PathAFL: Path-Coverage Assisted Fuzzing

Shengbo Yan
yanshb@gmail.com
Nankai University
College of Cyber Science

Chenlu Wu
wucl555@gmail.com
Nankai University
College of Cyber Science

Hang Li
leejuly30@gmail.com
Nankai University
College of Artificial Intelligence

Wei Shao
wei.shao@mail.nankai.edu.cn
Nankai University
College of Cyber Science

Chunfu Jia*†
cfjia@nankai.edu.cn
Nankai University
College of Cyber Science

ABSTRACT

Fuzzing is an effective method to find software bugs and vulnerabilities. One of the most useful techniques is the coverage-guided fuzzing, whose key element is the tracing code coverage information. Existing coverage-guided fuzzers generally use the the number of basic blocks or edges explored to measure code coverage. Path-coverage can provide more accurate coverage information than basic block and edge coverage. However, the number of paths grows exponentially as the size of a program increases. It is almost impossible to trace all the paths of a real-world application.

In this paper, we propose a fuzzing solution named PathAFL, which assists a fuzzer by path identification. It can effectively identify and utilize the important *h-path*, which is a new path but whose edges have all been touched previously. First, PathAFL only inserts one assembly instruction to AFL's original code to calculate the path hash, and uses a selective instrumentation strategy to reduce the tracing granularity of an execution path. Second, we design a fast filtering algorithm to choose higher weight paths from a large number of *h-paths* and add them to the seed queue. Third, both the seed selection algorithm and the power schedule are implemented based on the path weight. Finally we implemented PathAFL based on the popular fuzzer AFL and evaluated it on 10 well-fuzzed benchmark programs. In 24 hours, PathAFL explored 38% more paths and 9.3% more edges than AFL. Compared with CollAFL-x, the number is 25% and 5.9% correspondingly. Moreover, PathAFL found the more bugs on the LAVA-M dataset, even four unlisted bugs. The results show that PathAFL outperforms the previous fuzzers in terms of both code coverage and bug discovery. In well-tested programs, PathAFL found 8 new security bugs with 6 CVEs assigned.

*Corresponding author.
†Also with TianJin Key Laboratory of Network and Data Security Technology, TianJin 300350, China.

ASIA CCS '20, October 5–9, 2020, Taipei, Taiwan

ACM ISBN 978-1-4503-6750-9/20/10…$15.00
https://doi.org/10.1145/3320269.3384736

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

Fuzzing, Vulnerability, Path Coverage, Instrumentation

ACM Reference Format:
Shengbo Yan, Chenlu Wu, Hang Li, Wei Shao, and Chunfu Jia. 2020. PathAFL: Path-Coverage Assisted Fuzzing. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIA CCS '20), October 5–9, 2020, Taipei, Taiwan.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3320269.3384736

1 INTRODUCTION

Software bugs are everywhere. Serious software vulnerabilities may affect the security of the entire system. Fuzzing is an effective technique for finding security bugs and vulnerabilities in software. It generates a large number of inputs to test the target program and monitors the running status of the program to discover the defects exposed. In 1990, Miller et al.[26] developed the first fuzzing tool, which was originally designed to test the reliability of UNIX tools. Traditional fuzzing tools bring too much randomness in generating test cases, thus largely decrease efficiency. To tackle the problems in fuzzing, researchers have proposed many new methods to improve the efficiency of fuzzing, and found a large number of vulnerabilities in real-world software[11][25].

Coverage-guided fuzzing is one of the most effective methods. It obtains the code coverage information generated by the instrumentation tools. This information is then used to guide the generation of the test cases in the next loop to maximize the code coverage. Currently, there are two basic measures of code coverage in fuzzers. One is to count execution basic blocks (BBLs). The other one is to use the transition between two BBLs, which is also known as the edge coverage. A BBL only has one single entry and one exit point, which is easy to count. For example, LibFuzzer[32] and honggfuzz[4] used the static instrumentation provided by a compiler (GCC or Clang) to trace the BBL coverage information. VUzzer[30] uses the dynamic instrumentation tool PIN[1] to trace BBL coverage information. American fuzzy lop (AFL)[3] is the first tool to introduce edge coverage into coverage-guided fuzzing. It supports both compile-time instrumentation and external instrumentation, with GCC/LLVM[23] mode and QEMU[7] mode. This method provides more accurate information than BBL coverage.

Recently, there have been some new ways to improve the measurement, such as Angora[13], which use context-sensitive branch counts to measure code coverage.

However, the above methods can not fully describe the path information of program execution. Some vulnerabilities can only be triggered under a specific execution path. Using only BBL or edge coverage may omit such paths and fail to trigger vulnerabilities. Compared with BBL and edge coverage information, path coverage information provides a more accurate execution tracing knowledge. In theory, this information can better guide a fuzzer. However, in real-world it is impossible to trace all program execution paths during the running time of a program because the number of paths grows exponentially, and the resources to store and analyze these paths are unaffordable.

In this paper, we propose a novel path-coverage assisted fuzzing solution, named PathAFL, which balances between the tracing path granularity and the fuzzing performance.

We use static instrumentation to trace program execution paths and calculate the path hash to distinguish different *h-paths*. Similar to AFL, PathAFL inserts codes at compile-time. These codes automatically calculate the path hash during the program execution and store it in shared memory, from which the PathAFL can read to retrieve the hash of program execution paths. In this way, it is not necessary to record the entire path details. As the number of paths grows exponentially, we have to reduce the tracing granularity of the paths. PathAFL performs selective instrumentation by only tracing larger functions and memory operation functions. The method reduces the number of paths actually being traced. Furthermore, we simplify the hash function to decrease the number of traced paths and reduce the computational load.

After using the above methods, the number of new paths found during fuzzing is still too large, which results in the decrease of performance of fuzzers. We design a fast path filtering algorithm to determine which *h-paths* to be added to the seed queue. We refer to and improve a key point of the algorithm in the method of CollAFL[17]. PathAFL uses static analysis to extract program branch information. During fuzzing, it dynamically calculates the weight of the path based on the number of untouched neighbor branches and function calls in the branches. Only when the weight of the newly discovered path is high will the test case corresponding to the path be added to the seed queue.

The path filtering algorithm used by PathAFL adds a lot of high weight paths to the seed queue. To further utilize these paths, we design a new seed selection algorithm that preferentially selects higher weight seeds and fixes the inherent defects in the original algorithm. Besides, we add a power schedule[9] to assign more energy to the path with higher weight, and conversely assign less energy to the lower weight path.

In order to calculate the number of edge coverage and path weight accurately, we extend AFL to implement two main algorithms in CollAFL which reduce edge hash collisions. On this basis, we implement two fuzzers which are PathAFL and CollAFL-x. CollAFL-x uses untouched-neighbor-branch guided policy, which shows the best experimental result among the three policies in the paper of CollAFL. We use it as a baseline fuzzer in experiments.

We tested PathAFL using 10 widely-used test programs. The results show that the number of both path and edge coverage PathAFL explored is greater than that by AFL and CollAFL-x. After 24 hours, compared with AFL and CollAFL-x, the average number of path coverage of PathAFL increased by 38% and 25% respectively, and the edge coverage increased by 9.3% and 5.9% respectively. Besides, PathAFL found more bugs than AFL on the LAVA-M dataset[16], and even found four unlisted bugs (bugs that the LAVA authors injected but were unable to trigger). Furthermore, PathAFL found 8 new bugs in three popular toolkits for fuzzing, 6 of which were assigned CVE IDs.

