IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received July 19, 2019, accepted July 30, 2019, date of publication August 23, 2019, date of current version September 3, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2936235

LearnAFL: Greybox Fuzzing With Knowledge Enhancement

TAI YUE, YONG TANG, BO YU, PENGFEI WANG, AND ENZE WANG
College of Computer, National University of Defense Technology, Changsha 410073, China

Corresponding author: Bo Yu (yubo0615@nudt.edu.cn)

This work was supported in part by the Program for New Century Excellent Talents in University, in part by the National High-Level Personnel for Defense Technology Program under Grant 2017-JCJQ-ZQ-013, in part by the Hunan Province Science Foundation under Grant 2017RS3045, in part by the Natural Science Foundation of Hunan Province in China under Grant 2019JJ50729, and in part by the National Science Foundation China under Grant 61472437, Grant 61902412, and Grant 61902416.

ABSTRACT Mutation-based greybox fuzzing is a highly effective and widely used technique to find bugs in software. Provided initial seeds, fuzzers continuously generate test cases to test the software by mutating a seed input. However, the majority of them are ''invalid'' because the mutation may destroy the format of the seeds. In this paper, we present a knowledge-learn evolutionary fuzzer based on AFL, which is called LearnAFL. LearnAFL does not require any prior knowledge of the application or input format. Based on our format generation theory, LearnAFL can learn partial format knowledge of some paths by analyzing the test cases that exercise the paths. Then LearnAFL uses these format information to mutate the seeds, which is efficient to explore deeper paths and reduce the test cases exercising high-frequency paths than AFL. We compared LearnAFL with AFL and some other state-of-the-art fuzzers on ten real-world programs. The result showed that LearnAFL could reach branch coverage 120% and 110% of that of AFL and FairFuzz, respectively. LearnAFL also found 8 unknown vulnerabilities in GNU Binutils, Libpng and Gif2png, all of which have been reported to the vendors. Besides, we compared the format information learned from the initial seed of an ELF file with a format standard of ELF files. The result showed that LearnAFL learns about 64% part of the file format without any prior knowledge.

INDEX TERMS Input format learning, deep path fuzzing, greybox fuzzing, vulnerability detection.

I. INTRODUCTION

Fuzzing is a modern and practical approach to software vulnerability detection. As an automated software testing technique, fuzzing was first developed by Barton Miller to test the robustness of UNIX applications in 1989 [1], [2]. Since then, fuzzing has been developed rapidly and widely used in software testing and vulnerability detection, and exposed a large number of vulnerabilities in many programs [2]. The key idea behind fuzzing is to generate and feed the target program with plenty of test cases that are hopeful of triggering software errors [2].

There are three main types of fuzzing techniques in use: blackbox fuzzing, whitebox fuzzing, and greybox fuzzing [3], [4]. Blackbox fuzzing a technique of testing without having any knowledge of the internal working of the application [5]. Conversely, whitebox fuzzing is based on an analysis of the internal structure of the target program [3], [6]. Greybox fuzzing tests the program with limited knowledge of the structure of an application [5]. Nowadays, greybox fuzzing technique combined with machine learning, symbolic execution, dynamic taint analysis, static analysis, and other technologies is becoming a research hotspot in the field of fuzzing [7]–[10]. Compared with the whitebox fuzzing, greybox fuzzing has a shallower understanding of the program. However, since most of greybox fuzzers do not need the source code of the target program, the scope of greybox fuzzing will be more extensive than that of whitebox fuzzing. Compared with the blackbox fuzzing, greybox fuzzing takes some time for program analysis. Nevertheless, the lightweight greybox test can better understand the information of the target program and make the test more efficient. Today most vulnerabilities were exposed by lightweight fuzzers [11].

As a classic and efficient mutation-based greybox fuzzer, AFL (American fuzzy lop) [12] is preferred by many

The associate editor coordinating the review of this article and approving it for publication was Fan Zhang.

researchers. By using lightweight (binary) instrumentation to determine a unique identifier for the path that is exercised by an input, AFL classifies the exercised paths of the target program and uses *evolutionary algorithms* to screen out seeds with higher coverage, and then mutates them to generate new tests [11]. By its lightweight program analysis approach, AFL can generate and execute a large number of tests in a short period. Moreover, evolutionary algorithms ensure that AFL can select tests as seeds that are easier to explore new paths. As an effective tool used in file application fuzzing, AFL had found hundreds of high-impact vulnerabilities discoveries [13]. Thus, enhancing AFL is a significant work.

However, AFL sometimes shows insufficient performance in some respects. First, the ability to perceive the input formats of the target program of AFL is not reliable. Although AFL can perceive some interesting characters of the seeds by doing the deterministic mutation strategies, it cannot learn the complicated formats of files inputted to programs, such as `pdf` and `png`. Besides, in doing the random mutation strategies, AFL may destroy the formats of seed and mutate the crucial parts of the seed, which are crucial to satisfy the conditions necessary to exercise this path. As a result, many test cases are generated to exercise the high-frequency paths [11]. Only a few test cases exercise some low-frequency paths. However, test cases executing these low-frequency paths are more interesting than others [11]. In addition, some results have shown that the efficiency of deterministic mutation strategies in AFL is lower than that of random mutation strategies and the deterministic mutation strategies take more energy of AFL than random mutation strategies, which decreases the efficiency of AFL [14], [15].

To solve the above problems, we present LearnAFL, a *knowledge-learn evolutionary fuzzer*. LearnAFL is built on a top of AFL without deterministic mutation strategies (i.e., FidgetyAFL [14]). As the random mutation strategies are more efficient than the deterministic strategies to improve the test coverage, the ability to explore new paths of LearnAFL is stronger than AFL. More importantly, based on the *equivalence-classes-based format generation theory*, LearnAFL can learn the format of the seed files and keep format attribute of seeds unchanged in the random mutation by *format-based path transition model*. In detail, similar to FairFuzz [15], LearnAFL can identity certain parts of an input, which are crucial to satisfy the path constraints. LearnAFL avoids mutating them to reduce the frequency of executing invalid paths, which increases the probability of exploring deeper paths.

However, different from FairFuzz [15], LearnAFL achieves this goal without implementing deterministic mutation strategies. LearnAFL classifies test cases into different sets based on the paths they exercise. Then LearnAFL learns the format features of the test cases in each set and mutates the seeds according to these format features. This mechanism can effectively reduce the number of invalid test cases and generate more test cases that can cover deep paths and trigger in-depth bugs. We provide some measures to evaluate the mutation efficiency in exploring deep paths when comparing techniques. The experiment shows that LearnAFL could learn about 64% part of the data struct of the target file format without any prior knowledge. Besides, LearnAFL can find the vulnerabilities which are hard to be found by other state-of-the-art fuzzers and generate more test cases to cover them than other tools. More importantly, compared to FairFuzz, LearnAFL does not rely on the implementation of deterministic mutation strategies, which could improve the test efficiency. We perform our evaluation of LearnAFL with other state-of-the-art fuzzers (e.g., FairFuzz, AFLFast, FidgetyAFL [11], [14], [15]) on ten real-world software, nine of them with the latest version. Our evaluation shows that LearnAFL could reach branch coverage 120% and 110% of that of AFL and FairFuzz, respectively. Specifically, our paper makes the following *contributions*:

- **Equivalence-classes-based Format Generation Theory**. Based on the mapping theory, we regard the target program as a map and prove that test cases can be divided into several subsets according to the paths exercised by test cases. Test cases in each subset satisfy the same path constraints and are consistent in the format. Therefore, we could deduce the format of a path if the number of test cases is sufficient. Particularly, this approach allows LearnAFL to identity the crucial part of seeds without relying on deterministic strategies.
- **Format-based Path Transition Model**. Based on our equivalence-classes-based format generation theory, we point out that the essence of the path transition in gerybox fuzzing is to modify the format features of the seeds in the mutation. By destroying some format features, we may generate test cases to exercise high-frequency paths. In contrast, keeping the magic bytes and other crucial parts unchanged in the mutation is possible for us to explore deeper paths than AFL. The experiment shows that LearnAFL finds some in-depth vulnerabilities which are not found by some other tools.
- **Enhanced Expression of Magic Bytes Generation Algorithm**. We define the *enhanced expression of magic bytes* as the target knowledge we aim to learn, which is more helpful to assist mutation than simple magic bytes. Based on the longest common substring searching algorithm, we propose our algorithm to generate the enhanced expression of magic bytes. We evaluate the efficiency of our algorithm. The result shows that LearnAFL can identify 64% of the target files' data structure.
- **Tool and Evaluation**. We implement our approach on a top of AFL, named LearnAFL. We evaluated LearnAFL on ten real-world software against the other five AFL-type fuzzers. The results have shown that LearnAFL reaches the 120% and 110% coverage reached by AFL and FairFuzz, respectively. Furthermore, LearnAFL has found eight previously unknown vulnerabilities. Similarly, we published LearnAFL as a fork of AFL. To foster further research in the area, we open sourcing

LearnAFL on GitHub:https://github.com/MoonLight-SteinsGate/LearnAFL.

