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

Abstract

Recently, directed grey-box fuzzing (DGF) becomes much more 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 test only the potential buggy code whose location is known (e.g., testing the high-risk code such as string operations). For the efficiency, all the inputs generated by an ideal DGF should reach the buggy code, hoping to trigger the bug. Any unreachable input will waste the time spent on execution. Unfortunately, in real situation, large numbers of the generated inputs miss the target, greatly impacting the efficiency of testing, especially when the buggy code is hard to reach.

In this paper, we propose a deep-learning based approach to predict the reachability of inputs and filter out those unreachable ones, which works together with DGF fuzzers instead of replacing them. In the process of combining deep learning with fuzzing, we design a suite of new techniques (e.g., *step-forwarding approach*, *representative data selection*) to solve the problems of unbalanced labelled data and low efficiency. We implemented a prototype called FuzzGuard and evaluated it using 45 real vulnerabilities. The results show that FuzzGuard boosts the fuzzing efficiency of the state-of-the-art DGF (e.g., AFLGo) up to 17 times. We also found 19 undisclosed bugs, and 4 zero-day bugs (we have got CVE numbers). Finally, we design an approach to understand the extracted features of the deep neural network model, and find them correlate with the constraints in target programs, which indeed impacts the execution on code level.

1 Introduction

Fuzzing is an automated program testing technique, generating a good number of inputs to a computer program (called *target program*), and monitoring the target program for exceptions such as crashes. It can usually be 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) for a given CVE [5] whose buggy code is known. There are many directed fuzzing tools such as AFLGo [11], SemFuzz [45] and Hawkeye [15]. 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 situation, a large number of the generated inputs could miss the target, especially when the buggy code is hard to trigger. Running the unreachable inputs wastes lots of time since they can never trigger the bug. Facing this situation, various techniques (e.g., symbolic execution [24]) are designed to generate reachable inputs. However, even for the state-of-the-art DGF tools, the ratio of unreachable inputs is still high. Based on our evaluation using a state-of-the-art DGF fuzzer (AFLGo [11]), on average, over 91.7% of the inputs cannot reach the buggy code (Section 6).

Is it possible to know whether an input could reach the buggy code before executing the program? If so, there is no need to run the target program with the unreachable inputs, which further saves the overall time of fuzzing. However, without executing the target program, it is very challenging to tell whether an input never seen before could reach the buggy code or not. Traditional program analysis approaches such as symbolic execution [24], theoretically, could build a model using the constraints of the target program to judge the reachability of an input. However, due to the complexity of programs (e.g., path explosion) and low efficiency of symbolic execution, building such models is almost impossible.

Inspired by the success of pattern recognition [35] 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 labelled inputs (i.e., the reachability of inputs) from previous executions, we could utilize the model to predict the reachability of the newly generated inputs without running the target program. However, due to the differences between traditional pattern recognition of images and classification of program inputs, generating a suitable model for DGF is highly challenging.

Challenges. **C1:** Lack of balanced labelled data. As we know, in the field of pattern recognition, to generate a model for object classification (e.g., cats and dogs), the training process needs enough and balanced labelled data. 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 training a model for program input classification, the labelled data are usually unbalanced. Especially, in the early stage of fuzzing, the situation is even more severe since the first reachable input may show up very late (e.g., in Figure 3, the first reachable input is generated after more than 22.6 million executions, which means the dataset contains only 1 reachable input while having over 22.6 million unreachable ones). Without the balanced labelled data, the trained model will be prone to overfitting. One may think of extending the labelled data just like the way of image transformation (e.g., resizing, distortion, perspective transformation). As we know, transforming an image will not change the identified object, which could increase the number of the object's images to balance the training data. However, such transformation cannot be applied on program inputs since even one bit flip may change the execution path of an input and further impact its label (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 pre-trained model fail to predict the reachability of the new input. 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 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. So it is highly challenging to decide the suitable time for training the model while keeping its high accuracy.

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 the reachability of a newly generated input to the buggy code by learning from the execution with previously generated inputs. If the result of prediction is unreachable, the fuzzer shouldn't execute this input any more, which saves the time spent on real execution. Note that FuzzGuard is not meant to replace a DGF 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, prediction, and model updating. (1) In the first phase, the DGF fuzzer works as usual, generating a bunch of inputs and testing whether a bug is triggered. At the same time, FuzzGuard saves the reachability of each input and trains the initial model using the labelled data. Note that the labelled data are unbalanced (C1). To solve this problem, we design a *step-forwarding* approach: choosing the pre-dominating nodes (of the buggy code) as the mid-term 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 DGF fuzzer generates a new bunch of inputs, FuzzGuard utilizes the model to predict the reachability of each input. As mentioned in C2, the pre-trained model may not work for the newly generated inputs. To solve this problem, we design a *representative data selection* approach, which randomly samples some inputs for training before letting the model make decision. We also design an approach to minimize the number of samples by seed comparisons. (3) In the third phase, FuzzGuard updates the model using the labelled 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 decreasing the number of updating through observing the model accuracy and whether new untouched pre-dominating nodes are reached.

We implement FuzzGuard on the base of AFLGo [11] (an open-source state-of-the-art DGF fuzzer), and evaluate the performance using 45 real vulnerabilities on 10 popular programs (randomly chosen from CVEs in last 3 years). The results show that FuzzGuard boosts the fuzzing performance by $1.3\times$ to $17.1\times$. Interestingly, we find that the longer the fuzzing process taken, the more performance FuzzGuard could gain. Also, more time could be saved if a unreachable input shows earlier. We also find 23 undisclosed bugs, and successfully get 4 CVE numbers. 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 impact the execution on code level.

Contribution. The contributions of this paper are as follows:

• *New technique*. We designed and implemented FuzzGuard which helped DGF fuzzers 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, *representative data selection*. Evaluation results show that up to 94.7% of fuzzing time can be saved for state-of-the-art DGF fuzzers (e.g., AFLGo). We also plan to release our FuzzGuard for helping researchers in the community.

• *New understanding*. We designed an approach to study the features utilized by FuzzGuard's deep learning model for prediction, and found them correlate with the branches in target programs. The understanding of such relationship help to explain the deep learning model and further help to improve FuzzGuard.

2 Background

In this section, we give a brief background of fuzzing and deep learning, and show recent studies utilizing deep learning to improve the fuzzing process.

2.1 Fuzzing

Fuzzing [34] is one of the classical software testing techniques to expose exceptions of a computer program [42]. The main idea of fuzzing is to feed a massive number of inputs (i.e., test cases) to the target program, and to capture exceptions in the program execution. Such exceptions may expose bugs or severe vulnerabilities. Fuzzing can usually be divided into three categories: black-box, grey-box and white-box fuzzing. Black-box fuzzing feeds programs with randomly inputs without aware of the internal program structures. White-box fuzzing leverages program analysis to study the program internals for systematically facilitating fuzzing. Grey-box fuzzing mainly utilizes instrumentation to glean program information to increase code coverage or to reach certain critical program statements. Among those three categories, grey-box fuzzing becomes very popular due to the high efficiency and reasonable performance overhead.

