Energy Distribution Matters

***Abstract*—AFL based greybox fuzzer is one of the most effective and scaled fuzzing tools to explore vulnerabilities or crashes in commercial off-the-shelf software. Existing research are trying to improve the abilities of these fuzzers to increase the coverage of program. However, none of existing fuzzing tools can explore exhaustively the entire space of realistic programs in practices, especially when the testing resource (time or computation) is limited. Thus, the strategy of distributing the fuzzing energy is important.**

Existing energy distribution strategies of AFL based fuzzers has two problems: (1) focus on the balance of energy distribution and increase coverage, lack of directiveness to strengthen fuzzing specific kinds of vulnerability promising areas. (2) focus on the distribution of the amount of mutations, but lack considering the schedule on proportions of different granularity mutation operators;

In this paper, we leverage two new insights to improve existing AFL's fuzzing energy distribution in a principled way. We directed AFL to stress fuzzing toward vulnerability-promising areas based on program static metrics. More specific, four kinds of vulnerability-promising areas (sensitive areas, complexity areas, deep areas and rare-to-reach areas) directed fuzzing are evaluated; And we schedule the distribution proportion of different granularity mutation operators. The proportion of mutations operations which has better ability to trigger new path will be increased gradually over time; Furthermore, all the improvements are integrated and implemented into an open source fuzzing tool named TAFL. Large scale experimental evaluation have show the effectiveness of each improvement and the performance of TAFL.

***Index Terms*—GreyBox Fuzzing, Directed fuzzing, Mutator Schedule**

I. Introduction

Fuzzing [21], as an automatic security testing technique, has become one of the most effective and scalable approaches to exploring vulnerabilities or crashes in commercial off-the-shelf software since its first introduction to analyze the reliability of UNIX utilities in early 1990s [25]. Nowadays, it has been widely used by mainstream software companies such as Google, Microsoft to improve their software's reliability and security. The core idea of fuzzing is to feed massive semi-valid inputs to the target program, and try to trigger unintended program behaviors (e.g. crash) by monitoring the performance of system under testing (SUT).

State of the art fuzzing approaches [36] [4] could be classified by three dimensions.

- ***Mutation vs. Generation.*** According to the ways of generating testcase, fuzzers could be divided into generation-based fuzzer or mutation-based fuzzer. Generation-based fuzzers such as Sulley [1] and Peach [8] start without inital seeds. They generate test case based on specification of input format and grammar (e.g protocol session model) that specify the message format and session states. They are more suitable to fuzzing programs that process highly-structured inputs. Mutation-based fuzzers such as AFL [16], AutoFuzz [15] and SecFuzz [38]) generate test inputs by modifying pre-provided seeds using various kinds of mutation operators (e.g byte flipping or adding anomalies). They can effectively fuzz program process compact and unstructured data formats.
- ***Stdin vs. File vs. Network.*** According to the channels of delivering test cases, fuzzing can be classified as stdin fuzzing, file fuzzing and network(or protocol) fuzzing. Where stdin fuzzing deliver input by standard input and output, file fuzzing read test case by opening and reading files, and protocol fuzzing send and receive test cases based on network communication.
- ***BlackBox vs. WhiteBox vs. GreyBox.*** According to the levels of utilizing interal knowledge of target program, fuzzers could be classified as BlackBox fuzzer, WhiteBox fuzzer and GreyBox fuzzers. Black-box fuzzers [1] [8] [12]) have no knowledge about the internals of the SUT and thus are less effective. White-box fuzzers usually apply heavy-weight program analysis such as taint-analysis [11] [40] or symbolic execution [35] [13] to improve the effectiveness, but may have scalability problems. GreyBox fuzzers such as AFL [16] and its variations (e.g. AFLFast [3], FairFuzz [20], AFLGo [2], CollAFL [10], etc.), LibFuzzer [18] are in the between. They usually apply light-weight program analysis and instrumentation to collect partial information of the SUT as feedback to drive fuzzing without sacrificing the fast execution speed and scalability ability.

Nowadays, feedback driven and mutation based greybox fuzzing tools, typical like AFL [16] and its variations (e.g. AFLFast [3], FairFuzz [20], AFLGo [2], CollAFL [10], etc.), LibFuzzer [18], honggfuzz [37], etc. have prove effective and promising in academic and industry areas based its mechanism of feedback based evolutionary loop. The main goal of greybox fuzzing is to detect more vulnerabilities effectiveness and efficiency in the process of search the entire program path states .