II. BACKGROUND

In this section, we primarily introduce the operating mechanism of AFL and the *mapping theory* for programs. Notably, in Subsection II-A, we elaborate on the advantages of AFL in its efficiency, and then point out the drawbacks of AFL in terms of mutation and power schedule. In Subsection II-B, we focus on how to get a set of equivalence classes of the test cases according to *mapping theory*, which is the theory basement of our *format-based path transition model*.

A. AMERICAN FUZZY LOP (AFL)

AFL(American fuzzy lop), as a state-of-the-art greybox fuzzer, has exposed serious vulnerabilities in many important software programs, which is also the basis of many greybox fuzzers (e.g., AFLGo, CollAFL, PTFuzz [16]–[18]).

AFL uses lightweight instrumentation to capture basic block transitions and gain coverage information [19]. According to coverage information, AFL is able to determine a unique identifier for the path that is exercised by an input, and then employs genetic algorithms to automatically discover test cases that likely trigger new internal states in the targeted program. After that, these test cases will be added to the queue of seeds.

Algorithm 1 provides a general overview of the process and is illustrated in the following by means of AFL's implementation [19]. First, we must provide AFL with initial seeds to start it. If AFL is provided with initial seeds S, AFL will add the initial seeds to the queue of seeds Q. The seeds are chosen in a continuous loop until a timeout is reached or the fuzzing is aborted. For the seed s_i, AFL classifies it as a favorite if it is the fastest and smallest input for any of the control-flow edges it exercises [11]. If s_i was a non-favorite seed, AFL ignores it and chooses the next seed in queue Q to fuzz; otherwise, AFL transfers to the fuzzing stage and s_i is going to be mutated to generate new test cases to test program. During the fuzzing stage, for each execution of test case t generated by s_i, if t exercised a new path which was never exercised, the test case t is regarded as an interesting seed and added to seeds queue Q. Particularly, t is going to be added to the set of crash T_c when it triggers bugs in the program. After the fuzzing stage, AFL sequentially chooses the next seeds in queue Q according to the order of being added to Q. The above is an overview of the AFL mechanism, and then we focus on its *mutation strategies*.

Mutation strategies. There are two categories of mutation strategies in AFL: *deterministic strategies* and *random strategies* [20]. The deterministic strategies include: *bitflip, arithmetic, interest, dictionary*, which mutate the seeds without any randomness. Especially in *bitflip*, the first strategy to be implemented, AFL flips the seeds in the bit-level, from 1 bit to 32 bits. During this strategy, AFL makes a heuristic judgment on the file format of test cases by observing that whether the test case exercise a new path if it is generated by flipping in one byte, which is able to help AFL to distinguish the *token* that is also called "*magic byes*" in the seeds. If AFL regarded some bytes as the token, AFL adds them in the *effector map* and skip to mutate the seeds on these positions in subsequent deterministic strategies. Therefore, it is capable of AFL to perceive partial formats of inputs. However, since the deterministic strategies don't have any randomness, a seed is only mutated by AFL with deterministic strategies when it is the first time for the seed to be fuzzed. After that, AFL effectuates the random strategies which include *havoc* and *splice*. In the *havoc* stage, AFL would mutate the seed by

Algorithm 1 AFL's Mechanism

```
Require: Initial Seeds Set S
  T_c = ∅
  if S = ∅ then
    return
  end if
  Q = S
  i = 0
  repeat
    if i > |Q| then
      i = 0
    end if
    Choose s_i from Q
    if IsFavored(s_i) = 0 then
      i = i + 1
    else if WasFuzzed(s_i) or PassDeterministic then
      n = AssignEnergy(s_i)
      for j from 1 to n do
        t = Mutate(s_i, RANDOM STRATEGIES)
        res = Execute(t)
        if res = CRASH then
          add t to T_c
        else if IsInteresting(res) then
          add t to Q
        end if
      end for
      i = i + 1
    else
      n = AssignEnergy(s_i)
      for j from 1 to n do
        t = Mutate(s_i, ALLSTRATEGIES)
        res = Execute(t)
        if res = CRASH then
          add t to T_c
        else if IsInteresting(res) then
          add t to Q
        end if
      end for
      i = i + 1
    end if
  until timeout reached or abort-signal
Ensure: T_c
```

randomly choosing a sequence of mutation operators from the deterministic strategies and apply them to random locations in the seed file. As a result, a new test case is generated, which is significantly different from the seed. At last, the *splice* strategy allows AFL to randomly choose another seed from the seeds queue Q and recombine it with the current seed.

Although AFL can detect some magic bytes, there are some shortcomings. In the random mutation strategies, the magic bytes of the seeds may be mutated, which means it is significantly possible for AFL to destroy the format of the seeds. Even though destroying the seeds' format may help AFL to find a new path, it reduces the number of test cases conforming to the format, which makes AFL less effective in discovering deeper bugs. Besides, taking most energy on deterministic strategies decreases the efficiency of AFL. We will discuss these issues in detail in Section III.

B. MAPPING THEORY

In mathematics, the term *mapping*, sometimes shortened to *map*, refers to the relationship between elements of two sets, which is usually used to mean a function in many branches of mathematics. In category theory, *mapping* is often used as a synonym for morphism or arrow, thus for something more general than a function [21].

Formally, a mapping F from a set X to a set Y is defined by a set G of ordered pairs (x, y) such that $x \in X$, $y \in Y$, and every element of X is the first component of precisely one ordered pair in G [22]. Specific to fuzzing, notice that software can also be regarded as a mapping. Assuming that there are no random functions in the software, inputted a test case, the software is going to execute a specific path.

More formally, given a program M and a set of tests $T = \{s_1, s_2, s_3, s_4, s_5, \ldots\}$, in which s_i denotes a test case for some $i \in \mathbb{N}$, we input test case s_i to the program M, and then s_i will exercise a certain path j. After that we can get a mapping $F : T \to P, P = \{1, 2, 3, \ldots, N\}$, $P = \{1, 2, 3, \ldots, N\}$ for some $N \in \mathbb{N}$ in which $j \in P$ is the identifier of a path of the program M.

$\forall s_i \in T$, s_i can be represented as a sequence of characters from the set of 2^8 ASCII characters (i.e. $s_i = (x_{i_1}, x_{i_2}, x_{i_3}, x_{i_4}, \ldots, x_{i_l}, Null, Null, Null, Null, \ldots)$, where l is the byte length of s_i), for the test case is stored in bytes [23]. Further more, $\forall n \in \mathbb{N}$, the number of test cases with lengths equal to n bytes is limited, up to 256^n. According to the theorem that the union set of countably infinite countable sets is still a countable set, we can conclude that T is a countable set [25]. That is, denoting T by $\{s_1, s_2, s_3, s_4, \ldots\}$ is rational.

While $\forall j \in P$, j stands for a path in M, which is restrained by a set of conditions C_j in M. If s_i exercises the path j in M, we can denote this as

$$F(s_i) = j \tag{1}$$

We can get the relationship between the input and the execution path of the program by (1). However, from $|T| > |M|$, we could infer that F is a surjection, not a bijection. In Section III, we state how to get a bijection F' based on *mapping theory* and deduce the formats of the test cases that exercise the same paths.

III. FORMAT-BASED PATH TRANSITION MODEL

In Subsection II-B we introduce the *mapping theory*, formularize the relationship between test cases and paths of the program as (1). In this section, we discuss the path transition in fuzzing and propose the *equivalence-classes-based format generation theory*. Furthermore, we elaborate on how to explore new paths by mutation based on the format of test cases, which is formulated as *format-based path transition model*.

A. EQUIVALENCE-CLASSES-BASED FORMAT GENERATION THEORY

In Subsection II-B, we denote that the path j is exercised by test case s_i as (1), which maps the program M to the mapping F. Notice that, F is a surjection, not a bijection. In order to get a bijection, we divide T into several equivalence classes T_i by the exercised path of tests,

$$T_i = \{s_{i_j} | F(s_{i_j}) = i\} \tag{2}$$

Further more, $\forall s_{i_{j_1}}, s_{i_{j_2}} \in T_i$, as $F(s_{i_{j_1}}) = F(s_{i_{j_2}})$, $s_{i_{j_1}}$ and $s_{i_{j_2}}$ are both restrained by a set of conditions C_j (i.e., they both satisfy the same pattern in format). For instance, their first four characters are "*%PDF*".

According to the equivalence classes of the test cases, we can get a bijection $F' : T' \to P, T' = \{T_1, T_2, T_3, T_4, \ldots, T_N\}$,

$$F'(T_i) = i \tag{3}$$

In order to distinguish between F and F', we define F as the *simple mapping* of M and F' as the *bijection* of M. Notice that, given the restrained conditions C_i of the path p_i in program M, it is able to get the input format of T_i by solving restrictions C_i, which is the principle of symbol execution. On the contrary, given T_i (i.e., all the test cases exercising path i), it is also possible to get the format by observing the regular pattern of the elements in T_i.

We illustrate this conclusion using the simple program in Listing 1 which takes as input a 4-character string and crashes for the input "bad!".

```
1 void test1(char *buf) {
2   int n = 0;
3   if(buf[0] == ''b'')
4     if(buf[1] == ''a'')
5       if(buf[2] == ''d'')
6         if(buf[3] == ''!'')
7           raise(SIGSEGV);
8 }
```

LISTING 1. Motivating example to illustrate that it is possible to get the format by observing the test cases in T_i.