Coverage-based Grey-box Fuzzing. One main goal of fuzzing is to expose as many bugs as possible. Therefore, some fuzzers [4, 12, 16, 17, 30, 31] aim to achieve high code coverage of the target program, hoping to accidentally trigger the bug, namely *Coverage-based Grey-box Fuzzing* (CGF). Typically, to achieve higher code coverage, a coverage-based grey-box fuzzer generates test cases by mutating the seed input which could traverse previous undiscovered program statements. AFL [4], as a representative of CGF, employs light-weight compile-time instrumentation technique and genetic algorithms to automatically discover interesting test cases, select seed inputs that trigger new internal states in the fuzzing process, and mutate seed inputs in various ways (e.g., bit and byte flips, simple arithmetics, stacked tweaks and splicing [28]). There are some fuzzers aim to improve the efficiency of CGF by various strategies of seed inputs selection [12] or mutation [30, 31].

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 [11, 15, 45] 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 becomes very common since some kinds of code may have higher probability 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 fuzzers are utilized to generate a proof-of-concept exploits toward the buggy code [45]. With different goals from CGF, DGF usually chooses the seed inputs that are closer to the buggy code for mutating, hoping to quickly reach the buggy code and trigger the bug. For example, AFLGo [11] calculates the distance between the execution trace of a test case with the path from program start to the buggy code in the control flow graph; then utilizes the distance to choose suitable inputs for mutation. SemFuzz [45] chooses the seed inputs whose execution traces are with the shortest distance to the buggy code.

However, even for state-of-the-art fuzzers, still lots of time is spent on unnecessary executions. All the fuzzers rely on executing the target program to judge whether an input could trigger the bug. Actually, if an input cannot reach the buggy code, it will never trigger the bug. Never till now, to the best of knowledge, could a fuzzer removes unreachable input without executing the target program, which greatly wastes the time. In our experiments (see Table 2), 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, huge amount of time could be saved. In this paper, we design such filter called FuzzGuard, which leverages 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.

2.2 Deep Learning

Recently, deep learning [29] shows its power in many fields such as computer vision, speech recognition, natural language processing, etc. It is able to extract features from raw inputs without human involvement, which could discover intricate features from large data sets. There are some recently impressive improvements made by deep learning [9, 10, 32]. For example, Convolutional Neural Network [21] (CNN) models have demonstrated their outstanding classification performance on image recognition [27], including face recognition [44, 46] and object recognition [14, 23] and so on. They are good at handling the problem of image recognition, image detection, speech recognition because they can directly input the raw image or text data, avoiding complicated preprocessing. Recurrent Neural Network [40] (RNN) models have good abilities to learn features from the sequence data and to solve relative problems (e.g., machine translation [8] and speech recognition [22]). Generative adversarial network [20] (GAN) models have achieved great success in generating incredibly realistic fake data [19] by cooperative game between a generator and a discriminator.

Apply deep learning on Fuzzing. Thanks to recent advances of deep learning, security researches apply deep learning to fuzzing, which provides new insights for solving difficult problems in previously researches. For example, some studies utilize RNNs to generate program inputs which have higher code coverage [18]. Some studies [37] 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. GANs could be used to predict the executed path of an input [36] to improve the performance of the AFL [4] fuzz testing framework. Angora [16] and NEUZZ [41] adapt gradient descent algorithm (along with byte-level taint tracking) and CNNs 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 fuzzers (e.g., AFLGo), which means we could further increase the fuzzing efficiency together with the performance boost by other fuzzers.

3 Motivation

As mentioned above, current DGF fuzzers rely on executing the target program to determine whether an input could reach the buggy code, and leverage the reachability for further mutating the input. In the fuzzing process, lots of inputs cannot reach the buggy code in the end (impossible to trigger the bug). Based on our evaluation, no more than 91.7% of the inputs can hit the buggy code on average (see Table 2). Executing millions of unreachable inputs could cost very long time (e.g., 76 hours for a million of inputs when fuzzing Podofo, a library to work with the PDF file format with a few tools [2]). Especially, when the execution of the program is very slow or the buggy code is embedded deep in the program, the wasted time is even more. If there is an approach that can determine the reachability of an input without executing the program with the input, lots of time could be saved.

