Shi *et al. Cybersecurity* (2022) 5:17
https://doi.org/10.1186/s42400-022-00116-x

Cybersecurity

RESEARCH Open Access

CAMFuzz: Explainable Fuzzing with Local Interpretation

Ji Shi[1,2,3,4], Wei Zou[1,2,3,4], Chao Zhang[5*], Lingxiao Tan[1,2,3,4], Yanyan Zou[1,2,3,4], Yue Peng[1,2,3,4] and Wei Huo[1,2,3,4]

Abstract

Grey-box fuzzing techniques have been widely used in software bug finding. In general, there are many decisions to make in the fuzzing process, including which code block in the target program should be explored first, which bytes of an input seed should be mutated to reach the target code block, and how to mutate the chosen input bytes. However, existing solutions usually rely on random exploration or certain heuristics to choose where and how to fuzz, which limits the efficiency of fuzzing. In this paper, we propose a novel solution CAMFuzz to guide the fuzzing process with explainable decisions in explainable artificial intelligence (XAI). First, we propose a dynamic weight adjustment algorithm, which considers both the difficulty of reaching a block and the number of unvisited blocks nearby, to find code blocks worthy to explore first. Second, we utilize a widely used local interpretation technique, i.e., class activation mapping (CAM), to recognize which part of an input seed should be mutated to reach a given target code block. Therefore, CAMFuzz can distinguish which part of code in the program is more important and which positions in the input file should be mutated first, in order to achieve a better code coverage and bug finding efficiency. Third, to further help the fuzzer increase fuzzing efficiency, we leverage a lightweight static program analysis to help the fuzzer identify magic values. We implement a prototype of CAMFuzz and evaluate it on 13 real-world programs (including 11 open source targets, 2 closed-source commercial products including a Microsoft component and Hancom Office) Results show that CAMFuzz outperforms state-of-the-art fuzzers in both code coverage and bug finding. To detail, CAMFuzz on average achieves 2.07× more bugs and 1.17× coverage improvements. In total, it found 19 previously unknown vulnerabilities, of which 6 have been assigned by CVE so far.

Keywords: Fuzzing, Explainable artificial intelligence, Grey-box fuzzing

Introduction

Recently, grey-box fuzzing techniques have become the most popular solution in finding bugs. One of the unique tools is American Fuzzy Lop (AFL) (Zalewski 2014). It has been proven effective when finding bugs in real-world software.

Challenges: Despite the success of its genetic algorithm, AFL has many blind spots and makes random decisions in the fuzzing process, including which bytes of an input seed should be mutated to trigger the unvisited code block, how to mutate the chosen input bytes, and it does not distinguish the code blocks with different weights. These all limit the efficiency of fuzzing (Gan et al. 2020; Lemieux and Sen 2018; Wang et al. 2020). There are several solutions proposed to address these problems. Generally, they will (1) figure out more significant parts such as branches or code with higher weight, (2) analyze which input bytes may have a relation with the code using taint analysis, or (3) mutate these bytes with higher priority and replace them with specific values.

First, to prioritize code blocks to explore, many studies focus on counting unvisited code blocks, e.g., used in CollAFL (Gan et al. 2018) and FairFuzz (Lemieux and Sen 2018). However, we believe this strategy is not sufficient. For example, although more unvisited child nodes

*Correspondence: chaoz@tsinghua.edu.cn
[5] Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing, China
Full list of author information is available at the end of the article

connect to some code blocks, the unvisited nodes may hardly be reached. For example, the code responsible for handling the failure of *malloc* operation with a fixed size is useless in fuzzing. Thus, it is a waste of power if the fuzzer focuses on those blocks.

Second, given target code blocks to explore, the most straightforward solution to find out related critical input bytes is data-flow or symbolic execution analysis. Nevertheless, it is time-consuming. For example, TaintScope (Wang et al. 2010) uses dynamic instrumentation to perform traditional data-flow analysis (e.g., taint analysis), which will decrease the fuzzing efficiency. Some lightweight data-flow analyses have been proposed to speed up the taint analysis, e.g., GreyOne (Gan et al. 2020), ProFuzzer (You et al. 2019). However, they still face much manual work to define rules when doing data-flow analysis (e.g., defining a rule to infer the type of input field) or under-taint issues (Kang et al. 2011).

Recently, machine-learning-based methods have been proposed to guide the fuzzer on where to fuzz, such as NEUZZ (She et al. 2019) and Rajpal et al. (2017). However, existing techniques are still in the early stage and have limitations. For example, Rajpal et al. (2017) use LSTM to predict good locations to fuzz. However, the training stage of the model lasts long. NEUZZ uses the gradient derived from the neural network to locate critical bytes in the input to mutate and outperforms state-of-the-art fuzzing techniques. However, as shown in RQ3 in the *Evaluation* section, the gradient is unstable and will introduce much noise. The noise itself may cause the fuzzer to waste power on mutating unrelated input positions when given a code block.

Third, given input bytes to mutate, several solutions employ data-flow analysis to infer which values should be used for mutation. For instance, VUzzer (Rawat et al. 2017) relies on static and dynamic analysis to infer critical values, including magic numbers, etc., to replace target input bytes. Nevertheless, it may produce false negatives in calculating constant values, and it needs traditional data-flow analysis to calculate the positions related to these values.

Our Solution: To solve the challenges, we present a novel solution named CAMFuzz to guide the fuzzing process with explainable decisions. Specifically, it leverages the explainable artificial intelligence (XAI) and program comprehension to determine which code blocks in target programs should be explored first, which input bytes should be prioritized to mutate to reach target code blocks, and how to mutate these bytes (i.e., which value to use).

First, we propose a dynamic weight adjustment algorithm to prioritize code blocks (i.e., nodes in the control flow graph) to explore. For each candidate node, our algorithm considers not only the number of its unvisited children in the CFG but also the difficulty of reaching it. We intend to explore those easier-to-reach blocks first, then the rest. Second, we utilize XAI techniques to determine which input bytes to mutate to explore target code blocks. XAI is used to help humans understand why the AI model makes the specific decision. For example, given a picture of a cat, XAI can help us understand which part in the picture plays the most crucial role when the model predicts it as a cat. We utilize local interpretation technique in XAI field to speculate the input positions with a strong relationship given a target code block. After a thorough consideration (as shown in *Explainable Artificial Intelligence* section), we choose one of the local interpretation techniques, Class Activation Mapping (CAM), to interpret which input bytes are more valuable to mutate to reach a given code block. Lastly, we leverage static program analysis to extract magic value, enumeration, and other values of interest from target programs, to determine how to mutate the chosen input bytes (i.e., what values to use during mutation). We use the term "magic value" in this paper to represent not only the signature of a file format (e.g., "0x5A4D" stands for a PE file), but also constant values of other fields that have a set of predefined values in the file (e.g., the MARKER field in JPEG format, or E_TYPE in ELF format)

Results: We implement a prototype of CAMFuzz and evaluate it on 13 real-world programs. We compare it with AFL, AFLFast (Böhme et al. 2017), FairFuzz, NEUZZ, Angora (Chen and Chen 2018), TortoiseFuzz (Wang et al. 2020), WinAFL (Fratric 2017), and Winaflfast (Bohme 2018). The results show that CAMFuzz outperforms the other fuzzers in both code coverage and bug finding. We find 19 previously unknown vulnerabilities consisting of 13 from open-source programs and 6 from commercial products. Six of them have been assigned with CVEs so far. In this paper, we make the following contributions:

* We propose a solution CAMFuzz, which utilizes the local explanation method (i.e., CAM) to locate critical input bytes and guides fuzzers to spend more energy on mutating these bytes.
* We propose a dynamic weight adjustment (DWA) algorithm to determine which code blocks should be explored first and guide fuzzers to skip hard to reach blocks.
* We implement a prototype of CAMFuzz and evaluate it on programs with and without source code. It has found 19 unknown vulnerabilities and we have reported them to vendors, among which six have been assigned with CVEs.

Background

Grey-box fuzzing

Current grey-box fuzzing techniques mainly utilize code coverage to figure out good seeds. Researchers use this technique to test different programs such as browsers (Aschermann et al. 2019a), language engines (Groß 2018), and kernels (Jeong et al. 2019). The fundamental intuition behind the grey-box fuzzing technique is that increasing code coverage likely leads to more bugs (Takanen et al. 2018). Although grey-box fuzzing is very efficient in increasing code coverage, there are still some limitations. For example, some fuzzers do not distinguish which code block should be explored first; the power schedule is not intelligent enough to increase the fuzzing efficiency (Yue et al. 2020). Several methods based on symbolic execution and taint analysis have been proposed to help the fuzzer decide which bytes should be mutated first and which code is more important. However, symbolic execution based methods such as S2E (Chipounov et al. 2011) and Angr (Shoshitaishvili et al. 2016) have been proven to suffer from path explosion and have difficulties solving complex constraints. These problems have hindered its application in fuzzing, especially in complex real-world programs. In addition to symbolic execution, many researchers utilize machine learning to guide fuzzers towards specific input bytes. This paper uses the explainable AI technique to guide the fuzzer to concentrate on valuable input positions.

Explainable artificial intelligence

Machine learning, especially deep learning, has been used to solve many complex tasks. For example, in computer vision (Pishchulin et al. 2016), natural language processing (Dong et al. 2019), and autonomous driving (Casanova et al. 2018). Although machine learning has achieved great success in solving many complex tasks, it is still difficult for humans to understand the working principles of the model. This is due to a complex model structure and a large number of hyper-parameters, which makes the results obtained by the model difficult to understand. The interpretability of neural networks (Ghorbani et al. 2019) can help people understand the working principles of the model more directly. At present, interpretability can be divided into global interpretation and local interpretation (Guidotti et al. 2018). Local interpretability helps us understand the reason why a model makes a specific decision. This paper focuses on the local interpretability: we train a special model to map an input to its code coverage. Different from existing AI-based methods, we power the model with the ability to locate promising input parts. When given a block of code, we expect the local interpretation to tell us which part of the input contributes to the code block the most.