This paper makes the following main contributions:

- We propose a novel method for tracing the execution path. We demonstrate defects tracing edge coverage information used by AFL and discuss some problems in path-coverage. To address these problems we make a trade-off between tracing path coverage granularity and fuzzing performance. It is found that we can trace important paths with very little overhead.
- We design a path filtering algorithm. It makes a quick judgment on the new path. Only those paths that meet certain conditions and own a high weight will be added to the seed queue.
- We improve AFL based on the path weight. We discover a flaw in the AFL's seed selection algorithm and design a new algorithm according to the path weight. At the same time, a power schedule based on the path weight is added, which can be enabled by the parameter setting.
- We develop an open-source implementation of PathAFL, and publish it on Github[1] as a fork of AFL, hoping that our work can be useful for readers of this paper.

The rest of this paper is structured as follows. Section 2 gives the overview and our motivation. Section 3 respectively elaborates the details of PathAFL. Section 4 presents the implementation details. Section 5 evaluates our approach. Section 6 discusses our work and some limitations. Section 7 introduces the related work before. Section 8 draws the conclusions.

2 OVERVIEW

Our proposed solution, PathAFL, is built on the state-of-the-art fuzzers, AFL and CollAFL. In this section, we first present an overview of these two fuzzers. We also introduce the advantages and disadvantages of three code coverage measurements and use an example to illustrate the motivation of PathAFL at the end.

2.1 Overview of AFL

AFL is an edge coverage-guided grey-box fuzzer that employs a novel compile-time instrumentation and genetic algorithms to automatically discover interesting test cases that trigger new internal states in a target binary. Fig 1 shows the working process of PathAFL. The purple components illustrate our improvement on the original AFL. In this section we introduce AFL and its main loop, which includes the steps as follows:

(1) Instrument target application.
(2) Provide initial inputs to the seed queue.

[1] https://github.com/yanxxd/PathAFL

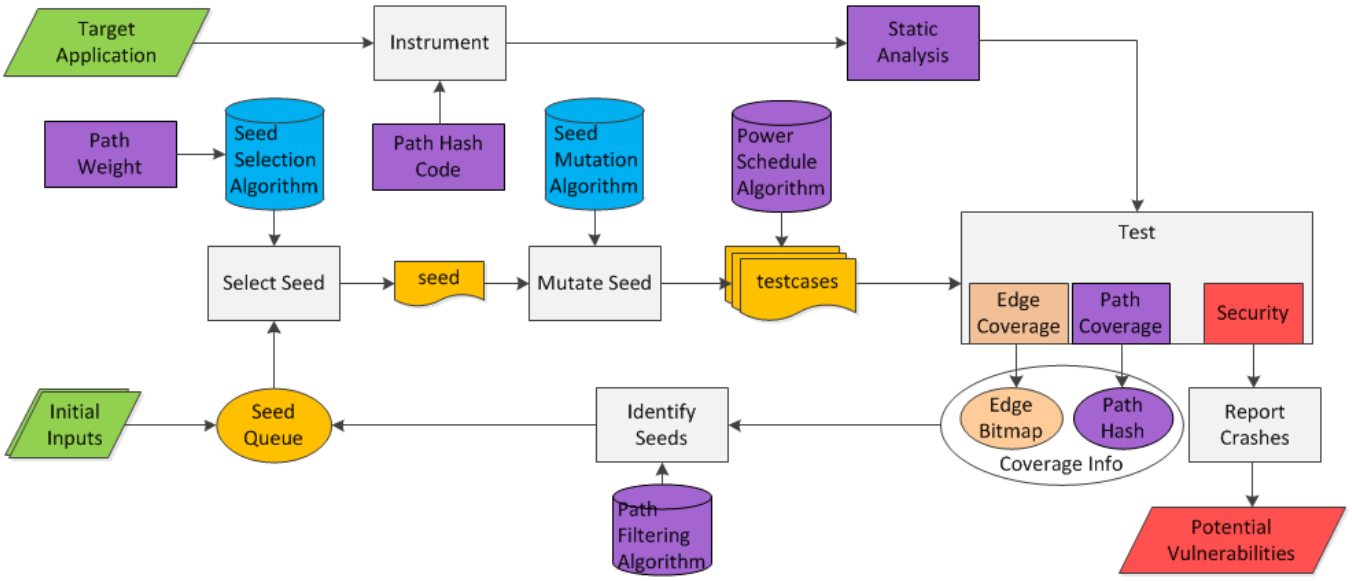

Figure 1: Working process of PathAFL.(Except for the purple components, it is the original AFL.)

(3) Select seed. AFL selects a seed from the seed queue according to the seed selection algorithm, which prefers the faster and smaller ones.
(4) Mutate seed. This step uses multiple mutation algorithms to mutate the seed file and generate a large number of test cases in a loop.
(5) Test and trace. This step takes test case as input, executes and traces the instrumented target application.
(6) Report crashes. If a crash is found, a potential vulnerability may have been triggered.
(7) Identify seed. If a new edge coverage state is found, adding the test case to the seed queue for the next loop.
(8) If fuzzing this seed is over, go to step (3), otherwise go to step (4).

The entire fuzzing is an infinite loop and it only end the loop when it is manually terminated.

2.1.1 *Seed Selection.*

In AFL, the fastest and smallest seed that covers an edge is called the *top rated seed* for this edge. The seed selection algorithm generates $favored\ seed\ set$ from all *top rated seeds*. Seeds in this set are tested preferentially. The $favored\ seed\ set$ is selected based on the constraint that it must cover all discovered edges. Seed selection algorithm is implemented using greedy algorithm, as shown in Algorithm 1.

2.1.2 *Coverage Calculation.*

AFL records edge coverage information, including which edges were hit and how many times they were hit. It is stored in an array with a fixed size, usually is 65536 bytes. The instrumented code calculates the edge hash, which is used as the index of the array. The value in the array is the number of times the edge is hit. AFL uses a novel method to calculate the edge hash. It inserts a random number called BID (id of BBL) from 0 to 65535 in each BBL.

Algorithm 1 Seed Selection Algorithm of AFL

Input: $T[MAP\_SIZE]$ - top rated seed array, edge hash as index.
Output: F - favored seed set

```
1: C[MAP_SIZE] ← ∅                          ▷ edges covered by F
2: for 0 <= h < MAP_SIZE do                 ▷ h: hash of edge
3:     if T[h] and not C[h] then
4:         F.Add(T[h])
5:         UpdateBitmap(C, T[h])
6:     end if
7: end for
```

The edge hash is calculated with the random numbers in the head BBL and tail BBL of the edge through a simple hash algorithm. The calculation formula is as follows:

$$hash\_edge(\langle bb1, bb2\rangle) = (BID(bb1) >> 1) \oplus BID(bb2) \quad (1)$$

$\langle bb1, bb2\rangle$ represents an edge. $bb1$ is the head BBL of the edge, and $bb2$ is the tail BBL of the edge. The above algorithm ensures that edge $\langle bb1, bb2\rangle$ has a different hash from edge $\langle bb2, bb1\rangle$.

