IDFuzz: Intelligent Directed Grey-box Fuzzing

Yiyang Chen, *Tsinghua University;* Chao Zhang, *Tsinghua University and JCSS, Tsinghua University (INSC) - Science City (Guangzhou) Digital Technology Group Co., Ltd.;* Long Wang, *Tsinghua University;* Wenyu Zhu, *Tsinghua University and AscendGrace Tech;* Changhua Luo, *Wuhan University;* Nuoqi Gui, Zheyu Ma, and Xingjian Zhang, *Tsinghua University;* Bingkai Su, *Hunan University*

https://www.usenix.org/conference/usenixsecurity25/presentation/chen-yiyang

This paper is included in the Proceedings of the 34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

IDFuzz: Intelligent Directed Grey-box Fuzzing

Yiyang Chen[1], Chao Zhang[1,5*], Long Wang[1*], Wenyu Zhu[1,4], Changhua Luo[2], Nuoqi Gui[1],
Zheyu Ma[1], Xingjian Zhang[1], Bingkai Su[3]
[1]*Tsinghua University,* [2]*Wuhan University,* [3]*Hunan University,* [4]*AscendGrace Tech,*
[5]*JCSS, Tsinghua University (INSC) - Science City (Guangzhou) Digital Technology Group Co., Ltd.*
chenyy23@mails.tsinghua.edu.cn, {chaoz, longwang}@tsinghua.edu.cn

Abstract

Directed grey-box fuzzing aims to test target code in programs and is widely utilized in various scenarios, including patch testing, candidate vulnerability confirmation, and known vulnerability reproduction. However, we find that existing directed fuzzers generally lack effective input mutation strategies and resort to the randomness and empiricism inherent in AFL-based strategies, which prove to be inefficient in directed fuzzing contexts.

This paper presents IDFuzz, an intelligent input mutation solution for directed fuzzing. Our key insight is to leverage a neural network model to learn from historically mutated inputs and extract useful experience that can guide input mutation towards the target code. We introduce several novel techniques in model construction and model training, which help build a model that well captures experience on how to cover both explored and unexplored code relevant to the target. We further devise a refined model gradient-guided scheme that leverages the experience to locate critical input fields and develop a directed input mutation strategy. We implement IDFuzz as an input mutation module that complements most open-source state-of-the-art directed fuzzers. In our evaluation, IDFuzz significantly accelerates existing directed fuzzers by over 2.48x in reproducing target vulnerabilities on the Google Fuzzer Test Suite. Moreover, we demonstrate that IDFuzz helps existing directed fuzzers reduce ineffective mutations by 91.86%. Lastly, we detected 6 previously unknown vulnerabilities with 4 CVE IDs assigned so far and 1 incomplete fix of a high-severity vulnerability in well-tested real-world software using IDFuzz.

1 Introduction

Fuzzing [35] has become one of the most extensively utilized techniques to find vulnerabilities in cyberspace. Typically, fuzzing is categorized into two branches [28, 33], distinguished by their goals. Coverage-guided fuzzing aims to achieve higher coverage of the program code, whereas directed fuzzing is designed to reach specific target sites within the code. In recent years, directed grey-box fuzzing (DGF) has gained increased attention as it is widely implemented in crash reproduction [37, 40, 46, 51, 57], vulnerability verification [10, 36, 56, 61], and patch testing [18, 42].

Numerous directed fuzzing solutions have been proposed, which often utilize the distance or path to the target code as feedback to schedule seed inputs or optimize input execution. For instance, some efforts [6, 11, 14, 25, 32] leverage distance metrics to prioritize inputs closer to the target code and apply more mutations to them. Some other efforts [19, 63] leverage path reachability to prune inputs and save their execution runtime, and works [22, 32] skip collecting coverage information of irrelevant code during input execution to narrow down the evolutionary direction.

However, limited attention has been devoted to the input mutation step in directed fuzzing. Existing directed grey-box fuzzers [6, 11, 14, 19, 22, 25, 32] often adopt a mutation strategy based on AFL's approach [60], which randomly chooses offsets and lengths during mutation. On the other hand, DGF aims to reach specific target sites with corresponding control flow requirements, often necessitating mutations at specific input fields (*critical fields*) to approach these target sites [11, 32]. Therefore, mutation strategies that randomly select offsets and lengths lack directedness and often lead to a large number of ineffective mutations in directed fuzzing. As demonstrated by recent results [63], over 65.1% of mutated inputs fail to approach the target sites.

Input mutation/generation has been extensively studied in the context of coverage-guided fuzzing. However, finding an effective solution of input mutation for directed fuzzing remains non-trivial due to its unique challenges. Two widely used optimization approaches, correlation analysis [5, 12, 26] and concolic execution [43, 59], face significant limitations when applied to directed fuzzing tasks.

Correlation analysis techniques, such as taint tracking [12], aim to match influencing input bytes to a given constraint. These techniques are highly effective in handling early sanity

*Corresponding authors.

checks, such as magic byte checking [5], which can greatly benefit coverage-guided fuzzing. However, they encounter issues such as over-tainting and under-tainting when handling many constraints [29, 50]. Directed fuzzing targets are often located deep in programs, guarded by intricate constraints [14, 25]. Correlation analysis is either computationally expensive or insufficiently accurate for addressing many of these constraints in directed fuzzing tasks.

Concolic execution leverages constraint solvers to resolve path constraints and can effectively assist fuzzers in bypassing tight constraints. However, there can be numerous paths that lead to directed fuzzing targets in real-world programs [11, 20], often causing concolic execution to suffer from high costs and scalability issues such as path explosion [12, 52].

Thus, a lightweight and scalable method capable of handling various constraints is desirable for directed fuzzing.

In this work, we propose a novel mutation strategy for directed fuzzing. Our key insight is that *historically mutated inputs encompass a wealth of experience that can guide input mutation towards the target code*. For example, if an input *a* is mutated into an input *b* that is closer to the target (*e.g.*, *b* advances by one basic block along the required path to the target), by comparing the byte-level differences between *a* and *b*, we can extract some experience for approaching the target (*i.e.*, where the critical field is). When mutating other inputs that are stuck at the same constraint as *a*, we can leverage this experience to directly mutate the critical fields of the inputs, thereby reducing ineffective attempts. This data-driven approach, which leverages the experience from past inputs, enables us to efficiently locate the critical fields corresponding to various constraints without the need to explicitly solve the constraints.

However, to leverage experience from historically mutated inputs in developing input mutation strategies, we have two key questions to answer.

Q1: Is such experience applicable to future mutations? We aim to apply the experience learned from *a* and *b* to the mutation of other inputs, specifically by identifying the offset and length of the critical field, mutating which can bring the input closer to the target. However, for different inputs, the critical field may vary significantly. Even for fields that correspond to the same part of the file format, most of them have different offsets across different inputs [7, 13, 17]. As a result, the experience gained from *a* and *b* is often only applicable to *a* as a specific input.

Q2: Can such experience improve the efficiency of directed fuzzing? Suppose input *a* has a distance of `d`, and input *b* has a distance of `d-1`. The experience gained from *a* and *b* theoretically includes at best how to move from `d` to `d-1`, but does not provide insights on how to generate inputs with a distance smaller than `d-1`. In other words, the knowledge we gain only helps guide mutations to reach distances that have already been achieved, while it does not offer guidance towards unexplored code regions. Covering new code is necessary to reach the target, so it remains questionable whether such experience can contribute to directed fuzzing progress.

Our Solution. We propose an intelligent neural network-based mutation method tailored for directed fuzzing that provides affirmative answers to both of the above questions.

For *Q1*, we found that while it is difficult to derive generalizable experience from a single pair of inputs, it is possible to extract a pattern between *how the input bytes are organized* and *the input distance* from the large volume of fuzzing inputs as a whole. This is because the correlation between input bytes and distance essentially reflects the execution logic of the program under test (PUT). The PUT reads input bytes into variables, generates the execution path, and determines distance. The PUT's execution logic is deterministic and stable, resulting in a strong pattern between input bytes and distance that is independent of specific inputs. While such a pattern is difficult to summarize using traditional program analysis methods [63], we found that neural network (NN) models have strong potential for learning this pattern, owing to their great promise in pattern recognition [9, 54].

We employed a simple multilayer perceptron (MLP) model [45] to learn the pattern between input bytes and distance, which is cost-effective for our task. We use the byte values of the fuzzing input as the model's input. To provide richer features for better model fitting, we did not directly use distance values as the model's output. Instead, we devised a novel *branching encoding* technique that encodes *relevant branching behaviors* as the model's output. We define relevant branching behaviors as the coverage information of the target code's dominating basic blocks (referred to as *dom-BBs*) in the program's control flow graph (CFG). By definition, dom-BBs are a sequence of basic blocks that must be covered sequentially to reach the target code. Therefore, encoding their coverage as the model's output includes information not only about "whether the execution path is close to the target" (*i.e.*, distance) but also "how to get closer to the target" by enabling the model to learn how to sequentially cover these dom-BBs. Additionally, we developed an *adaptive dataset generation* technique that identifies and samples a small yet high-quality training set from the large volume of fuzzing inputs in an online manner.

For *Q2*, we found that experience can benefit directed fuzzing in two aspects. First, guiding the generation of inputs to reach explored code is already very beneficial. This is because, once the target is reached, we can utilize the experience from existing reachable inputs to generate more reachable inputs, which helps accelerate the reproduction of the target vulnerability [20]. Second, we found that experience from certain covered branches can be used to infer how to reach unexplored code. Our observation is that one control flow edge shares the same critical field as its sibling edges [53] that correspond to the same conditional statement. By leveraging *branching encoding* to associate the coverage of all branches at a conditional statement, the model can learn how to reach the unexplored edge from explored sibling edges. For example, if our target is a specific `switch` case, even if the target case has not been explored, we can infer the critical field corresponding to the `switch` statement from other explored cases. Thus, our approach not only retains close inputs but also continuously guides the generation of closer inputs.

Once the NN model finish training, it simulates the target function f_i in the following equation.

$$br_i = f_i(x_1, x_2, ..., x_n) \quad (1)$$

In the equation, br_i represents the encoded branching behavior of a dom-BB's associated program conditional statement, and x_1, x_2, ... represent the values of input bytes. The model input is the vector $< x_1, x_2, ..., x_n >$, and the model output is the vector $< br_1, br_2, ..., br_m >$, where br_i are ordered based on the dominance levels of the corresponding dom-BBs. We compute the partial derivative of br_i with respect to x_j. According to the mathematical definition, $\frac{\partial br_i}{\partial x_j}$ reflects how much the j-th input byte would impact the branching behavior br_i.

Thus, in theory, the input bytes that influence the conditional statement can be identified by locating the input bytes with the largest partial derivative absolute values $|\frac{\partial br_i}{\partial x_j}|$, and these partial derivatives can be effortlessly obtained by computing *the gradients of the model's output with respect to its input*. This idea of leveraging model gradients is inspired by neural program smoothing [49], introduced in §7. When determining which input bytes to mutate, we can specify the first dom-BB unreached by the current input and then identify its influencing bytes. Mutating these bytes can guide execution towards the dom-BB, thereby approaching the target code.