There are several studies on local interpretation. As we intend to figure out which input bytes contribute to the decision, this is a kind of outcome explanation (Guidotti et al. 2018). Simonyan et al. (2013) use the gradient to build a saliency map to help understand which part of the input the model focuses on. However, as Smilkov et al. (2017) claim, the gradient-based method will introduce noise. In the fuzzing task, these noises will not contribute to fuzzing efficiency, which we will explain in RQ3 in *Evaluation*. Zhou et al. (2016) use the global average pooling (GAP) layer to restore the partial information in the input, which can be called Class Activation Mapping (CAM). The limitation of CAM is that it has to change the network structure and replace the fully connected layer with GAP. In our task scene, since we can build the neural network, we use CAM to be our interpretation technique.

Approach

Overview. The overall architecture of our framework is shown in Fig. 1. We have four core modules: Model Training, Explanation, Node Selection, and Magic Value Identification. The term "Magic Value" in this paper not only means the signature of a file format, but it also represents constant values(e.g., enumeration type) in other fields. The core idea of our fuzzer is to utilize the local explanation technique in XAI to recognize critical input bytes related to code blocks and guide the fuzzer to focus on a worthy part of the input to mutate first. Since a neural network is the premise of explanations, we need to train a model to map an input to the coverage map. Besides, we use a Node Selection module to help the fuzzer focus on the more critical code, which may bring new coverage. Finally, we use the Magic Value Identification module to extract particular values that will guide the fuzzer on how to fuzz during mutation.

Training. We aim to train a model that can simulate the behavior of a program: given a seed input, the model can predict the code coverage map. We use a classification model to achieve the goal. First, we collect the seeds and their corresponding coverage map to preprocess. Then we train a convolution neural network with global average pooling (GAP) instead of a fully connected layer next to the last convolution layer.

Local Explanation. The local explanation is the core module we use to locate critical positions of the input to mutate. Here we consider the critical positions as the part of the input, which could unlock unvisited CFG nodes with mutations. In this module, we use the GAP layer to help us in interpretation. GAP brings the ability to localize an object in the input (Zhou et al. 2016). In the fuzzing task, since we have trained a model to map the input (similar to pictures in computer vision) to the

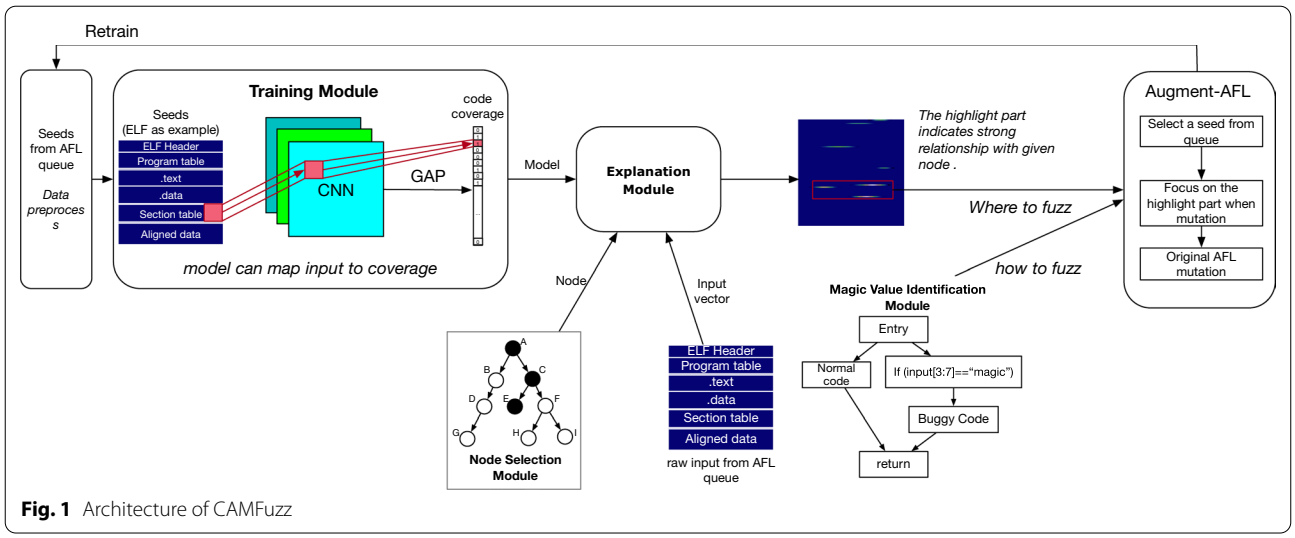


Fig. 1 Architecture of CAMFuzz

coverage map (similar to labels), our intuition is to feed the explanation module with a block in the coverage map. Then the module may guide us with positions which have a tight connection with it. Once the input positions are known, we can guide the fuzzer to focus on mutations to this part with higher priority.

Node Selection. In the Local Explanation module, we intend to feed the model with a code block. But which block should be chosen first? We have two considerations when choosing the code to explore:

(i) Many nodes in the CFG have not been visited, and we cannot choose them directly because the explanation module does not understand how to explain this "label" since the trained model does not know this feature.

(ii) Some CFG nodes are hard to reach in fuzzing. For example, in a call to *malloc* with a fixed size, the failure branch may never be reached in fuzzing since the memory might be big enough to ensure the *malloc* a success. Focus on these hard-to-reach nodes first will waste time.

Based on the above considerations, our idea is to feed the model with visited nodes. These nodes should have more unvisited code blocks connected to them and are easier to reach. We have two main steps to select code to be explored.

The first step is inspired by CollAFL. We analyze the CFG of the program and choose the nodes which directly connect unvisited child nodes. Then we initialize them with weights depending on how many untouched child nodes they have. The nodes chosen in this step are visited ones, but they have unvisited nodes directly connected to, which means mutation to input bytes related to this code may affect the conditional branch of the node. Therefore we are more likely to reach a new path.

The second step is to consider the difficulty of reaching the code. During the mutation process by local explanation, we found some nodes are hard to be explored. It means even if we mutate the positions extracted by local interpretation many times, we still cannot cover its directly connected children nodes. We apply the Dynamic Weight Adjustment (DWA) algorithm to decrease the weight of these nodes to focus on easier-to-reach nodes first.

Magic Value Identification. Since we have figured out which code blocks should be explored and which bytes of the input have a strong relationship with the blocks, we use a lightweight static analyzer to identify if there are constant values we can extract to help the fuzzer know how to fuzz, for instance, file signature, enumeration value, and loop count.

Training

Data preprocess

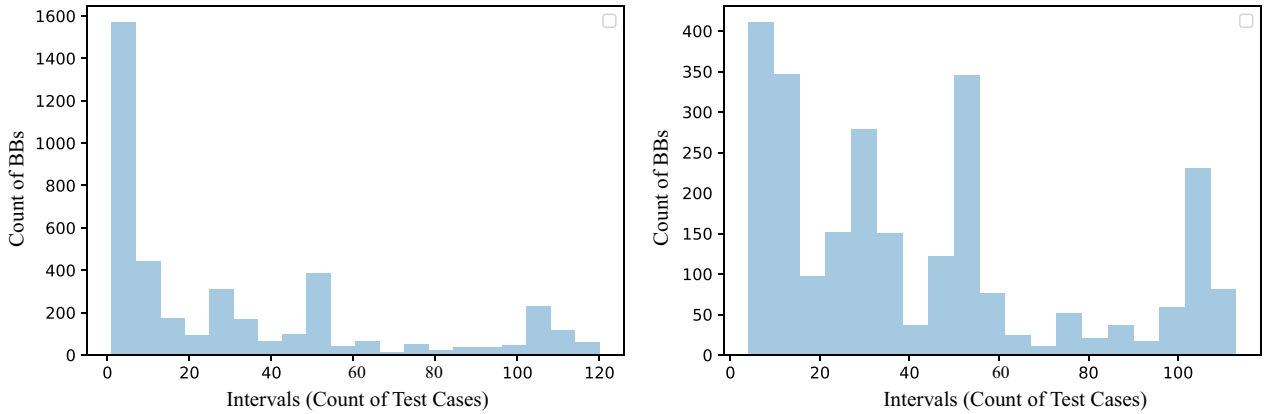
Data preprocessing is fundamental in the training process as the training data quality will directly affect the performance of the model (García et al. 2015). We need to collect two data types from the original fuzzing progress: seed file and coverage map. For the seed file, since we cannot clarify every file structure, we treat the input as raw bytes, the value of each byte is from 0 to 255. For the coverage map, we use binary instrumentation to obtain the map. If a specific block is covered, we set it to 1, otherwise 0.

Algorithm 1 Data Preprocess

Require: SeedSet
Ensure: COV, Data
1: $S_{max} = MaxFileSize(SeedSet)$
2: $COV = []$
3: $Data = []$
4: **for** $i = 0$; $i < Count(SeedSet)$; $i++$ **do**
5: $File_{padded} = Pad(SeedSet_i)$
6: $Cov_i = Run(File_{padded})$
7: $COV.append(Cov_i)$
8: $Data_i = Vectorize(File_{padded})$
9: $Data.append(Data_i)$
10: **end for**
11: $UnderSampling(COV)$
12: $DiscardMinClass(COV)$

We describe the algorithm of preprocessing in Algorithm 1 . We first figure out the largest file in the training data, set the file size with S_{max}, then pad the rest of the data to S_{max} with byte 0x00. Note the sequence of line 5 and line 6 in Algorithm 1. After padding the file, we run the program because the code coverage may be different while parsing the padded file and original file.