In this program, there are five execution paths, which is listed in Table 1. If we had already known the determine statements in the program, it is easy for us to infer the input

TABLE 1. Paths and input format of code in listing 1.

Paths	Input format
1	****
2	b***
3	ba**
4	bad*
5	bad!

formats of each path. Supposing, we are fuzzing the program by AFL; we only have some test cases that execute each path. Can we get the input format based on these test cases? It is not necessary if test cases are few. For instance, ''beda'' and ''bed1'' both exercise the path 2. However, we may infer that the input format of path 2 is ''bed*'', which is an incorrect conclusion. In order to get the correct format, we need more test cases for reference. Fortunately, mutation-based greybox fuzzing can bring us many test cases to observe. Overall, the more test cases, the closer we can get to the correct format.

Based on that, we proposed the *equivalence-classes-based format generation theory*.

(1) Assuming that there are no random functions in the program M, and the state of M is consistent before each testing, M could be represented to the *simple mapping* $F : T \to P$, which T is the set of test cases and P is the set of identifiers of paths.

(2) According to 1, we can deduce the bijection $F' : T' \to P$ by dividing T into several equivalence classes T_i by the exercised path of tests.

(3) According to 2, we get the sets T_i of test cases corresponding to different paths i. Furthermore, $\forall s_{i_j} \in T_i$, as $F(s_{i_j}) = i$, test case s_{i_j} must be restrained by a set of conditions C_i, which determines whether a test case exercises the path i.

(4) According to 3, if there are sufficient test cases of the set T_i, it is possible for us to deduce the format of test cases exercising the path i.

B. FORMAT-BASED PATH TRANSITION MODEL

AFLFast proposed the *transition probability* of mutation-based fuzzing and modeled it in a *Markov chain* [11]. For mutation-based greybox fuzzing, the transition probability p_{ij} is defined as the probability to generate an input t that exercises path j by randomly mutating the seeds s that exercises path i [11]. Formally, we can denote this process as

$$\begin{cases} F(s) = i \\ s \xrightarrow{mutation} t \\ F(t) = j \end{cases} \tag{4}$$

Notice that, the path transition is a process that generates a test case of the equivalence class T_j from the seed of the equivalence class T_i. More substantially, the essence of the path transition is to modify the format of the seed s in the mutation. The new test case t generated by seed s satisfies the constraint conditions C_j of path j rather than C_i of path i, which determines test case t to exercise path j. In other words, if we can get the format of the path i by collecting and observing a large number of test cases exercising i, we could solve the constraint C_i of i. Furthermore, inverting the constraint and modifying the format of test cases, we may explore a new path.

In order to illustrate this conclusion, we walk through a more complex code presented in Listing 2.

```
9  void test(char *buf) {
10   int n = 0;
11   if(buf[0] == ''b'')
12     if(buf[1] == ''a'')
13       if(buf[2] == ''d'')
14         if(buf[3] == ''!'')
15           if(buf[4] >= ''0'' && buf[4] <= ''8'')
16             if(buf[5] == buf[4]+1)
17               raise(SIGSEGV);
18 }
```

LISTING 2. A more complex code than that in listing 1.

There are seven paths in the code snippet, among which only the deepest path could trigger a crash. Observing the code, it is easy to get the format of each path by solving the constraints, which is listed in Table 2.

TABLE 2. Paths and input format of code in listing 2.

Paths	Input format
1	****
2	b***
3	ba**
4	bad*
5	bad!
6	bad!x(0≤x≤9)
7	bad!xy(0≤x≤9, y=x+1)

Provided that a seed s = ''bast'' exercises the path 3, $s \in T_3$, if we mutate the magic bytes ''a'' of the seed s and generate a test case t = ''best'', according to Table 2, the test case t will exercise the path 2, which is a good illustration of the fact that the path transition is essentially the equivalence classes transition.

However, this is a path transition from a deeper path to a high-frequency path, which is easy to reach. Informally, we call this way of path transition *decrease-transition*. In fact, for mutate-based greybox fuzzing, exploring the deeper paths is more challenging than decrease-transition, which is called *increase-transition* informally. The main reason is that most of the time, we do not know the determine statements so that we could not deduce the format of the deeper path. Moreover, even if we generate some test cases exercising the deeper paths, that does not mean we can deduce the exact format.

Figure 1 shows the CFG of the code in Listing 2, which is used for illustrating our conclusion. For path 6, we may only be able to deduce its format as ''bad!''. If we modify the first four characters, we will generate test cases exercising the high-frequency path, which is not helpful for us to explore deeper paths.

Notice that, though ''bad!'' is not the exact format of path 6, it is the public format of path $5-7$. Therefore, aiming to explore deep paths based on some exercised paths, we need to keep the learned format unchanged.

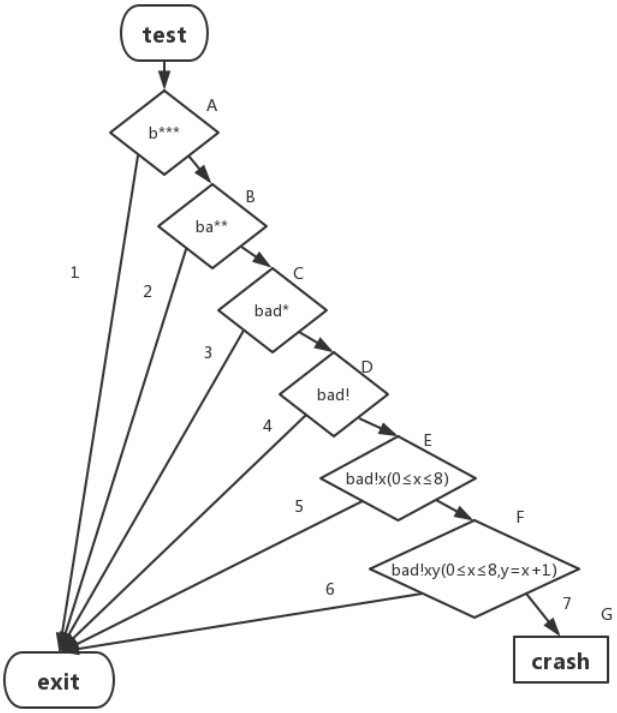

FIGURE 1. CFG of the code in listing 2.

Supposing we had explored the paths $1-5$, and now we are fuzzing the seed "bad!t" which exercises path 5, our goal is to pass the determined statement of block E. If we implement the initial indeterministic strategies of AFL, it couldn't ensure that the valid format "bad!" of the seed "bad!t" is not changed in the mutation, so that it may generate a lot of test cases which isn't able to reach block F. This means that AFL generates a large number of test cases that perform high-frequency paths in the mutation, which makes AFL not efficient in exploring the deeper paths. In contrast, if we had learned the valid format "bad!" of the seed "bad!t", we only need to keep the valid format unchanged and mutate the other positions in the seed. It will be easier for us to find the deeper paths 6, 7 than implementing the initial indeterministic strategies of AFL. That is, by keeping the formats of exercised paths unchanged, we are able to decrease the number of test cases exercising high-frequency paths and focus on the deeper paths, which means the transition probability is increased.

Above all, we proposed the format-based path transition model. Given some seeds and format of the paths exercised by these seeds, we can exercise the paths which are no deeper than the original paths by destroying the formats of seeds, which is decrease-transition. In contrast, we can keep the formats unchanged in mutation, which is helpful to generate test cases with good quality and reach increase-transition. Based on this theory and model, we implement LearnAFL. It can learn the formats of paths we have exercised and help us to explore deeper paths.

IV. DESIGN AND IMPLEMENTATION

In this section, we introduce the architecture of LearnAFL and detail the algorithm of learning formats.

A. ARCHITECTURE OF LearnAFL

Similar to AFLFast, LearnAFL is also based on the AFL 2.52b, especially the mutation strategies, power schedules, and execution engine. However, LearnAFL only implements the *haovc* mutation strategy. Besides, we have added a python script to learn the formats of paths and an assist mechanism to use the formats to mutate seeds. The main idea of LearnAFL is to collect the information of each fuzzing in the fuzzing process, divide the test cases into several sets according to the exercising paths, deal with the test case set and learn formats of each path, and then use the formats as an auxiliary in mutation. Figure 2 provides an overview of its main components.

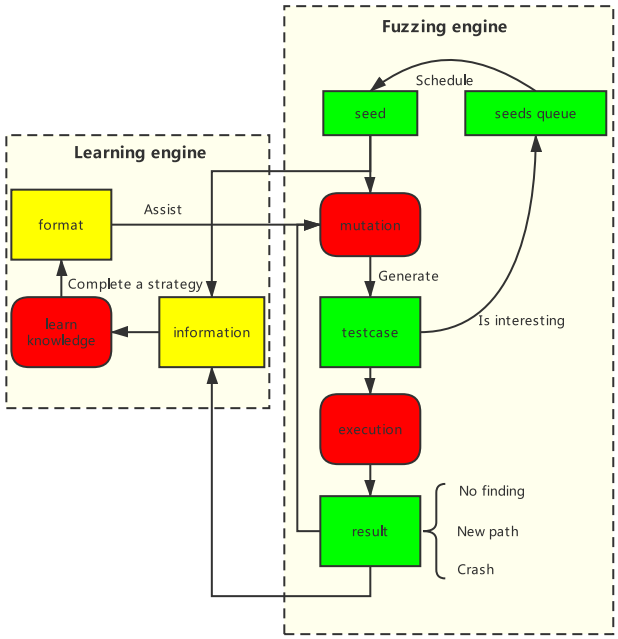

FIGURE 2. Overview of LearnAFL.