Based on this theoretical foundation, we further designed a novel *gradient filtering* technique to eliminate real-world noise in the model gradients. The noise is caused by uneven byte distributions in practical scenarios and can significantly reduce the accuracy of critical field identification. Then, we applied the Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm [15] on the gradient values of input bytes to group them and identify the offset and length of critical fields. Based on these two techniques, we developed an intelligent mutation strategy that efficiently identifies and mutates critical fields, guiding inputs towards the target code.

We implemented our approach as an input mutation module named **IDFUZZ** that can replace the AFL-based mutation strategy used in existing directed fuzzers. We integrated IDFUZZ into 4 state-of-the-art directed fuzzers [6, 14, 19, 32] and conducted experiments on the Google Fuzzer Test Suite (FTS) [1], a benchmark widely used to evaluate directed fuzzers. The results show that IDFUZZ accelerates the reproduction of target vulnerabilities with a speedup of over 2.48x and outperforms existing optimized mutation approaches. Moreover, we find that IDFUZZ helps existing directed fuzzers reduce ineffective mutations by 91.86% with only 6% increase in overhead. In addition, we detected 6 new vulnerabilities with 4 CVE IDs assigned so far and 1 incomplete fix in popular real-world software with AFLGo-IDFUZZ.

In summary, our contributions are as follows:

- We investigated the insufficiency of existing directed fuzzers and proposed leveraging experience from past inputs to guide input mutation towards the target code.
- We introduced an intelligent neural network-based mutation strategy tailored for directed fuzzing. We devised several novel techniques, including *branching encoding*, *adaptive dataset generation*, and *gradient filtering*, which help build a model that effectively learns from experience and uses it to improve the directedness of input mutation.
- We implemented a prototype input mutation module IDFUZZ and conducted extensive experiments. Our experiments showed that IDFUZZ significantly enhances the performance of existing directed fuzzers.

2 Motivation

We use a real-world example to motivate our work. The code snippet in Listing 1 is part of the `jhead` program for reading a JPEG format input file. In this example, *line 4* checks the file's magic bytes, and *lines 5-29* sequentially read each JPEG section through a `for` loop. In the loop, *lines 8-15* filter out the `0xff` padding bytes before the section and read the marker byte that determines the type of the section. *Lines 18-28* then use a `switch` to further process the section according to the value of the marker byte. Our target in this example is located in "`case M_COM`" at *line 24*.

2.1 Limitations of Existing Directed Fuzzers

Existing directed fuzzing approaches can be summarized into three main streams: optimizing input prioritization, optimizing input execution, and optimizing input generation.

Optimizing Input Prioritization. Most existing efforts [6, 11, 14, 25, 47], as exemplified by AFLGo [6], use a fitness metric to measure the distance from seed inputs to the target and prioritize seeds closer to the target for mutation. In this example, they manage to prioritize seeds that reach the `switch` statement, but they lack awareness of the marker byte "`M_COM`" during seed mutation, leading to a significant waste of mutations on other irrelevant bytes. Even if a few inputs successfully reach "`case M_COM`", their mutation strategy is oblivious to this success: when mutating unreachable inputs in the future, they cannot leverage this experience to prune ineffective mutations, and they may even mistakenly mutate the marker byte in reachable inputs, causing the execution path to deviate from the target.

Optimizing Input Execution. FuzzGuard [63] uses an NN model to predict the reachability of inputs and prunes unreachable inputs. Beacon [19] saves execution time by pre-

```
1  static int SectionsRead;
2  int ReadJpegSections(FILE* infile, ReadMode_t ReadMode) {
3      int a = fgetc(infile);
4      if (a != 0xff || fgetc(infile) != M_SOI) return FALSE;
5      for(;;) {
6          int prev = 0;
7          int marker = 0;
8          for (a=0;;a++) {
9              marker = fgetc(infile);
10             if (marker != 0xff && prev == 0xff) break;
11             if (marker == EOF) {
12                 ErrFatal(''Unexpected end of file'');
13             }
14             prev = marker;
15         }
16         Sections[SectionsRead].Type = marker;
17         SectionsRead += 1;
18         switch(marker) {
19             case M_SOS: ...
20             case M_DQT: ...
21             case M_DHT: ...
22             ...
23             case M_COM:
24                 // target
25             ...
26             case M_SOF15: ...
27             default: ...
28         }
29     }
30     return TRUE;
31 }
```

Listing 1: A motivating example.

maturely terminating the execution of unreachable inputs. MC^2 [47] employs Monte-Carlo execution to remodel directed fuzzing and estimates target-reaching inputs based on the execution frequency of program branches. SelectFuzz [32] and DAFL [22] use selective instrumentation to ensure that the feedback from input execution only includes coverage information of relevant code, thereby avoiding evolutionary direction deviate from the target. Although they effectively cull a large number of unreachable inputs or narrow down the search space, they are unable to directly generate inputs that reach "`case M_COM`".

Optimizing Input Generation. Halo [20] learns from executed inputs to infer the conditions for reaching the targets and mutates input fields into specific values based on these conditions. WAFLGo [55] selects the closest edge to the target as target edge and uses mutation masking to mutate only the input bytes that do not deviate from the target edge. However, neither Halo nor WAFLGo can infer the offset of the marker byte. They rely on sequentially testing each byte [20, 55], which is compared to taint tracking and similarly faces the issues of over-tainting and under-tainting. Additionally, Halo can infer the conditions for reaching "`case M_COM`" only if there are already inputs that reach "`case M_COM`".

2.2 Limitations of Constraint Solving

Another possible solution is to use constraint-solving techniques, such as correlation analysis [5, 12, 26] or concolic execution [8, 59], to bypass the `switch` statement.

Correlation analysis techniques face challenges when applied to identify the marker byte, whose offset can vary across different inputs: due to the `loop` at *line* 8 that checks padding bytes before the marker byte, taint tracking [12,29] may suffer from over-tainting, while input-to-state relationships [5, 17] might lead to inaccurate identification due to erroneous modifications of padding bytes during "colorization".

For concolic execution, its inherent scalability issues limit its effectiveness in reproducing bugs in large programs, particularly in this example, which involves program structures with nested `for` loops.

2.3 Towards Intelligent Directed Fuzzing

In this example, existing efforts struggle to reach "`case M_COM`". Even after successfully generating a few reachable inputs, they find it difficult to generate more such inputs [20]. We believe this is due to their *mutation-oblivious* nature: they fail to leverage fuzzing experience in their mutation.

Our intuition is to leverage the experience from mutated inputs to identify the marker byte and directly mutate it. We extract experience in two levels:

- At the first level, once a few inputs have successfully reached "`case M_COM`", we learn from these inputs and those stuck at `switch` to extract the experience that the marker byte is the critical field for reaching "`case M_COM`".
- At the second level, if "`case M_COM`" has not been reached, those inputs stuck at `switch` would take other cases. While existing approaches typically do not pay attention to which case these inputs take (assigning them the same distance [6, 11, 14, 22, 25, 32] or treating them uniformly as unreachable [19,20,63]), the critical field for reaching these cases is also the marker byte, holding the potential to infer the critical field for "`M_COM`". By capturing and learning the different branching behaviors at `switch` from these inputs, we identify that the critical field for all cases, including "`M_COM`", is the marker byte.

What exactly is "*experience*"? We leverage the experience learned by the model to identify critical input fields, where the "experience" is essentially *the gradients of the model's output* (*i.e.*, relevant branching behaviors) *with respect to its input* (*i.e.*, input bytes).

In this example, we use *branching encoding* to encode different branching behaviors of the `switch` statement as different *br* values in Equation 1, precisely representing different execution paths that ① do not reach `switch`, ② reach `switch` and take "`case M_COM`", and ③ reach `switch` but take other cases. Our model is designed to effectively learn the relationship between input bytes and these encoded branching behaviors. After model training, we compute the model gradients for a given input. The gradient $\frac{\partial br}{\partial x_j}$ corresponding to the marker byte theoretically has the largest absolute value. Thus, we can determine that the offset of the marker byte is j. We are fortunate to find that this approach can effectively

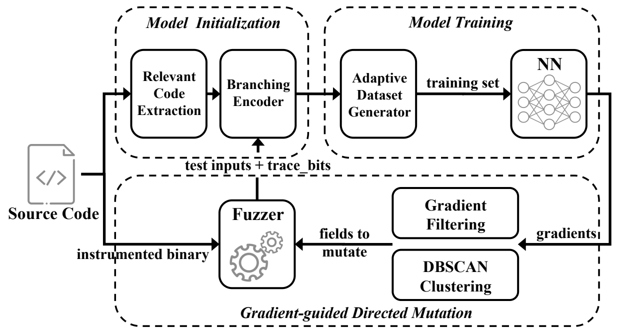


Figure 1: Workflow of IDFUZZ.

locate the marker byte even though its offset may vary across different inputs, as demonstrated in §5.4.

Challenges. We still face two challenges in achieving efficient and effective directed mutation.

C1: Efficiency in extracting experience. NN approaches may either consume significant resources or result in trivial models due to improperly defined machine learning tasks or low-quality training data [38, 63]. For our model, we need an output representation that encapsulates the required two-level experience while maintaining low dimensionality to reduce the difficulty of model fitting. Additionally, given the large volume of mutated inputs in fuzzing, we require an effective method to sample a dataset that retains the necessary experience while remaining as small as possible to minimize training overhead.

C2: Accuracy in identifying critical fields. In real-world fuzzing, uneven input byte distributions introduce noise into the model, leading to inaccurate identification of critical fields when following the naive principle that "the larger the gradient, the more critical the byte". In the motivating example, this results in mistakenly identifying the magic bytes of JPEG files, which are irrelevant, as critical fields (we will further explain this in §3.4). Therefore, a refined method of using the gradients is required for accuracy in identifying critical fields.

3 Design

3.1 Architecture Overview

The workflow of IDFUZZ can be divided into three phases: model initialization, model training, and gradient-guided directed mutation, as shown in Figure 1. In the model initialization phase, we use lightweight static analysis to extract the target code's dom-BBs and develop a Branching Encoder that can encode an input's branching behaviors at these dom-BBs (*i.e.*, relevant branching behaviors) into a tensor as the model output. In the model training phase, we employ an Adaptive Dataset Generator to collect fuzzing inputs, use the Branching Encoder to encode their execution paths, and generate the model training set. After model training, we feed seed inputs into the model and compute model gradients. Through gradient filtering and DBSCAN clustering techniques, we identify critical input fields and guide directed seed mutation. We will introduce the details of the three phases below.

3.2 Model Initialization

When initializing our NN model, we need to define the model's input and output. We use the byte values of the fuzzing input as the model's input. The model output is carefully designed to achieve two goals: it must effectively encapsulate the two-level experience discussed in §2.3, and it should maintain low dimensionality to minimize the training burden. We construct the model output through *relevant code extraction* and *branching encoding*.

Relevant Code Extraction. We utilize target dominance analysis to extract dom-BBs of the target site in the program's CFG. Target dominance analysis is a well-established lightweight static analysis technique used to efficiently extract the dominators of a given basic block from a program's CFG. It leverages the classic Lengauer-Tarjan [27] algorithm and is readily available in static analysis frameworks such as LLVM [23]. This approach condenses the entire program space to code strongly relevant to directed fuzzing, successfully reduced the dimensionality of the model output by 95% compared to representing the entire program space. The final output dimensionality is usually below 50.