2.2 Overview of CollAFL

It is obvious that formula 1 will cause a hash collision issue, where two different edges may have the same hash, resulting in inaccurate records. The collision rate may reach 30%[3] when the number of edges reaches 50000, even reaches 75% in some programs[17]. AFL uses inaccurate edge coverage information to identify test cases and implement seed selection, which affects the performance of AFL.

CollAFL generates three new hash calculation formulas, which reduce the collision rate to nearly 0 and improve the accuracy of edge coverage information. It uses three hash functions, i.e., *Fsingle*,

Fmul and *Fhash* to resolve the edge hash collision issue afterto-wards. The functions are defined as follows,

$$Fsingle(\langle bb1, bb2\rangle) = c(bb2) \quad (2)$$

$$Fmul(\langle bb1, bb2\rangle) = (BID(bb1) >> x) \oplus (BID(bb2) >> y) + z \quad (3)$$

$$Fhash(\langle bb1, bb2\rangle) = hash\_table\_lookup(\langle bb1, bb2\rangle) \quad (4)$$

where c, x, y and z are parameters to be determined, which could be different for different edges.

CollAFL divides BBLs into two categories, one with multiple precedent BBLs and the other with only one precedent BBL. Then it uses *Fmul* and *Fhash* to handle the BBLs in the first category, and *Fsingle* to handle the BBLs in the second category. Performance overhead using *Fhash* is the largest, but the number of BBLs that needs to use *Fhash* is close to 0 because the vast majority of BBLs in the first category can be solved by *Fmul*.

In addition, CollAFL proposes three novel seed selection policies. The first policy is that seeds with more untouched neighbor branches will be prioritized to fuzzing. The second policy prefers the seeds with more untouched neighbor descendants. The third prefers the seeds with more memory access operations.

2.3 Motivation of PathAFL

Tracing code coverage is crucial for coverage-guided fuzzing since detailed coverage information always provide abundant feedback. According to the granularity of coverage, methods used to measure code coverage are mainly divided into three types, that is, BBL coverage, edge coverage and path coverage. Although CollAFL has improved the accuracy of coverage information, it still only uses edge coverage information.

In this section, we briefly review three measuring methods of code coverage and demonstrate how path coverage help us to improve the efficiency of fuzzer with a simple example.

2.3.1 *BBL Coverage.*

BBL is a code snippet with one single entry and one exit point, instructions in BBL will be sequentially executed and will only be executed once. BBL is the smallest coherent unit of program execution, which could be identified by the address of the first instruction. BBL information could be easily extracted through code instrumentation and static analysis. Because of these advantages, BBL coverage information is widely used by fuzzers.

While typical coverage-guided fuzzers based on BBL only trace whether each block is hit, not the order which blocks are hit during fuzzing, so the detailed information is lost. As shown in Fig 2, during fuzzing, if the program path 1(A,B,C,D) is executed first, test cases associated with program path 2(A,B,D) will never be added to seed queue since path 2 doesn't hit new BBL, so the information of edge BD will lose.

2.3.2 *Edge Coverage.*

To tackle the disadvantage of BBL coverage, edge coverage traces whether each edge is hit or not. In the previous example in Fig 2, if the program path 1(A,B,C,D) is executed first, fuzzer will record edges AB, BC and CD, a new edge BD will be recorded when path 2(A,B,D) is executed, so test cases associated with path 2 will now be added to seed queue. A well-known method that uses edge coverage is AFL.

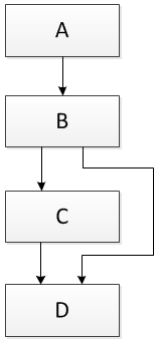

Figure 2: A sample of BBL coverage

However, edge coverage doesn't trace the order that edges are hit, so some detailed information may lose. We illustrate this problem using the simple program in Listing 1 which takes 8 characters as input. The program will crash when input is "abcd**!!" or "**cdef!!". In practice, this kind of code is very common.

Listing 1: A simple program

```
1 void vul(short *s){
2   if(s[0] == 0x6261) //ab
3     s[2] = 0x6665;   //ef
4
5   if(s[1] == 0x6463) //cd
6     if(((int*)s)[1]==0x21216665) //ef!!
7       abort();
8 }
```

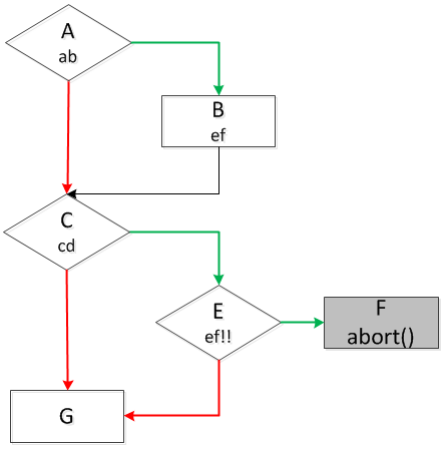

Figure 3: A sample of edge coverage. The green arrow indicates success, and the red indicates failure.

The control flow graph corresponding to Listing 1 is shown in Fig 3. It has 6 BBLs, 7 edges and 6 execution paths. Input "abcd**!!" or "**cdef!!" will trigger crash, with execution path p_0(A,B,C,E,F) or p_1(A,C,E,F). Obviously, compared to p_1, p_0 is easier to find. A common approach is to firstly generate the input "ab******" or "**cd****", and then mutate it into "abcd****", finally change it to "abcd**!!".

The probability of each step is $1/65536$ even we use the random variation strategy, which is easy for edge coverage-guided fuzzer. However, things become hard when taking the following situation into consideration:

(1) Fuzzer starts fuzzing using random seed s_0 with one execution path p_2(A,C,G) and covered edges E{AC, CG}.
(2) Fuzzer takes "ab*****" as test case s_1 with execution path p_3(A,B,C,G) and edges {AB, BC, CG} by mutating s_0. This test case will be added to the seed queue because new edges {AB, BC} are hit, now E is updated to {AB, AC, BC, CG}.
(3) Another input "**cd****" as test case s_2 with execution path p_4(A,C,E,G) and edges {AC, CE, EG} is generated by mutating s_0. This test case will also be added to the seed queue because new edges {CE, EG} are hit, now E is updated to {AB, AC, BC, CE, CG, EG}.
(4) The problem comes up when "abcd****" as test case s_3 with execution path p_5(A,B,C,E,G) is generated. Since p_5 doesn't cover any new edges, this test case won't be added to the seed queue. In order to generate test case "abcd**‼" or "**cdef‼", edge coverage-guided fuzzer has to select seed from s_0, s_1 and s_2, and all of them have at least four bytes different from target data, so the probability is $1/2^{32}$, which is very difficult to trigger crash.

We write a simple test program with 26 lines code to verify our previous analysis. We uploaded it to our github repository as well, and AFL failed to trigger the crash during one week fuzzing, so the edge coverage method does have some disadvantages.