After running all the padded files and get code coverage, we then prune the code coverage using the undersampling method and directly discard some coverage labels which has few seeds covered. This step is essential because the original code coverage is usually imbalanced. We choose readelf as a target to illustrate, we randomly choose 120 ELF files as seeds, then run readelf with the seeds. The total block number we traced is 4001. We then draw a histogram of code coverage distribution in Fig. 3. The X-axis indicates the intervals that show the count of test cases. The max value is 120, the same as the total amount of the seeds. The Y-axis indicates the count of distinct blocks. Before preprocessing, the sum of all the

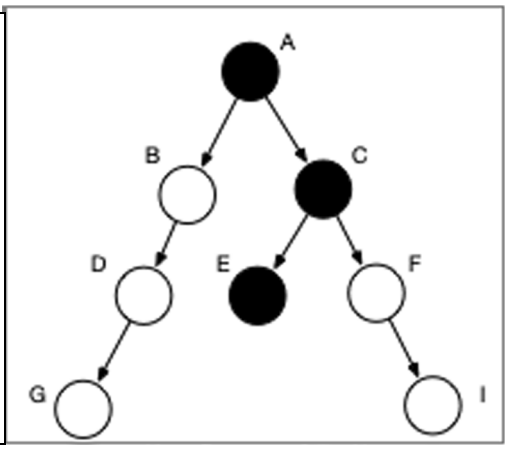


(a) before preprocessing (b) after preprocessing

Fig. 2 Coverage distribution histograms

```
obj *o=malloc(0x30);
    if (o==NULL){
     perror("malloc error");
     cleanup();
     return o;
     }
    else{
     if(initialize(o)==False){
      buggyfunction();
      return NULL;
     }
     else{
      return o;
     }
    }
```

(a) Sample Code



(b) Sample CFG

Fig. 3 Sample code and CFG

columns in Fig. 2a equals to 4001, which is the total number of basic blocks covered by the seeds. In Fig. 2a, we observe that the distribution data is hugely imbalanced. At the very left of Fig. 2a, There are almost 1600 blocks covered by a small number (about less than 10) of cases. At the right of Fig. 2a, some nodes are covered by almost all cases. It is easy to uncover the reason: some common blocks can be covered by most of the seeds while some of the unique seeds may cover a number of rare blocks.

Imbalanced class distribution data will negatively affect the training model. There are many studies to deal with such problem, such as Castillo et al. (1997) and Provost (2000). We follow the basic principles to process the imbalanced data: we use the under-sampling (Barandela et al. 2004) approach to reduce the data in majority classes and discard some minority classes data to rebalance the training data.

Figure 2b shows the distribution of the coverage after being pruned. The data is more balanced than those in Fig. 2a and the training model will benefit from the pruning process.

Model

We use a convolution neural network (CNN) model in the training process, and the model is used to map seed input to coverage map. That is to say, given a well-trained model f and a corresponding input (consists of bytes $x_1, \ldots x_i \ldots, x_n$), the model can predict if a node cov_i can be covered or not.

Denote that x_i is between [0x0, 0xFF] and cov_i is the probability which is between [0, 1].

$$f\left(\frac{x_1}{255}, \cdots \frac{x_i}{255} \cdots, \frac{x_n}{255}\right) = (cov_0, \ldots, cov_i, \ldots, cov_m) \tag{1}$$

Here we detail three critical points in our model.

(i) We use a GAP layer instead of a fully connected layer.
(ii) We set the shape of our feature map produced by the last convolution layer to 128.
(iii) As we are clear that the count of code blocks may be huge when we run the seeds. To improve efficiency, we design a segmental training algorithm shown in Algorithm 2.

In this algorithm, we firstly count the preprocessed nodes and sort them by the number of occurrences. Then for each turn, we select the top *window* of labels to train the model. Note that the sort procedure is essential since each turn, there might be new edges after sorting, which will be added to the tail, and these features can be trained later.

We choose CNN as our model for the following two reasons: First, a CNN is usually used to do classification tasks. For example, given several panda pictures, a CNN automatically learns the hidden feature in these pictures. Later, when we feed the model with a new image, the model may tell us whether there is a panda in the picture. In our task, given a seed input, we want the model to predict whether a specific node can be covered or not. Both of these two scenarios are classification tasks.

Second, a CNN can easily filter out the background information in the target input. The model will focus on the main object by using the convolution and pooling operation, which means the background has less effect on the final decision. That is very similar to program analysis: the input file usually consists of metadata and data. Mutations to metadata will significantly influence the code coverage, whereas mutating data may bring less coverage variance. CNN has been demonstrated its ability to filter out the background in the picture. We utilize this characteristic in fuzzing: filter out useless parts of data and focus on those which may have a significant influence on code coverage.

Local interpretation

Local interpretation helps us understand why the model gives the decision in classification tasks, especially which part of input contributes to the prediction. In the fuzzing progress, we can collect a large amount of training data, which are input files and the corresponding covered code blocks. We can train a model to map the input to *visited* code blocks. After the model is well trained, we provide a block that is visited but has *unvisited* nodes connected. We expect local interpretation to tell us which bytes are highly related to this block, then by mutating the input bytes instructed by the interpretation, we have a higher possibility to reach those unvisited nodes.

Note that, although the model only learns the mapping between input bytes and visited blocks, it can report input bytes related to visited blocks, including those related to the entry condition and the exit condition of the visited block. Mutating input bytes related to the entry condition may lead the program to explore the sibling blocks of the visited block, while mutating input bytes related to the exit condition may lead the program to explore children blocks of the visited block, including those unvisited children.

Algorithm 2 Segmental Training

Require: Coverage Map Nodes, Seed
Ensure: Model
$window = 2000$
$index = 0$
while True **do**
  $C =$ Count($nodes$)
  for $i = 0$ to C **do**
    $Counter[i] =$ Count(occurrences for $node_i$)
  end for
  Sort $Counter$ by number of occurrences
  $Train_{node} = Counter[index : index + window]$
  Train($Vector(Seed)$,$Train_{node}$)
  $index = index + window$
  Fuzz($Seed$)
end while

If we know which byte is the variable related to the branch, we can reach the unvisited branch by mutating the enumeration bytes. For example, the header parser function will generally process the header part of the input file rather than the data section. If we feed the local interpretation module with a block in this function, this block has unreached nodes connected. It will guide the fuzzer to mutate bytes in the header to the unvisited blocks covered.

We will review one of the local explanation techniques named CAM (Zhou et al. 2016). The core ideas behind CAM are:

(i) Use a GAP layer to replace the fully connected layer.
(ii) Take advantage of partial information in the feature map produced by the last convolution layer behind GAP and the weight produced by GAP to generate localization information.

We will look at how the localization information is calculated: suppose we have an input with shape (h, w). The last convolution layer locates at L_{th} layer in the network, and its output shape is (m, n, k), where k means the number of features, (m,n) means the shape of one feature map. The GAP accepts the (m, n, k) feature maps and produces (1, 1, k) values which represent the global average value of each feature in the L_{th} layer. w_i^c is the weight regarding class c. The final score, S, of class c is calculated as:

$$S = \sum_{i=1}^{k} F_i * w_i^c \tag{2}$$

In this formula, F_i is the feature map produced by the L_{th} layer. Note that after the CAM is calculated, the size of the heatmap is (m, n), which means we should restore this map to the shape of the original input. The general way is to upsampling the CAM to the input size, which is the final heatmap. In our framework, we design the network and set the size of the last convolution layer as 128 of the input size. We have the following considerations when we choose the size. Foremost, we do not have to keep the shape of the feature map the same as raw input since only parts of the input significantly influence the code coverage. Simultaneously, if we keep the original size, there would be lots of convolution calculations, which would cause the training of the network to slow down.

Node selection

By using the local interpretation, we can guide the fuzzer on the part of the input strongly related to the given code. Here we aim to use the Node Selection to choose the more valuable code as an explanation target.

Consider the sample code in Fig. 3a and its CFG in Fig. 3b. The buggy function will be called if the initialization of object o fails. In the CFG, black nodes are covered while white ones have not been covered yet. Node B, D, and G will be called if the *malloc* fails, Node F represents the buggy function call. We should focus on C more than A because it is almost impossible to cover B, D, and G, although more new nodes are connected to A than C.

Initial Selection. We will first choose those nodes with more untouched nodes connected through initial selection. As general knowledge, a trained model can only map and explain the features that it has learned. For those unknown features, the model cannot do anything. That is, we cannot use the model to help us increase the code coverage if we feed the model with unvisited nodes.

To solve the gap, we select the nodes which are near unvisited ones. In Figure 3b, we will choose nodes A and C (untouched node B directly connected to A and F directly connected to C). Inspired by CollAFL, we first count the unvisited nodes connected to A and C. Then assign them with the initial weight based on the number of untouched nodes.

Dynamic Weight Adjustment (DWA). By using the DWA algorithm, we intend to decrease the weight of the hard-to-reach code. Considering the code snippet in Fig. 3a and its CFG in Fig. 3b. Nodes B, D, and G represent lines 3-5, and the code will be executed if *malloc* fails. Generally, almost all the cases will succeed in *malloc* operation during fuzzing. Thus Node B, D, G are very hard to be touched. If we only use the initial weight to evaluate the importance of the node, we may waste the power on useless nodes. So our solution is to decrease the weight of node A dynamically even it has more untouched child nodes at the beginning.

Suppose we have N cases that can cover node A and node C at the beginning. The initial weights for these two nodes are 3 and 2. Choosing a case N_i to mutate, we feed the interpretation module with node A and get the related positions in the input. If node B is not covered after the fuzzer mutates the positions, we will subtract the current weight with $\frac{1}{N}$, then the weight of node A will be updated as $3 - \frac{1}{N}$. As fuzzing goes on, the weight of node A will be decreased during mutation. Note that the N keeps the same during one iteration, which means it will only change after the model is retrained.