There are two engines in LearnAFL, the *fuzzing engine* (shown on the right) and the *learning engine* (shown on the left). The fuzzing engine follows that of AFL. When the fuzzing engine is working, LearnAFL selects a seed s from seeds queue and prepares to fuzz it. Before doing fuzzing on s, LearnAFL checks whether there has been the format of the path exercised by the seed s. In order to facilitate the storage and modifying of the format knowledge, we store the format knowledge in a text file format. Informally, we call this file *format model file*. Especially, if LearnAFL had done fuzzing on s and learned the format, LearnAFL would read the format model file and use the format knowledge to assist LearnAFL to mutate s. Otherwise, LearnAFL would fuzz s like AFL. LearnAFL collects the information of each fuzzing, which includes seed, test case, the identifier of the path exercised by test case (i.e., *cksum* defined by AFL).

If LearnAFL fuzzes a seed for some time, LearnAFL will transfer to the learning engine. In this mode, LearnAFL calls the python script to process the information file and learns the format of the path exercised by current seed *s*. If there has been a format model file of the path, LearnAFL will read the prior model file and regulate the format according to the latest collected information. Then LearnAFL transfers to the fuzzing engine and uses the latest format knowledge to mutate the seed *s*. We detail the algorithms used in LearnAFL in Subsection IV-C.

B. DEFINITION AND EXPRESSION OF FORMAT

In this subsection, we state the definition and expression of the format knowledge.

In Section III, we had listed some examples to illustrate our model, which also contains some simple formats of seeds. Nevertheless, due to the complexity of the determine statements, the paths constraints in the real-world program are very complicated. Therefore, the formats of these paths are also very complicated (e.g., the paths in software to handle PDF). Compared to them, formats of paths like path 5 in Table 2 are simple examples. Generally speaking, because of the complexity of software in the real world, it is challenging to learn the formats of the paths in them accurately. Therefore, we propose a method to use the *enhanced expression of magic bytes* instead of the actual format of the path to assist the mutation.

The enhanced expression of magic bytes is built on the top of the regular expression of all test cases exercising the same path. It is a 2-dimensional array including the array of *regular* and the array of *position*, in which *regular* is an array of the substrings of all test cases exercising a path and *position* is an array of the positions of each substring in the test cases. Table 3 is an example to show the enhanced expression of magic bytes.

TABLE 3. The enhanced expression of magic bytes.

Test Cases
mail:badword@gmail.com mail:thankyou@qq.com mail:sorry@163.com mail:wonderful@126.com
Regular
[“mail:”, “o”, “@”, “.com”]
Position
[0, -2, -2, -1]

There are four test cases in Table 3. It is easy to deduce all substrings, which are listed in the array of regular in order. The *pos* of the position array shows the position of a substring in the regular array. If *pos* equal to -2, it means that the position of this substring is variable in all test cases (e.g., position of ‘‘@’’ in all strings). If *pos* equal to -1, that means the substring is at the end of all test cases (e.g., all strings ended with ‘‘.com’’). Otherwise, it means the fixed position of this substring in all test cases (e.g. ‘‘mail:’’ is the first four characters in all strings). However, given some test cases, there may be several expressions for these test cases, especially if the number of test cases is too scanty. For these strings in Table 3, the [‘‘mail:’’, ‘‘@’’, ‘‘.com’’] is also an array of regular of an expression. Notice that the [‘‘mail:’’, ‘‘@’’, ‘‘.com’’] is the subarray of [‘‘mail:’’, ‘‘o’’, ‘‘@’’, ‘‘.com’’]. Therefore, if there is an expression *exp*1, which includes the expression *exp*2, we use *exp*1 instead of *exp*2 as the expression of the paths.

There are several reasons for us to choose this enhanced expression as the format we learned from the test cases. First, among the determine statements of software, comparing variables to a fixed string or magic bytes (i.e., ‘‘if(buf[0]==‘‘a’’)’’) is more complicated than these determine statements to pass, such as the determine statements which are satisfied by a range of values (i.e., ‘‘if(a<10)’’). Therefore, learning the magic bytes is more helpful for the mutation to pass some determine statements than some formats. Second, learning the regular expression of a path is more accessible than learning other formats. For the determine statement ‘‘if(a<10)’’, if we want to learn the format, we need to determine the boundary value, which is complex and worthless. Though there are some determine statements more difficult for comparing magic bytes to pass. That means they are helpful for mutation-based fuzzing. However, learning such a format is tough. For instance, some software of reading files or accepting network packets will check the length of the content of files or packets and execute different paths according to the result of comparing the character at a specific location with the length. Unless we have some prior knowledge of files of the target software, we often have difficulties learning the format that satisfies this constraint even by manual learning.

C. ALGORITHMS IN LearnAFL

There are two algorithms we implement in LearnAFL for learning format and assisting mutation.

The first algorithm is the enhanced expression of magic bytes generation algorithm. It is based on the longest common substring searching algorithm.

In more detail, we choose the hardcode as the highest priority substrings, whose position is stationary in all test cases of a set. Then we divide the set into several subsets according to the hardcode, similar to [27]. For the strings in each subset, we select the most extended and leftmost substring as the second-highest priority substring. If there has been a format model of some test cases, we use the new strings to generate new expression to regulate the format. Figure 3 shows the whole process of generating the expression of strings in Table 3 and utilizing new strings to recorrect format.

After introducing the first algorithm, we illustrate the second algorithm, *assistant mutation algorithm*. This algorithm is mainly used to mutate the seeds with the format knowledge.

In Subsection IV-A, we point out that it is an ongoing process for us to learn the format of a seed. The more the number of test cases exercising the same path with the

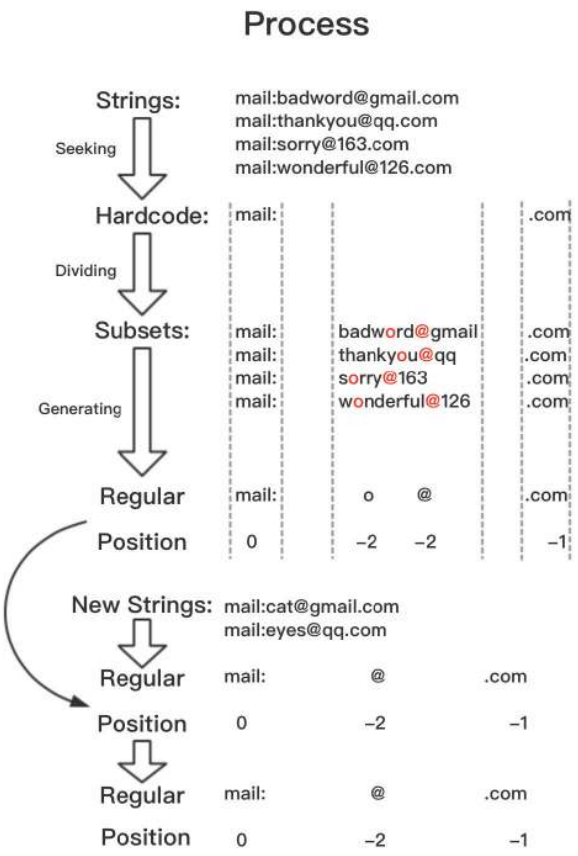

FIGURE 3. Process of generating the enhanced expression of magic bytes.

seeds is, the more precise the format knowledge we will learn. Based on these and the format-based path transition model, we implement the assistant mutation algorithm.

In detail, for fuzzing a new seed s, which means there is no format model file of the path exercised by s, LearnAFL effectuates the same mutation strategies with AFL at the initial phase. After that LearnAFL learns the format, generates the enhanced expression of the path and turns to the second stage. At the second stage, LearnAFL only mutates the characters in the enhanced expression of the seed and generate some test cases to fuzz (e.g. "mail:", "@" or ".com" in the example). It will be efficient for us to save the energy, generate exacter enhanced expression and explore paths, according to format-based path transition model. If a test case does not exercise the same path with the seed, we will infer that the characters which we changed in mutation affect the result of a determined statement. It means the characters must be components of the format. Until all substrings of the enhanced expression are changed, we will determine which substring is crucial to this format and obtain a precise format model. Then LearnAFL turns to the third stage, mutates the characters whose positions are not in the array of positions of enhanced expression during implementing the deterministic strategies (e.g. arithmetic) and utilizes the format to regulate the test cases generated by the indeterministic strategies (e.g. havoc). As a result, test cases generated in mutation with the assist of format model all satisfy the enhanced expression of the path, which brings more possibilities to LearnAFL to explore the deeper paths. The algorithm is detailed in Algorithm 2.