2.3.3 *Path Coverage.*

Methods based on path coverage trace the whole execution path, including the order edges are hit, so the most abundant information are recorded. If we go back to the fourth step in the last section and use path coverage to measure code coverage, when fuzzer covers path p_5, it will add this test case s_3 to the seed queue even if p_5 doesn't hit any new edges, and mutating from s_3 to "abcd**‼" is much easier than mutating from s_0, s_1 and s_2 to "abcd**‼".

It is almost impossible to implement path coverage for real-world applications, because there are so many loops and conditions in the program that the number of paths will explode. Massive paths will bring large run-time overhead and may decrease efficiency during fuzzing. To overcome this issue, we divide new explored paths into two categories:

- path with previously untouched edges, we denoted it as *e-path*. "*e*–" means that the path has new **e**dges.
- path that all the edges have been touched, like path p_5, we denoted it as *h-path*. "*h*–" means that the path has a new **h**ash.

The critical problem is how to handle massive *h-paths*, our solution is that instead of adding all *h-paths* to the seed queue, we only add those **high-weight** *h-paths* to the seed queue. It is kind of a trade-off between efficiency and tracing granularity.

3 TECHNOLOGY

The main challenge is the number of paths grows exponentially as the program size increases. Therefore, it is strateforward to add only important paths to the seed queue. Following this idea, we summarize the three key issues in path-coverage assisted fuzzer:

(1) How to identify *h-paths*?
(2) How to reduce the number of *h-paths*?
(3) Which *h-paths* will be added to the seed queue?

In this section, we discuss the challenges and propose our methods.

3.1 How to Identify *h-paths*?

The most intuitive way to trace a program execution path is to record the whole program execution flow, which can be represented by BBL sequences. However, it will cost lots of memories when the execution path is too long. Path hash is used to identify if a new path have been explored. To reduce memory consumption, we propose a novel approach that calculates the program execution path hash automatically during run-time.

Static instrumentation. AFL instruments the target program at compile-time. The instrumentation code calculates the edge hash and updates the edge coverage record to shared memory. PathAFL maintains a similar global hash table as AFL. The table index represents the path hash and the value represents whether the path is covered. PathAFL instruments the target program in a similar way. It only inserts a small piece of code to AFL's original instrumentation code to calculate the path hash. Moreover, PathAFL extends the original shared memory by 4 bytes to store the path hash value. During the execution of a program, the hash of the execution path is calculated in real-time and appended to the shared memory. When the running of the program is finished, the execution path hash can be used to determine whether a new path has been explored.

Compared with AFL, PathAFL only adds the hash calculation function in instrumentation code, leading to little overhead increseases.

3.2 How to Reduce the Number of *h-paths*?

We construct a hash function by common "shift" and "xor" operations, and instrument a program according to the method in section 3.1. With a few simple tests, it is easy to find that there are still too many *h-paths* in fuzzing, resulting in a sharp decline in the performance of a fuzzer. It once again proves that it is not advisable to trace the full execution path. PathAFL adopts two approaches to reduce the tracing granularity of the execution path so that the fuzzer can limit the number of new *h-paths*.

3.2.1 *Selective Instrumentation.*

AFL instruments all edges in the target program by default, allowing us to trace almost all the execution paths. To reduce *h-paths*, PathAFL has to reduce tracing granularity, only instrumenting partial BBLs and tracing the paths which have a higher probability to reach new BBLs or trigger new bugs. We have three intuitions that can help us:

(1) To trace the whole path approximately, the instrumentation locations should be evenly distributed in the execution path. Instrumenting function entry is a good choice and is easier to implement technically.

(2) A larger function body contains more code numbers, which means more chances to trigger a crash. In other words, tracing a larger function and mutating *h-path* passing through these functions will cover more code and have higher probability of founding a crash.
(3) The ultimate goal of fuzzing is to improve the efficiency of vulnerability discovery. There is an intuition that if *h-path* goes through functions with more memory operations, it is more likely to trigger potential memory corruption vulnerabilities, so does its mutations.

Following the above intuitions, PathAFL only instruments some functions for the tracing of the execution path. The following four strategies are put forward to select key functions:

(1) Instrument the larger functions, default the largest 20%. Hence many *h-paths* passing through these functions can be found.
(2) Instrument the functions that have memory operations, such as function whose name includes "alloc/free".
(3) Neglect the functions that are too small. Similar to (1), the smaller the function is, the less code there is.
(4) Instrument 10% of other functions.

According to the above strategies, the number of newly discovered *h-path* can be effectively reduced.

3.2.2 Hash Algorithm.

The original intention of using hash functions is to quickly distinguish different paths and judge whether the path has been executed. Due to the fact that the number of execution paths is large, there are massive path hash collisions. A strong hash algorithm can effectively reduce hash collisions, but it requires more instrumentation code, which will slow down the running speed. In addition, not all new explored paths are added to the seed queue.(We will discuss in detail in Section 3.3). Therefore, we decide on using a path hash function that are not too strong.

We adopt a very simple scheme, which directly uses the total addition operation as the path hash algorithm and distinguish execution paths in a coarse-grained manner. The hash algorithm is as follows:

$$hash\_path(p) = \sum_{bb \in BS} BID(bb) \tag{5}$$

p denotes the execution path. BS denotes the execution sequence of all instrumented BBLs. It only needs to insert an "add" assembly instruction in the existing instrumentation code, which has bare effect on the running efficiency of the tested programs. The size of the hash table is the same as that of the bitmap in AFL. Note that this method is only used to calculate the path hash and does not affect the original edge hash calculation of AFL. After testing several real programs, it is found that this method is indeed feasible.

3.3 Which *h-paths* Will Be Added to the Seed Queue?

Unlike other fuzzers that only pay attention to *e-paths*, PathAFL considers *h-paths* in the meanwhile. Even though tracing granularity has been reduced, there are still massive amounts of *h-paths* in the seed queue which will largely increase run-time overhead and decrease edges coverage. Hence, to further reduce the number of *h-paths* added to the seed queue, a strategy is designed to determine whether or not to add an *h-path* to the seed queue. This strategy should meet the following requirements:

- Efficiency is a crucial factor.
- The *e-path* is more important than the *h-path*.
- The *h-path* that is more likely to touch a new edge after mutating should be chosen.

A fast filtering algorithm is designed fufilling those requirements as shown in Algorithm 2.

Algorithm 2 Path Filtering Algorithm

```
1:  function PathFilter(p)                          ▷ p - path
2:      if HasNewEdgeCoverage(p) then
3:          AddtoSeedQueue(p)
4:      else if NumOfPathInQueue() < MIN_PATHS then
5:          return
6:      else if IsContinuousPath(p) then
7:          return
8:      else if not IsNewPathHash(p) then
9:          return
10:     else if CalcWeight(p) > weight_high then
11:         AddtoSeedQueue(p)
12:     end if
13: end function
```

(1) As shown in line 2 and 3, if there is a new *e-path*, add it to the seed queue (same as AFL).
(2) If the current total number of paths in the seed queue is less than *MIN_PATHS* which is a predefined value, do not add it. Since in the initial stage, fuzzers usually find a large number of *e-paths*, *h-paths* do not need to be added to the queue. Moreover, if there are not enough paths, it is difficult to determine whether an *h-path* is good or bad.
(3) To keep the diversity of *h-paths*, three consecutive *h-paths* should not appear, due to the fact that they are generally generated by mutating the same seed and will be too similar.
(4) *CalcWeight* requires a little calculation. It must be placed in the last step and will be called by fuzzers as few times as possible.