Inspired by recent success of deep learning in pattern recognition [14, 23, 44, 46], we are wondering whether deep learning could be applied to identify unreachable inputs. We carefully compare the process of pattern recognition and input identification, and find they are quite similar: they both classify data (a certain object v.s. unreachable inputs) based on either a priori knowledge or on statistical information extracted from the patterns (many labelled images of the object v.s. many labelled inputs from previous executions). However, they do have essential differences (e.g., the distribution of labelled data, efficiency requirements, etc.) which make the process of unreachable input identification very challenging (see Section 1).

```
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.

Example. List 1 gives an example. The vulnerable code is at Line 6. So the goal of DGF is to generate as many as inputs that could reach there and hope to trigger the bug. The seed input chose by the state-of-the-art fuzzer (AFLGo) is `not_kitty.png`. It takes 13 hours to generate 16 million inputs and fuzz 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 there are several paths that 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 feathers of reachable inputs and to identify future reachable ones. Based on our evaluation, nearly 14 million inputs (84.1%) could be 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%, respectively.

Scope and Assumption. Different from previous researches on fuzzing using deep learning [16, 18, 36, 37], 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 fuzzer and can work together

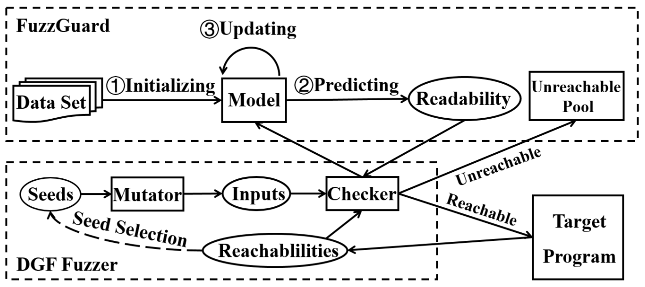


Figure 1: An overview of FuzzGuard.

(The Checker in DGF fuzzer is a function added by FuzzGuard.)

with them, without replacing them. Also note that FuzzGuard is designed to find the bugs hard-to-trigger. If a bug can be easily triggered, there is no need to make a model for filtering unreachable inputs for the bug. Based on our evaluation, actually, 80% of the bugs are hard to trigger[1].

4 Methodology

We propose the design of FuzzGuard, a deep-learning-based approach to felicitate DGF to filter out unreachable inputs without really executing the target program with them. Such 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 phrase: model initialization (MI), model prediction (MP) and model updating (MU). A DGF fuzzer is also shown in the figure. Firstly, the fuzzer generates a good number of inputs and observes any exceptions. At the same time, by observing the reachability of the inputs, MI works step by step to generate an online deep learning model. Then MP utilizes the model to judge whether a newly generated input could reach the buggy code. If so, the input is fed into the program for real execution. When a number of inputs are executed by the program, MU further updates the model to increase its performance. Otherwise, the input should not be executed and the time spent on execution could be saved. Note that these inputs are not abandoned. Instead, they are saved for further analysis after the model is updated, in case of missing any PoC. Although generating and using the model looks like the normal process of pattern recognition, combining deep learning with fuzzing brings in several new challenges.

Still, we use the example in List 1 to illustrate. A DGF fuzzer first generates a bunch of inputs (e.g., "Inputs" in the Figure 2) and runs the target program with them. In this process, the inputs (referred to as *"data"*) and their reachabilities (referred to as *"label"*) are recorded[2]. An ideal situation for training a deep learning model is that the labelled data are balanced. That is, about half of the inputs could reach the buggy code while the other cannot. Unfortunately, in real situation, the data are usually extremely unbalanced in the initial process of fuzzing. To solve this problem and make the model useful even before the balanced data are collected, we design a step-forwarding approach which lets the inputs reach the pre-dominating nodes of the buggy code (i.e., B_0, B_1 and B_2 in the Figure 2) step-by-step to the destination (i.e., B_3 in the Figure 2). Particularly, FuzzGuard chooses a pre-dominating node as a middle-stage destination (i.e., referred to as *"target"*) and generates a model to filter out unreachable inputs to the node. The pre-dominating node should have balanced labelled data and also be closest to the buggy code. In this example, we find the balance that could reach at the node B_1 which is also the nearest node to the destination. Then MP judges whether a newly generated input could reach there using the model. For reachable inputs (e.g., <0,1,0,0> in the figure), FuzzGuard runs the program with it and records whether it can really reach the buggy code. Such information is further leveraged to continuously update the model by MU. After more inputs are tested, a closer pre-dominating node having the balanced labelled data will appear (e.g., B_2 in this case). Such 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 challenging 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 and gain balanced data. 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 start point and the buggy code is in B_3. B_1 and B_2 are the pre-dominating nodes of B_3. In the beginning of fuzzing, no input could reach B_3, while half of the inputs could reach B_1. 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 going further, B_2 and B_3 will get balanced data for training.

[1]We consider those vulnerabilities that are hard to trigger when they cannot be triggered in 3 hours in the fuzzing process.

[2]If any input could trigger the bug, the fuzzing process will end. Unfortunately, in real situation, the fuzzing usually lasts for several hours to even tens of days after testing on billions of inputs.

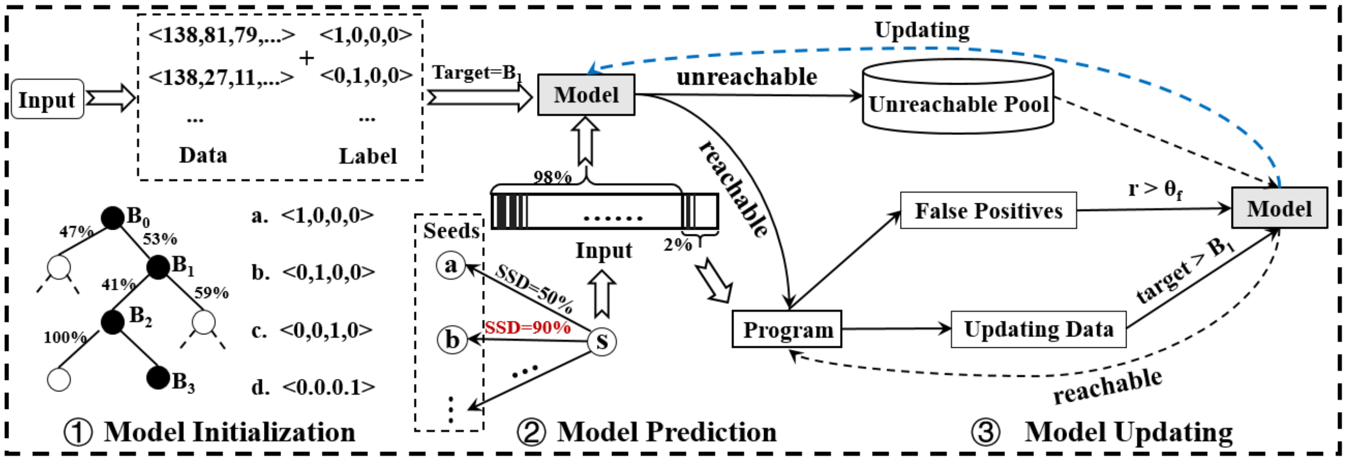


Figure 2: An example of filtering unreachable inputs by FuzzGuard.

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 design to label the reachability of the nodes (i.e., $B = \{B_1, B_2, ..., B_n\}$) in the array L. For each element $l_i \in L$, it represents the input could let the program execute to B_i, but not $B_{i+1} (i < ||L||)$. $||L||$ represents the length of L, which equals the number of the pre-dominating nodes. As shown in Figure 2, for the input a, the label is represented as $L_a =< 1,0,0,0 >$, which means that B_0 is reached by the input but others are not. Similarly, $L_b =< 0,1,0,0 >$ means that the input b can let the execution reach B_0 and B_1, but not B_2. $L_d =< 0,0,0,1 >$ means that the buggy code is finally reached. For an input, we map it to a vector I: each byte of the input ($byte_i$) is mapped to an element in the vector (I_i).

After designing the format of inputs and labels, we carefully choose a deep learning model. Such model should be good at extracting features from inputs and make 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 [14, 23, 44, 46]. Similarly, the features of inputs which 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. 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 latter two lays 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.

In this way, we can learn the reachability to all nodes of the inputs in a CNN model, which has shown a good performance on classification tasks [14, 23, 44, 46]. The goal of the model is to learn a target function F, which could predict the label Y of the data X, i.e., $Y = F(X)$. The data X is represented as a n dimensional vector like I, and the predicted label Y is represented as a $||Y||$ dimensional vector like L. The function F is composed of a number of filters, for each filter w in the model, the output at the i-th vertex (location) is