Branching Encoding. Encoding execution paths is not a new concept, but existing encoding methods fail to meet our goals. One straightforward encoding approach is to use distance, as adopted by most directed fuzzers. However, distance is not precise enough for the model to learn how to bypass specific constraints. For instance, using AFLGo's distance definition, two execution paths with the same distance can be stuck at entirely different constraints. Moreover, the scalar nature of distance inherently prevents us from performing *gradient filtering* (introduced in §3.4). Another encoding approach is to map each edge to an independent dimension in the model output, as adopted by neural program smoothing [49]. This method can accurately represent execution paths, but it overlooks the relationships between sibling edges [53] and fails to incorporate the second-level experience, which is crucial for targeting unexplored code.

We introduce *branching encoding* to overcome the above limitations, as shown in Algorithm 1. By mapping the branching behavior of a dom-BB to one dimension of the output, we not only accurately capture the critical parts of the execution path but also associate uncovered branches with sibling branches, effectively leveraging second-level experience.

We first map the coverage of each dom-BB to one dimension of the output vector. If a dom-BB ($dom\_bbs[i]$) is covered, we consider the branching behavior of the preceding dom-BB ($dom\_bbs[i-1]$) to be appropriate and set the corresponding dimension for $dom\_bbs[i]$ to 1. Otherwise, if

Algorithm 1 Branching encoding.

```
Input: trace_bits
Output: output_vector

$output\_vector \leftarrow [0,\dots,0]$ // the dimensionality equals dom-BB number
$dom\_bbs \leftarrow target\_dominance\_analysis()$
for $0 \le idx < len(dom\_bbs)$ do
    for all $edge$ in $dom\_bbs[idx].edges$ do
        if $trace\_bits[edge] \neq 0$ then
            $output\_vector[idx] \leftarrow 1$
        end if
    end for
    if $output\_vector[idx] == 0$ then
        if $dom\_bbs[idx-1].is\_class\_a$ then
            $b \leftarrow 0$ // the branching behavior
            for all $edge$ in $dom\_bbs[idx-1].ndedges$ do
                $b \leftarrow b \ll 1$ // traverse each non-dom-edge
                if $trace\_bits[edge] \neq 0$ then
                    $b \leftarrow b+1$
                end if
            end for
            $b \leftarrow b/(1 \ll len(dom\_bbs[idx-1].ndedges))$
            $output\_vector[idx] \leftarrow b$
        end if
    end if
end for
return $output\_vector$
```

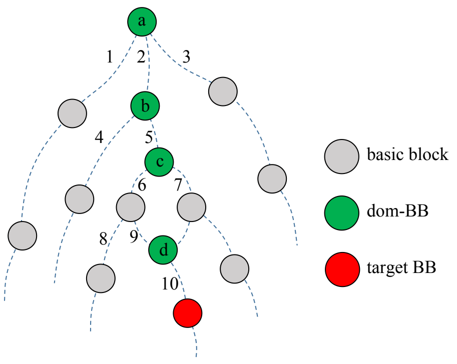


Figure 2: Dom-BBs in the program CFG.

$dom\_bbs[i]$ is not covered but $dom\_bbs[i-1]$ is, according to the characteristics of dominator tree, $dom\_bbs[i-1]$ will be the last covered dom-BB and $dom\_bbs[i]$ will be the first uncovered dom-BB. Covering $dom\_bbs[i]$ will bring us closer to the target site. In this case, we attempt to cover $dom\_bbs[i]$ by adjusting the branching behavior of $dom\_bbs[i-1]$.

We further use a fraction between 0 and 1 to represent the branching behavior of the last covered dom-BB ($dom\_bbs[i-1]$). Specifically, we categorize the dom-BBs into two classes, A and B. A-class dom-BBs are those that are adjacent to the next dom-BB, connected by a single edge termed as a "*dom-edge*", as exemplified by the basic blocks a, b, and d in Figure 2. In contrast, B-class dom-BBs are those that are separated from the next dom-BB by other non-dom-BBs, exemplified by the basic block c in Figure 2. If the final covered dom-BB of an execution path is A-class, a single change in its branching behavior is possible to cover the next dom-BB. Thus, the branching behaviors of A-class dom-BBs are highly relevant to reaching the target sites. We mark the outgoing edges of A-class dom-BBs. For an A-class dom-BB with n outgoing edges, we employ an $(n-1)$-bit binary number to indicate the coverage status of its $n-1$ "non-dom-edges", where coverage is denoted as 1 and non-coverage as 0. We then divide this binary number by $1 \ll (n-1)$ and place the result in the output vector dimension corresponding to the next dom-BB.

Conversely, if the final covered dom-BB is B-class, altering the branching behavior of the final dom-BB might not help to approach the target site. For instance, changing the branching behavior of dom-BB c would not directly benefit path "2→5→6→8" in Figure 2. To keep the analysis lightweight, we disregard the branching behavior of B-class dom-BBs and simply set the corresponding dimension to 0.

For example, we use a 5-dimensional output vector to represent the coverage of dom-BBs in Figure 2. The path "1" is encoded as $< 1, 0.25, 0, 0, 0 >$, the path "2→4" as $< 1, 1, 0.5, 0, 0 >$, and path "2→5→6→8" as $< 1, 1, 1, 0, 0 >$.

It is notable that when encoding an A-class branching behavior, we randomly map the edges to the binary bits. Larger binary values do not necessarily indicate greater proximity to reaching the next dom-BB. In fact, quantifying how close a branching behavior is to reaching the next dom-BB is both difficult and unnecessary, as it only affects the sign of model gradients and does not impact the identification of critical fields based on the magnitude of the absolute gradient values.

3.3 Model Training

We utilize an *adaptive dataset generation* technique to obtain a small yet sufficient dataset for model training.

Adaptive Dataset Generation. A straightforward approach to collect the training dataset is to directly use the fuzzing seed queue [49]. However, since only inputs that cover unique paths are retained as seed inputs, the seed queue often lacks sufficient samples with the same label, which can cause issues such as model underfitting [48,63]. Another possible approach is to include inputs generated by every seed in the dataset to ensure that the model can guide the mutation of any seed. However, this would result in an oversized training dataset and increase training costs. Therefore, based on the second approach, we reduce the required inputs in two steps.

First, the seed queue contains a large number of early accumulated inputs that cover very few dom-BBs. They are far from the target and typically receive minimal mutations in power scheduling [6]. Therefore, their experience is less valuable. After obtaining all labels (*i.e.*, relevant branching behaviors) from the seed queue, we select the top 20% of dom-BB counts and only focus on seed inputs of these labels.

Second, we can reduce the number of required samples by leveraging the similarity in byte distributions among neighboring seed inputs. Except for the initial seeds, all seed inputs are generated by mutating existing seeds. For a given seed, we refer to those seeds that are within two mutation steps (excluding splicing) as *neighboring seeds*. We observed that the byte distributions of neighboring seeds are typically very similar. Thus, if neighboring seeds share the same relevant branching behavior, we only need to collect the inputs generated by one representative seed.

We use a simple greedy algorithm to select the representative seeds that cover all the important labels. Based on the recommended sample size [3, 16], for each label, we evenly collect 100-150 inputs generated by these seeds during fuzzing. The final training set size is approximately 1000.

Timing of Training. We hope to train the model as early as possible to provide effective guidance for fuzzing mutations. However, model training takes time. If fuzzing covers a significant number of new dom-BBs during the training period, the experience contained by the model will become "outdated" by the time the model finishes training. Therefore, we should avoid starting training during periods when new dom-BBs are frequently discovered, which typically occurs in the early stages of the fuzzing process. Instead of hardcoding the start time for model training, we designed a simple adaptive method to accommodate different programs. After fuzzing begins, we continuously monitor the frequency of new dom-BB discoveries. If no new dom-BBs are covered within a threshold time θ, we proceed to generate the training set and begin training.

After the initial model training, new seed inputs may still emerge that either cover new dom-BBs or have significantly different byte distributions compared with existing seeds (more than two mutation steps away or generated by splicing). The model may not effectively guide the mutation of these seeds. Therefore, if the number of newly covered dom-BBs reaches n, or if the proportion of seeds with significantly different byte distributions exceeds R, we select an appropriate time to retrain the model based on the θ threshold.

Based on our experimental experience, we set θ to 60 seconds, n to 1, and R to 5%.

3.4 Gradient-guided Directed Mutation

After training the model, we calculate gradients in the NN to guide directed seed mutation.

Given a model input of dimensionality n and output of dimensionality m, the gradient of the output with respect to the input is an $m \times n$ Jacobi matrix J.

$$\begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \frac{\partial y_m}{\partial x_2} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

Each column of the matrix corresponds to one seed byte and each row corresponds to one dom-BB. We refer to the i-th row of the matrix as the *gradient vector* for the i-th dom-BB.

Algorithm 2 Gradient filtering.

```
1:  Input: Jacobi_matrix, n, training_set // locate the $n$-th dom-BB's critical bytes
2:  Output: hot_bytes
3:
4:  $hot\_bytes \leftarrow \{\}$
5:  $g_0 \leftarrow Jacobi\_matrix[n]$ // gradient vector for the $n$-th dom-BB
6:  $g_{-2} \leftarrow Jacobi\_matrix[n-2]$
7:  $g_{+2} \leftarrow Jacobi\_matrix[n+2]$
8:  for $0 \leq byte\_idx < \text{len}(g_0)$ do
9:      if $g_0.rank(byte\_idx) < g_{-2}.rank(byte\_idx)$ and
           $g_0.rank(byte\_idx) < g_{+2}.rank(byte\_idx)$ then
10:         $hot\_bytes.append(byte\_idx)$ // retain bytes with relatively higher ranked gradient values
11:     end if
12: end for
13: $hot\_bytes \leftarrow sort(hot\_bytes, -1)[:20]$ // top 20 bytes with the largest gradient values
14: return $hot\_bytes$
```

Theoretically, J_{ij} demonstrates how much the j-th seed byte affects the branching behavior of the i-th dom-BB.

However, we observed in our experiments that the seed bytes with the largest gradient values often do not correspond to critical bytes (*i.e.*, bytes that make up the critical field), but rather bytes from fixed fields (bytes with the same values across almost all samples, *e.g.*, magic bytes). We find a mathematical explanation: the value of J_{ij} in the matrix, $\frac{\partial y_i}{\partial x_j}$, can be interpreted as the change in the coverage of the i-th dom-BB divided by the change in the j-th seed byte value, as both changes approach 0. For a seed byte with almost the same value across all samples, the change in the byte value is near 0, resulting in a large J_{ij}. Since fuzzing inputs typically need to conform to certain file formats, fixed fields like magic bytes are very common. As a result, the "critical bytes" identified by locating the input bytes with the largest absolute gradient values are often these fixed bytes.

Meanwhile, we also observed that aside from fixed bytes, critical bytes for neighboring dom-BBs can correspond to large values in the current dom-BB's gradient vector. This can be explained by the sequential nature of dom-BBs' coverage, where covering the n-th dom-BB requires first covering the $(n-1)$-th, and not covering the n-th dom-BB requires first not covering the $(n+1)$-th. Therefore, during model training, critical bytes for the $(n-1)$-th and $(n+1)$-th dom-BBs are also important for the n-th dom-BB. Hence, in addition to excluding fixed bytes, gradient analysis needs to also exclude critical bytes for neighboring dom-BBs.