Algorithm 2 Assistant Mutation Algorithm

Require: The Seeds Queue Q
  Choose s_i from Q
  if s_i has a format model **then**
    $reg, pos, use\_model = \text{ReadModelFile}(s_i)$
  else
    $use\_model = 0$
  end if
  $n = \text{AssignEnergy}(s_i)$
  for j from 1 to n **do**
    if $use\_model = 0$ **then**
      $t = \text{Mutate}(s_i)$
    else if $use\_model = 1$ **then**
      Choose mutation position in pos
      $t = \text{Mutate}(s_i)$
    else if $use\_model = 2$ **then**
      Keep position in pos unchanged
      $t = \text{Mutate}(s_i)$
    end if
    $res = \text{Execute}(t)$
    $\text{RecordInformation}(s_i, t, res)$
    if $res = \text{CRASH}$ **then**
      add t to T_c
    else if $\text{IsInteresting}(res)$ **then**
      add t to Q
    end if
    if $j\%1024 = 0$ **then**
      $\text{LearnFormat}(s_i)$
      $reg, pos, use\_model = \text{ReadModelFile}(s_i)$
    end if
  end for

V. EVALUATION

We evaluated LearnAFL on ten different real-world utility programs and libs, nine of which are the latest version [28]–[33] (since the latest version of `libjpeg` is compiled of CMake, we choose the 1.5.3 version to test [31]). We selected these from those favored for evaluation by some AFL-type fuzzers. We ran all the evaluation without dictionaries to level out the playing field. The configuration of all experiments is listed in Table 4. For each case we seeded the fuzzing run with the inputs provided from the `testcases` directories of AFL; for `PNG` we used only `not_kitty.png`, which is the same as that of [15].

We compare five popular versions of AFL against LearnAFL, listed as

(1) AFL is a classic file-type coverage-based greybox fuzzer.
(2) FidgetyAFL [14] is AFL running without deterministic mutation strategies.

TABLE 4. The configuration of experiments.

Subjects / Version	File Size	Function os Program
nm -C @@ / Binutils-v2.322	48kb	Decode low-level symbol names into user-level names
objdump -d @@ / Binutils-v2.32	116kb	Display assembler contents of executable sections
readelf -a @@ / Binutils-v2.32	532kb	Display information about the contents of ELF format files
size @@ / Binutils-v2.32	16kb	Displays the sizes of sections inside binary files
c++filt @@ / Binutils-v2.32	8kb	Filter to demangle encoded C++ symbols
djpeg @@ / libjpeg-turbo-1.5.3	28kb	Decompress a JPEG file to an image file
gif2png @@ / gif2png-2.5.13	28kb	Convert files from the GIF to PNG
pdfimages @@ / xpdf-4.01.01	8kb	Extract images from PDF files
tcpdump -nr @@ / tcpdump-4.9.2	72kb	Print out a description of the contents of packets on a network interface
readpng @@ / libpng-1.6.37	4kb	Read PNG files

(3) AFLFast is an outstanding fuzzer implementing the monotonous power schedule without the adaptive mechanism.
(4) AFLFast.new [15] is AFLFast running without deterministic mutation strategies.
(5) FairFuzz is a state-of-the-art greybox fuzzer implementing targeted mutation strategies [15].

We ran our experiments on a 64-bit machine with 40 cores (2.8 GHz Intel R Xeon R E5- 2680 v2), 64GB RAM, and Ubuntu 16.04 as server OS. According to [34], we ran each experiment 5 times for 24 hours, which was longer than that in AFLFast. Fuzzing is a random variation. By taking the average value of many experiments, we can reduce the contingency during our experiments. In addition, time is measured using *Unix time stamps*. The total hours of our experiments are over 300 CPU days.

A. RESULT OF BRANCH COVERAGE

We choose the basic block transitions coverage achieved by different techniques through time as the primary metric, which is the same as the evaluation in FairFuzz [15]. Some researchers may choose the path coverage of AFL as the main metric. However, Lemieux and Sen [15] pointed out that the basic block transitions coverage is close to the notion of branch coverage used in real-world software testing. Notably, the creator of AFL also favors branch coverage as a performance metric [15]. Besides, AFL provides the map coverage C_M as a metric. From the technical details [19], we can calculate the basic block transitions coverage C_B as

$$C_B = C_M * (2^{16} - 1) \tag{5}$$

1) RESULTS

For each subject and technique, Figure 4 plots the average branch coverage reached overall 5 runs at each time point. As Figure 4 shows, on all programs except on `pdfimages`, LearnAFL reaches the maximum branch coverage, which is the blue line shown in Figure 4. The basic block transitions coverages reached by AFL and AFLFast are the lowest among these tools. Table 5 shows the specific values of the coverage reached by each tool on each subject in detail. According to Table 5, LearnAFL achieves average branch coverage of 120% of that of AFL (average 20.06% increase). However, FairFuzz only increases 10% coverage of AFL [15]. The growth of coverage achieved by LearnAFL is 200% as that of FairFuzz.

More specifically, on `pdfimages`, the basic block transitions reached by all techniques are almost the same. On other programs, particularly on `nm`, `objdump` and `djpeg`, LearnAFL performance significantly better than some other techniques. In detail, LearnAFL reaches the basic block transitions coverages of 135.79%, 142.36% and 123.79% of these achieved by AFLFast, respectively on `nm`, `readelf`, `objdump` and `djpeg`. Besides, the gap of coverage between LearnAFL and other tools is not very large on `readpng` and `pdfimages`. One of the most important reasons is that these two programs are used to read and convert target files. That means, they do not analysis the formats of files deeply. Therefore, learning the format knowledge to fuzz these programs is not as effective as that of others. For the five programs of GNU Binutils, LearnAFL performs much better than other techniques, including FidgetyAFL and AFLFast.new. Particularly, LearnAFL reaches the average coverage of 110% of that of FidgetyAFL on these five programs. Since LearnAFL, FidgetyAFL, and AFLFast.new all runs without deterministic strategies, these results show that obtaining format knowledge to assist mutation could improve the test efficiency for these programs. In addition, on most programs (e.g., `nm`, `objdump`, `readelf`, `size`), LearnAFL performs worse than other AFL techniques in the beginning of fuzzing. The reason is that at the beginning, LearnAFL spent a certain amount of time on learning knowledge. With the gradual increase of coverage, the discovery of low-frequency paths brings more benefits to coverage growth than that of high-frequency paths. LearnAFL's advantage in exploring low-frequency paths makes it better than other tools in the later stage.

Compared to FairFuzz, in general, LearnAFL reaches higher branch coverage than FairFuzz on all programs except `pdfimages`. The average coverage reached by LearnAFL is about 110% of that reached by FairFuzz. On `tcpdump`, the basic block transitions coverage of LearnAFL is slightly higher than that of FairFuzz. However, LearnAFL and FairFuzz reach significantly higher coverage than the other four tools. Particularly, compared to AFLFast, LearnAFL and FairFuzz reach branch coverage of 155% and 153% of that of AFLFast respectively. The main reason is that

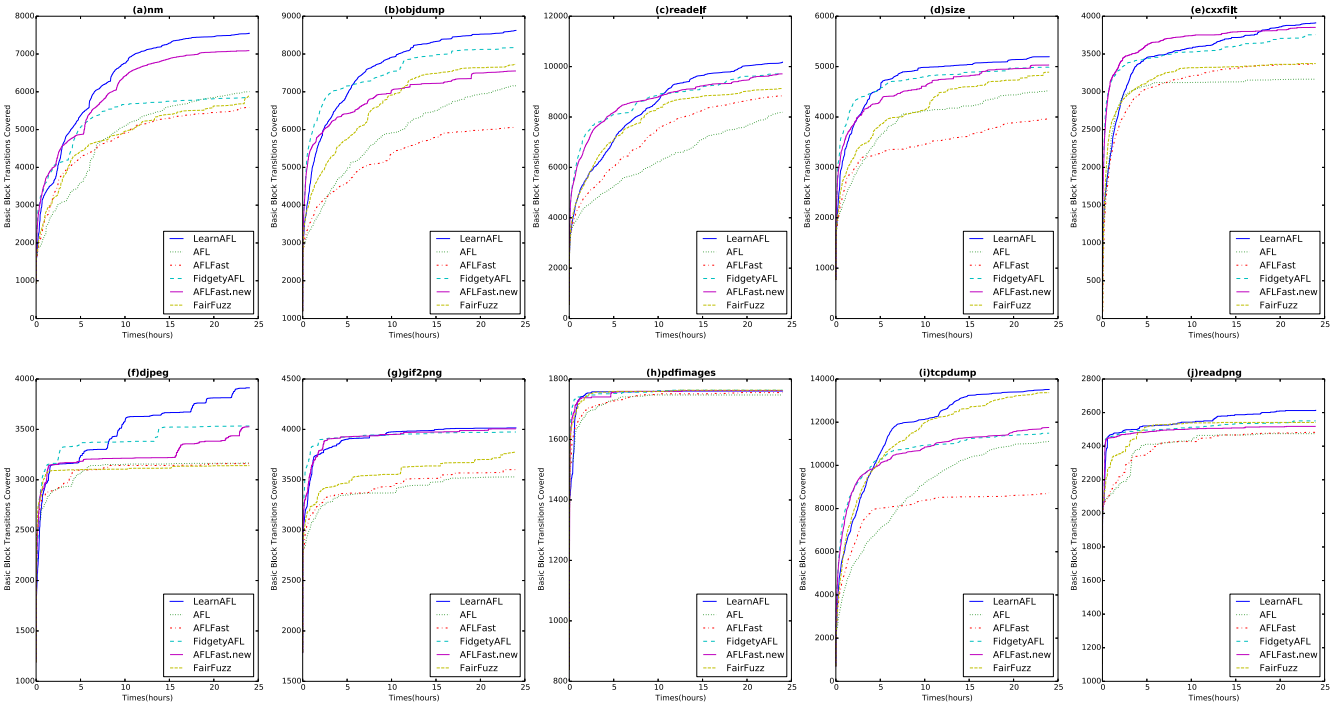

FIGURE 4. Branch coverage reached by different AFL techniques averaged over 5 runs, where the X axis represents the testing time.