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

where $x_i = [x_{i-k+1}, \ldots, x_{i+k+1}]^T$ is the local input at the i-th vertex, $i-k < j < i+k$ indicates that the j-th vertex is a neighbor of the i-th vertex, and the k is the width of the filter w. Gradient descent algorithm will update weights of each filter w_i to decrease the loss value $loss = \frac{1}{N}\sum_{i=1}^{N}(L_i - Y_i)^2$, which indicates a criterion that measures the mean squared error between each element of the predicted Y_i and actual label L_i of X_i. When the loss value is close to 0, we believe that the target function in the classification model has been convergent and the model could be used to predict new 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 labelled data for model updating. In particular, for an input I, assume that the model can only predict the pre-dominating nodes before B_k (i.e., *the target*), and the prediction result is P, where only an element in P equals 1 (i.e., $P_i = 1$ and $P_j = 0 (j \neq i)$). The following function f is used to judge whether the input I is reachable.

$$f(I) = \begin{cases} reachable & P_i = 1 \land i \geq k \\ unreachable & P_i = 1 \land i < k \end{cases}$$

However, in real situation, we find the prediction results are not accurate enough, even after many labelled 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. As we know, the mutation strategy of DGF generates inputs from a seed input. Most of the inputs mutated from the same seed only have few differences with each others, while lots of 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_a mutated from the seed a may failed to predict the labels of the data in S_b mutated from the seed b.

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. Then FuzzGuard utilizes the trained model to predict the reachability of the rest inputs. Suppose for two mutations, a DGF fuzzer generates m_1 and m_2 inputs. A simple solution of selecting representative inputs is to select a fixed number of inputs from m_1 and m_2 (e.g., 5% of the data set). However, in real execution, even 5% of the data set is a big number (e.g., over 20 thousand inputs), and execution using these inputs cost lots of time. Our idea is to select less inputs from m_2 if m_2 is similar enough to m_1. Note that we cannot directly say m_1 and m_2 are similar only through the fact whether they are mutated from the same seed. 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 (as shown in Figure 2). So our idea is to directly compare the seed mutating m_1 and the seed mutating m_2 only if the two seeds are mutated using the same strategy. We define the Seed Similarity Degree *SSD* between a previous seed R and a new seed S as follows:

$$SSD = \frac{min(||R||,||S||) - \sum_{i=1}^{min(||R||,||S||)} R_i \oplus S_i}{max(||R||,||S||)}$$

In this way, we could measure the similarity between two sets of inputs through their predecessor seeds. When SSD is over a threshold (θ), we consider that the seed S is similar to the seed R, and less data from the inputs mutated from S could be selected. For example, in Figure 2, we select less data (e.g., 2%) from the inputs set that generated by seed the s, because s is similar to the seed b (e.g., SSD=90%). In this way, we could select less inputs for real execution and training without impacting the model's accuracy. Based on our evaluation, 70.5% of the fuzzing time could be saved using this approach.

4.4 Model Updating

To realize online model updating, we utilize *incremental learning* [33] 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 the model's knowledge. The aim of incremental learning is to adapt to new data without forgetting its existing knowledge for the learning model, and it does not retrain the model. It can be applied when training data becomes available gradually over time as the fuzzer generates and exercises new test case continuously. Also incremental learning could help us decrease the time of waiting for data collecting, and filter out more unreachable test cases.

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

We perform model updating when the model is getting "old". We say the model is old when it is not accurate enough or 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 set a buffer to 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 clear the data in the buffer and record false positive rate again. In the second situation, when a new pre-dominating node (i.e., $L_i = 1 \land i > target$) is reached by executing the program with new inputs, the model needs to be updated. 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 low frequency.

To avoid missing a PoC (i.e., filtering out a PoC), we temporarily store unreachable inputs in a buffer, called the unreachable pool, instead of dropping them. When the model is updated, we use the new model to check the inputs in the pool again, and pick out the reachable ones for executing. Based on our our evaluation, the model is accurate enough that no PoC is even put into the unreachable pool.

Table 1: State-of-the-art Fuzzers.

Fuzzers	Category	Target Program	Open Source?
libFuzzer [3]	CGF	Source Code	Yes
AFL [4]	CGF	Binary/Source Code	Yes
AFLFast [12]	CGF	Binary/Source Code	Yes
FairFuzz [30]	CGF	Binary/Source Code	Yes
Steelix [31]	CGF	Binary	No
VUzzer [38]	CGF	Binary	Yes
Skyfire [43]	CGF	Source Code	Yes
Angora [16]	CGF	Source Code	Yes
CollAFL [17]	CGF	Source Code	No
NEUZZ [41]	CGF	Source Code	Yes
AFLGo [11]	DGF	Source Code	Yes
SemFuzz [45]	DGF	Source Code	No
Hawkeye [15]	DGF	Source Code	No

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 (i.e., implementing the data sharing interfaces between the DGF fuzzer and FuzzGuard).

Model Initialization. For the inputs, we use the same setting as the DGF fuzzer. For example, AFLGo limits the length of program inputs to 1 KB as default. So FuzzGuard also uses 1 KB as the length of inputs. In particular, it maps each byte of an input to an integer number in a vector. If the length of the input is less than 1 KB, FuzzGuard pads the rest part of the input with 0, to make sure the inputs to FuzzGuard have equal length for training. For the labeled data (i.e., the classification results), FuzzGuard creates corresponding vectors (as shown in Figure 2). The length of an vector equals the number of the pre-dominating nodes of the buggy code (including the buggy code itself). Usually, the DGF fuzzer (e.g., AFLGo[3]) itself also calculates the pre-dominating nodes for targeted fuzzing. So there is no extra efforts for FuzzGuard to calculate the nodes again. For the training process, we create a CNN model, consisting of three 1-dimensional convolution layers as well as three pooling layers and a fully-connected layer. Every layer uses ReLU [6] as activation function. We add Dropout layers with 20% disabling rate after convolution layers to avoid overfitting. Also we use the Adam optimizer [25] to help the learning function converge to the optimal solution rapidly and stably. We implement the model using PyTorch[4]. The training process ends when the loss value of the learning function becomes stable.

Model Prediction. In this process, recall that FuzzGuard needs to sample a number of inputs from the newly generated

[3]AFLGo starts fuzzing with a target file containing the pre-dominating basic blocks and the buggy basic block, since it needs to calculate the distance between the executed path to the buggy code according to the pre-dominating basic blocks.

[4]PyTorch [7] is an open-source Python machine learning library with GPU acceleration.

Algorithm 1 Function Checker()