CollAFL uses the number of untouched neighbor branches as the weight of a path and proves that in most cases test cases with higher weights are better. However, CollAFL only takes the number of neighbor branches into consideration and ignores the number of function calls that directly correspond to the size of the branch. In PathAFL, the number of instruction "call" is used to adjust the weight of the neighbor branch in a path. The weight of edges and paths is given by:

$$\begin{aligned} \text{weight\_edge}(\langle bb1, bb2\rangle) = \text{IsUntouched}(\langle bb1, bb2\rangle)? \\ (1 + \text{CountCall}(bb2) \cdot 2) : 0 \end{aligned} \tag{6}$$

$$\text{weight\_path}(p) = \sum_{\substack{bb \in p \\ < bb, bb_i > \in E}} \text{weight\_edge}(< bb, bb_i >) \tag{7}$$

CountCall returns the number of instruction "call" in $bb1$. *IsUntouched* returns 1 if the edge <bb1, bb2> is not covered by any previous test case, otherwise 0. E denotes the set of all edges. p denotes the execution path.

PathAFL only adds test case whose path weight is greater than a predefined value *weight_high* which is given by:

$$\text{weight\_high} = \text{weight\_avg} + (\text{weight\_max} - \text{weight\_avg})/w \quad (8)$$

weight_avg denotes the average weight of seeds in seed queue and *weight_max* denotes the maximum weight. w is an input parameter, which is set to 3 by default. In general, *weight_high* is greater than the mean value, which ensures *h-paths* added to the seed queue have enough neighbor branches.

3.4 Seed Selection Algorithm

How to select seeds from the seed queue in fuzzing is a very critical problem. Previous work has proved that a good seed selection strategy can significantly improve the fuzzing efficiency and help to find more bugs faster[25]. MoonShine[29] introduced seed distillation. Pailoor et al.[31] compared six different seed selection algorithms and the results showed that heuristics employed by seed selection algorithms perform better than fully random sampling.

We adopt the same idea as CollAFL of preferentially selecting

Algorithm 3 Fuzzing Loop of AFL

```
1:  for TRUE do
2:      //start one fuzzing loop
3:      q = queue
4:      while q do                          ▷ Traverse queue
5:          F = cull_seed(q)                ▷ Call Algorithm 1
6:          if IsTest(q) then               ▷ Fuzzing q?
7:              fuzzing_one(q)
8:          end if
9:          q = q->next
10:     end while
11: end for
```

the seeds with higher weight. However, there is a small problem in the original algorithm of AFL. Algorithm 1 ensures that F(favored seed set) covers all explored edges, but it does not ensure that the tested favored seeds cover all the discovered edges in one fuzzing loop. Specifically, as shown in Algorithm 3, AFL contains an infinite fuzzing loop and each loop traverses the seed queue. Suppose the following example, the seed queue is

$$Q = \{q_0, q_1, ..., q_i, ..., q, ..., q_n\}$$

q represents current position in queue. If F does not contain q_i before current loop, but it does from now until current loop end. In that way, q_i will be omitted and unique edges in q_i may not be covered by tested seeds in this fuzzing loop. This problem is easily spotted as it happens so frequently.

We propose a new seed selection algorithm as shown in algorithm 4. It fixes the defects mentioned above to ensure that favored seeds that are tested in one fuzzing cycle will cover all discovered edges. It is divided into 4 steps, as follows:

(1) As shown in line 2 and 3, we split seeds queue into two subsets Q_1 and Q_2 by current position. Q_1 contains seeds before current position, and Q_2 contains others.
(2) Update C which records all edges covered by F.
(3) Sort the seeds in Q_2 in descending order by weight.
(4) Select seeds by the above order, and add those that cover new edges to F, otherwise remove from F.

Algorithm 4 Seed Selection Algorithm of PathAFL

Input: F - favored seed set
Output: F

```
1:  C[MAP_SIZE] ← ∅                         ▷ edges covered by F
2:  Q1 ← Seeds before queue_cur
3:  Q2 ← Current and subsequent seeds
4:
5:  for q in Q1 do
6:      if q in F then
7:          UpdateBitmap(C, q)
8:      end if
9:  end for
10: SortByWeight(Q2)
11: for q in Q2 do
12:     if HasNewCoverage(q) then
13:         F.Add(q)
14:         UpdateBitmap(C, q)
15:     else
16:         F.Remove(q)
17:     end if
18: end for
```

3.5 Power Schedule

Böhme et al.[9] first proposed the method of power schedule in AFLFast. A power schedule decides how many test cases are generated when fuzzing a seed. AFLFast explains the challenges and opportunities of CGF (Coverage-based Greybox Fuzzing) using a Markov chain model and divide the paths into high-frequency ones and low-frequency ones. It assigns more fuzzing energy to low-frequency paths. However, the experiments of Klees et al.[22] show that AFLFast has even less edge coverage than AFL dose over a long period of time.

PathAFL implements a new power schedule according to the weight of the path, and assigns more energy to higher weight paths. Specifically, it divides all paths into six parts by weight, each of which is assigned different energy. In Section 5, PathAFL with power schedule and the one without power schedule are tested respectively.

4 IMPLEMENTATION

The latest open source version of AFL as of this writing was 2.52b, so we extended it to PathAFL. The purple components in Figure 1 are newly added. We implemented them using Assembly, C, C++, and Python. In this section we explain the implementation of some components.

Instrumentation. To calculate path weight accurately, we implemented *Fsingle* and *Fmul* that were proposed in CollAFL. We didn't implement *Fhash* because it is used in a very low probability. According to CollAFL's experiment results, untouched-neighbor-branch guided policy is the best method among the three guided

Table 1: Time overhead of static analysis

Application	Time(s)
binutils-objdump	265
binutils-readelf	79
bison	84
cflow	12
exiv2	245
libnurses-captoinfo	31
libpng-pngtest	28
libtasn1-asn1Parser	8
libtiff-pal2rgb	37

policies. Therefore, on the basis of the above, we implemented a fuzzer with untouched-neighbor-branch guided policy, named it CollAFL-x, as the baseline fuzzer for comparison with PathAFL in Section 5, because PathAFL also uses the same methods to avoid edge hash collision.

We use static instrumentation to achieve *Fsingle* and *Fmul*. In addition, PathAFL inserts only one assembly instruction at the end of AFL's instrumentation code for tracing program execution path, which has illustrated in Section 3.2.2. In this step, We instrument all function entry during compilation.

Static Analysis. We use IDA for static analysis. Referring to Shudrak et al.[34]'s method, the IDAPython[5] script is written to automatically analyze program structure, obtain the edge information, and generate a data file for the target program. The file records all the edge information, as an input of PathAFL to calculate path weight. Static analysis will take a litte time. We analyzed it in a Windows virtual machine with Intel Core i7-2600 CPU. The final result is shown in Table 1. Compared with time overhead by day during fuzzing, this time overhead is insignificant.

In additon, in this step we remove the instrumentation for some functions according to the strategy in section 3.2.1.

Path Filtering. We implement Algorithm 2 in the original *save_if_interesting* function in *afl-fuzz.c*. It requires the parameter *weight_high*. *weight_high* is updated only when a new seed is found, to reduce the computational overhead.