Algorithm 3 Extract Magic Value

Require: Target Program
Ensure: candidateValues
1: $SET\ candidateValues = []$
2: $SET\ threshold = K$
3: //identify constant value
4: **for** each $instruction \in node$ **do**
5: **if** the instruction has cmp **then**
6: append constant value to $candidateValues$
7: **end if**
8: **end for**
9: //identify switch-case
10: **if** CountOfSuccessors(node) $>$ 3 **then**
11: extract jmp_{base}
12: **for** each $offset \in successors$ **do**
13: $value = offset$ - jmp_{base}
14: append the $value$ to $candidateValues$
15: **end for**
16: **end if**
17: //expand more values
18: **for** each $value \in candidateValues$ **do**
19: append [value-K,value) to $candidateValues$
20: append (value,value+K] to $candidateValues$
21: **end for**

Magic value identification

We augment the original mutator of AFL with external magic values embedded in CFG nodes. The term "magic value" in this paper not only means the signature of a file format, but it also represents constant values(e.g., enumeration type) of other fields in the file. In our design, magic values can be extracted in two ways: constant values and switch cases. For constant values, we implement a static analyzer similar to Vuzzer (Rawat et al. 2017). For switch cases, since the assembly code will use a jump table rather than *cmp* instructions, we design a method to identify switch cases in the assembly code and extract the magic values. Different as REDQUEEN (Aschermann et al. 2019b): they try to find subtractions to identify switch-case, which can be improved by a more generalized way to extract *case* values in switch-case.

Here we detail the idea on how to extract *case* values embedded in switch-case. First, choose the nodes which have more than three successors since we consider switch-case usually has more than three branches. Then, convert the code in the node to intermediate representation and find the base address of the jump table. At last, for each successor of this node, calculate the difference between the offset of the successor and the offset of the base address of the jump table. The results are the *case* values, which can guide the fuzzer on how to mutate. The extraction algorithm is shown in Algorithm 3.

From line 18 to line 21 in Algorithm 3, instead of just using the constant values directly, we set a threshold to mutate sufficiently. The reason behind this is that if the value represents a loop count or a size, we may want to mutate the value around.

Implementation

Main framework

We implement the AI component of CAMFuzz with Python. While for the fuzzing part, we rewrite the original AFL to support open source targets. To support binary-only fuzzing on Windows, we build CAMFuzz on top of WinAFL (Fratric 2017), which is a port of AFL for the Windows platform. CAMFuzz utilizes the bitmap provided by AFL and WinAFL as feedback respectively. We also add extra AI mutation and magic value mutation strategy to the original AFL. The detailed algorithm of CAMFuzz is shown in Algorithm 4.

Model training

We collect the training data, such as seeds and coverage, with the help of DynamoRIO (2020) during fuzzing. We use online training along with fuzzing. Specifically, CAMFuzz collects training data during fuzzing, then trains the model with Keras to guide the fuzzer. To increase the fuzzing efficiency, we first trim the code coverage as described in *Data Preprocess* section. Then we use a segmental training method as described in Algorithm 2. Both of the two steps can help to decrease the dimension of the label in the model, and then the training time can be short enough as the experiment shows in *GPU resource occupation* section. We implement CNN with five hidden layers embedded. The last hidden layer is the GAP layer, which we use to keep the partial information produced by the convolution layer. The optimizer we use is *adam*, and we use *binary_crossentropy* as the loss function.

Algorithm 4 CAMFuzz Implementation

```
Require: CFG, seeds
Ensure: NewCoverage
MinimumTrainingCount = 200
for each node ∈ CFG do
    MagicValues[node] = ExtractMagicValue(node)
end for
cnt=0
raw_seeds=[]
for each seed ∈ seeds do
    TraditionalFuzz(seed)
    raw_seeds.append(seed)
    if cnt > MinimumTrainingCount then
        training_data = PrepareTrainingData(raw_seeds)
        InitializeWeight(CFG)
        Train(training_data)
        for each input_seed ∈ training_data do
            SET cov = QueryCoverageData(input_seed)
            SET Nodes = NodeSelection(cov)
            for each node ∈ Nodes do
                SET input_pos = Calculate_Critical_Positions(node)
                SET values = MagicValues[node]
                AI_Mutate(input_pos, values)
                UpdateWeight()
            end for
        end for
    end if
    cnt+=1
end for
```

Table 1 Summary of 13 applications to test

Program	Descirption	Parameter
giflib	Image parser	./gif2rgb @@
jhead	Image parser	./jhead @@
libjpeg	Image parser	./djpeg @@
tcpdump	Network packet parser	./tcpdump -nr @@
libde265	Video processor	./dec265 @@
zlib	Compression tool	./minigzip -c @@
catdoc	Document parser	./catdoc @@
catppt	Document parser	./catppt @@
xls2csv	Document parser	./xls2csv @@
Hancom Office	Document parser	N/A
readelf	ELF parser	./readelf -a @@
objdump	Binary file parser	./objdump -x @@
Microsoft JetEngine	Database engine	N/A

AI mutation

Here we use the local explanation to guide the fuzzer on where to fuzz, and we use values extracted from CFG nodes on how to fuzz. For those valuable positions produced by the local explanation, we choose the top 256 related ones. During magic value extraction, we transform the assembly code to intermediate representation with VEX (Shoshitaishvili 2014).

Node selection

To calculate the weight of the nodes, we use two levels of iteration to calculate the number of unvisited nodes with the help of Angr (Wang and Shoshitaishvili 2017). For instance, if there is a *call* instruction in an unvisited

Table 2 Visited basic blocks of different fuzzers in 24 h (median value)

Program	version	AFL	AFL-MAGIC	AFLFast	NEUZZ	AFL-rb	Angora	TortoiseFuzz	WinAFL	WinAFLFast	CAMFuzz (AI+MAGIC+DWA)
giflib	5.2	1633	1601	1691	1879	1722	1821	1735	N/A	N/A	**2050 (+ 9.10%)**
jhead	3.04	1161	1161	1160	N/A*	1158	1207	1160	N/A	N/A	**1164 (- 3.56%)**
libjpeg	2.0.4	3361	3215	3431	4922	4053	3745	3543	N/A	N/A	**5910 (+ 20.07%)**
tcpdump	4.10.0	25304	22982	25442	29806	28335	30501	26533	N/A	N/A	**33729 (+ 10.58%)**
libde265	1.0.5	3633	3557	3719	N/A*	4007	3802	4002	N/A	N/A	**4096 (+ 2.22%)**
zlib	1.2.11	1903	1881	1898	1906	1920	1955	1912	N/A	N/A	**1919 (- 1.84%)**
catppt	0.95	577	555	601	610	686	721	732	N/A	N/A	**898 (+ 22.68%)**
catdoc	0.95	1313	1213	1351	N/A*	1415	1458	1521	N/A	N/A	**1786 (+ 17.42%)**
readelf	2.32	11323	10404	12359	13393	13122	14782	14192	N/A	N/A	**16223 (+ 9.75%)**
objdump	2.32	7232	6502	7572	8267	8260	8182	7801	N/A	N/A	**9016 (+ 9.15%)**
xls2csv	0.95	2113	1902	2298	2301	2318	2323	2435	N/A	N/A	**2517 (+ 3.37%)**
Hancom	9.6.1.7749	N/A	N/A	N/A	N/A	N/A	N/A	N/A	5322	5728	**8943 (+ 56.13%)**
Microsoft Jet	Multiple	N/A	N/A	N/A	N/A	N/A	N/A	N/A	7430	7882	**13406 (+ 70.08%)**
Average											+ 17.32%

*The fuzzer failed to run on the target over 24 h

node, we will count the number of nodes in the *call*. The DWA algorithm will be invoked on the completion of every AI mutation cycle for efficiency reasons.

Evaluation

In this section, we evaluate CAMFuzz and answer the following questions:

- **RQ1:** What is the performance of CAMFuzz, comparing to other state-of-the-art fuzzers?
- **RQ2:** What is the performance of the neural network model used in CAMFuzz?
- **RQ3:** Can the CAM help the fuzzer focus on related positions better than the gradient-based method?

Evaluation setup

Fuzzers.To answer the questions above, we compare CAMFuzz with both AFL and WinAFL. Additionally, we compare it with six other state-of-the-art fuzzers: Winaflfast, NEUZZ, AFLFast, AFL-rb, Angora, and TortoiseFuzz. Here we choose NEUZZ because it is the first fuzzer to use the gradient-based method in the AI field to guide fuzzing for improvement, and it significantly improved in finding new paths. For AFL, being the most popular fuzzer and extensively studied, it is valuable to be included as a baseline benchmark comparison. For AFLFast, TortoiseFuzz, and AFL-rb, they are all based on AFL and equipped with additional interesting mutation strategies. For Angora, it uses taint analysis and gradient descent to increase fuzzing efficiency. These are some of the state-of-the-art traditional fuzzing tools. It is important to compare our AI-based fuzzer with traditional fuzzing techniques. For WinAFL and Winaflfast, since CAMFuzz can be applied in closed-source targets, we choose WinAFL and Winaflfast to experiment on closed-source targets in Windows.