```
Input: The parameter to run_target $argv$, $timeout$
1: GlobalVar $trace\_bits, input$
2: $fault \leftarrow 0$
3: if $FuzzGuard(input)$ is $reachable$ then
4:     $fault \leftarrow run\_target(argv, timeout)$
5:     $reachability \leftarrow check\_trace(trace\_bits)$
6:     $Save(reachability)$
7: end if
8: return $fault$
```

ones in each round of mutation, and sample less inputs if its seed is similar to previous seeds. Here, we set the threshold of SDG as 0.85 (i.e., $\theta = 0.85$), which means that we combine two sets if the seeds of the two sets have no more than 15% different bytes. Based on our evaluation, setting the threshold using this value has the best performance. For the sampling rate, we choose to use 5% of the total inputs in each round of mutation. For those which could be combined, we choose to sample $(1-\theta)/5$ (i.e., no more than 3%). In this way, the more similar the seed in the current mutation to the previous ones, the less inputs need to be chosen for real execution.

Model Updating. We update the model when it is viewed as old. Considering that models' accuracies vary a lot for different programs, we do not use a fixed value of θ_f in different fuzzing processes. Instead, we dynamically change θ_f according the previous executions: $\theta_f = 1 - acc_{avg}$, where acc_{avg} represents the average accuracy of the models updated in previous executions. To save the memory in the fuzzing process (hundreds of millions of inputs are generated in the fuzzing process), we combine the buffer used to store the training set with the unreachable pool. After updating the model, the inputs in the training set could be abandoned, which is suitable for storing the unreachable inputs.

Deployment of FuzzGuard. As mentioned previously, FuzzGuard is designed to assist DGF instead of replacing DGF. It works as a library and the only modification on a DGF fuzzer is to add a function call to FuzzGuard, which helps to share DGF's data with FuzzGuard for training. Table 1 lists the-state-of-art fuzzers presented in recent years. In our implementation, we choose to use AFLGo[5] in Table 1 since it is the only open source DGF. To achieve the data sharing with the fuzzer, we add a function *Checker()* to afl-fuzz.c in AFLGo. Algorithm 1 shows the details of *Checker()*. This function works instead of the function *run_targe()* in afl-fuzz.c. It will handle parameters to *run_target()* and visit global variables (trace_bits and input). Before feeding them to the target program, an input will be checked by FuzzGuard. Only when it is reachable, the target program will run (see Line 3 to 4 in algorithm). After executing the target program, the reachability of the input will be saved to a buffer (see Line 5 to 6 in algorithm) for futher training. Otherwise, the input is put

[5]https://github.com/aflgo/aflgo

to the unreachable pool of FuzzGuard[6] and there's no need to execute. 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 AFLGo, which is an open source DGF fuzzer in Table 1. According to the experiment results, FuzzGuard boosts the performance of fuzzing up to 17 times faster than that of AFLGo. Then we break down the performance overhead to measure the time cost of FuzzGuard. We also analyze the accuracy of FuzzGuard and show our findings including 23 newly discovered undisclosed bugs (we got 4 CVE numbers).

6.1 Settings

We collect 10 real world programs with different input formats (e.g., BMP, MP4, XML, PDF and PCAP) and randomly choose 27 hard-to-trigger vulnerabilities[7] for these programs from CVEs [1]. In the fuzzing process using our approach, we further find 18 hard-to-trigger undisclosed vulnerabilities. We also include them in our evaluation. Table 2 shows the details of each vulnerability, including program names and line number of the vulnerable code (the column Vuln.). 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[8].

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 2). The ideal comparison is to compare the time of fuzzing using AFLGo (T_{AFLGo}) and the corresponding time when equipping AFLGo with FuzzGuard (T_{+FG}). However, one problem making the comparison unfair is that the fuzzing process is random. Even for the same seed input in two different executions, the crashes will appear randomly, making it very hard to fairly compare T_{AFLGo} and T_{+FG} (the generated inputs in different executions could be quite different). So our idea is to make the mutations of seed inputs be the same in

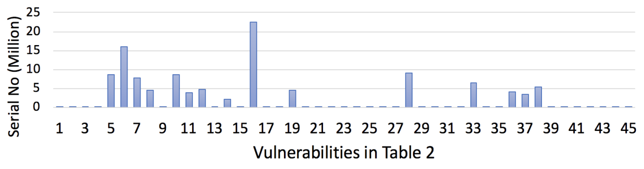


Figure 3: The serial number of the 1st reachable input shows in I_{AFLGo} when fuzzing different vulnerabilities.

the two different executions. Particularly, for a vulnerability of a target program, we use AFLGo to fuzz and record all the mutated inputs I_{AFLGo} in order (the number of the inputs N_{AFLGo} is shown in Table 2) until an exception (e.g., a crash) appears or timeout (200 hours in our evaluation). In this process, the time of fuzzing T_{AFLGo} (as shown in Table 2) is also recorded. Then we utilize the same inputs I_{AFLGo} to test AFLGo equipped with FuzzGuard, and record the number of the inputs $N_{filtered}$ that are filtered by FuzzGuard. 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 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 $I_{filtered}$ are the inputs filtered out by FuzzGuard and t_i stands for the time spent on executing the target program with input i.

Landscape. The results are shown in Table 2, which includes two parts. The upper part shows the vulnerabilities that eventually end with crashes in the fuzzing process. The lower part shows the vulnerabilities that end with timeout. From the table, we find that for all the vulnerabilities, FuzzGuard can increase the runtime performance of AFLGo from 1.3× to 17.1× (see the "Speedup" column in Table 2, which is calculated as follows: $Speedup = T_{AFLGo}/T_{+FG}$). The average performance is increased by 5 times. Note that such performance boost is further added to a DGF (i.e., AFLGo) which has already been optimized.

Understanding the performance boost. To understand the performance of different vulnerabilities, we further study the relationship among the speedup, the time spent on fuzzing, the sizes of programs, etc. The findings are as follows.

- The speedup increases when the time spent on fuzzing goes longer. From the fuzzing process of CVE-2018-17000, we found that in the first 1 hour, the speedup is 2.3 times; and in the last 2 hours, the speedup increases to 8.5 times. This is mainly because the increased number of labelled data could train a more accurate model. Also with the execution going further, more inputs could be filtered out for saving the time on unnecessary executions.
- The quicker the first pre-dominating node is reached, the more time could be saved. From Table 2, we find that for some vulnerabilities, the speedup is not too much (e.g., 1.4 times for

[6]The function Checker dose not need interact with the unreachable pool.

[7]We remove those vulnerabilities that are easy to trigger (can be triggered in 3 hours in the fuzzing process). Such easy-to-trigger vulnerabilities are not our goal (see the assumption in Section 3).