Power Schedule. We add the "-p" argument to PathAFL to indicate whether the power schedule is enabled in the test. We use "PathAFL-np" to indicate the version without power schedule.

5 EVALUATION

In this section, we performed a set of experiments on different applications and compared with four fuzzers AFL, CollAFL-x, PathAFL-np, and PathAFL. The experiments were designed to answer the following three research questions:

- RQ1. Is tracing execution path feasible for fuzzing?
- RQ2. How good is the code coverage of PathAFL?
- RQ3. How good is the bug detection capability of PathAFL?

We ran all experiments on two 64-bit machines with 32 cores (Intel(R) Xeon(R) E5-2620 v4 @ 2.10GHz), 64GB of main memory, 12TB hard disk, and Ubuntu 16.04 as host OS.

Table 2: Time to expose the bug.

Fuzzer	Total time	Crashes	First crash
AFL	7 days	0	N/A
CollAFL-x	7 days	0	N/A
PathAFL-np	1 days	6	1h 4m
PathAFL	1 days	8	1h 3m

5.1 Result on a Simple Example(RQ1)

To illustrate the significance of path coverage, we built a test program in Section 2.3.3. Because the program is very simple, we instrumented all edges to calculate the path hash when test with PathAFL. The results are shown in Table 2. AFL and CollAFL-x were unable to trigger a crash in 7 days, while PathAFL and PathAFL-np triggered it for only an hour. Moreover, PathAFL and PathAFL-np triggered 6 and 8 crashes in one day respectively.

We observed that AFL and CollAFL-x were stuck after finding two paths. By contrast, in the first minute PathAFL and PathAFL-np found the third path which corresponds to the input "abcd****", because them can effectively identify the *h-path*. Then they quickly mutated the seed, triggered the crash an hour later. This experiment shows that PathAFL can effectively identify *h-path*, and it is feasible to use *h-path* to guide the fuzzer.

5.2 Code Covereage and Crashes Measurements(RQ2)

Klees et al.[22] pointed out some common problems in current fuzzing experiments and proposed testing methods in the hope of establishing unified testing standards. This paper refers to the testing ideas proposed by them, and considering the hardware resources we have, we conducted the experiments in the following ways:

(1) We selected 9 popular applications as shown in Table 1 and conducted 10 experiments, in which two different parameters of `objdump` were tested.
(2) For each target program, some common inputs are selected, which are empty seed, random seed, normal files, public test samples and POC (Proof of Concept) of vulnerabilities. For each experiment, only one seed input is provided.
(3) Because fuzzing is an inherently non-deterministic process, we ran each experiment at least ten times and averaged the results, and each experiment for 24 hours.
(4) We evaluated PathAFL-np and PathAFL, with CollAFL-x and AFL as baseline fuzzer.
(5) AFL supports the master-slave fuzzing paradigm. In real vulnerability discovery process, master and slave fuzzers usually run in parallel. So, in order to get closer to the real situation, we ran each experiment with one master fuzzer and one slave fuzzer in parallel.
(6) The main measurement is the number of paths and edges explored. The fuzzer that grows with both paths and edges is better. In addition, we also measured the number of unique crashes triggered.

The number of Paths and Edges. Table 3 shows the number of paths and edges explored by four fuzzers. Columns 2-5 show the

Table 3: The number of paths and edges explored by four fuzzers. The data before the comma represents the percentage increase compared to AFL, and the data after the comma represents the percentage increase compared to CollAFL-x.

Command	AFL		CollAFL-x		PathAFL-np		PathAFL	
	Paths	Edges	Paths	Edges	Paths	Edges	Paths	Edges
asn1Parser -c	682	813	791	859	+32.3%, +14.1%	+5.44%, -0.19%	+31.4%, +13.3%	+7.83%, +2.07%
bison	1058	2262	1229	2402	+70.0%, +46.4%	+6.34%, +0.14%	+66.0%, +42.9%	+29.55%, +22.00%
captoinfo	4485	3695	4052	3678	+21.8%, +34.9%	+0.82%, +1.27%	+30.4%, +44.4%	+2.24%, +2.70%
cflow	2586	2562	2796	2636	+53.0%, +41.4%	+3.71%, +0.81%	+49.6%, +38.3%	+3.84%, +0.93%
exiv2	3908	7727	4661	8112	+45.4%, +21.9%	+6.47%, +1.41%	+36.6%, +14.5%	+9.92%, +4.71%
tiff2pdf	3815	7556	4558	7668	+43.3%, +20.0%	+9.45%, +7.86%	+44.0%, +20.5%	+8.54%, +6.97%
objdump -d	6772	7890	7649	8295	+66.6%, +47.5%	+8.76%, +3.45%	+64.2%, +45.3%	+7.18%, +1.95%
objdump -x	1827	3283	1942	3316	+24.2%, +16.9%	+7.09%, +6.01%	+23.5%, +16.2%	+6.15%, +5.08%
pngtest	440	1543	469	1600	+22.0%, +14.6%	+11.47%, +7.50%	+17.5%, +10.5%	+11.77%, +7.78%
readelf -a	12053	9487	12628	9639	+7.2%, +2.4%	+2.69%, +1.07%	+13.6%, +8.4%	+6.24%, +4.57%
Average	3763	4682	4077	4820	+38.6%, +26.0%	+6.22%, +2.93%	+37.7%, +25.4%	+9.33%, +5.87%

Table 4: The number of unique crashes found by four fuzzers. Dark gray indicates the fuzzer found the most unique crashes.

Application	AFL	CollAFL-x	PathAFL-np	PathAFL
bison	0	0.6	1.2	0.7
captoinfo	123	67	222	307
cflow	73	179	134	77
exiv2	1	1.4	0.9	1.5
tiff2pdf	1.4	1.5	1	2.1
readelf	0	1.1	1.1	0.9
Total	198.4	250.6	360.2	389.2

Table 5: Bugs found by fuzzers on LAVA-M dataset.

Program	Listed bugs	found by each fuzzer	
		AFL	PathAFL
base64	44	16	48
md5sum	57	0	0
uniq	28	9	19
who	2136	0	0
Total	2265	25	67

number of paths and edges explored by AFL and CollAFL-x, and the other columns show the percentage increase of PathAFL-np and PathAFL compared to each of them, respectively. Comparing with CollAFL-x, PathAFL-np on average explored 26% more paths and 2.93% more edges, while PathAFL explored 25.4% more paths and 5.87% more edges. PathAFL and PathAFL-np explored almost the same number of paths, but PathAFL explored more edges than PathAFL-np. We think PathAFL is better, it might touch more new code branches.

Path Coverage Growth over Time. Figure 4 shows the growth of code coverage of different fuzzers in 10 experiments. From Figure 4, we can see that the path coverage of PathAFL and PathAFL-np grows faster than AFL and CollAFL-x respectively. PathAFL and PathAFL-np have similar growth trends except `captoinfo`. They exceed AFL and CollAFL-x, with CollAFL-x growing faster than AFL. Again, it shows that our solution is effective.