We propose a *gradient filtering* method to remove fixed bytes and critical bytes of neighboring dom-BBs. We present the gradient filtering algorithm in Algorithm 2.

3.4.1 Gradient Filtering

Our intuition is twofold: first, fixed bytes will have large gradient values across gradient vectors of all dom-BBs, while critical bytes will only have large gradient values in the gradient vectors of the current dom-BB and neighboring dom-BBs; second, as the distance between two dom-BBs increases, the influence of their critical bytes on each other's gradient vector diminishes. Therefore, we consider the gradient vectors of the $(n-2)$-th and n-th dom-BBs. In the gradient vector of the $(n-2)$-th dom-BB, the dimensions with large values correspond to fixed bytes and the critical bytes from the $(n-3)$-th to $(n-1)$-th dom-BBs. The critical bytes for other dom-BBs can be neglected since those dom-BBs are farther away. Similarly, in the gradient vector of the n-th dom-BB, the dimensions with large values correspond to fixed bytes along with the critical bytes from the $(n-1)$-th to the $(n+1)$-th dom-BBs. Thus, by removing the intersection of these two sets of bytes, we can effectively filter out the fixed bytes and the critical bytes for the $(n-1)$-th dom-BB.

3.4.2 Gradient-guided Directed Mutation

We apply the above idea to the gradient-guided mutation strategy. Given a seed, gradient-guided mutation aims to generate a test case that covers one more dom-BB (a single mutation is unlikely to drastically alter the execution path [11], so we do not expect covering multiple deeper dom-BBs through one mutation). Based on the seed's execution path, we adopt two different types of gradient-guided mutation: *aggressive mutation* and *conservative mutation*.

Aggressive Mutation. Suppose the execution path of the seed to mutate covers n dom-BBs. If the model training set contains samples covering $n+1$ dom-BBs (first-level experience) or n dom-BBs but with different branching behaviors (second-level experience), then the trained model will learn the experience of how to cover the $(n+1)$-th dom-BB. In this case, we apply aggressive mutation to cover the $(n+1)$-th dom-BB: we take bytes corresponding to the largest dimensions in the $(n+1)$-th dom-BB's gradient vector, filter them using the $(n-1)$-th and $(n+3)$-th dom-BBs' vectors for critical bytes, and mutate these critical bytes.

Conservative Mutation. If the deepest coverage in the training set is n dom-BBs and samples covering n dom-BBs exhibit only one branching behavior (or the n-th dom-BB is a B-class), then the model cannot learn anything about how to cover the $(n+1)$-th dom-BB. In this case, we apply conservative mutation to avoid mutations that regress already covered dom-BBs: we avoid mutating bytes corresponding to large values (the top 20) in the n-th dom-BB's gradient vector.

Havoc Mutation. The gradient filtering method is not absolutely accurate. In rare cases, it may incorrectly filter out critical bytes (discussed in §5.2). Therefore, after the above stages, IDFUZZ enters a brief AFL_{havoc} mutation stage [60] to compensate for possible error filtering. Additionally, if the model is not ready, we run AFL_{havoc} and skip other mutations.

3.4.3 Field Recognition

Building upon the identification of critical bytes, we utilize DBSCAN clustering [15] to further recognize critical fields. DBSCAN clustering algorithm is a widely used density-based clustering algorithm that identifies clusters as regions of high point density. In IDFUZZ, we treat a seed input as a one-dimensional space, where each byte is positioned according to its offset, and its associated gradient value is used as the density. Clustering in this space yields segments of contiguous bytes exhibiting consistently high gradient magnitudes. We used the existing implementation of the DBSCAN algorithm provided by the `scikit-learn` package [41] in Python 3.

IDFUZZ applies AFL's mutators, where the choices for lengths are 1, 2, and 4. However, clustered contiguous byte lengths are not necessarily base-2 integers. Hence, we take the smallest base-2 integer greater than the segment length as the final length of the partitioned field. For instance, if the length of a segment is 3, the partitioned field would be the 4 bytes including these 3 bytes as well as a preceding or succeeding byte. During aggressive mutation, we not only mutate individual critical bytes in 0-255 but also collectively mutate the entire identified field to interesting values [60].

4 Implementation

We observed that nearly all state-of-the-art directed fuzzers [6, 14, 19, 22, 32, 63] are based on the AFL framework [60]. To validate the efficacy of our design and its complementarity with existing directed fuzzers (*i.e.*, IDFUZZ optimizes input mutation, whereas existing directed fuzzers focus on other aspects), we implemented IDFUZZ as a replacement for the original input mutation module in the AFL framework. The Relevant Code Extractor was implemented in C++. The Branching Encoder, the Adaptive Dataset Generator, and the NN model were implemented in Python. The refined mutation strategy was implemented as a C patch to "`afl-fuzz.c`".

Relevant Code Extraction. We extract the call graph of the program from its LLVM IR bitcode [23] and identify the function call chain for the target site. We then utilize the `DominatorTreeWrapperPass` class [2] to extract the dominator tree for each function and identify the dominator chain from the function entry basic block to the call site basic block. Finally, we connect these dominator chains to derive the interprocedural dominator chain for the target site. If a callee function is invoked by multiple call sites, we take the intersection of the dominator chains of these call sites. During instrumentation, we keep track of the instrumentation indexes for outgoing edges of dom-BBs.

NN Architecture. There are already related studies [49, 53] that explore the use of different models to simulate program behaviors. Thus, we did not prioritize the exploration of model architectures as a contribution in our work. Based on the model selections in these related works, we chose a 3-layer MLP implemented in PyTorch [39] as the model for IDFUZZ, which is cost-effective for our task. We use `ReLU` as the activation function for the hidden layers and `sigmoid` for the output layer to normalize each output dimension. The model has 512 hidden neurons. In our evaluation, it achieved an accuracy that exceeded 95% in all programs within 10 epochs.

We also experimented with increased numbers of neural network layers and more complex models like CNNs [24]. However, these endeavors incurred higher overhead without a corresponding improvement in gradient accuracy, which is consistent with prior results [53]. We leave the exploration of more potential models for future work.

5 Evaluation

In this section, we conduct a series of experiments to evaluate IDFUZZ and answer the following four research questions:

- **RQ1:** How efficiently can IDFUZZ accelerate existing directed fuzzers in triggering known vulnerabilities? (§5.1)
- **RQ2:** How does each component affect IDFUZZ's performance? (§5.2)
- **RQ3:** What is the runtime overhead of IDFUZZ? (§5.3)
- **RQ4:** How well can IDFUZZ locate critical fields? (§5.4)

5.1 Efficiency in Triggering Known Vulnerabilities

Baselines and Benchmark. Vulnerability reproduction is a typical application scenario for directed grey-box fuzzing. To evaluate the efficacy of IDFUZZ in vulnerability reproduction, we equipped AFLGo [6], WindRanger [14], Beacon [19] and SelectFuzz [32] with IDFUZZ, replacing their original input mutation modules. AFLGo and WindRanger are representative directed fuzzers that focus on optimizing input prioritization, while Beacon and SelectFuzz are state-of-the-art directed fuzzers that focus on optimizing input execution. We did not include other directed fuzzers, as they are either not publicly accessible (*e.g.*, FuzzGuard [63] and CAFL [25]) or have their own optimized mutation strategies (*e.g.*, WAFLGo [55]).

Moreover, we compared IDFUZZ with other input generation solutions. We selected Angora [12] and SymCC [43], which are widely used as baselines [20,32], representing correlation analysis and concolic execution solutions, respectively. We also included WAFLGo [55], a state-of-the-art directed fuzzer that employs *target edge selection* and *mutation masking* methods to optimize input mutation. Another advanced directed fuzzer that utilizes optimized input generation strategies, *i.e.*, Halo [20], was not included as it was not publicly accessible. Nevertheless, we believe that Halo's approach complements IDFUZZ, as Halo infers the values of critical input fields [20], while IDFUZZ determines their offsets and lengths. We integrated IDFUZZ into raw AFL for comparison to eliminate the influence of irrelevant directed strategies.

Experimental Setup. For each program in the FTS, we set the crash point (listed in Table 7) as the target site and used the seeds provided in the FTS as the initial seeds for fuzzing. If no seeds were given, a file with the content "hi" was used as the initial seed. All experiments were conducted 10 times with a time budget of 24 hours in Docker environments on a 64-bit Ubuntu 18.04 LTS server equipped with Intel(R) Xeon(R) Gold 6140 CPU @ 2.30GHz*72 and 256 GB of RAM. We did not use any GPUs and we restricted the NN model to run on a single CPU throughout the entire process.

Results. The results of different directed fuzzers equipped with IDFUZZ are shown in Table 1. We calculated the average time-to-exposure (TTE) of vulnerabilities and applied the Mann-Whitney U test [34] between the results with and without IDFUZZ. In Table 1, p indicates the p-value between the original fuzzer and the fuzzer equipped with IDFUZZ. T.O. represents that the fuzzer could not trigger the vulnerability within 24 hours. N/A indicates failure in building the program. *Ratio* indicates the improvement ratio after being equipped with IDFUZZ. When calculating the improvement ratios and p-values for cases involving a timeout, we treat the timeout as 86400 seconds and add a ">" symbol to the *Ratio* value to indicate that the actual improvement could be even greater.

Overall, equipping IDFUZZ provides speedups of 1.87x, >2.01x, >1.95x, and >1.64x for AFLGo, WindRanger, Beacon, and SelectFuzz, respectively. We found that the improvement brought by IDFUZZ is strongly correlated with the difficulty of reproducing the target vulnerability. For easy-to-reproduce targets, such as those reproduced within 5 minutes, the improvement brought by IDFUZZ is highly unstable and may even result in performance degradation. However, for hard-to-reproduce targets, such as those requiring more than an hour, IDFUZZ consistently delivers substantial improvements. The relationship between the improvement ratios and the TTE results (without IDFUZZ) is shown in Figure 3. Using 10 minutes as a threshold, for targets with TTE <10 minutes, the speedups for AFLGo, WindRanger, Beacon, and SelectFuzz are 1.02x, 1.06x, 0.97x, and 0.99x, respectively. Even when improvements are observed, the corresponding p-values are greater than 0.05, indicating a lack of statistical significance. For targets with TTE >10 minutes, the speedups are 2.88x, >2.63x, >2.38x, and >2.01x for AFLGo, WindRanger, Beacon, and SelectFuzz, respectively, with all the p-values less than 0.05, indicating statistically significant improvements.

The primary factor behind this phenomenon is the time required for the NN model to take effect. The model typically finishes its first training around 10 minutes after fuzzing starts. Therefore, if the TTE of the target vulnerability is less than 10 minutes, the model often has not had enough time to provide guidance. As the TTE increases, the time available for the model to take effect increases, leading to a more significant contribution to the target vulnerability reproduction.

For hard-to-reproduce targets (TTE >10 minutes), which are the actual focus of directed fuzzing, IDFUZZ demonstrates stable and significant efficacy, providing an average speedup of >2.48x for existing directed fuzzers. Notably, there are

Table 1: Time-to-exposure (seconds) for each target in Google Fuzzer Test Suite. "∅" represents the original directed fuzzer and "*I*" represents the directed fuzzer equipped with IDFUZZ.