Target Programs. We consider multiple factors when choosing the target programs to be tested, such as popularity, functionality, and frequency of updates. From Klees et al. (2018), the median number of real-world programs tested by 32 different papers in recent years is 7. We evaluate the fuzzers with 13 real-world programs listed in Table 1. We choose these targets because they are popular and are used to process different kinds of input, such as pictures, network traffic, compressed files, and multimedia files. They come from different developers, which can ensure the variety of code and the generality of CAMFuzz. CAMFuzz can not process textual inputs which rely much on semantics, such as XML, HTML, PDF, etc. We leave this kind of input in further work. Despite the real-world programs, we also evaluate

Table 3 Unique bugs found by different fuzzers (known/unknown)

Program	version	AFL	AFLFast	NEUZZ	AFL-rb	Angora	TortoiseFuzz	WinAFL	CAMFuzz	WinAFLFast
giflib	5.2	0/0	0/0	0/0	0/0	0/0	0/0	N/A	**0/1**	N/A
jhead	3.04	3/0	1/0	N/A*	0/1	2/1	2/0	N/A	**1/2**	N/A
libde265	1.0.5	2/0	3/0	N/A*	1/0	3/0	4/0	N/A	**7/1**	N/A
catdoc	0.95	0/1	0/1	0/2	0/1	0/1	0/1	N/A	**0/3**	N/A
catppt	0.95	2/1	3/1	3/1	2/0	2/2	1/1	N/A	**2/3**	N/A
xls2csv	0.95	1/2	1/0	0/2	1/1	1/2	0/2	N/A	**0/3**	N/A
Hancom	9.6.1.7749	N/A	N/A	N/A	N/A	N/A	N/A	0/1	**0/3**	0/0
Ms. Jet	Multiple	N/A	N/A	N/A	N/A	N/A	N/A	0/1	**0/3**	0/1
Total	N/A	8/4	8/0	3/5	4/3	8/6	7/4	0/2	**10/19**	0/1

*The fuzzer failed to run on the target over the whole run

Table 4 Bugs found in LAVA-M test suites

Program	AFL	AFLFast	AFL-Magic	AFLFast-Magic	NEUZZ*	Vuzzer	CAMFuzz	Angora
base64	1	1	13	18	48	16	48	48
md5sum	0	0	17	18	60	19	57	56
uniq	0	2	15	14	29	24	28	28
who	4	3	41	49	1582	48	1623	1541

*We use the experiment results from the paper of NEUZZ since it is not open-sourced

CAMFuzz on the LAVA-M benchmark, a popular performance benchmark suite for fuzzers.

Seeds. We randomly choose the initial seed files provided by the target program, or from the Internet if they do not provide the sample seeds. We use the same thresholds described in *Data Preprocess* section.

RQ1: What is the fuzzing performance

To determine the effectiveness of CAMFuzz, we measure two metrics: basic block coverage and crashes found.

Basic block coverage

The experiment lasted for 24 h, and we fed all the fuzzers with the same seed set when testing one target. For the hardware configuration, we use a machine with 8GB RAM, Intel Xeon CPU E5-2650, and GTX 1060 6GB GPU to train the AI model for NEUZZ and CAMFuzz. For the closed-source targets, we run the fuzzers on Windows 10. For the open-source targets, we use Ubuntu 18.04 to finish the experiment.

We repeat the experiment 5 times to mitigate the randomness issue. The final amount of basic block coverage after 24 h of fuzzing is shown in Table 2. We calculate the mean amount of basic blocks using DynamoRIO (Bruening 2020) in the experiment. In Table 2, we also measure the improvement of CAMFuzz (DWA+MAGIC+AI) over the second-best fuzzer, on average coverage increased **17.32%**. From the final result, we can see that our fuzzer outperforms the rest of the fuzzers.

For programs responsible for parsing complex file structures, such as readelf, libjpeg, and Microsoft Jet, CAMFuzz outperforms other fuzzers significantly. The reason is that the file format consists of several metadata sections, such as headers, data definitions, tables, etc. Different parts of code are responsible for processing different file sections, and CAMFuzz can learn the relationship between the input and the code coverage. At the same time, our node selection module will pick nodes that are easier to touch first, along with critical values such as magic value and enumeration. Then the local interpretation may produce the prioritized fuzzing positions given a node from CFG.

To determine how the magic values extracted from the code affect the final result, we extend the original AFL with magic values extracted from our Magic Value Identification module. Interestingly, the final coverage of AFL with magic is less than the original AFL in several targets. When we analyze the reason, we find that AFL does not know where to place these magic values properly. It randomly selects positions to mutate with the magic value, which will waste much power in useless mutation. However, for CAMFuzz, since it is aware that where the magic value comes from and which bytes of the input have a strong relationship with the code, it can mutate the input more precisely.

From the result, we can conclude that when we apply the magic value and DWA strategies, the fuzzer performs much better.

This is because DWA helps the fuzzer focus on those unvisited nodes which may be covered with higher

Table 6 The impact on different seeds sizes

Size	Coverage (AFL/CAMFuzz)	Accuracy (%)
10kb	9787/14336 (**+ 46.48%**)	81
20kb	10122/13312 (**+ 31.52%**)	82
40kb	10634/13124 (**+ 23.42%**)	82

Table 5 Contribution of different components (basic blocks in 24 h)

Program	AFL/WinAFL	CAMFuzz(MAGIC)	CAMFuzz(AI+DWA)	CAMFuzz(AI)	CAMFuzz(AI+MAGIC)	CAMFuzz(AI+MAGIC+DWA)
giflib	1618	1594 (− 1.48%)	1980 (22.37%)	1881 (16.25%)	1923 (18.85%)	2051 (26.76%)
jhead	1161	1161 (0.0%)	1166 (0.43%)	1160 (− 0.09%)	1157 (− 0.34%)	1164 (0.26%)
libjpeg	3352	3208 (− 4.3%)	5744 (71.36%)	5302 (58.17%)	5686 (69.63%)	5899 (75.98%)
tcpdump	25280	22854 (− 9.6%)	32991 (30.5%)	31213 (23.47%)	32132 (27.1%)	33729 (33.42%)
libde265	3617	3539 (− 2.16%)	4079 (12.77%)	3902 (7.88%)	4088 (13.02%)	4092 (13.13%)
zlib	1901	1878 (− 1.21%)	1907 (0.32%)	1911 (0.53%)	1908 (0.37%)	1919 (0.95%)
xls2csv	2102	1996 (− 5.04%)	2469 (17.46%)	2400 (14.18%)	2438 (15.98%)	2517 (19.74%)
Hancom	5229	4857 (− 7.11%)	8683 (66.05%)	8093 (54.77%)	8422 (61.06%)	8943 (71.03%)
readelf	11201	10397 (− 7.18%)	15933 (42.25%)	14687 (31.12%)	15443 (37.87%)	16211 (44.73%)
objdump	7204	6973 (− 3.21%)	8863 (23.03%)	8500 (17.99%)	8644 (19.99%)	9006 (25.01%)
Microsoft Jet	7390	6946 (− 6.01%)	13085 (77.06%)	12437 (68.29%)	12901 (74.57%)	13406 (81.41%)
Average		**− 4.30%**	**33.05%**	**26.60%**	**30.74%**	**35.67%**

Table 7 GPU time during 24 h testing

Target	Duration (mm:ss)	Train times
giflib	17:03	10
tcpdump	35:15	10
jhead	22:24	10
libjpeg	32:05	13
libde265	31:33	12
zlib	29:21	13
catdoc	30:11	11
catppt	29:40	11
objdump	31:01	12
xls2csv	30:21	11
readelf	31:42	8
Hancom	45:33	11
Microsoft Jet	44:22	10
Average	**31:34**	**11**

possibility. Then in the mutation stage, the fuzzer knows how to mutate.

Bugs found by different fuzzers

Since the purpose of a fuzzer is to find bugs in software, the number of unique bugs found is a critical metric when we assess the fuzzers. Table 3 represents the unique bugs found by CAMFuzz comparing with other fuzzers. Some fuzzers do not apply to the targets. For example, only CAMFuzz supports both open-source targets and binaries, so some fuzzers regarding Hancom and Microsoft Jet are marked as N/As. We found 19 previously unknown bugs in 7 days. **6** of them got CVEs so far: 3 from Microsoft and 3 from Hancom Office. From the results, we can conclude that CAMFuzz has a good ability to find new vulnerabilities in both closed-source and open-source targets.

Evaluation on LAVA-M Database There are four programs in the LAVA-M database suites, each of which contains several bugs injected. We also compare the number of bugs found by AFL, AFLFast, NEUZZ, Vuzzer, Angora, and CAMFuzz with the same seeds for 5 h. Since CAMFuzz is equipped with magic values, we also compare AFL and AFLFast with magic values. The result is shown in Table 4. We use the result from the paper of NEUZZ directly since they use a custom LLVM to find the magic value, and that part is not open source. From the result, we can see that AFL and AFLFast found fewer bugs than others. This is because most of the bugs in LAVA-M are injected and triggered with magic values. However, when we extend both fuzzers with magic values, CAMFuzz still outperforms them. That is because the local interpretation can help the fuzzer to put magic value in the right place. For Angora and NEUZZ, we consider they have a competitive mutation strategy targeting such bugs in LAVA-M. For CAMFuzz, the static analysis helps CAMFuzz identify magic values automatically, and the AI can provide valuable information on positions to mutate.

