32] Michael Sutton, Adam Greene, and Pedram Amini. Fuzzing: Brute Force Vulnerability Discovery. Pearson Education, 2007.

[33] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In Proceedings of the 38th IEEE Symposium on Security & Privacy (S&P 2017). IEEE, 2017.

[34] Xiang Wu, Ran He, Zhenan Sun, and Tieniu Tan. A light cnn for deep face representation with noisy labels. IEEE Transactions on Information Forensics and Security, 13(11):2884–2896, 2018.

[35] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pages 2139–2154. ACM, 2017.

[36] Zhedong Zheng, Liang Zheng, and Yi Yang. A discriminatively learned cnn embedding for person reidentification. ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM), 14(1):13, 2018.

Appendix

```
0x665abb in WriteOnePNGImage coders/png.c:7061
0x677891 in WriteMNGImage coders/png.c:9881
0x479f3d in WriteImage magick/constitute.c:2230
0x47a891 in WriteImages magick/constitute.c:2387
0x42bb9d in ConvertImageCommand magick/command.c:6087
0x43672e in MagickCommand magick/command.c:8872
0x45eeaf in GMCommandSingle magick/command.c:17393
0x45f0fb in GMCommand magick/command.c:17446
0x40c895 in main utilities/gm.c:61
```

Listing 2: The sequence of calls to trigger CVE-2017-17501.

```
0x548b71 in WriteOnePNGImage coders/png.c:7263
0x551d97 in WriteMNGImage coders/png.c:9881
0x450f60 in WriteImage magick/constitute.c:2230
0x4515da in WriteImages magick/constitute.c:2387
0x4215bc in ConvertImageCommand magick/command.c:6087
0x427e48 in MagickCommand magick/command.c:8872
0x44113e in GMCommandSingle magick/command.c:17393
0x441267 in GMCommand magick/command.c:17446
0x40be26 in main utilities/gm.c:61
```

Listing 3: The sequence of calls to trigger the zero-day vulnerability.