TABLE 5. The average branch coverage of each fuzzer on each subject.

Subjects	Basic Block Transitions					
	AFL	AFLFast	FidgetyAFL	AFLFast.new	FairFuzz	LearnAFL
nm -C	6,004	5,585	5,850	7,091	5,894	**7,584**
objdump -d	7,159	6,058	8,170	7,551	7,726	**8,624**
readelf -a	8,187	8,839	9,712	9,712	9,133	**10,178**
size	4,521	3,978	4,990	5,033	4,890	**5,197**
cxxfilt	3,167	3,370	3,756	3,852	3,374	**3,912**
djpeg	3,167	3,160	3,534	3,526	3,140	**3,912**
gif2png	3,528	3,600	3,974	4,005	3,774	**4,015**
pdfimages	1,747	1,755	1,763	1,762	**1,763**	1,760
tcpdump -nr	11,109	8,712	11,491	11,754	13,368	**13,516**
readpng	2,475	2,482	2,551	2,518	2,541	**2,614**

`tcpdump -nr` performs an in-depth analysis of network packets. Therefore, avoiding to mutate some crucial parts of the seeds is valid to improve the coverage. Moreover, LearnAFL performs significantly better on these programs than AFL, which is mainly because LearnAFL only implements random mutation strategies and utilizes the format knowledge to assist mutation.

In particular, as suggested by [34], we do the statistical tests and use the p value to measure the performance of these fuzzers. Specifically, $p1$ is the p value yielded from the difference between the performance of LearnAFL and AFL. $p2$, $p3$, $p4$ and $p5$ are the p value yielded from the difference between the performance of LearnAFL and AFLFast, FidgetyAFL, AFLFast.new, FairFuzz, respectively. The results of p value are shown in Table 6.

From Table 6, on these programs except `pdfimages`, $p1$ is smaller than 10^{-3}, which means that the distribution of the branch coverage reached by LearnAFL and AFL is significantly different. The difference demonstrates statistical evidence for that LearnAFL can achieve higher branch coverage than that of AFL.

TABLE 6. The p in each evaluation of branch coverage.

Subjects	Basic Block Transitions				
	$p1$	$p2$	$p3$	$p4$	$p5$
nm -C	$9.8*10^{-4}$	0.0025	0.0285	0.0209	0.0119
objdump -d	$3.7*10^{-5}$	$6.6*10^{-6}$	0.0266	$4.7*10^{-4}$	0.0032
readelf -a	$6.6*10^{-5}$	0.0028	0.151	0.1508	0.0214
size	$7.5*10^{-6}$	$2.9*10^{-5}$	0.0019	0.0601	0.0182
cxxfilt	$1.8*10^{-5}$	0.0096	0.1688	0.5917	$2.7*10^{-4}$
djpeg	$2.7*10^{-8}$	$4.4*10^{-8}$	0.0608	0.0556	$2.3*10^{-8}$
gif2png	$1.4*10^{-4}$	$8.5*10^{-4}$	0.0214	0.6813	0.0266
pdfimages	0.0582	0.4468	0.5716	0.7845	0.5716
tcpdump -nr	$2.1*10^{-4}$	$1.8*10^{-4}$	0.0165	0.001	0.016
readpng	$1.5*10^{-4}$	$1.4*10^{-6}$	$4.4*10^{-4}$	0.0016	0.724

In general, LearnAFL achieves branch coverage of 120% of that of AFL over 24 hours of testing (average 20.06% increase). However, FairFuzz only increases the coverage reached by AFL about 10% [15]. This result has proved that LearnAFL can significantly improve the testing efficiency of AFL among these techniques.

B. DEEP PATHS AND VULNERABILITY DISCOVERY

In this subsection, we evaluate the ability to explore deep paths and discovery vulnerabilities of LearnAFL against other techniques. Since most of the programs we test are the latest version and the initial seed is simple, LearnAFL and other five techniques all have not found any crashes on the ten programs except on `pdfimages` and `gif2png`. The average number of unique crashes found by each technique on these two programs is listed in Table 7.

TABLE 7. The average number of unique crashes in fuzzing gif2png and pdfimages.

Fuzzer	gif2png	pdfimages
LearnAFL	11	**45**
FairFuzz	12	1
AFLFast.new	20	15
FidgetyAFL	**29**	13
AFLFast	1	3
AFL	1	4

As Table 7 shows, LearnAFL finds the most number of unique crashes on `pdfimages` among these fuzzers, which is significantly more than others. On `gif2png`, though the number of unique crashes found by LearnAFL is not the most, LearnAFL finds more crashes than AFL and AFLFast. Besides, we further analyze the crashes and find a heap-buffer-overflow vulnerability in the *writefile* function. Only LearnAFL and FidgetyAFL trigger this vulnerability. Moreover, FidgetyAFL only generates 1 test case to trigger this vulnerability over 5 runs. In contrast, LearnAFL generates 12 test cases in total. The results show that LearnAFL is useful in detecting vulnerabilities and generating more test cases to explore deep paths and bugs. Especially, compared to AFL, LearnAFL finds 10 times more unique crashes than AFL.

1) EXPLORING DEEP PATHS

We use the heap-buffer-overflow vulnerability found by LearnAFL in `gif2png` to illustrate that LearnAFL can explore deep paths which other tools are hard to find. This heap-buffer-overflow vulnerability is triggered in the *writefile* function of `gif2png.c`, which is listed in Listing 3.

In detail, the heap-buffer-overflow vulnerability occurs when `gif2png` executes the statement on lines 15 in Listing 3. The variable *s* in *writefile* function represents a data structure of the `gif` file. Only the *s->data* is a null pointer and the statement on lines 15 is executed a second time, this vulnerability is triggered. However, the condition for executing this statement is that the value of *s->GIFtype* must match the *GIFcomment*, which is preset to "0xfe". That is,

```
1  static int writefile(.....)
2  {
3    .....
4    while(lastimg ? s != NULL : s != e->next) {
5      .....
6      switch((byte)s->GIFtype) {
7      case GIFimage:
8        .....
9        break;
10
11     case GIFcomment:
12       j = (int)s->size;
13       .....
14       comment.text = (png_charp)s->data;
15       comment.text_length = strlen(comment.text);
16       png_set_text(png_ptr,info_ptr,&comment,1);
17       /*@=observertrans =statictrans@*/
18       break;
19
20     case GIFapplication:
21       .....
22       break;
23     .....
24     }
25     /*@=branchstate@*/
26     s = s->next;
27   }
28   .....
29 }
```

LISTING 3. Code of *writefile* function.

only if one test case can pass two times of type validations for *GIFcomment* in succession, it is possible for this test case to trigger this vulnerability. Therefore, triggering this heap-buffer-overflow vulnerability needs to generate well-format test cases exercising deep paths. In the term, LearnAFL performs significantly better than the other five tools, with the total number of unique crashes triggering this vulnerability listed in Table 8 over 5 runs. The result in Table 8 shows that LearnAFL is more effective in exploring deep paths and vulnerabilities than other tools.

TABLE 8. The total number of crashes triggering the heap-buffer-overflow vulnerability in fuzzing gif2png.

Fuzzer	Triggering This Crashes	Total Unique Crashes
LearnAFL	**12**	53
FairFuzz	0	61
AFLFast.new	0	98
FidgetyAFL	1	**147**
AFLFast	0	7
AFL	0	5

2) DISCOVERING VULNERABILITIES

Moreover, we use the AFL_ASAN mode to compile these programs [35], which can detect more crashes than normal mode. The initial seeds we choose are the crashes that were exposed in previous versions of these programs. We recompile all programs where we find crashes with *AddressSanitizer* and reevaluate them with the discovered crash inputs [36], [37]. AddressSanitizer can trail the stack trace and locate the bugs. This is a common way to find unique vulnerabilities in practice [35]. After inputting the crashes and observing the results, we find 8 unknown vulnerabilities in

TABLE 9. The discovered vulnerabilities.