[8]https://developer.nvidia.com/cuda-zone

Table 2: Effectiveness of FuzzGuard.

ID	CVE	Program	Vuln.	UR.	T_{AFLGo}	T_{+FG}	Speedup	N_{AFLGo}	N_{+FG}	Filtered
1	CVE-2017-8366	Ettercap	ec_strings.c:182	99.0%	80.5 h	6.1 h	13.3	49.2 M	3.0 M	93.9%
2	CVE-2016-9399	Jasper	jpc_dec.c:1700	99.7%	46.9 h	3.7 h	12.7	11.3 M	0.6 M	94.3%
3	CVE-2016-9397	Jasper	jpc_dec.c:1881	99.9%	19.7 h	1.6 h	12.0	6.1 M	0.4 M	94.0%
4	-	GraphicsMagick	tiff.c:2375	95.9%	94.8 h	11.1 h	8.6	32.1 M	3.1 M	90.5%
5	CVE-2018-17000	Libtiff	tif_dirwrite.c:1901	99.9%	9.6 h	1.3 h	7.4	8.6 M	0.7 M	91.4%
6	CVE-2018-20189	GraphicsMagick	png.c:7503	99.9%	13.2 h	3.9 h	3.4	16.4 M	2.6 M	84.1%
7	CVE-2017-14733	GraphicsMagick	rle.c:753	99.4%	30.8 h	10.4 h	3.0	17.7 M	5.2 M	70.5%
8	-	Libming	parser.c:3232	99.8%	6.1 h	2.8 h	2.2	12.6 M	4.9 M	61.3%
9	-	Libtiff	tif_swab.c:289	99.7%	29.6 h	15.0 h	2.0	44.7 M	21.1 M	52.8%
10	-	Libtiff	tiffcp.c:1386	99.9%	8.9 h	4.6 h	1.9	15.6 M	7.5 M	51.7%
11	-	Libming	parser.c:3221	99.9%	14.0 h	8.2 h	1.7	13.0 M	7.4 M	43.2%
12	-	Libming	parser.c:3250	99.9%	7.3 h	4.4 h	1.7	16.6 M	9.0 M	46.0%
13	CVE-2016-10266	Libtiff	tif_read.c:346	79.5%	77.9 h	49.8 h	1.6	60.6 M	38.6 M	36.3%
14	-	Libming	parser.c:3089	99.9%	5.2 h	3.4 h	1.5	19.6 M	11.1 M	43.3%
15	CVE-2018-14584	Bento4	Ap4AvccAtom.cpp:83	38.1%	44.0 h	29.9 h	1.5	1.8 M	1.2 M	32.3%
16	-	GraphicsMagick	png.c:3810	99.9%	31.9 h	22.4 h	1.4	22.6 M	15.2 M	32.8%
17	-	Libming	parser.c:3061	99.8%	3.4 h	2.5 h	1.4	18.9 M	11.8 M	37.2%
18	-	Libming	parser.c:3071	99.9%	3.8 h	2.9 h	1.3	17.6 M	11.7 M	33.6%
19	-	Libming	parser.c:3209	99.9%	8.9 h	6.9 h	1.3	30.7 M	22.2 M	27.7%
20	CVE-2016-9827	Libming	outputtxt.c:143	65.5%	7.7 h	6.1 h	1.3	27.3 M	20.6 M	24.6%
21	CVE-2016-9829	Libming	parser.c:1645	99.8%	200.0 h	11.7 h	17.1	32.3 M	1.7 M	94.7%
22	CVE-2017-17880	ImageMagick	webp.c:769	99.1%	200.0 h	12.6 h	15.9	11.5 M	0.7 M	93.9%
23	CVE-2017-17913	GraphicsMagick	webp.c:716	99.5%	200.0 h	15.2 h	13.2	67.5 M	4.4 M	93.4%
24	CVE-2018-18544	ImageMagick	msl.c:8353	99.2%	200.0 h	15.4 h	13.0	7.3 M	0.6 M	92.4%
25	CVE-2017-17501	GraphicsMagick	png.c:7061	98.4%	200.0 h	16.2 h	12.3	56.9 M	3.8 M	93.3%
26	CVE-2016-1835	Libxml2	SAX2.c:2035	99.9%	200.0 h	17.6 h	11.3	92.6 M	5.2 M	94.4%
27	-	ImageMagick	webp.c-403	96.0%	200.0 h	19.8 h	10.1	19.1 M	1.8 M	90.7%
28	-	GraphicsMagick	png.c:6945	96.6%	200.0 h	23.4 h	8.5	30.0 M	3.4 M	88.8%
29	CVE-2016-9831	Libming	parser.c:64	91.9%	200.0 h	27.2 h	7.3	16.1 M	2.2 M	86.6%
30	CVE-2017-8354	ImageMagick	bmp.c:894	98.5%	200.0 h	36.4 h	5.5	9.4 M	1.7 M	82.0%
31	CVE-2017-7378	Podofo	PdfPainter.cpp:1945	99.3%	200.0 h	40.7 h	4.9	2.6 M	0.5 M	79.7%
32	-	Libming	parser.c:3381	99.7%	200.0 h	61.3 h	3.3	38.4 M	11.6 M	69.9%
33	CVE-2018-20591	Libming	decompile.c:1930	99.9%	200.0 h	63.0 h	3.2	38.6 M	11.5 M	70.2%
34	CVE-2017-18028	ImageMagick	tiff.c:1934	64.3%	200.0 h	65.0 h	3.1	12.9 M	4.2 M	67.6%
35	CVE-2017-18229	GraphicsMagick	tiff.c:2433	75.3%	200.0 h	66.7 h	3.0	78.4 M	23.9 M	69.5%
36	-	Libming	parser.c:3095	92.9%	200.0 h	70.0 h	2.9	46.8 M	16.1 M	65.7%
37	-	Libming	parser.c:2993	97.2%	200.0 h	71.8 h	2.8	45.9 M	16.1 M	64.8%
38	-	Libming	parser.c:3126	92.9%	200.0 h	75.3 h	2.7	77.0 M	28.1 M	63.6%
39	CVE-2017-6851	Jasper	jas_seq.c:254	62.4%	200.0 h	89.0 h	2.2	22.3 M	9.8 M	56.0%
40	CVE-2018-12600	ImageMagick	dib.c:1306	99.9%	200.0 h	91.4 h	2.2	3.2 M	1.5 M	54.4%
41	CVE-2018-12599	ImageMagick	bmp.c:2062	99.9%	200.0 h	95.6 h	2.1	3.0 M	1.4 M	52.3%
42	CVE-2018-20570	Jasper	jp2_enc.c:309	99.4%	200.0 h	99.0 h	2.0	28.0 M	13.8 M	50.9%
43	CVE-2018-13112	Tcpreplay	get.c:174	53.2%	200.0 h	105.4 h	1.9	203.3 M	102.8 M	49.5%
44	-	GraphicsMagick	png.c:5007	99.9%	200.0 h	132.3 h	1.5	16.0 M	10.5 M	34.3%
45	CVE-2017-15238	GraphicsMagick	list.c:232	37.2%	200.0 h	146.4 h	1.4	73.0 M	52.3 M	28.4%

the 16th vulnerability). The reason is that most of the inputs cannot even reach the first pre-dominating node, especially before the time when FuzzGuard generates the initial model. In this stage, FuzzGuard is unable to filter out any inputs, which makes no performance boost. We also draw Figure 3 to show the serial number of the first reachable input to the buggy code of each vulnerability, which verifies our guess (the first reachable input of the 16th vulnerability appears after 22.6 million inputs).

• FuzzGuard is effective on both large and small programs. Based on initial feelings, it usually takes more time for a large program to execute once than a small program. So if the number of filtered inputs are the same, the large program could spend less time on executing. Contrary to the initial feeling, from the results, we can find even for small programs (e.g., Libming, Libtiff), there is still lots of time saved in the fuzzing process. This is mainly because a high percentage of inputs have been filtered out.

Based on the understandings above, we find that when the fuzzing process goes longer, the speedup could be increased more. If a DGF fuzzer could generate inputs to reach the first pre-dominating node earlier, the performance could also be increased. Also, our approach can apply to both large and small programs.

Cost. The time cost rate T_{FG}/T_{+FG} of FuzzGuard in each

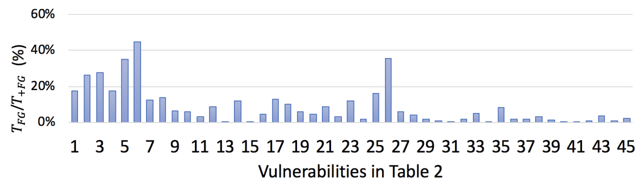


Figure 4: Time Cost of FuzzGuard T_{FG} in T_{+FG}.

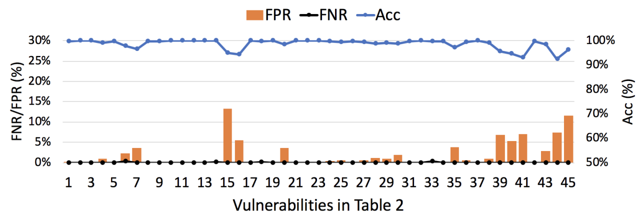


Figure 5: Accuracy of FuzzGuard.

fuzzing process is shown in Figure 4. FuzzGuard averagely takes 9.2% of the total time of fuzzing (T_{+FG} in Table 2). 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. We also evaluate whether the length of an input will impact the cost. Based on our evaluation, the length of the inputs will only slightly impact the cost of FuzzGuard since the training process by GPU will support longer data.

6.3 Accuracy

We measure the accuracy of FuzzGuard. The more accurate it behaves, the more unreachable inputs could be filtered out. Note that no PoC will be missed since FuzzGuard saves the filtered inputs in the unreachable pool for further checking by updated model. Figure 5 shows the false positive rate (orange bar) and false negative rate (black broken line) of FuzzGuard. The broken line in blue represents the accuracy of FuzzGuard for the vulnerabilities in Table 2. 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 $acc = \frac{N_p+N_n-N_{fp}-N_{fn}}{N_p+N_n}$.

From the figure, we can find that FuzzGuard is very accurate (ranging from 92.5% to 99.9%). The average accuracy is 98.7%. The false positive rates for all vulnerabilities are less than 13.3%, and 1.9% on average. Note that false positives will not let a PoC be missed. Nor will it increase the time spent on executing the inputs (such inputs will always be executed by the program if there is no FuzzGuard). The false negative rate is negligible. The false negative rates for most vulnerabilities are 0. The highest one is less than 0.3%. There are only 4 vulnerabilities have false negatives. We further check those false negatives manually and confirm that there are no PoC in those inputs. Even if a PoC is included, as mentioned previously, FuzzGuard will save it to the unreachable pool for further testing by updated models (no PoC will be missed). Such accurate model enables FuzzGuard to have high performance.