The number of Unique Crashes. In these experiments, fuzzers trigger many program crashes. We counted the average number of triggered unique crashes, as shown in Table 4. Programs without crashes are not shown in this table. Dark gray indicates the fuzzer found the most crashes. It shows that CollAFL-x found the most unique crashes in `cflow` and `readelf`. PathAFL found the most unique crashes in `captoinfo`, `exiv2` and `readelf`, and it found the most total number of unique crashes in all the programs. Although unique crashes can dramatically over-count the number of actual bugs, resulting in inaccurate data, the number of crashes is still relevant.

In summary, PathAFL outperforms AFL and CollAFL-x in terms of paths and edges discovery. And it is better to use the power schedule. These expriments prove that the proposed solution could help improve the paths and edges coverage, and trigger more crashes. Notably, sometimes CollAFL-x is better. We think it's because PathAFL put too many seeds into the queue, which sometimes reduces efficiency.

5.3 Results on LAVA-M Dataset(RQ3)

LAVA-M is a well-known benchmark used for evaluating the fuzzers. It consists of four GNU coreutils programs: uniq, base64, md5sum, and who. Each program is injected with many bugs. Each injected bug has an unique ID, which is printed when the bug is triggered.

We evaluated AFL and PathAFL on the LAVA-M dataset for seven days. We ran each experiment five times and counted all of the found bugs. The evaluation results are presented in Table 5.

The results show that PathAFL found more bugs compared with AFL. Specifically, PathAFL found all the bugs found by AFL. In addition, PathAFL found all bugs injected in base64, and even found four new bugs that the author of LAVA-M did not list. The four unlisted bugs are 274, 521, 526, 527.

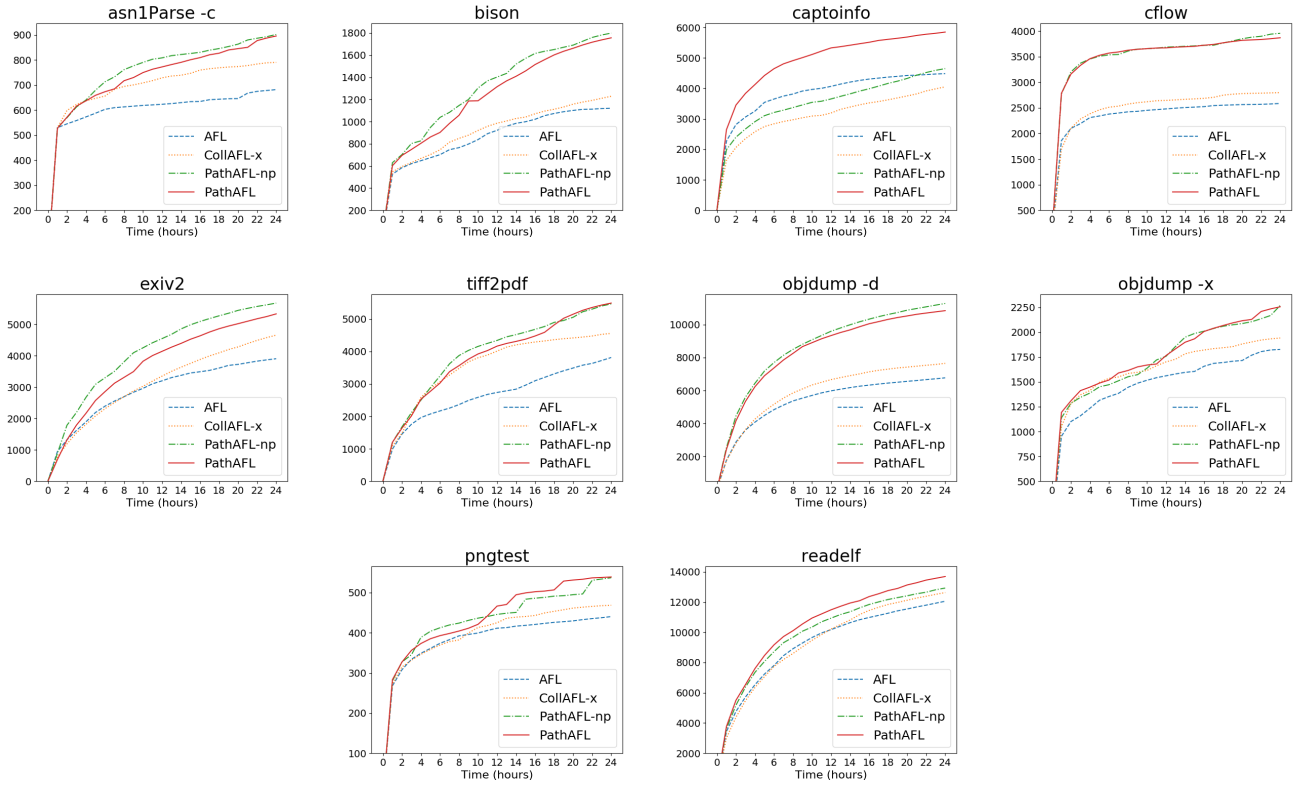

Figure 4: Number of paths explored over time in 24 hours.

5.4 Discovery of Real-world Vulnerabilities(RQ3)

We chose `binutils`, `libav` and `libtiff` for experiments. They are all maintained by the open source community. `Binutils` is widely used for the analysis of program binaries. It consists of several tools including `nm`, `objdump`, `readelf` and more. `Libav` provides cross-platform tools and libraries to convert, manipulate and stream a wide range of multimedia formats and protocols. `Libtiff` provides support for the TIFF (Tag Image File Format), a widely used format for storing image data. We found a lot of serious vulnerabilities and several bugs using PathAFL (listed in Table 6).

All of the bugs and vulnerabilities were previously unreported and could raise some security risks. We contacted the developers of these tools and submitted bug details.

Table 6: Bugs and vulnerabilities found by PathAFL

Bugs	Application	Comment
CVE-2018-1000876	Binutils-objdump	heap overflow
CVE-2018-20673	Binutils-libiberty	heap overflow
CVE-2019-14444	Binutils-readelf	interger overflow
Bug 1	Binutils-readelf	interger overflow
CVE-2019-14441	Libav-avconv	read access exception
CVE-2019-14442	Libav-avconv	hang, exhaust resource
CVE-2019-14443	Libav-avconv	floating point exception
Bug 2	Libtiff-tiffcp	write access violation

6 DISCUSSION

In this section, we discuss the limitations of our current design and possible future directions.

Tracing Path. Because of the well-known path explosion problem, PathAFL uses selective instrumentation and a simple hash algorithm that effectively reduces path tracing granularity. Given the available computing resources, we believe that this should be a good method. However, PathAFL only instruments some function entries. The selection of instrumented functions have some randomness, and some important paths may be lost. We think there might be other ways to choose the positions of instrumentation and trace to a better path.

Path Hash. The hash scheme uses only "additions". However, additions are commutative, meaning that the ordering between covered branches is ignored. Some other schemes can be used to better identify the ordering, despite that they may add more code and their effect needs to be further tested.

Filtering Path. PathAFL proposes a fast path filtering algorithm that reduces computation. It only uses path weight as the evaluation index sometimes results in limitations. The experiments in Section 5.2 shows that the number of paths increases very quickly, but the proportion of edges increases is less than that of paths. How to choose a good *h-path* is still a direction that needs further research.