Program	AFLGo				WindRanger				Beacon				SelectFuzz			
	∅	*I*	*Ratio*	*p*	∅	*I*	*Ratio*	*p*	∅	*I*	*Ratio*	*p*	∅	*I*	*Ratio*	*p*
boringssl	T.O.	T.O.	-	-	T.O.	T.O.	-	-	T.O.	T.O.	-	-	T.O.	T.O.	-	-
c-ares	32	53	0.60x	-	47	30	1.57x	>0.05	19	24	0.79x	-	31	32	0.97x	-
guetzli	1835	828	2.22x	<0.01	6192	1383	4.48x	<0.01	1513	820	1.85x	0.01	756	692	1.09x	0.04
harfbuzz	T.O.	T.O.	-	-	T.O.	T.O.	-	-	T.O.	30790	>2.81x	<0.01	T.O.	27172	>3.18x	<0.01
json	177	208	0.85x	-	327	293	1.12x	>0.05	181	185	0.98x	-	218	197	1.11x	>0.05
lcms	14256	3927	3.63x	<0.01	8966	3300	2.72x	<0.01	6626	2021	3.28x	<0.01	4279	2110	2.03x	<0.01
libarchive	T.O.	T.O.	-	-	T.O.	T.O.	-	-	T.O.	T.O.	-	-	T.O.	52241	>1.65x	<0.01
libssh	477	409	1.17x	>0.05	2355	1170	2.01x	<0.01	1035	697	1.48x	0.03	253	290	0.87x	-
libxml2	1115	876	1.27x	<0.01	1039	843	1.23x	<0.01	858	666	1.29x	0.03	684	593	1.15x	0.03
openssl$_{1.0.1f}$	24	17	1.41x	0.04	36	49	0.73x	-	24	19	1.26x	>0.05	21	30	0.70x	-
openssl$_{1.0.2d}$	284	343	0.83x	-	252	305	0.83x	-	147	171	0.86x	-	179	139	1.29x	>0.05
pcre	353	277	1.27x	>0.05		N/A			1273	648	1.96x	0.02	681	599	1.14x	0.04
re2	55548	14920	3.72x	<0.01	T.O.	30921	>2.85x	<0.01	59962	16932	3.54x	<0.01	53826	17755	3.03x	<0.01
vorbis	T.O.	T.O.	-	-	T.O.	T.O.	-	-	T.O.	31117	>2.37x	<0.01	T.O.	36389	>2.78x	<0.01
woff	10872	3048	3.57x	<0.01	8813	3510	2.52x	<0.01	9320	3836	2.43x	<0.01	12200	5012	2.43x	<0.01
TTE Range	**<600**	**>600**	***overall***		**<600**	**>600**	***overall***		**<600**	**>600**	***overall***		**<600**	**>600**	***overall***	
Avg. Ratio	1.02x	2.88x	1.87x		1.06x	>2.63x	>2.01x		0.97x	>2.38x	>1.95x		0.99x	>2.01x	>1.64x	

Table 2: Time-to-exposure (seconds) for each target in the Google Fuzzer Test Suite compared with other optimized input generation strategies. The symbols in the table have the same meaning as those in Table 1.

Program	AFL-IDFuzz	Angora			SymCC			WAFLGo		
	TTE	*TTE*	*Ratio*	*p*	*TTE*	*Ratio*	*p*	*TTE*	*Ratio*	*p*
boringssl	T.O.	T.O.	-	-	T.O.	-	-	T.O.	-	-
c-ares	31	33	1.06x	>0.05	84	2.71x	0.04	79	2.55x	<0.01
guetzli	1089	1937	1.78x	0.02	39397	36.18x	<0.01	3918	3.60x	<0.01
harfbuzz	T.O.	T.O.	-	-	T.O.	-	-	T.O.	-	-
json	128	189	1.48x	>0.05	73	0.57x	-		N/A	
lcms	5821	46810	8.04x	<0.01	T.O.	>14.84x	<0.01	12938	2.22x	<0.01
libarchive	T.O.	T.O.	-	-	T.O.	-	-	T.O.	-	-
libssh	823	466	0.57x	-	T.O.	>104.98x	<0.01	285	0.35x	-
libxml2	1010	3125	3.09x	<0.01	23347	23.12x	<0.01	1698	1.68x	0.01
openssl$_{1.0.1f}$	18	16	0.89x	-	545	30.28x	<0.01	36	2.00x	<0.01
openssl$_{1.0.2d}$	404	570	1.41x	>0.05	742	1.84x	0.03	158	0.39x	-
pcre	395	810	2.05x	<0.01	32513	82.31x	<0.01	2723	6.89x	<0.01
re2	23092	T.O.	>3.74x	<0.01	T.O.	>3.74x	<0.01	T.O.	>3.74x	<0.01
vorbis	T.O.	T.O.	-	-	T.O.	-	-	T.O.	-	-
woff	2953	5277	1.79x	<0.01	14864	5.03x	<0.01	T.O.	>29.26x	<0.01
Avg. Ratio			>2.35x			>27.78x			>5.27x	

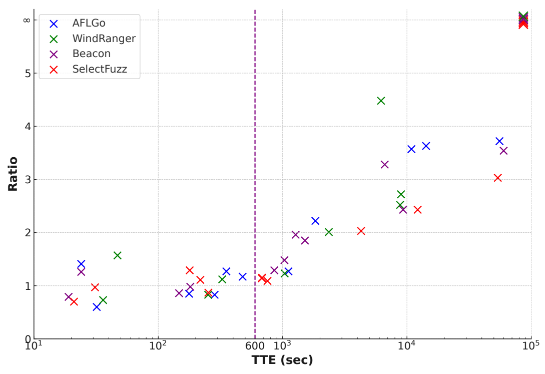


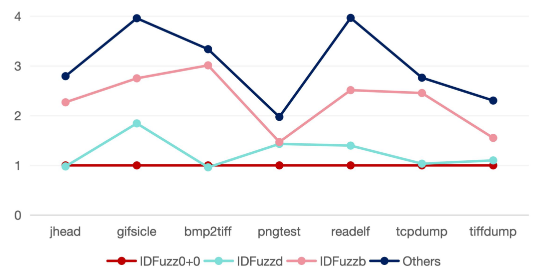
Figure 3: Relationship between *Ratio* and TTE.

3 targets (harfbuzz, libarchive, and vorbis) that can only be reproduced by fuzzers equipped with IDFUZZ.

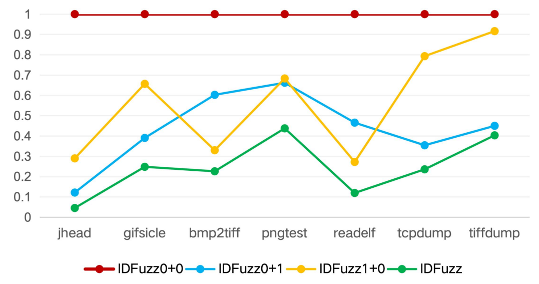
The results of AFL-IDFUZZ compared with other fuzzers that utilize optimized input generation are shown in Table 2. AFL-IDFUZZ significantly outperforms Angora, SymCC, and WAFLGo, achieving speedup ratios of >2.35x, >27.78x, and >5.27x, respectively. For targets where AFL-IDFUZZ performs worse than other fuzzers, *e.g.*, `libssh` and `openssl`$_x$, we found that the short reproduction times for these targets were also the main reason: other fuzzers reproduced these targets before IDFUZZ's model began functioning. We observed that the SymCC hybrid fuzzer outperformed other fuzzers only on the `json` program, which has fewer than 14k lines of code, but performed poorly on other large programs due to the inherent scalability limitations of concolic execution. Notably, AFL-IDFUZZ even outperforms WAFLGo, which is based on AFLGo and adopts multiple directed strategies. Our analysis indicates that this is because the mutation masking technique employed by WAFLGo is inefficient when handling long inputs, and its heavy instrumentation significantly reduces fuzzing throughput. However, this does not diminish WAFLGo's contributions, as it has unique strengths in testing commits and handling cases where the directed fuzzing target (code changed by the commit) is not the crash point [55].

Table 3: Dataset for ablation study. N$_{cond.}$, N$_{testcases}$, and Len$_{avg.}$ denote the number of conditional statements, the number of test cases, and the average byte length of test cases.

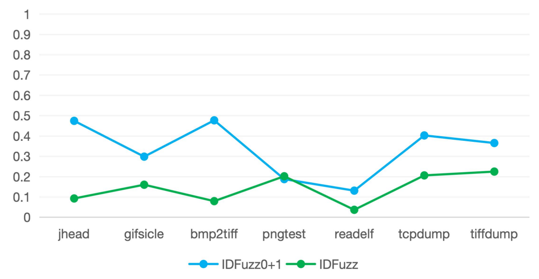
Program	Format	N$_{cond.}$	N$_{testcases}$	Len$_{avg.}$
jhead	JPEG	3	4334	482.29
gifsicle	GIF	4	10512	365.94
bmp2tiff	BMP	4	6000	403.90
pngtest	PNG	3	8565	403.56
readelf	ELF	8	7517	448.97
tcpdump	PCAP	5	9611	581.73
tiffdump	TIFF	4	9499	564.99



(a) Relative attempts required for *Others*, IDFUZZ$_b$, and IDFUZZ$_d$ compared with IDFUZZ$_{0+0}$.



(b) Relative attempts required for different IDFUZZ versions compared with IDFUZZ$_{0+0}$.



(c) Error filtering rates of IDFUZZ$_{0+1}$ and IDFUZZ.

Figure 4: Ablation study results.

The results above indicate that IDFUZZ can significantly accelerate existing directed fuzzers in triggering known vulnerabilities and outperforms other optimized mutation strategies in directed fuzzing tasks.

5.2 Ablation Study

IDFUZZ mainly employs three novel techniques: branching encoding, adaptive dataset generation, and gradient filtering. We conducted ablation experiments on these three techniques to investigate their roles in IDFUZZ.

Experimental Setup. For a deeper insight into how IDFUZZ works, we use the average number of attempts required to successfully mutate the critical field (referred to as "*attempts required*") as the evaluation metric in the following evaluations. Given the diversity of specific mutation operations, it is hard to measure the required number of mutations. Therefore, we define an "attempt" as *selecting a position within the seed to mutate*, a shared step of all mutation operations.

Though FTS is a popular benchmark used to evaluate directed fuzzers, we found that over half of the targets could be reached within 10 minutes, where IDFUZZ's model was trained only once or did not play a role at all. In order to test IDFUZZ's capabilities more comprehensively, we chose targets that are harder to reach. We chose 7 real-world programs that handle different file formats and were evaluated in prior works [14, 50, 63] as the evaluation dataset, listed in Table 3.

We first used AFL [60] to fuzz each program for 24 hours and then randomly selected conditional statements that required more than an hour to bypass. All selected targets are listed in Table 8. We determined the ground-truth critical fields by referring to the file format standards and testing. We then set these conditional statements as targets and used AFLGo [6], equipped with different versions of IDFUZZ, to conduct directed fuzzing. When targets were reached, we recorded the seed queues and the NN model.