```
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
```

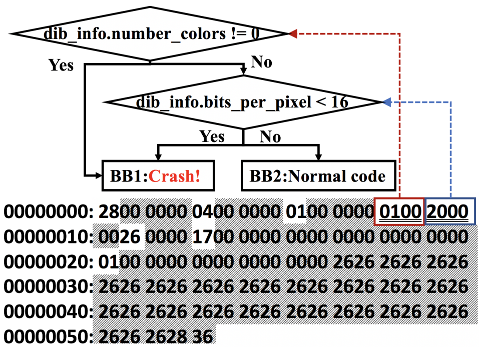
Listing 2: The sequence of calls to trigger CVE-2017-17501.

```
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.

6.4 Findings

Interestingly, in our evaluation, we find 23 undisclosed bugs and 4 of them are zero-day vulnerabilities. The undisclosed bugs are patched in the new versions of the corresponding programs. For the four zero-day vulnerabilities, we applied and successfully gained the CVE numbers (i.e., CVE-2018-20189, CVE-2019-7581, CVE-2019-7582 and CVE-2019-7663). The vulnerabilities are triggered when we perform target fuzzing on other vulnerabilities. For example, CVE-2018-20181 was found in the fuzzing process of CVE-2017-17501, CVE-2019-7663 was found in the fuzzing process of CVE-2016-10266, and we discovered CVE-2019-7581 and CVE-2019-7582 when verifying CVE-2016-9831. After manually analyzing the undisclosed bugs and zero-day vulnerabilities, we found 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



```
00000000: 2800 0000 0400 0000 0100 0000 0100 2000
00000010: 0026 0000 1700 0000 0000 0000 0000 0000
00000020: 0100 0000 0000 0000 0000 2626 2626 2626
00000030: 2626 2626 2626 2626 2626 2626 2626 2626
00000040: 2626 2626 2626 2626 2626 2626 2626 2626
00000050: 2626 2628 36
```

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

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[9]. Therefore, FuzzGuard's ability (i.e., letting the execution be quick to reach the code area near the buggy code) helps to shorten the time to trigger them.

7 Understanding

Our evaluation results show that FuzzGuard is highly effective to filter out unreachable inputs, with the 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 is reasonable, the results of FuzzGuard are also trustworthy. To achieve this goal, our idea is to find the connections among features, program inputs and the target program. However, as we know, the high-dimensional features extracted by deep neural network are hard to be understood directly. So our idea is to project the features to inputs (referred to as f_I), and check whether the f_I could impact the execution of the target program.

In particular, to get f_I, we design a mask-based approach to obtain the corresponding fields of an input used by the features. The basic idea is as follows. We use a mask m (i.e., a vector with the same length as the input) to cover the input x (the covered fields are set to 0). If the covered input has the same prediction result $p = F(m \cdot x)$ (F is the CNN model in FuzzGuard) as the original one, the covered fields will not impact the prediction result, which means that they are not used by the features. By increasing the number of covered fields in the input step by step, we could acquire f_I in the end. The mask at this time is referred to as *maximum mask*. For example, an input is shown in Figure 6. The mask m sets the value of the shaded part of the input to 0. If $F(x)$ and $F(m \cdot x)$ are the same, the shaded part will not impact the reachability. 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 other words, we adjust m according to the deviation between the predicted label p and the ground truth l (the label of x) until the predicted label is consistent with the ground truth. To utilize this approach, we design a loss function *loss* that not only consider the deviation between the predicted and actual values, but also coverage rate in the mask.

$$loss = \frac{\sum_{i \in ||l||}(p_i - l_i)^2}{||l||} + \frac{\sum_{i \in n} m_i}{n}$$

where n is the number of bytes of x before padding. When the gap between p and l is minimal and the number of covered bytes is maximum, the uncovered bytes in x are f_I, which are the fields in the input affecting the reachability viewed by FuzzGuard. In this way, f_I could be compared with the constraints in the target program to check whether f_I can really impact the execution.

For example, the PoC of CVE-2018-20189 is shown in Figure 6. f_I has been calculated through the above approach. The upper part of the figure shows the conditions impacting the execution of the target program (i.e., GraphicsMagick). After manual analysis, we verify that the 15th and the 16th bytes in the input (also in f_I) represents `bits_per_pixel` of the DIB (Device Independent Bitmap) file, which affects the if-statement in the function ReadDIBImage() (see Line 6 of the code in List 1). The 13th and the 14th bytes represent `number_colors` of the DIB file, also impacting the execution. When `bits_per_pixel` < 16 or `number_colors` $\neq 0$, the input can reach the buggy code, but it will trigger the crash only when `bits_per_pixel` > 8. For other identified fields in f_I, our analysis verifies that they really impact the execution, which means that the features learned by FuzzGuard are reasonable.

8 Discussion

Benefit for the strategy of input mutation. Most of current fuzzers focus on mutating inputs for enhancing the performance of fuzzing (e.g., AFL [4], AFLFast [12] and AFLGo [11]). Different from them, our idea to help DGF is to filter out unreachable inputs. Interestingly, we find our approach could also potentially help them to optimize the strategy of input mutation. As we know, DGF would like to identify the fields in inputs impacting the execution, and mutate them for letting the program execution reach the buggy code. Modification on 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). In this way, FuzzGuard could further help the DGF in the process of input mutation.