No.	Softwares	Files	Function	Type	Status
1	Binutils-v2.32	libiberty/cp-demangle.c	d_expression_1	Heap-buffer-overflow	Patched
2	Binutils-v2.32	bfd/bfd.c	_bfd_doprnt	Heap-buffer-overflow	Acknowledged
3	Binutils-v2.32	bfd/hash.c	bfd_hash_hash	Heap-buffer-overflow	Acknowledged
4	Binutils-v2.32	binutils/objdump.c	slurp_symtab	Memory leaks	Won't Fix
5	Libpng-1.6.37	libpng16.so.16	png_malloc_warn	Memory leaks	Submitted
6	Libpng-1.6.37	libpng16.so.16	png_create_info_struct	Memory leaks	Submitted
7	Gif2png-2.5.13	gif2png	ReadImage	Memory leaks	Submitted
8	Gif2png-2.5.13	gif2png	writefile	Heap-buffer-overflow	Submitted

these programs. The analysis of these vulnerabilities is listed in Table 9.

In GNU Binutils, we have found 4 unknown vulnerabilities. Three of them are heap-buffer-overflow vulnerabilities, and the last one is a memory leak. In detail, the memory leak is triggered when the *slurp_symtab* function is called. it is due to `objdump` hitting a fatal error and calling *bucomm.c:bfd_fatal*. Besides, this problem leaks 2013265920 bytes. We have submitted all vulnerabilities to vendors. Vendors also acknowledge the other three vulnerabilities. However, the vendor of GNU Binutils refused to fix this memory leak vulnerability.

On other programs, LearnAFL also found some vulnerability. Since `gif2png` is linked to `libpng`, they are all triggered when we are testing `gif2png`. We also reported these problems to their vendors.

From these results, we could conclude that LearnAFL is efficient in triggering crashes and detecting vulnerabilities. Particularly, in some programs (e.g., `gif2png`), it finds some deep bugs more effective than the other five tools.

C. ACCURACY OF FORMAT MODEL

In previous subsections, we analyzed the ability of LearnAFL to detect vulnerabilities and compared it with other techniques. In this subsection, we focus on the ability of LearnAFL to learn formats knowledge of target programs.

We begin our evaluation with the `ELF` file type. Since LearnAFL can obtain the format features of a seed after it has done some mutation on this seed, the format knowledge of the initial seed with well-format is the closest to the standard of `ELF` among all seeds. Therefore, we choose the format file of the path which is exercised by the initial seed and compare it to the standard of `ELF`. Moreover, We analyze the format model corresponding to the initial seed when using LearnAFL to test `objdump` and show this in Figure 5. In addition, LearnAFL generated the format file after doing one turn of fuzz on the initial seed, less than 2 minutes. Notably, the initial seed is an i386 type file, whose size is 324 bytes. We compiled `objdump` to the 32-bit LSB version in the x86-64 Ubuntu 16.04 services.

According to the structures of the `ELF` headers shown in Listing 4, the first four bytes in the seed stands for the magic number of the `ELF` files, which is "\x7fELF" of the file [38]. As Figure 5 shows, LearnAFL identities this magic number of the seed and keeps it unchanged in the

\x7f	E	L	F	\x01	\x01	\x01	\x00	\x00	\x00		\x00		\x00		\x00
	\x00	\x03	\x00			\x00			\x00			\x34	\x00	\x00	\x00
\xa4	\x00	\x00	\x00	\x00	\x00	\x00	\x00				\x00	\x02	\x00	\x28	\x00
\x04	\x00	\x03	\x00	\x01	\x00	\x00	\x00	\x00	\x00	\x00	\x00		\x80	\x04	\x08
				\x80	\x00	\x00	\x00		\x00			\x00			\x00
		\x00		\x01	\x00	\x00	\x00	\x80	\x00	\x00	\x00		\x90	\x04	\x08
				\x0c								\x00			
				\x00								\x00			
s			t			\x00					t	\x00			a
t		\x00		\x00	\x00	\x00	\x00	\x00	\x00	\x00	\x00	\x00	\x00	\x00	\x00
\x00		\x00		\x00		\x00		\x00	\x00	\x00	\x00	\x00	\x00	\x00	\x00
\x00	\x00	\x00	\x00		\x00		\x00		\x00		\x00	\x0b	\x00	\x00	\x00
\x01	\x00	\x00	\x00	\x06	\x00	\x00	\x00		\x80	\x04	\x08	t	\x00	\x00	\x00
\x0c	\x00	\x00	\x00	\x00	\x00	\x00	\x00	\x00	\x00	\x00	\x00			\x00	
	\x00			\x11	\x00	\x00	\x00	\x01	\x00	\x00	\x00	\x03	\x00	\x00	\x00
\x80	\x90	\x04	\x08	\x80	\x00	\x00	\x00		\x00	\x00	\x00	\x00	\x00	\x00	\x00
\x00	\x00	\x00	\x00									\x01	\x00	\x00	\x00
\x03	\x00	\x00	\x00	\x00	\x00	\x00	\x00	\x00		\x00	\x00	\x8c	\x00	\x00	\x00
\x17	\x00	\x00	\x00	\x00	\x00	\x00	\x00	\x00	\x00	\x00	\x00		\x00		
\x00			\x00												

FIGURE 5. Part of format LearnAFL learned from the ELF file whose size is 324 bytes provided by AFL. Red part stands for the substrings whose position are not fixed.

```
#define EI_NIDENT 16
typedef struct {
        unsigned char e_ident[EI_NIDENT];
        ELF32_Half e_type;
        ELF32_Half e_machine;
        ELF32_Word e_version;
        ELF32_Addr e_entry;
        ELF32_Off e_phoff;
        ELF32_Off e_shoff;
        ELF32_Word e_flags;
        ELF32_Half e_ehsize;
        ELF32_Half e_phentsize;
        ELF32_Half e_phnum;
        ELF32_Half e_shentsize;
        ELF32_Half e_shnum;
        ELF32_Half e_shstrndx;
}Elf32_Ehdr;
```

LISTING 4. The data structure of the ELF headers.

mutation, which makes the test cases generated by this format knowledge be regarded as an `ELF` file by `objdump`. Besides, the *e_machine* stands for the architecture required to run the program. More specifically, "\x03\x00" represents that this file should run on the i383 architecture, which is consistent with the type of initial seed. Similarly, we could deduce the corresponding data structures in the learned format and show them in Table 10.

Comparing the data structure in Listing 4 with the values in Table 10, LearnAFL has identified 9 variables of

TABLE 10. The value of aata structures in the format model.

Data Structure	Value
e_ident	\x7fELF\x01\x01\x01\x00\x00
e_machine	\x03\x00
e_phoff	\x34\x00\x00\x00
e_shoff	\xa4\x00\x00\x00
e_flags	\x00\x00\x00\x00
e_phnum	\x02\x00
e_shentsize	\x28\x00
e_shnum	\x04\x00
e_shstrndx	\x03\x00

the ELF headers' data structure, which is about 64% of the total variables. By learning these format knowledge and utilizing them to assist mutation, LearnAFL can generate more test cases that pass some complicated program verifications than AFL and other tools.

Based on this result, we can get a conclusion that LearnAFL can learn a certain degree of the ELF file format accurately, about 64%, which is helpful to generate test cases with valid formats to exercise deep paths.

VI. RELATED WORK

In the previous sections, we have already produced some of the significant differences between LearnAFL and AFL. In this section, we survey recent work in the area of fuzzing and learning knowledge, which enables us to highlight some of the features and differences concerning existing work.

A. COVERAGE-BASED GREYBOX FUZZING

Coverage-based Greybox Fuzzing plays an important role in detecting vulnerabilities. As a typical representative among them, AFLFast [11] modeled Coverage-based Greybox Fuzzing as a Markov chain and proposed the transition probability p_{i_j} that fuzzing the seed exercising path i generates an input exercising path j. Based on these, AFLFast implements several power schedules and produces more unique crashes than AFL. However, AFLFast didn't modify the mutation operators or improve the effectiveness of the mutation strategy, which means the probability p_{i_j} does not change from AFL to AFLFast. In contrast, our work implements format-assistance mutation strategies. By learning formats of paths and utilizing the knowledge to assist mutation, the transition probability to explore deeper paths has been increased, which improves the effectiveness of AFL.

B. APPLICATION-AWARE EVOLUTIONARY FUZZING

Generally speaking, application-aware evolutionary fuzzing (e.g., VUzzer) mostly used some program analysis techniques to get information and learn the knowledge of the program, such as static analysis, symbolic execution, and dynamic taint analysis [8], [10]. According to the information and knowledge, application-aware evolutionary fuzzing accurately determines where and how to mutate seeds to explore deep and interesting paths. However, the main drawback of these techniques is that the test speed is significantly slower than AFL. Compared to these techniques, LearnAFL is built on the top of AFL and follows the high-speed feature of AFL. Besides, LearnAFL is more convenient to start and available for most targets of real-world programs than these techniques.

C. GRAMMAR-BASED FUZZING

Grammar-based fuzzing (e.g., Peach and SPIKE [39], [40]) is valid for fuzzing software with complex structured inputs. Provided an input grammar, grammar-based fuzzing can generate test cases satisfying the grammar and exercising deep paths, which is similar to the format-assistance mutation of LearnAFL. However, it is necessary for users to define an input gramma manually before doing fuzzing. Compared to grammar-based fuzzing, LearnAFL does not need to be provided a model in advance, just generating the format model during the processes of fuzzing. This mechanism improves the practical of LearnAFL.