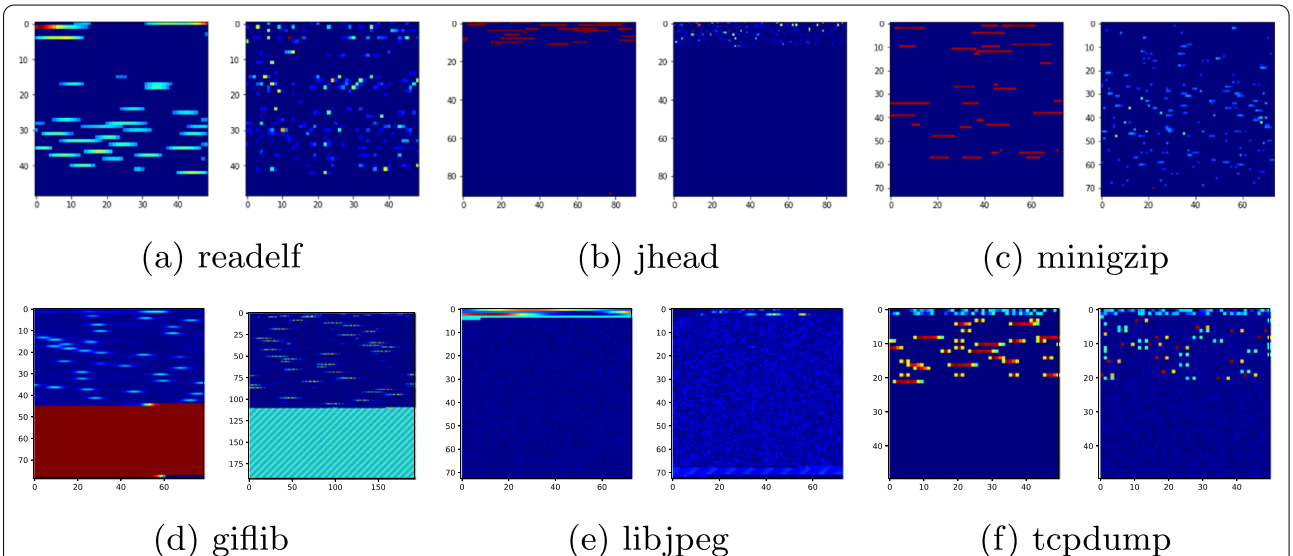


Fig. 4 Heatmap produced by the CAM (left) versus the gradient (right). Highlight parts represent importance

Further analysis

To further understand the contribution of different components, we first evaluate CAMFuzz with different component combinations.

We run 11 programs for 24 h due to a time limit. We calculated the amount of reached basic blocks. We set the original AFL/WinAFL as a baseline to calculate the percentage of basic block improvement. The result is shown in Table 5. From the result, we have the following conclusions:

First, local interpretation contributes the most regarding DWA and Magic Value Identification strategies: the average improvement of CAMFuzz with AI outperforms 26.60% than AFL/WinAFL, while AI+DWA and AI+Magic outperform AFL/WinAFL by 33.05% and 30.74% respectively. This means the AI strategy in CAMFuzz makes the most contribution. Second, when we equip the fuzzer with magic values only, the fuzzer performs worse. The reason is that the fuzzer does not know where to put this magic value when mutating the input, and it wastes much time in the magic value mutation stage.

RQ2: Performance of the NN model

Since the model itself is fundamental, we will train the model multiple times during fuzzing. We will evaluate how the seed size could affect the training, how much GPU resource it consumes, and the accuracy of the training model.

Size of seeds

The cost of model training is proportional to seed sizes, and this phenomenon is similar to the probing/reprobing stage in You et al. (2019). This section will evaluate how the seed sizes could affect the accuracy of model and code coverage.

We use readelf as the target, and we split the seeds into different groups by size. Then we run CAMFuzz and AFL with the seeds, respectively. The result is shown in Table 6. We can conclude from the result: 1) there is an inverse relationship between seed sizes and the code coverage growth. The reason which causes the growth to decrease is: as the size of seeds grows, the training stage will get slower. 2) The accuracy of the model stays stable among different sizes, and the accuracy is calculated as:

$$Accuracy = (C_{TP} + C_{TN})/C_{nodes} \tag{3}$$

Where C_{TP} is the number of nodes covered and predicted as 1 by the model. C_{TN} is the number of nodes that are uncovered and predicted as 0 by the model. C_{nodes} is the total number of nodes used in the training progress.

GPU resource occupation

We use the extra resource, i.e., GTX 1060 GPU, to help improve our fuzzer, while other traditional fuzzers can not use GPU in fuzzing. We will evaluate how much GPU time our fuzzer takes up during fuzzing to ensure CAMFuzz consumes reasonable GPU resources and the design of online training is practical. We run CAMFuzz with the target programs for 24 h, and we calculate the accumulated GPU occupation time. From the result in Table 7, the average training time during 24 h is 31 min, and there are on average 11 training iterations. Note that the re-train happens when all the model labels are enumerated and interpreted, and the number of re-train iterations changes with different target programs.

To conclude, we use an extra 2.15%(31min/24*60min) GPU time, but we achieve 117% code coverage improvement and 207% extra bugs found.

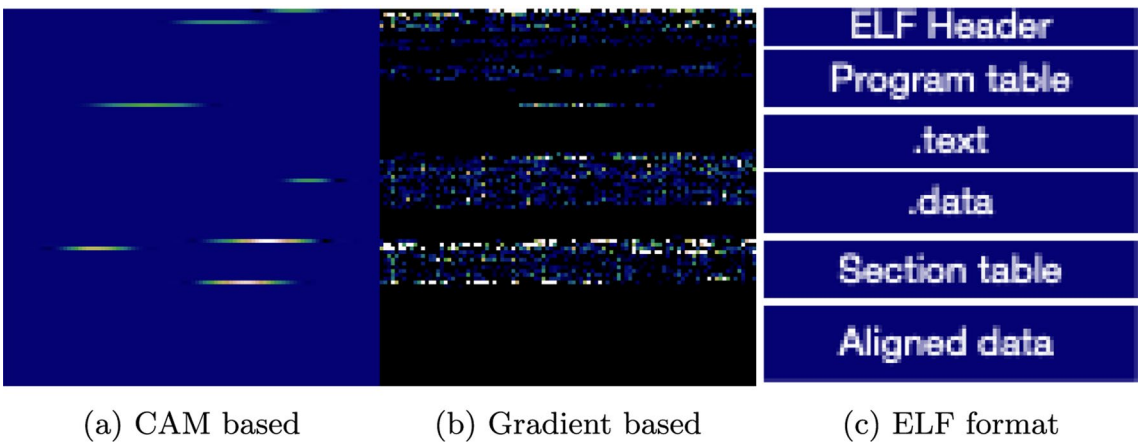


Fig. 5 Manually analyzed case

RQ3: Can the CAM help the fuzzer focus on related positions better than the gradient-based method?

Here we want to see how local interpretation helps the fuzzer locate those critical input positions. Does our method outperform the gradient-based method in position locating? To answer this question, we visualize the positions that our fuzzer focused on during fuzzing. Then we analyze some cases manually.

We choose six targets to draw the heatmap, and we observe similar results on the rest programs. Our benchmark is the gradient-based method, and we design our experiment as follows: using the same training set and model to train for 50 epochs. After the model is well trained, we choose the nodes in distinguishable functions, such as header parsing functions, instead of those in library functions like *print*, *error*, and *malloc* from the training data. For each node, we use CAM and the gradient-based method to calculate positions related to this node. We select the top 256 related positions and virtualize the importance of positions with a heatmap. We reshape the original input size to squares and the results are shown in Fig. 4.

We first conclude that both CAM and the gradient-based method can effectively help the fuzzers focus on metadata from the heatmap.

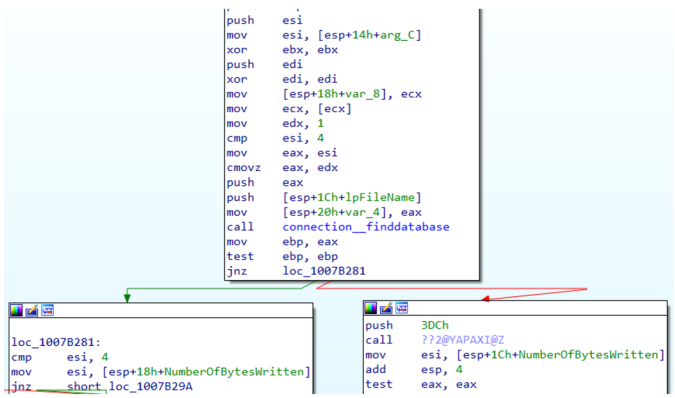
For readelf, we know that the ELF file header is located at the front of the file and the section header table at the end of the file. In the heatmap, we can see the front and the tail are highlighted. Note that we pad the heatmap at the tail for proper display, so there is a dark portion in our picture at the end. For the jhead, we can see the highlighted concentrates at the front, which is the jpeg file header. That means for this program, we need to mutate the front part with high priority rather than the rest of the file. Interestingly, the highlighted part in giflib mostly locates at the tail part. When we investigate it, we find that the nodes we choose are responsible for processing the data part of the gif file. For minigzip, as we use the command line to compress the target file, we can see the highlighted is uniformly distributed across the file.

To further analyze the details, we can see that the CAM based method has better interpretation ability for fuzzing in the following aspects:

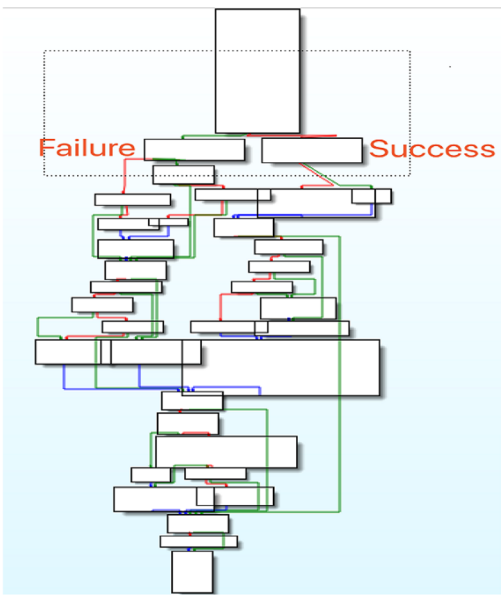
First, the continuity is better than the gradient-based method. This can help the fuzzer explore more adjacent positions. For example, many fields consist of multiple bytes, and these bytes are continuous for 2 bytes, 4 bytes, etc.