Seven mutation strategies were included: IDFUZZ without adaptive dataset generation, represented as IDFUZZ$_{0+1}$, where the seed queue was simply used as the model training set; IDFUZZ without gradient filtering, denoted as IDFUZZ$_{1+0}$, where gradient filtering was omitted and the ranking of the first byte within the ground truth critical field was used as the average attempt; IDFUZZ without both, represented as IDFUZZ$_{0+0}$; the complete version of IDFUZZ, represented as IDFUZZ; and the AFL-based strategy employed by other directed fuzzers (random mutation), represented as *Others*. Since branching encoding cannot be independently removed, we evaluate the effectiveness of branch encoding by replacing the model outputs in IDFUZZ$_{0+0}$ with the outputs generated by two alternative encoding methods. The first encoding, denoted as IDFUZZ$_b$, uses a tensor converted from the input's corresponding coverage bitmap (provided by the AFL framework [60]) as the model output. This encoding has been adopted by neural program smoothing [49] to model the relationship between test inputs and code coverage. We identify the output dimension corresponding to the target edge and compute its gradient with respect to the input to determine the critical bytes necessary for reaching the target edge. The second encoding method, denoted as IDFUZZ$_d$, employs the input's corresponding distance value (as defined by AFLGo [6]) as the model output. Our aim here is to directly identify critical bytes that reduce this distance metric by computing the gradient of the distance value with respect to the input.

We computed *half of the seed length* as the "attempts required" for the AFL-based strategy. We ran AFLGo-IDFUZZ, collected the gradients produced by the NN model, and computed *the average gradient ranking of bytes in the critical field* as the "attempts required" for IDFUZZ$_*$.

IDFUZZ's gradient filtering method may erroneously filter out bytes from critical fields in some cases. Therefore, we calculated the error ratio of incorrect filtering. We adopted a strict calculation approach: if any byte within the ground truth critical field is filtered out, it is considered incorrect.

Experimental Results. The relative attempts required compared with IDFUZZ$_{0+0}$ are shown in Figure 4a and 4b. The error filtering rates of IDFUZZ$_{0+1}$ and IDFUZZ are shown in Figure 4c.

First, by comparing the attempts required for *Others* and IDFUZZ$_{0+0}$ in Figure 4a, we discovered that through branching encoding, the average number of attempts needed to locate critical fields is significantly reduced by 67%. Furthermore, by comparing the results of IDFUZZ$_{0+0}$ and IDFUZZ$_b$, we found that branching encoding also significantly outperforms using the coverage bitmap as the model output, reducing the attempts needed by 53%. This is because using the coverage bitmap as the model output results in excessively high dimensionality and substantial noise, which yields a trivial model. Moreover, IDFuzz$_{0+0}$ reduces attempts by 16% compared to IDFuzz$_d$. However, their performance is comparable on `jhead`, `bmp2tiff`, `tcpdump`, and `tiffdump`. This is because branching encoding does not offer a clear advantage over distance-based encoding in model fitting complexity. Its strength lies in scenarios involving complex control flow, where the same distance may correspond to different constraints, or when second-level experience is required. In such cases, branching encoding can generate more accurate and informative model outputs. For the programs mentioned above, the control flow is relatively simple, and few unexplored targets require second-level experience, resulting in similar performance across both methods. Nonetheless, it is important to note that IDFuzz$_d$ does not support gradient filtering, which, as we demonstrate later, plays a critical role.

Next, by comparing the attempts required for IDFUZZ$_{0+0}$ and IDFUZZ$_{1+0}$, we found that adaptive dataset generation improved the performance of IDFUZZ across all tested programs, reducing the attempts required by 44% on average. However, this improvement is not stable across different programs. Adaptive dataset generation brought notable enhancements in `jhead`, `bmp2tiff`, an `readelf`. This can be attributed to `jhead` and `bmp2tiff` generating a limited number of seeds (around 300), leading to model underfitting. Conversely, while the `readelf` program produced a substantial number of seeds (around 10,000), the majority of these seeds covered irrelevant code (i.e., covered few dom-BBs). As a result, the model's fitting performance saw significant improvement once we used adaptive dataset generation. Meanwhile, we observed that the fuzzing seed queues of certain programs, such as `tcpdump` and `tiffdump`, already constituted high-quality training sets. Consequently, adaptive dataset generation showed minimal improvement on these two programs.

We then study how gradient filtering affects IDFUZZ's performance. On the one hand, according to the attempts required for IDFUZZ$_{0+0}$ and IDFUZZ$_{0+1}$, we found that gradient filtering led to an average 56% reduction in the required attempts of all correctly filtered test cases. Notably, unlike the case of adaptive dataset generation, the magnitude of this improvement was relatively stable across different programs. On the other hand, we also observed that IDFUZZ$_{0+1}$ displayed a high rate of incorrect filtrations, peaking at 47.68% (as shown in Figure 4c). Test cases that are incorrectly filtered only have the opportunity to mutate the critical fields during the AFL_{havoc} phase we mentioned in §3.4.2, which is not effective. In summary, the results demonstrate the effectiveness of our gradient filtering and the importance of low error filtering rates.

Finally, by comparing the results of IDFUZZ$_{0+1}$, IDFUZZ$_{1+0}$, and IDFUZZ, we observed that adaptive dataset generation and gradient filtering worked well together in reducing the attempts required, while a high-quality training set could significantly lower the error filtering rate to 14.28% on average. Therefore, the techniques are complementary and both contribute to IDFUZZ's high efficiency, reducing the attempts required to locate critical fields by 75.47% and 91.86% compared with IDFUZZ$_{0+0}$ and *Others*, respectively.

We summarize the results of the ablation study in Table 4. We also include four metrics: false positive rate, false negative rate, precision, and recall, which are used to further evaluate gradient filtering. These four metrics are based on the binary classification task achieved by gradient filtering, which determines whether each input byte is critical or not. Among these results, we observe relatively low Precision values, primarily due to the fixed selection of 20 critical bytes identified by gradient filtering, whereas the ground-truth critical bytes typically number no more than 4. Consequently, the Precision is inherently limited to below 20% (4/20). However, we consider this low Precision acceptable, as it still represents an over 10× improvement compared to other AFL-based directed fuzzers, based on the average input length observed in our experiments. Furthermore, gradient filtering exhibits high Recall (*i.e.*, low false negative rate), and the integration of adaptive dataset generation effectively enhances this metric (*i.e.*, reduces the false negative rate). This is because adaptive dataset generation substantially improves the quality of the training data, allowing the trained model to compute more accurate gradients and thereby yielding higher gradient ranks for the critical bytes. Conversely, adaptive dataset generation has only a minor impact on the false positive rate, due to the substantial imbalance between the numbers of positive and negative samples for critical bytes.

Table 4: A summary of ablation study results. "EFR", "FPR", and "FNR" denote the error filtering rate, false positive rate, and false negative rate. "B", "A", and "G" refer to branching encoding, adaptive dataset generation, and gradient filtering.

	Attempts	**EFR**	**FPR**	**FNR**	**Precision**	**Recall**
Random	301.37%	-	-	-	-	-
B	100.00%	-	-	-	-	-
B+A	56.30%	-	-	-	-	-
B+G	43.53%	33.36%	3.94%	24.14%	11.64%	75.86%
B+A+G	24.53%	14.28%	3.85%	9.52%	13.65%	90.48%

Table 5: CPU seconds and proportions for training data generation, model training, and gradient analysis in a 24-CPU-hour fuzzing period. “Total Trainings” refers to the total number of NN trainings within 24 CPU hours.

Program	Generation	Training	Analysis	Total Trainings
jhead	18 (0.02%)	4381 (5.07%)	11 (0.01%)	19.8
gifsicle	35 (0.04%)	6040 (6.99%)	37 (0.04%)	18.2
bmp2tiff	17 (0.02%)	4597 (5.32%)	18 (0.02%)	15.8
pngtest	91 (0.11%)	1355 (1.57%)	5 (0.01%)	6.5
readelf	137 (0.16%)	5562 (6.44%)	325 (0.38%)	15.1
tcpdump	39 (0.05%)	6800 (7.87%)	21 (0.02%)	17.9
tiffdump	23 (0.03%)	4478 (5.18%)	29 (0.03%)	13.7
Avg.	51 (0.06%)	4745 (5.49%)	64 (0.07%)	15.3

5.3 Runtime Overhead

To evaluate the runtime overhead of IDFuzz, we recorded the time costs for the key components of IDFuzz while conducting experiments on the programs listed in Table 3. Specifically, we recorded the total CPU time spent on training data generation, model training, and gradient analysis during 24 CPU hours of fuzzing using AFLGo-IDFuzz. We did not select the FTS benchmark for evaluation because a significant portion of the FTS targets can be reproduced before the NN model takes effect. In such cases, the calculation of IDFuzz’s 24-CPU-hour fuzzing overhead is meaningless.

The results are presented in Table 5. Training data generation, model training, and gradient analysis account for only 0.06%, 5.49%, and 0.07% of the total runtime overhead, respectively. Note that all results represent the total overhead of the NN model within 24 CPU hours, including all retraining sessions, rather than the cost of a single training session. These results further demonstrate the efficiency of IDFuzz.

5.4 Case Studies

To further elucidate IDFuzz’s capabilities to guide input mutation, we conducted case studies on its performance in locating specific critical fields on real-world programs.

Based on prior works [7, 13, 17, 21, 50, 58] and our observations, we broadly categorized fields into two types: *fixed-offset fields*, which have a fixed offset, and *variable-offset* fields, which have variable offsets across different files.

Fixed-offset fields such as magic bytes can be easily and accurately identified by IDFuzz. However, other approaches such as correlation analysis [5, 12, 44] are also efficient in recognizing such fields. Thus, we selected more challenging cases, variable-offset fields, to highlight IDFuzz’s advantages.

According to [7] and our observations, the offsets of variable-offset fields can be typically determined by two elements: *direction fields*, which store the information of other fields’ offsets, and *markers*, which contain constant values signifying the beginning of other fields.

Fields Determined by Direction Fields. In `readelf`, we selected a conditional statement whose corresponding critical field was the `sh_size` field in the string table section of the ELF file. The related code is shown in Listing 2. The calculation formula for the critical field’s offset is $\texttt{e\_shoff} + \texttt{e\_shstrndx} \times \texttt{e\_shentsize} + 20$. In our training set, all samples were 32-bit ELF files with `e_shentsize` equal to 40. Therefore, the position of the critical field can be seen as determined by the two fixed-position direction fields `e_shoff` and `e_shstrndx` through linear calculation.

```
section = section_headers + elf_header.e_shstrndx;

if (section->sh_size != 0)
{
...
}
```

Listing 2: readelf.c of binutils-2.28.

IDFuzz performed well in identifying this field, with an average of 6.23 attempts required and a 7.70% error filtering rate. We found that the average gradient rankings for both direction fields were also high, with `e_shoff` at 26.25 and `e_shstrndx` at 2.08. The NN model discerned the importance of these two fields in relation to the conditional statement, which suggests that IDFuzz was able to comprehensively grasp the various factors affecting critical fields. Given an average seed length of 495.01, AFL-based mutation strategies would require an average of 247.51 attempts.

Fields Determined by Markers. The motivating example in §2 is another case of variable-offset field determined by a marker. IDFuzz required an average of 6.49 attempts with a 16.03% erroneous filtration rate, considering an average of 244.18 attempts for AFL-based mutation strategies.