Learning Models. We adopt CNN as the learning model for FuzzGuard. Actually, there are many other machine learning models. Traditional machine learning methods (e.g., Support Vector Machine [13], Logistic Regression [26] and Decision

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

Tree [39]) usually require manual efforts to select features for classification problems. This is not suitable for our problem since the features related to complex constraints in target programs are exactly what want to avoid handling manually. For the deep learning methods, CNN is good for handling image data, while RNN is famous for handling text data. In the fuzzing process, the fields in inputs are combined together, impacting the execution to the buggy code, which is just like image classification problems (different pixels in a figure combined together as features for object classification). Therefore, we choose to use CNN in our problem.

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 the memory while at the same time keeping the accuracy. Based on our evaluation, no PoC is dropped in this way.

9 Related Work

Since the fuzzing testing technique has been the most popular technique to find program bugs. In the rest part of this section, we will introduce different techniques applied in fuzzing, including traditional techniques and deep learning techniques, also we summarize state-of-the-art work on fuzzing briefly.

Traditional Fuzzers. As we can see in Table 1, a lot of state-of-the-arts are proposed in recent years. AFL [4] is a representative CGF fuzzer among them, who gave other fuzzers [11, 12, 15, 17, 30] a guidance. For example, AFLFast [12] uses the Markov model to construct the fuzzing process, it chooses the seeds which have low-frequency execution traces, and mutates them to cover more code to find bugs. FairFuzz [30] is similar to AFLFast, but it provides new mutation strategies (i.e., overwritten, deleted and inserted). CollAFL [17] fixes the problem of path collision in AFL by correcting the path coverage calculation in AFL. Another variant of AFL is AFLGo [11], it selects the seeds which have the execution path closer to the targets path, and mutates them to trigger the target bugs. Hawkeye [15] improves AFLGO in four aspects, including focusing on all paths to the targets, more precise program analysis and energy assignment strategies, and a adaptive mutation strategy (coarse-grained, fine-grained). Some researchers improve the effectiveness by traditional program analysis. For exmaple, Steelix [31] uses techniques such as static analysis and instrumentation to acquire the magic number position during execution and applies it to the mutation to improve the execution depth of the test case. Vuzzer [38] uses dynamic and static analysis techniques to obtain control flow and data flow information to improve the effectiveness of the mutation. 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, SemFuzz [45] extracts vulnerable information from network and trigger the bugs in linux kernel. Skyfire [43] learns the grammar and semantics features from a large number of program inputs through probabilistic contextsensitive grammar (PCSG), and then generates program inputs from that PCSG. Similarly, there are some previous works [18, 36, 37] training static models to improve the mutation strategy of the fuzzer by generating inputs that are more likely to trigger bugs. Godefroid et.al. [18] apply RNN to learn the grammar of program inputs through large number of test cases, and further leverage the learned grammar to generate new inputs consequently. The study [37] utilizes a LSTM model to predict suitable bytes in an input and mutates these bytes to maximize edge-coverage based on previous fuzzing experience. The previous work [36] trains a GAN model to predict the executed path of an input. Gradient descent algorithm is also applied in fuzzer. Angora [16] applies gradient descent algorithm to solve the path constraint problem and finds the key bytes in an input to the buggy code. NEUZZ [41] also utilizes gradient descent to smooth the neural network model and learns branches in the program to improve program coverage. Different from these works which mainly focus on input generation or increase code coverage, the goal of FuzzGuard is to discard the unreachable inputs to avoid executing them, which is complementary and compatible with other fuzzers, without replacing them.

10 Conclusion

Recently, DGF is efficient to find the bugs with potentially known locations. To increase the efficiency of fuzzing, most of 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 focus on filtering out unreachable inputs and propose an approach without executing a program to judge whether an input could reach the buggy code. In this way, the program could save the execution with the unreachable inputs. Note that this approach is complementary and compatible with other fuzzers. To achieve this goal, we design FuzzGuard, which utilizes deep learning to predict the reachability of the input. We also propose a suite of novel techniques to handle the unique problems when combining deep learning with testing. To evaluate the effectiveness of FuzzGuard, we equipped FuzzGuard on AFLGo. The results on 45 real vulnerabilities show that up to 17.1× speedup could be gained by FuzzGuard. After designing an approach to understand the extracted features, we find that they are correlated with the constraints in the if-statements in the target programs, which indeed impact the execution on code level.

References

[1] Common vulnerabilities and exposures (cve). https://cve.mitre.org, 1999.

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

[3] Libfuzzer. https://llvm.org/docs/LibFuzzer.html, 2016.

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

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

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

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

[8] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[9] Rohan Bavishi, Michael Pradel, and Koushik Sen. Context2name: A deep learning-based approach to infer natural variable names from usage contexts. *arXiv preprint arXiv:1809.05193*, 2018.

[10] Vandana Bhatia and Rinkle Rani. Dfuzzy: a deep learning-based fuzzy clustering model for large graphs. *Knowledge and Information Systems*, pages 1–23, 2018.

[11] 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.

[12] 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.

[13] Christopher JC Burges. A tutorial on support vector machines for pattern recognition. *Data mining and knowledge discovery*, 2(2):121–167, 1998.

[14] 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.

[15] 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.

[16] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. *arXiv preprint arXiv:1803.01307*, 2018.

[17] 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.

[18] 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.

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

[20] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[21] Jiuxiang Gu, Zhenhua Wang, Jason Kuen, Lianyang Ma, Amir Shahroudy, Bing Shuai, Ting Liu, Xingxing Wang, Li Wang, Gang Wang, et al. Recent advances in convolutional neural networks. *arXiv preprint arXiv:1512.07108*, 2015.

[22] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, 29(6):82–97, 2012.

[23] 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.

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

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

[26] David G Kleinbaum and Mitchel Klein. *Logistic regression: a self-learning text*. Springer Science & Business Media, 2010.

[27] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

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

[29] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[30] Caroline Lemieux and Koushik Sen. Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage. *arXiv preprint arXiv:1709.07101*, 2017.

[31] 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.

[32] Yuwei Li, Shouling Ji, Chenyang Lv, Yuan Chen, Jianhai Chen, Qinchen Gu, and Chunming Wu. V-fuzz: Vulnerability-oriented evolutionary fuzzing. *arXiv preprint arXiv:1901.01142*, 2019.

[33] 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.

[34] 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.

[35] Nasser M Nasrabadi. Pattern recognition and machine learning. *Journal of electronic imaging*, 16(4):049901, 2007.

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

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

[38] 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.

[39] S Rasoul Safavian and David Landgrebe. A survey of decision tree classifier methodology. *IEEE transactions on systems, man, and cybernetics*, 21(3):660–674, 1991.

[40] Hojjat Salehinejad, Julianne Baarbe, Sharan Sankar, Joseph Barfett, Errol Colak, and Shahrokh Valaee. Recent advances in recurrent neural networks. *arXiv preprint arXiv:1801.01078*, 2017.

[41] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program learning. *arXiv preprint arXiv:1807.05620*, 2018.

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

[43] 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.

[44] 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.

[45] 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.

[46] 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.