Second, there is less noise than the gradient-based method. This means that when we feed the model with a given code block and input vector, the gradient-based method will produce more unrelated positions, which will cause a waste of time in mutation. To illustrate this problem we manually analyzed nearly 100 cases in different programs and different nodes. Here we choose a readelf one to explain.

We pick a CFG node that is responsible for reading the **section type** structure in the **section header table**. We calculate the CAM based heatmap and gradient-based



(a) Part of CFG for OpenDatabase function



(b) CFG for OpenDatabase function

Fig. 6 OpenDatabase function

heatmap separately and the result is shown in Fig. 5. We find that the information CAM provided is more accurate and is with less noise. Figure 5a is calculated with CAM, Fig. 5b is built with gradient, and Fig. 5c is the file format structure. As we can see that the highlights provided by the CAM output are mostly in the **section table**, while the gradient-based provides us more noise, e.g. the front of the file and the middle of the file. The fuzzer may waste time in mutating the noise part.

Case study

Microsoft Jet database Engine type confusion vulnerability

In the experiment discussed in the *Evaluation* section, we found several Microsoft vulnerabilities and they have been fixed. We choose one of them to analyze, and the call stack at the program crash site is listed in Listing 1.w

Listing 1: Call Stack

```
1 msjet40!TblPage::CreateLvSMLocs+0x76
2 msjet40!TblPage::InitTable+0x277
3 msjet40!Table::Open+0x15a
4 msjet40!Instance::Instance+0x213
5 msjet40!Session::OpenDatabase+0x14c
6 msjet40!ErrIsamOpenDatabase+0xb0
7 msjet40!ErrOpenDatabase+0x208
```

To trigger the bug, we first need to guide the program to call *msjet*40!*ErrOpenDatabase*. Then we have to craft a **TblPage** object in the PoC to trigger the final crash. Since the *MDB* file format is very complicated, we do not analyze the format itself. Instead, we focus on how the DWA and local interpretation help the fuzzer find it.

We try recalling how our fuzzer finds the bug and record the intermediate data, such as the changes of weight for different nodes under the DWA strategy and the input positions our fuzzer focused on with local interpretation.

When we analyze the *Session* : : *OpenDatabase* (line 5 in Listing 1)function, we find that the DWA strategy helps the fuzzer avoid wasting time in useless nodes in this function very well. Part of the CFG for this function is shown in Fig. 6 while the whole CFG is shown in Fig. 6b. There is a call to **Connection::FindDatabase** in the first node. During the entire fuzzing process, we find that **FindDatabase** always returns 0, which we consider as a success status. The call to **FindDatabase** seems to ensure that the database exists and can be connected to before further processing. We count the 13 nodes on the left and 11 on the right separately in Fig. 6b. The number represents the initial weight.

If we do not apply DWA, more power will be spent on the left since it has more unvisited blocks than that on the right. Nevertheless, in our case, DWA helps the fuzzer decrease the weight of the left part dynamically. When fuzzing progresses, DWA finds it too hard to reach the left part, and the fuzzer will decrease the weight. When we finally reach the crash, we find that the weight for the first block has been decreased to a negative value, which means this block is not as important as initially marked. Also, we find that when feeding our trained model with the first node of *InitTable*, the Local Interpretation module provides us positions mostly in the **Table Definition**, an essential structure in *.MDB* file. It is also where the fuzzer mutates and causes the final crash. In this case, the DWA strategy helps the fuzzer focus on more valuable blocks, while the Local Interpretation module helps the fuzzer identify where to fuzz.

Jhead out of bounds read bug

Jhead is a command-line tool for processing photo EXIF information and it is shipped with Fedora by default.

Listing 2: Key code around the bug

```
switch(marker){
    case M_DQT:
        process_DQT(Data, itemlen);
        break;
    case M_DHT:
        process_DHT(Data, itemlen);
        break;
    ...
}
```

We take the discovery of a known vulnerability in jhead, CVE-2020-6624, as an example to demonstrate the methodology of CAMFuzz. The code in Listing 2 shows the critical part to trigger the bug. It is responsible for using the marker type of Jpeg *section* and traversing the **Data** to build different tables.

To trigger the bug, the magic bytes of the *section* must be equal to **M_DQT**, then *Data* will be processed as the DQT section. We check the critical values found by the static analysis module and find it interesting that the fuzzer finds both **M_DQT** and **64**, which is a loop count. As we have claimed in Magic Value Identification, after identifying the critical values, we will add several adjacent values to critical values to mutate the space sufficiently. Taking this case as an example, we choose a random threshold of T and add $(64 - T, 64 + T)$ as candidate values.

Related work

The previous grey-box fuzzing studies have achieved great success, including various learning-based and guided fuzzing methods.

Learning-based fuzzing

Several learning-based methods have been proposed to help improve the fuzzing efficiency.

The first kind studies the relationship between input and code coverage. Rajpal et al. (2017) train the model to predict which part of the modification is promising to mutate. The intuition is that during fuzzing, mutations to some positions may trigger new path coverage while others do not. Instead of just predicting which input part may bring new coverage, CAMFuzz knows which part of the input has a high relationship with a given CFG node, and by assigning different weights to the nodes, it can mutate the input more targeted. NEUZZ (She et al. 2019) uses a neural network to make a smooth approximation of the branching behavior in target programs. However, the gradient may bring noise and thus cause a waste of time in mutating useless positions. MTFuzz (She et al. 2020) considers multiple factors such as context and mutation approach when training the NN to predict coverage. FuzzGuard (Zong et al. 2020) uses the NN to predict whether the seed file can achieve the target. It is a kind of Directed Grey-box Fuzzing (DGF) technique. CAMFuzz tries to solve different problems in fuzzing than both of these two fuzzers.

Another type is to use algorithms to learn the file format (Blazytko et al. 2019; Lee et al. 2020; Wang et al. 2017; You et al. 2019; Godefroid et al. 2017; Sivakorn et al. 2017; Mathis et al. 2020; Fioraldi et al. 2020). These algorithms could automatically learn the format or the grammar and then make changes within the specification scope. Different from these solutions, our intuition is to determine the relationship between input and CFG. Although CAMFuzz does not infer the input format, it can know which part of code is easier to be covered and which input part is highly related to a given code block. By knowing the information, CAMFuzz can mutate the input better.

Guided fuzzing

Some research works try to make the fuzzing more efficiently based on different guidance methods. Angora (Chen and Chen 2018), TaintScope (Wang et al. 2010), Steelix (Li et al. 2017), and some other works like Wagner (2009); Cadar et al. (2008); Godefroid et al. (2012, 2008); Rawat et al. (2017); Yun et al. (2018); Stephens et al. (2016); Choi et al. (2019) use taint analysis or a combination of symbolic execution to obtain coverage and other information to guide the fuzzing based on the detailed feedback information.

VUzzer (Rawat et al. 2017) proposes a fuzzing method without any prior knowledge of the application or input format. It collects information based on the control flow and data flow results provided by Pin (Luk et al. 2005) and then uses it to generate more interesting seeds. These traditional methods rely on the CPU heavily, but there is a disproportion between the growth of CPU and fuzzing performance, since there are nonlinear constraints in the execution path for the solver. Traditional methods may get stuck when solving the constraints.

Unlike these approaches, with a prediction-based method, we do not need symbolic execution or data-flow analysis which may be considered as heavy and cause the fuzzing stuck. Our method relies on GPU and we have proven that a low-cost GPU can achieve more improvement than traditional fuzzers. We use static analysis to extract particular values in the code blocks, while during fuzzing, we use the DWA strategy to determine which block is more important than others.

Conslusion

This paper proposes a novel fuzzing technique mainly based on the local interpretation technique in machine learning. Combined with static analysis and DWA strategy, CAMFuzz can guide the fuzzer on where and how to fuzz. By the experiment, it shows that CAMFuzz outperforms state-of-the-art fuzzers, and CAMFuzz also found 19 previously unknown vulnerabilities in the Microsoft Jet engine, Hancom Office, and several open-source programs.

Acknowledgements
Thanks for the anonymous reviewers.

Author contributions
All authors read and approved the final manuscript.

Funding
This work was supported in part by Innovative Research Group Project of the National Natural Science Foundation of China (62032010), National Key R&D Program of China (2021YFB2701000) and National Natural Science Foundation of China under Grant 61972224.

Availability of data and materials
Not applicable.

Declarations

Competing interests
No potential competing interest was reported by the authors.

Author details
[1]Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China. [2]School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China. [3]Key Laboratory of Network Assessment Technology, CAS, Beijing, China. [4]Beijing Key Laboratory of Network Security and Protection Technology, Beijing, China. [5]Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing, China.