5.5 New Vulnerabilities Discovered

We also evaluated the capability of IDFuzz in assisting existing directed fuzzers to detect new vulnerabilities. We selected popular projects that have been tested in prior fuzzing works [32, 63], referred to their existing CVEs or bug issues, and specified the original trigger points of these vulnerabilities in the latest versions of the projects as target sites. We aimed to discover incomplete fixes or new bugs sharing root causes with known issues. Since AFLGo is the only directed fuzzer capable of building all the target projects, we equipped AFLGo with IDFuzz (AFLGo-IDFuzz) for this experiment.

Overall, we have detected 6 new vulnerabilities with a time budget of 24 hours. We have 4 CVE IDs assigned so far including 1 high-severity vulnerability and 1 incomplete fix for a high-severity vulnerability. We show the results in Table 6. We repeated the experiments 5 times, and AFLGo-IDFuzz consistently reproduced all 6 vulnerabilities and the incomplete fix (33 successes out of 35 trials), while raw AFLGo could only successfully reproduce one (ID#1, 5 successes out of 35 trials) under the same settings and time budget.

6 Discussion

6.1 Limitations

Extracting Call Graphs Statically. IDFUZZ utilizes Andersen's points-to analysis to handle indirect calls [4]. This can sometimes lead to an incomplete inferred call graph, potentially resulting in missing segments in the dom-BB chain extracted by IDFUZZ. When exploring the missing parts of the extracted dom-BB chain, IDFuzz cannot provide effective mutation guidance and has to rely on random mutations to reach them. The other extracted parts are not affected. In fact, incompleteness in call graph inference is an open challenge faced by static analysis [31, 32]. To address this, we plan to explore dynamic tracing or AI-based techniques such as CALLEE [62] to complement call graphs in future work.

Predicting Values of Critical Fields. IDFUZZ effectively identifies the offsets and lengths of critical fields, but it still needs to blindly guess the proper values of critical fields. We believe that the capabilities of neural networks have the potential to further predict proper values (or value ranges) for critical fields. For instance, we could leverage the existing values of critical fields and the signs of gradients across multiple samples to narrow the value ranges. However, this necessitates accurate and effective signs of gradients, which poses a challenge for the current design (explained in §3.2). This is an important direction for our future work.

Guiding to Reach Any Unexplored Code. As introduced in §2.3, IDFUZZ can predict how to cover an unexplored dom-edge by observing the coverage behavior of its sibling edges. We refer to this as second-level experience. However, a necessary condition for leveraging second-level experience is that the corresponding conditional statement must contain at least three branches. If the conditional statement has only two branches, with just one sibling edge besides the dom-edge, IDFUZZ observes only a single coverage outcome during training. Due to this lack of variation, the model's gradients cannot guide how to switch between branches and target the unexplored dom-edge. We found that most relevant conditional statements (55% in our experiments) only have two branches, preventing IDFUZZ from extracting second-level experience from them. In fact, guiding fuzzing to reach unexplored code has always been a core challenge [5, 26]. To the best of our knowledge, no technique so far can guarantee effective guidance to reach unexplored code in all cases. As a lightweight approach, IDFUZZ can extract second-level experience from a considerable number of relevant conditional statements. For other conditional statements, we can effectively extract first-level experience to improve the efficiency of directed fuzzing.

6.2 Hyperparameter Selection

We employed several hyperparameters in the design of IDFUZZ. Specifically, in the gradient filtering technique described in §3.4, we selected 20 as the "critical threshold" (Algorithm 2, line 13) for identifying critical bytes. This choice represents a trade-off between recall and false positive rate. Additionally, during model training outlined in §3.3, we introduced hyperparameters θ, n, and R to determine the timing of model updates, which further influences overhead and the overall performance of IDFUZZ. A sensitivity analysis regarding these hyperparameters is provided in Appendix B.

Table 6: New vulnerabilities and incomplete fixes detected.

Project	ID	Vuln. Type	Status	CVE ID
tcpreplay	#1	NULL Pointer Dereference	patched	CVE-2023-43279
	#2	Infinite Loop	patched	CVE-2024-22654
gifsicle	#3	Floating Point Exception	patched	CVE-2023-46009 (High Severity)
yasm	#4	Memory Leak	patched	
	#5	NULL Pointer Dereference	patched	CVE-2024-22653
w3m	#6	OOB Read	reported	
	#7	OOB Write	patched	Incomplete fix for CVE-2022-38223 (High Severity)

7 Related Work

Neural Program Smoothing. To improve the efficiency of coverage-guided fuzzing, She et al. proposed neural program smoothing via NEUZZ [49]. NEUZZ constructs a machine learning model to capture the relationship between seeds and code coverage. When mutating a seed, NEUZZ computes the model gradients and designates seed bytes with the highest gradient values as "hot bytes", which are assumed with more potential to achieve higher coverage through mutation.

To address the data sparsity and lack of diversity problem, MTFuzz [48] further introduces *approach-sensitive edge coverage* and *context-sensitive edge coverage* as model outputs to construct a multi-task NN (MTNN) model, which learns important features shared across related tasks from limited training data in the fuzzing seed queue.

PreFuzz [53] refines NEUZZ and MTFuzz by proposing a resource-efficient edge selection mechanism for gradient computation, thereby enhancing diversity in edge exploration.

Neural program smoothing is a novel and intelligent technique that applies machine learning to fuzzing, yet it is not perfect. A recent study [38] pointed out that a potential major flaw associated with it is the difficulty in training an effective model for the entire program, given the inherent complexity of the program and the limited training data available. IDFUZZ adopts a gradient-guided input mutation approach inspired by neural program smoothing. However, our approach is fundamentally different due to its directed fuzzing purpose: IDFUZZ's model distills only the knowledge relevant to reaching the target code, rather than considering the entire program. This simplified machine learning task significantly reduces the burden of training effective models and underpins the efficiency of IDFUZZ. We also devised novel techniques to serve this purpose, achieving a 89.28% reduction in ineffective mutations compared with a neural program smoothing-based baseline (IDFUZZ$_b$), as demonstrated in §5.2.

Machine Learning for Directed Fuzzing. The application of machine learning to directed fuzzing has emerged as a prominent research direction. A notable prior work is DeepGo [30], proposed by Lin et al. in 2024. They observed that existing DGFs lack foresight into unexplored execution paths, making it difficult to reach target code when such paths involve complex constraints. To overcome this limitation, DeepGo leverages reinforcement learning to predict an optimal path to the target and introduces a corresponding action group to guide execution towards that path.

While IDFUZZ also employs machine learning, it targets a fundamentally different phase of directed fuzzing. It adopts supervised learning to model the relationship between input bytes and dom-BB coverage, and utilizes model gradients to identify critical input fields for mutation. We consider our approach orthogonal to DeepGo.

8 Conclusion

In this work, we present IDFUZZ, an intelligent input mutation solution that leverages a neural network model to learn from past inputs and effectively guide input mutation towards the target code. IDFUZZ adopts new techniques in model construction, model training, and gradient analysis. Our evaluation demonstrated that IDFUZZ boosts the performance of existing directed fuzzers by 2.48x in speed for reproducing target vulnerabilities on FTS and reduces fruitless mutations by 91.86%, with only 6% increase in overhead.

9 Ethics Considerations

For responsible vulnerability disclosure, we reported each vulnerability to the developers immediately upon discovery, following the security policies listed in the corresponding GitHub repositories. We applied for CVE identifiers after the vulnerabilities were patched. We did not disclose the bugs to any other third-parties apart from the developers and CVE maintainers.

For the three vulnerabilities that do not have CVEs (#4, #6, and #7):

- For #4, we reported it, and the developers patched it. However, since it was a relatively minor memory leak, we did not apply for a CVE.
- For #6, we have not yet received a meaningful response from the developers, but we will continue to follow up.
- For #7, we reported it, and the developers have patched it. However, since it was an incomplete fix of an existing CVE, we did not apply for a new one.

10 Open Science

Our artifacts are available at https://doi.org/10.5281/zenodo.13753907, including the source code and configurations of IDFUZZ and the log files of the experimental results. We will also maintain our code at https://github.com/vul337/IDFuzz.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd for their constructive comments and efforts to help improve this paper. This work is supported by the National Natural Science Foundation of China under grant U24A20337 and 62132009, and Joint Research Center for System Security, Tsinghua University (Institute for Network Sciences and Cyberspace) - Science City (Guangzhou) Digital Technology Group Co., Ltd.

References

[1] Google Fuzzer Test Suite, 2024. https://github.com/google/fuzzer-test-suite.

[2] LLVM DominatorTreeWrapperPass class, 2024. https://llvm.org/doxygen/classllvm_1_1DominatorTreeWrapperPass.html.

[3] Ahmad Alwosheel, Sander van Cranenburgh, and Caspar G Chorus. Is your dataset big enough? sample size requirements when using artificial neural networks for discrete choice analysis. *Journal of choice modelling*, 28:167–182, 2018.

[4] Lars Ole Andersen. Program analysis and specialization for the c programming language. 1994.

[5] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *NDSS*, volume 19, pages 1–15, 2019.

[6] 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, 2017.

[7] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 317–329, 2007.

[8] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[9] 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.

[10] Sadullah Canakci, Nikolay Matyunin, Kalman Graffi, Ajay Joshi, and Manuel Egele. Targetfuzz: Using darts to guide directed greybox fuzzers. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, pages 561–573, 2022.

[11] 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, 2018.

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

[13] Weidong Cui, Marcus Peinado, Karl Chen, Helen J Wang, and Luis Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 391–402, 2008.

[14] Zhengjie Du, Yuekang Li, Yang Liu, and Bing Mao. WindRanger: a directed greybox fuzzer driven by deviation basic blocks. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2440–2451, 2022.

[15] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd*, volume 96, pages 226–231, 1996.

[16] Rosa L Figueroa, Qing Zeng-Treitler, Sasikiran Kandula, and Long H Ngo. Predicting sample size required for classification performance. *BMC medical informatics and decision making*, 12:1–10, 2012.

[17] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. WEIZZ: Automatic grey-box fuzzing for structured binary formats. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, pages 1–13, 2020.

[18] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[19] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. Beacon: Directed grey-box fuzzing with provable path pruning. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 36–50. IEEE, 2022.

[20] Heqing Huang, Anshunkang Zhou, Mathias Payer, and Charles Zhang. Everything is good for something: Counterexample-guided directed fuzzing via likely invariant inference. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 142–142. IEEE Computer Society, 2024.

[21] Vivek Jain, Sanjay Rawat, Cristiano Giuffrida, and Herbert Bos. TIFF: Using input type inference to improve fuzzing. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 505–517, 2018.

[22] Tae Eun Kim, Jaeseung Choi, Kihong Heo, and Sang Kil Cha. {DAFL}: Directed grey-box fuzzing guided by data dependency. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4931–4948, 2023.

[23] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[24] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[25] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. Constraint-guided directed greybox fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3559–3576, 2021.

[26] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, pages 475–485, 2018.

[27] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):121–141, 1979.

[28] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):1–13, 2018.

[29] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jiaguang Sun. Pata: Fuzzing with path aware taint analysis. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1–17. IEEE, 2022.