7 RELATED WORK

In this paper, we introduced *h-path* and improved the effectiveness of fuzzing by adding high weight *h-path* to the seed queue. In this section, we focus on other related technologies.

Providing more guided information is a new trend on tuning the effectiveness of fuzzing. SeededFuzz[37] uses various program analysis techniques to facilitate the generation and selection of initial seeds which helps to achieve the goal of directed fuzzing. AFLGo[8] is a directed grey-box fuzzing system using a simulated annealing[21] power schedule to generate inputs that hold the trace closer to target program locations. Hawkeye[12] uses static analysis to collect information about the call graph, functions, and BBL distances to the target then evaluates exercised seeds based on both these information and the execution traces to generate the dynamic metrics, which are then used for seed prioritization, power scheduling and adaptive mutating. PTrix[14] works at the binary level by Intel Processor Tracing (PT)[2] technology, and directly decodes and splits PT trace as feedback for fuzzing, maintaining a new stronger feedback than edge-based code coverage. Differently, PathAFL uses light-weight instrumentation to trace path information, which is not very precise but can provide sufficient guidance to the fuzzers efficiently.

Some tools using artificial intelligence to generate good seeds. Learn&fuzz[18] shows how to automate the generation of an input grammar suitable for input fuzzing using sample inputs and neural-network-based statistical machine-learning techniques. Skyfire[36] learns a probabilistic context-sensitive grammar (PCSG) from the samples to describe grammatical characteristics and semantic rules, and generates well-distributed seed inputs for fuzzing programs that process highly-structured inputs. PathAFL chooses hige-weight *h-path*, actually adds more good seed files.

Symbolic execution[20] is another vulnerability discovery technique. Driller[35] uses selective concolic symbol execution to analyze large-scale programs by calling Angr[33] to generate good input when AFL is stuck. WildFire[28] first detects vulnerabilities by fuzzing independent functions in the program, and then validates these vulnerabilities using KLEE[10]. REDQUEEN[6] use input-to-state correspondence to solve magic bytes and checksums problem in symbolic execution without introducing any false positives. Symbolic execution has shown good effect in tests of small programs, but it is still difficult to scale up to large applications.

There are other motheds to generate good inputs. Fairfuzz[24] proposes a novel mutation mask creation algorithm to generate new mutation inputs that can hit a given rare branch. Choi et al.[15] propose grey-box concolic testing, which essentially casts grey-box fuzzing as a problem of path exploration. The proposed technique has the same motivation as PathAFL yet orthogonal to PathAFL. It presents a novel path-based test case generation method which leverages lightweight instrumentation to generate high coverage test cases.

In addition, some research focuses on limitations in the implementation of fuzzers. Xu et al.[38] design and implement three new operating primitives specialized for fuzzing that solve these performance bottlenecks and achieve scalable performance on multi-core machines. Jia et al.[19] find some limitations in the algorithm of AFL. To reduce fuzzing overhead, UnTracer[27] proposes coverage-guided tracing to trace only coverage-increasing test cases. Coverage-increasing test cases can self-report when a test case produces new coverage. PathAFL uses a new seed selection algorithm, avoiding an original disadvantage.

8 CONCLUSION

In this paper, we assist the fuzzer with path-coverage, discuss some of the challenges, and finally implement a new fuzzing system named PathAFL. It adopts coarse-grained path tracing and a fast filtering algorithm, which can effectively select higher weight *h-paths*, greatly increasing the number of path coverage. Moreover, we design a seed selection algorithm and a power schedule based on path weight. We evaluated it in several different experiments, and the results proved that PathAFL effectively improved edge coverage and found more unique crashes than AFL and CollAFL. In well-tested programs, PathAFL found 8 new security bugs with 6 CVEs assigned.

ACKNOWLEDGMENTS

Thanks to anonymous reviewers for feedback. This work is supported by the National Key R&D Program of China(2018YFA0704703), National Natural Science Foundation of China(61972215, 61702399, 61972073), Natural Science Foundation of TianJin(17JCZDJC30500).

REFERENCES

[1] 2012. Pin - A Dynamic Binary Instrumentation Tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.
[2] 2013. *Processor Tracing.* https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing.
[3] 2014. *American fuzzy lop.* http://lcamtuf.coredump.cx/afl/.
[4] 2015. *honggfuzz.* https://github.com/google/honggfuzz.
[5] 2019. *IDAPython.* https://www.hex-rays.com/products/ida/support/idapython_docs/.
[6] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence.. In *NDSS*, Vol. 19. 1–15.
[7] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX Annual Technical Conference, FREENIX Track*, Vol. 41. 46.
[8] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2329–2344.
[9] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2017. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering* 45, 5 (2017), 489–506.
[10] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.. In *OSDI*, Vol. 8. 209–224.
[11] Chen Chen, Baojiang Cui, Jinxin Ma, Runpu Wu, Jianchao Guo, and Wenqian Liu. 2018. A systematic review of fuzzing techniques. *Computers & Security* 75 (2018), 118–137.
[12] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2095–2108.
[13] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 711–725.
[14] Yaohui Chen, Dongliang Mu, Jun Xu, Zhichuang Sun, Wenbo Shen, Xinyu Xing, Long Lu, and Bing Mao. 2019. Ptrix: Efficient hardware-assisted fuzzing for cots binary. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. 633–645.
[15] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. 2019. Grey-box concolic testing on binary code. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 736–747.

[16] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. 2016. Lava: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 110–121.

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

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

[19] Chunfu Jia, Shengbo Yan, Zhi Wang, Chenlu Wu, and Hang Li. 2019. Method to improve edge coverage in fuzzing. *Journal on Communications* 40, 11 (2019), 76–85. https://doi.org/10.11959/j.issn.1000-436x.2019223

[20] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

[21] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. 1983. Optimization by simulated annealing. *science* 220, 4598 (1983), 671–680.

[22] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2123–2138.

[23] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 75.

[24] Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 475–485.

[25] Jun Li, Bodong Zhao, and Chao Zhang. 2018. Fuzzing: a survey. *Cybersecurity* 1, 1 (2018), 6.

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

[27] Stefan Nagy and Matthew Hicks. 2019. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 787–802.

[28] Saahil Ognawala, Fabian Kilger, and Alexander Pretschner. 2019. Compositional Fuzzing Aided by Targeted Symbolic Execution. *arXiv preprint arXiv:1903.02981* (2019).

[29] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. MoonShine: Optimizing {OS} Fuzzer Seed Selection with Trace Distillation. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 729–743.

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

[31] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing seed selection for fuzzing. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 861–875.

[32] Kosta Serebryany. 2016. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*. IEEE, 157–157.

[33] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 138–157.

[34] Maksim O Shudrak and Vyacheslav V Zolotarev. 2015. Improving fuzzing using software complexity metrics. In *ICISC 2015*. Springer, 246–261.

[35] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution.. In *NDSS*, Vol. 16. 1–16.

[36] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 579–594.

[37] Weiguang Wang, Hao Sun, and Qingkai Zeng. 2016. Seededfuzz: Selecting and generating seeds for directed fuzzing. In *2016 10th International Symposium on Theoretical Aspects of Software Engineering (TASE)*. IEEE, 49–56.

[38] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2313–2328.