Received: 8 December 2021 Accepted: 14 February 2022
Published online: 01 September 2022

References

Aschermann C, Frassetto T, Holz T, Jauernig P, Sadeghi A-R, Teuchert D (2019a) Nautilus: fishing for deep bugs with grammars. In: NDSS
Aschermann C, Schumilo S, Blazytko T, Gawlik R, Holz T (2019b) Redqueen: fuzzing with input-to-state correspondence. In: NDSS, vol 19, pp 1–15
Barandela R, Valdovinos RM, Sánchez JS, Ferri FJ (2004) The imbalanced training sample problem: Under or over sampling? In: Joint IAPR international workshops on statistical techniques in pattern recognition (SPR) and structural and syntactic pattern recognition (SSPR). Springer, pp 806–814
Blazytko T, Bishop M, Aschermann C, Cappos J, Schlögel M, Korshun N, Abbasi A, Schweighauser M, Schinzel S, Schumilo S et al (2019) {GRIMOIRE}: Synthesizing structure while fuzzing. In: 28th {USENIX} Security Symposium ({USENIX} Security 19), pp 1985–2002
Bohme M (2018) Winaflfast. https://github.com/mboehme/winaflfast
Böhme M, Pham V-T, Roychoudhury A (2017) Coverage-based greybox fuzzing as markov chain. IEEE Trans Softw Eng 45(5):489–506
Bruening D (2020) QZ: Dynamorio: dynamic instrumentation tool platform
Cadar C, Dunbar D, Engler DR et al (2008) Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI, vol 8, pp 209–224
Casanova A, Cucurull G, Drozdzal M, Romero A, Bengio Y (2018) On the iterative refinement of densely connected representation levels for semantic segmentation. In: Proceedings of the IEEE conference on computer vision and pattern recognition workshops, pp 978–987
Castillo B, Di Gennaro S, Monaco S, Normand-Cyrot D (1997) On regulation under sampling. IEEE Trans Autom Control 42(6):864–868
Chen P, Chen H (2018) Angora: efficient fuzzing by principled search. In: 2018 IEEE symposium on security and privacy (SP). IEEE, pp 711–725
Chipounov V, Kuznetsov V, Candea G (2011) S2e: a platform for in-vivo multi-path analysis of software systems. In: ACM SIGARCH computer architecture news, vol 39. ACM, pp 265–278
Choi J, Jang J, Han C, Cha SK (2019) Grey-box concolic testing on binary code. In: 2019 IEEE/ACM 41st international conference on software engineering (ICSE). IEEE, pp 736–747
Dong L, Yang N, Wang W, Wei F, Liu X, Wang Y, Gao J, Zhou M, Hon H-W (2019) Unified language model pre-training for natural language understanding and generation. In: Advances in neural information processing systems, pp 13042–13054
DynamoRIO (2020) DynamoRIO. https://dynamorio.org/
Fioraldi A, D'Elia DC, Coppa E (2020) Weizz: automatic grey-box fuzzing for structured binary formats. In: Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis, pp 1–13
Fratric I (2017) WinAFL: a fork of AFL for fuzzing Windows binaries
Gan S, Zhang C, Qin X, Tu X, Li K, Pei Z, Chen Z (2018) Collafl: path sensitive fuzzing. In: 2018 IEEE symposium on security and privacy (SP). IEEE, pp 679–696
Gan S, Zhang C, Chen P, Zhao B, Qin X, Wu D, Chen Z (2020) Greyone: data flow sensitive fuzzing. In: 29th USENIX security symposium (USENIX Security 20). USENIX Association, Boston. https://www.Usenix.Org/conference/usenixsecurity20/presentation/gan
García, S, Luengo J, Herrera F (2015) Data preprocessing in data mining, vol 72. Springer
Ghorbani A, Abid A, Zou J (2019) Interpretation of neural networks is fragile. In: Proceedings of the AAAI conference on artificial intelligence, vol 33, pp 3681–3688
Godefroid P, Levin MY, Molnar DA et al (2008) Automated whitebox fuzz testing. In: NDSS, vol 8, pp 151–166
Godefroid P, Levin MY, Molnar D (2012) Sage: whitebox fuzzing for security testing. Commun ACM 55(3):40–44
Godefroid P, Peleg H, Singh R (2017) Learn&fuzz: machine learning for input fuzzing. In: Proceedings of the 32nd IEEE/ACM international conference on automated software engineering. IEEE Press, pp 50–59

Groß S (2018) Fuzzil: coverage guided fuzzing for javascript engines. PhD thesis, TU Braunschweig
Guidotti R, Monreale A, Ruggieri S, Turini F, Giannotti F, Pedreschi D (2018) A survey of methods for explaining black box models. ACM Comput Surv 51(5):1–42
Jeong DR, Kim K, Shivakumar B, Lee B, Shin I (2019) Razzer: finding kernel race bugs through fuzzing. In: 2019 IEEE symposium on security and privacy (SP). IEEE, pp 754–768
Kang MG, McCamant S, Poosankam P, Song D (2011) Dta++: dynamic taint analysis with targeted control-flow propagation. In: NDSS
Klees G, Ruef A, Cooper B, Wei S, Hicks M (2018) Evaluating fuzz testing. In: Proceedings of the 2018 ACM SIGSAC conference on computer and communications security. ACM, pp 2123–2138
Lee S, Han H, Cha SK, Son S (2020) Montage: a neural network language model-guided Javascript engine fuzzer. arXiv preprint arXiv:2001.04107
Lemieux C, Sen K (2018) Fairfuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. In: the 33rd ACM/IEEE international conference
Li Y, Chen B, Chandramohan M, Lin S-W, Liu Y, Tiu A (2017) Steelix: program-state based binary fuzzing. In: Proceedings of the 2017 11th joint meeting on foundations of software engineering, pp 627–637
Luk CK, Cohn RS, Muth R, Patil H, Klauser A, Lowney PG, Wallace S, Reddi VJ, Hazelwood KM (2005) Pin: building customized program analysis tools with dynamic instrumentation. Acm Sigplan Notices 40(6):190–200
Mathis B, Gopinath R, Zeller A (2020) Learning input tokens for effective fuzzing. In: Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis, pp 27–37
Pishchulin L, Insafutdinov E, Tang S, Andres B, Andriluka M, Gehler PV, Schiele B (2016) Deepcut: joint subset partition and labeling for multi person pose estimation. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp 4929–4937
Provost F (2000) Machine learning from imbalanced data sets 101. In: Proceedings of the AAAI'2000 workshop on imbalanced data sets, vol 68. AAAI Press, pp 1–3
Rajpal M, Blum W, Singh R (2017) Not all bytes are equal: neural byte sieve for fuzzing. arXiv preprint arXiv:1711.04596
Rawat S, Jain V, Kumar A, Cojocar L, Giuffrida C, Bos H (2017) Vuzzer: application-aware evolutionary fuzzing. In: NDSS, vol 17, pp 1–14
She D, Pei K, Epstein D, Yang J, Ray B, Jana S (2019) Neuzz: efficient fuzzing with neural program smoothing. IEEE Secur Privacy 6:66
She D, Krishna R, Yan L, Jana S, Ray B (2020) Mtfuzz: fuzzing with a multi-task neural network. In: Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, pp 737–749
Shoshitaishvili Y (2014) Python bindings for Valgrind's VEX IR
Shoshitaishvili Y, Wang R, Salls C, Stephens N, Polino M, Dutcher A, Grosen J, Feng S, Hauser C, Kruegel C et al (2016) Sok:(state of) the art of war: offensive techniques in binary analysis. In: 2016 IEEE symposium on security and privacy (SP). IEEE, pp 138–157
Simonyan K, Vedaldi A, Zisserman A (2013) Deep inside convolutional networks: Visualising image classification models and saliency maps. arXiv preprint arXiv:1312.6034
Sivakorn S, Argyros G, Pei K, Keromytis AD, Jana S (2017) Hvlearn: automated black-box analysis of hostname verification in ssl/tls implementations. In: 2017 IEEE symposium on security and privacy (SP). IEEE, pp 521–538
Smilkov D, Thorat N, Kim B, Viégas F, Wattenberg M (2017) Smoothgrad: removing noise by adding noise. arXiv preprint arXiv:1706.03825
Stephens N, Grosen J, Salls C, Dutcher A, Wang R, Corbetta J, Shoshitaishvili Y, Kruegel C, Vigna G (2016) Driller: augmenting fuzzing through selective symbolic execution. In: NDSS, vol 16, pp 1–16
Takanen A, Demott JD, Miller C, Kettunen A (2018) Fuzzing for software security testing and quality assurance. Artech House
Wagner D (2009) Dynamic test generation to find integer bugs in x86 binary Linux programs. In: Proceedings of the 18th USENIX Security Symposium, Montreal, Canada, August 10–14, 2009
Wang F, Shoshitaishvili Y (2017) Angr—the next generation of binary analysis. In: 2017 IEEE cybersecurity development (SecDev), pp. 8–9. IEEE
Wang T, Wei T, Gu G, Zou W (2010) Taintscope: a checksum-aware directed fuzzing tool for automatic software vulnerability detection. In: 2010 IEEE symposium on security and privacy. IEEE, pp 497–512
Wang J, Chen B, Wei L, Liu Y (2017) Skyfire: data-driven seed generation for fuzzing. In: 2017 IEEE symposium on security and privacy (SP). IEEE, pp 579–594
Wang Y, Jia X, Liu Y, Zeng K, Su P (2020) Not all coverage measurements are equal: fuzzing by coverage accounting for input prioritization. In: Network and distributed system security symposium
You W, Wang X, Ma S, Huang J, Zhang X, Wang X, Liang B (2019) Profuzzer: on-the-fly input type probing for better zero-day vulnerability discovery. In: 2019 IEEE symposium on security and privacy (SP). IEEE, pp 769–786
Yue T, Wang P, Tang Y, Wang E, Yu B, Lu K, Zhou X (2020) Ecofuzz: adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In: 29th USENIX Security Symposium (USENIX Security 20). USENIX Association, Boston, MA. https://www.Usenix.Org/conference/usenixsecurity20/presentation/gan
Yun I, Lee S, Xu M, Jang Y, Kim T (2018) {QSYM}: a practical concolic execution engine tailored for hybrid fuzzing. In: 27th {USENIX} security symposium ({USENIX} Security 18), pp 745–761
Zalewski M (2014) American fuzzy lop
Zhou B, Khosla A, Lapedriza A, Oliva A, Torralba A (2016) Learning deep features for discriminative localization. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp 2921–2929
Zong P, Lv T, Wang D, Deng Z, Liang R, Chen K (2020) Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In: 29th {USENIX} Security Symposium ({USENIX Security} 20), pp 2255–2269

Publisher's Note