D. LEARNING GRAMMARS FOR GRAMMAR-BASED FUZZING

Recently, some researchers propose new algorithms to synthesize grammars given a set of input examples. Godefroid *et al.* [41] used neural-network-based statistical learning techniques to generate input grammars from sample inputs automatically. TreeFuzz [42] was a fuzz testing approach for tree-structured inputs (such as programs) by learning a generative model of tree structures from a corpus of example data. Therefore, though the grammars (e.g., tree structures) is more accurate and efficient than our file model, collecting example data becomes a significant issue for these approaches. Compared to them, LearnAFL takes full advantage of the high-speed features of AFL to get lots of test cases for generating format models.

E. TARGETED MUTATION GREYBOX FUZZING

Lemieux and Sen [15] proposed a targeted mutation strategy for increasing testing coverage of AFL, which is called FairFuzz. Similarly to LearnAFL, FairFuzz also can identify those crucial parts of the input that are crucial to satisfy the determined conditions and avoid mutating these parts in the random mutation. However, FairFuzz achieves this target depending on the implementation of deterministic strategies, which decreases the efficiency of testing. In contrast, LearnAFL only does random mutation strategies. Moreover, our evaluation has proved that LearnAFL is more efficient than FairFuzz in exploring paths and triggering bugs.

VII. CONCLUSION

In this paper, we propose a knowledge-enhancement fuzzer based on AFL. LearnAFL classifies test cases into different sets during the fuzzing process and obtains partial format features of each path that has been exercised. After that, the format models of paths are used to assist mutation. Through this, we enhanced the effectiveness and efficiency of AFL in producing crashes, as evidenced by our experiments.

More importantly, we introduce the equivalence-classes-based format generation theory to explain the relationship between inputs and paths. Moreover, we observe that AFL may destroy partial format attributes during mutation resulting in generating ineffective test cases. Based on this, we propose the format-based path transition model and enhance AFL's performance in the help of paths' format attributes. The most important thing is that the transition probability to explore deeper paths has been grown in LearnAFL, which means we improve the effectiveness of AFL. In other words, LearnAFL effectively exposes the vulnerabilities which are more in-depth than these of AFL.

ACKNOWLEDGMENT

The authors would like to sincerely thank all the reviewers for your time and expertise on this paper. Your insightful comments help us improve this work.

REFERENCES

[1] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990.
[2] R. McNally, K. Yiu, D. Grove, and D. Gerhardy, "Fuzzing: The state of the art," Defence Sci. Technol. Org., Edinburgh, Scotland, Tech. Rep., 2012.
[3] G. Zhang and X. Zhou, "AFL extended with test case prioritization techniques," *Int. J. Model. Optim.*, vol. 8, no. 1, pp. 41–45, 2018.
[4] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. London, U.K.: Pearson Education, 2007.
[5] M. E. Khan and F. Khan, "A comparative study of white box, black box and grey box testing techniques," *Int. J. Adv. Comput. Sci. Appl.*, vol. 3, no. 6, pp. 12–15, 2012.
[6] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Proc. NDSS*, 2008, pp. 151–166.
[7] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2018, pp. 711–725.
[8] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proc. NDSS*, 2016, pp. 1–16.
[9] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2017, pp. 579–594.
[10] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *Proc. NDSS*, vol. 17, 2017, pp. 1–14.
[11] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," *IEEE Trans. Softw. Eng.*, vol. 45, no. 5, pp. 489–506, May 2019.
[12] *American fuzzy Lop (AFL)*. Accessed: Jul. 18, 2019. [Online]. Available: http://lcamtuf.coredump.cx/afl/
[13] *AFL Vulnerability Trophy Case*. Accessed: Jul. 18, 2019. [Online]. Available: http://lcamtuf.coredump.cx/afl/#bugs
[14] *FidgetyAFL*. Accessed: Jul. 18, 2019. [Online]. Available: https://groups.google.com/forum/#!msg/afl-users/fOPeb62FZUg/CES5lhznDgAJ
[15] C. Lemieux and K. Sen, "FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, Sep. 2018, pp. 475–485.
[16] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 2329–2344.
[17] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "CollAFL: Path sensitive fuzzing," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2018, pp. 679–696.
[18] G. Zhang, X. Zhou, Y. Luo, X. Wu, and E. Min, "PTfuzz: Guided fuzzing with processor trace feedback," *IEEE Access*, vol. 6, pp. 37302–37313, 2018.
[19] *The Technical Details of AFL*. Accessed: Jul. 18, 2019. [Online]. Available: http://lcamtuf.coredump.cx/afl/technical_details.txt
[20] *The Implementation Details of AFL*. Accessed: Jul. 18, 2019. [Online]. Available: https://paper.seebug.org/496/
[21] H. Simmons, *An Introduction to Category Theory*. Cambridge, U.K.: Cambridge Univ. Press, 2011.
[22] A. G. Hamilton, *Numbers, Sets and Axioms: The Apparatus of Mathematics*. Cambridge, U.K.: Cambridge Univ. Press, 1982.
[23] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Synthesizing program input grammars," *ACM SIGPLAN Notices*, vol. 52, no. 6, pp. 95–110, 2017.
[24] *Countable Set*. Accessed: Jul. 18, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Countable_set
[25] P. Fletcher and C. W. Patty, *Foundations of Higher Mathematics*. Pacific Grove, CA, USA: Brooks/Cole, 1996.
[26] *Bijection-Wikipedia*. Accessed: Jul. 18, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Bijection
[27] S. Gorbunov and A. Rosenbloom, "Autofuzz: Automated network protocol fuzzing framework," *Int. J. Comput. Sci. Netw. Secur.*, vol. 10, no. 8, pp. 239–245, 2010.
[28] *GNU Binutils*. Accessed: Jul. 18, 2019. [Online]. Available: http://www.gnu.org/software/binutils/
[29] *GIF2PNG*. Accessed: Jul. 18, 2019.[Online]. Available: http://www.catb.org/~esr/gif2png/gif2png.html
[30] *XpdfReader*. Accessed: Jul. 18, 2019. [Online]. Available: http://www.xpdfreader.com
[31] *Libjpeg*. Accessed: Jul. 18, 2019. [Online]. Available: http://libjpeg.sourceforge.net
[32] *Libpng*. Accessed: Jul. 18, 2019. [Online]. Available: http://www.libpng.org
[33] *Tcpdump*. Accessed: Jul. 18, 2019. [Online]. Available: https://www.tcpdump.org
[34] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 2123–2138.
[35] *AFL and AddressSanitizer*. Accessed: Jul. 18, 2019. [Online]. Available: https://fuzzing-project.org/tutorial3.html
[36] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2012, pp. 309–318.
[37] *AddressSanitizer*. Accessed: Jul. 18, 2019. [Online]. Available: https://github.com/google/sanitizers/wiki/AddressSanitizer
[38] *The Format of ELF Files Hearder*. Accessed: Jul. 18, 2019. [Online]. Available: https://blog.csdn.net/dddxxxx/article/details/80347610
[39] *Peach Fuzzer Platform*. Accessed: Jul. 18, 2019. [Online]. Available: https://www.peach.tech/products/peach-fuzzer/peach-platform/
[40] D. Aitel, "MSRPC fuzzing with spike 2006," Immunity, Washington, DC, USA, Tech. Rep., Aug. 2006.
[41] P. Godefroid, H. Peleg, and R. Singh, "Learn&Fuzz: Machine learning for input fuzzing," in *Proc. 32nd IEEE/ACM Int. Conf. Automat. Softw. Eng.*, Nov. 2017, pp. 50–59.
[42] J. Patra and M. Pradel, "Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data," Dept. Comput. Sci., Technische Univ. Darmstadt, Darmstadt, Germany, Tech. Rep. TUD-CS-2016-14664, 2016.

TAI YUE received the B.S. degree from the Department of Mathematics, Nanjing University, Nanjing, in 2017. He is currently pursuing the M.S. degree with the College of Computer, National University of Defense Technology. His research interests include software testing and software security.

YONG TANG received the B.Sc., M.Sc., and Ph.D. degrees in computer science from the College of Computer, National University of Defense Technology, China, in 1998, 2002, and 2008, respectively, where he is currently an Associate Professor. His research interests include software security, vulnerability discovery, malware detection, and network security.

BO YU received the M.S. and Ph.D. degrees from the National University of Defense Technology, in 2010 and 2013, respectively, where he is currently a Researcher. His research interests include system security and network security.

PENGFEI WANG received the B.S., M.S., and Ph.D. degrees from the College of Computer, National University of Defense Technology, Changsha, in 2011, 2013, and 2018, respectively, where he is currently an Assistant Professor. His research interests include operating systems and software testing.

ENZE WANG received the B.S. degree from the College of Automation, Northwestern Polytechnical University, Xi'an, in 2018. He is currently pursuing the M.S. degree with the College of Computer, National University of Defense Technology. His research interests include operating systems and software testing.