[30] Peihong Lin, Pengfei Wang, Xu Zhou, Wei Xie, Gen Zhang, and Kai Lu. Deepgo: Predictive directed greybox fuzzing. In *NDSS*, 2024.

[31] Kangjie Lu and Hong Hu. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1867–1881, 2019.

[32] Changhua Luo, Wei Meng, and Penghui Li. Selectfuzz: Efficient directed fuzzing with selective path exploration. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2693–2707. IEEE, 2023.

[33] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2019.

[34] Patrick E McKnight and Julius Najab. Mann-whitney u test. *The Corsini encyclopedia of psychology*, pages 1–1, 2010.

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

[36] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices. In *NDSS*, 2018.

[37] Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. Binary-level Directed Fuzzing for Use-After-Free Vulnerabilities. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 47–62, 2020.

[38] Maria-Irina Nicolae, Max Eisele, and Andreas Zeller. Revisiting neural program smoothing for fuzzing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 133–145, 2023.

[39] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[40] Mathias Payer. The fuzzing hype-train: How random testing triggers thousands of crashes. *IEEE Security & Privacy*, 17(1):78–82, 2019.

[41] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[42] Jiaqi Peng, Feng Li, Bingchang Liu, Lili Xu, Binghong Liu, Kai Chen, and Wei Huo. 1dvul: Discovering 1-day vulnerabilities through binary patches. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 605–616. IEEE, 2019.

[43] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with {SymCC}: Don't interpret, compile! In *29th USENIX Security Symposium (USENIX Security 20)*, pages 181–198, 2020.

[44] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[45] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[46] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted feedback fuzzing for OS kernels. In *26th USENIX security symposium (USENIX Security 17)*, pages 167–182, 2017.

[47] Abhishek Shah, Dongdong She, Samanway Sadhu, Krish Singal, Peter Coffman, and Suman Jana. MC2: Rigorous and Efficient Directed Greybox Fuzzing. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2595–2609, 2022.

[48] Dongdong She, Rahul Krishna, Lu Yan, Suman Jana, and Baishakhi Ray. 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*, pages 737–749, 2020.

[49] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program smoothing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 803–817. IEEE, 2019.

[50] Ji Shi, Zhun Wang, Zhiyao Feng, Yang Lan, Shisong Qin, Wei You, Wei Zou, Mathias Payer, and Chao Zhang. AIFORE: Smart Fuzzing Based on Automatic Input

Format Reverse Engineering. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4967–4984, 2023.

[51] Mozhan Soltani, Annibale Panichella, and Arie Van Deursen. Search-based crash reproduction and its impact on debugging. *IEEE Transactions on Software Engineering*, 46(12):1294–1317, 2018.

[52] Mingzhe Wang, Jie Liang, Chijin Zhou, Zhiyong Wu, Jingzhou Fu, Zhuo Su, Qing Liao, Bin Gu, Bodong Wu, and Yu Jiang. Data coverage for guided fuzzing. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 2511–2526, 2024.

[53] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yuqun Zhang, Guowei Yang, Huixin Ma, Sen Nie, Shi Wu, Heming Cui, and Lingming Zhang. Evaluating and improving neural program-smoothing-based fuzzing. In *Proceedings of the 44th International Conference on Software Engineering*, pages 847–858, 2022.

[54] 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.

[55] Yi Xiang, Xuhong Zhang, Peiyu Liu, Shouling Ji, Hong Liang, Jiacheng Xu, and Wenhai Wang. Critical code guided directed greybox fuzzing for commits. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 2459–2474, 2024.

[56] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 818–834. IEEE, 2019.

[57] Jifeng Xuan, Xiaoyuan Xie, and Martin Monperrus. Crash reproduction via test case mutation: Let existing test cases help. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, pages 910–913, 2015.

[58] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *2019 IEEE symposium on security and privacy (SP)*, pages 769–786. IEEE, 2019.

[59] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 745–761, 2018.

[60] Michał Zalewski. American Fuzzy Lop, 2024. https://lcamtuf.coredump.cx/afl/.

[61] Yuyue Zhao, Yangyang Li, Tengfei Yang, and Haiyong Xie. Suzzer: A vulnerability-guided fuzzer based on deep learning. In *International Conference on Information Security and Cryptology*, pages 134–153. Springer, 2020.

[62] Wenyu Zhu, Zhiyao Feng, Zihan Zhang, Jianjun Chen, Zhijian Ou, Min Yang, and Chao Zhang. Callee: Recovering call graphs for binaries with transfer and contrastive learning. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2357–2374. IEEE, 2023.

[63] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. FuzzGuard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *29th USENIX security symposium (USENIX security 20)*, pages 2255–2269, 2020.

Appendices

A Partial Experimental Configuration

Table 7: Google Fuzzer Test Suite targets.

Program	Vuln. Code
boringssl	asn1_lib.c:459
c-ares	ares_create_query.c:196
guetzli	output_image.cc:398
harfbuzz	hb-buffer.cc:419
json	fuzzer-parse_json.cpp:50
lcms	cmsintrp.c:642
libarchive	archive_read_support_format_warc.c:537
libssh	messages.c:1003
libxml2	parser.c:10666
openssl-1.0.1f	t1_lib.c:2586
openssl-1.0.2d	target.cc:145
pcre	pcre2_match.c:1426
re2	nfa.cc:532
vorbis	codebook.c:407
woff	woff2_dec.cc:500

Table 8: Evaluation targets for ablation study.

Program	Target Code
jhead	jpgfile.c:126 jpgfile.c:168 jpgfile.c:189
gifsicle	gifread.c:428 gifread.c:575 gifread.c:787 gifread.c:819
bmp2tiff	bmp2tiff.c:291 bmp2tiff.c:317 bmp2tiff.c:403 bmp2tiff.c:412
pngtest	pngread.c:116 pngrutil.c:140 pngrutil.c:960
readelf	readelf.c:4077 readelf.c:5771 readelf.c:5866 readelf.c:5892 readelf.c:6421 readelf.c:6426 readelf.c:6451 readelf.c:16622
tcpdump	sf-pcap.c:213 sf-pcap.c:228 sf-pcap.c:365 sf-pcap.c:500 sf-pcap.c:530
tiffdump	tiffdump.c:214 tiffdump.c:224 tiffdump.c:282 tiffdump.c:319

B Hyperparameter Tuning

B.1 Critical Threshold

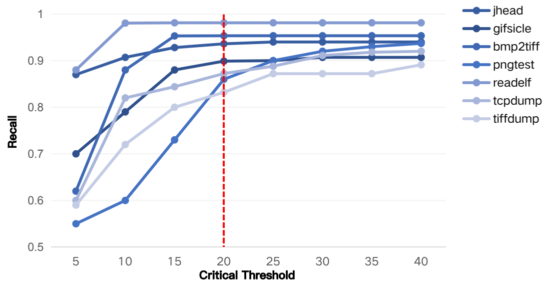
In the gradient filtering technique introduced in §3.4, we use a critical threshold to identify critical bytes. The choice of this threshold affects both the recall and the false positive rate of identifying critical bytes. Due to the significant imbalance between the number of critical and non-critical bytes in the input, the false positive rate increases almost linearly with the critical threshold. Therefore, we focus on analyzing the relationship between the critical threshold and recall, and select the threshold accordingly.

The impact of the critical threshold on recall across different programs and on the average recall is shown in Figure 5a and 5b, respectively. We observe that the marginal effect is small after the critical threshold reaches 20. Therefore, we select 20 as a trade-off between recall and false positive rate. Additionally, we note that the recall does not converge to 100% at the end, which is due to the presence of erroneous filtering.

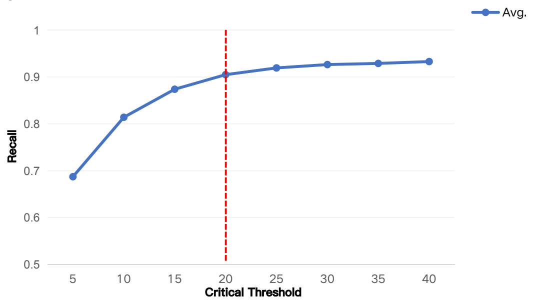
B.2 Hyperparameters for Model Update Frequency

We use a set of hyperparameters (θ, *n*, *R*) to determine the model update frequency, which in turn affects both the overhead and performance of IDFuzz. Specifically, θ denotes the time interval for discovering new dom-BBs, *n* denotes the number of newly discovered dom-BBs, and *R* represents the proportion of seed inputs with new structures. To perform a sensitivity analysis on these hyperparameters, we vary them in AFLGo-IDFuzz and evaluate the results on the FTS benchmark. All other settings follow those described in §5.1, except that each experiment is repeated only 5 times due to time constraints. For each configuration, we measure the average number of model trainings, the proportion of additional overhead, and the speedup relative to AFLGo. In each experiment, we fix two hyperparameters and vary the third, resulting in a total of 7 configurations, as shown in Table 9 (the configuration (60, 1, 5%) is listed 3 times to facilitate a clear comparison).

Our experimental results indicate that decreasing these hyperparameters, which leads to increased model update frequency, introduces additional overhead and, counterintuitively,



(a) Impact of critical threshold on recall for different programs.



(b) Impact of critical threshold on average recall.

Figure 5: Sensitivity analysis of the critical threshold.

Table 9: Sensitivity analysis of hyperparameters for model update frequency.

	(θ, *n*, *R*)	Total Trainings	Runtime Overhead	*Ratio*
θ	(30, 1, 5%)	20.5	8.01%	1.53x
	(60, 1, 5%)	15.3	5.62%	1.87x
	(90, 1, 5%)	12.9	4.60%	1.70x
n	(60, 0, 5%)	57.9	24.13%	1.25x
	(60, 1, 5%)	15.3	5.62%	1.87x
	(60, 2, 5%)	15.2	5.77%	1.88x
R	(60, 1, 2.5%)	26.8	9.34%	1.50x
	(60, 1, 5%)	15.3	5.62%	1.87x
	(60, 1, 7.5%)	11.8	4.77%	1.81x

uniformly reduces the performance of IDFUZZ. This is because, as introduced in §3.4.2, during model training, IDFUZZ executes AFL_{havoc} and skips other mutations. Excessively frequent training substantially reduces the effective time for model-guided fuzzing. Specifically, decreasing θ sharply increases the model update frequency in the early stages of fuzzing, affecting targets that are relatively easy to reproduce. Reducing n to zero significantly reduces the criterion for model updates in later fuzzing stages, causing a substantial increase in update frequency and thus impacting targets that are harder to reproduce. In contrast, R influences nearly all targets.

Furthermore, we observe that increasing different hyperparameters has varying effects. Increasing θ delays the initial training of the model and reduces the frequency of model updates in the early stages of fuzzing. For targets that are relatively easy to reproduce, an excessively large θ can result in insufficient time for the model to provide guidance, thereby degrading the performance of IDFUZZ. In contrast, increasing n from 1 to 2 yields no significant change in the results. This is because n and R jointly influence model updates via a logical `OR` condition. Once $n > 0$, R becomes the dominant factor. Increasing R reduces the frequency of model updates in the later stages of fuzzing, but has only a minor impact on the performance of IDFUZZ.

Ultimately, we find that the configurations (60, 1, 5%) and (60, 2, 5%) yield similarly favorable results across all metrics, making them good choices.