However, none existing greybox fuzzing tool could explore exhaustively the entire search space for realistic programs in practice, especially when the time and computation resource are limited. In order to make fuzzer more powerful, a lot of improvement work has been done, which can be catheterized into three directions:

- **Effectiveness**, which is aimed to improve the meta-ability of fuzzers to bypass obstacles and trigger vulnerabilities. It contains aspects such as (1) feedback accuracy [10] and

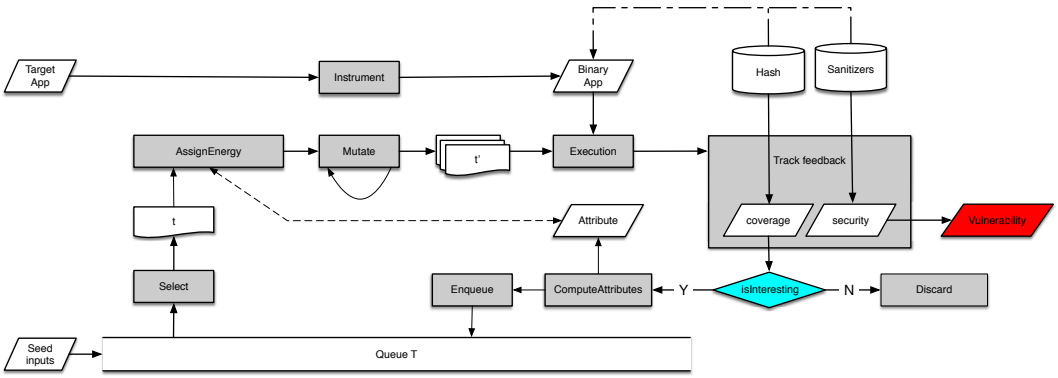

Fig. 1: The general workflow of AFL based greybox fuzzing

granularity [22]; (2) smart mutate strategies (i.e. where and what to mutate) [30] [27] [40] [6]; (3) sensitive to security violation [32] [34] and so on.

- **Efficiency**, which is aimed to improve performane to maximize the code coverage, in order to improve the probability of trigger vulnerabilities. It contains aspects like (1) providing high quality and diversity of intial seeds [39] [14] [26] [24]; (2) Improving the execution speed by prioritize faster and smaller seed, using new primitives [41] and utilizing system fork mechanism and hardware features (e.g Intel-PT) [31] [42]; (3) Balance the fuzzing energy distribution like low frequnce path and untouched path deserve more enery [3] [30] [10].
- **Directiveness**, which is aimed to make fuzzer coule be directed effectively for some specific goals. Typical, AFLGo [2] directed fuzzing to reach some specific location as soon as possible, and SlowFuzz [28] directed fuzzing to trigger specific kinds of bugs (i.e. algorithmic complexity vulnerabilities).

In this paper, we focus on AFL based greybox fuzzing. Two new insights about existing AFL based fuzzers' energy distribution are observed during the process of using these fuzzers.

Insight 1: The direct factor of vulnerability discovery is whether the area that vulnerability located are explored. If the vulnerability-related areas are prioritized to be fuzzed, then the vulnerability could be detected faster and more bugs may be found in same time. Existing improvement of the directiveness of greybox fuzzing is trying to direct fuzzing to reach a specific location (i.e. line of code), rather than towards promising kinds of areas where vulnerabilities may occur. There are some common institutions that sensitive areas, complex areas, deep areas, and rare-to-reach areas may have more chance to be vulnerable. Thus, directed fuzzing towards these vulnerability-promising areas could improve fuzzer's efficiency to detect more bugs.

Insight 2: Existing energy distribution strategy only consider the numbers of mutation for a seed. It does not take into account the schedule on distribution of different granularity mutation operations among assigned amount. Experimental studies have shown that coarse-grained mutations (e.g. extra mutators) has better ability to generate diversity test cases, which is more helpful to path growth; While fine-grained mutations will likely preserve the execution path to explore nearby. Thus, the proportion of different granularity mutation operators in the assigned mutation number should be considered and scheduled. The proportion of mutation operators which has better ability to trigger new path should be gradually increased over time.

Key Observation: All of the above limitations could be addressed by better energy distribution strategy.

Problem Statement: How to lavage the two insights to improve existing energy distribution strategy and improve the efficiency of AFL based greybox fuzzers ?

Our Work: We leverage two new insights to improve existing AFL's fuzzing energy distribution in a principled way. We directed AFL to stress fuzzing toward vulnerability-promising areas based on program static semantic metrics. More specific, four kinds of vulnerability-promising areas (sensitive areas, complexity areas, deep areas and rare-to-reach areas) directed fuzzing strategies are evaluated; And we schedule the distribution proportion of different granularity mutation operators. The proportion of mutations operations (i.e. extra mutators) which has better ability to trigger new path will be increased gradually over time; Furthermore, all the improvements are integrated and implemented into an open source fuzzing tool named TAFL. Large scale experimental evaluation have show the effectiveness of each improvement and the performance of TAFL.

In summary, our work makes following main contributions:

- We provided two new insights about the existing energy distribution limitation of AFL based greybox fuzzing.
- We improved the energy distribution strategy in a principled way by leverage the two new insights.
- We implemented and integrated our improvements into a new fuzzing tool named TAFL based on AFL-2.56b.
- We performed large scale experimental evaluation to show to show the effectiveness of our improvements and performance of integration.

The remainder of this article is structured as follows: Section II is introduction of background knowledge including the related work and detailed description of two new insights.

Section III give the details of the improvements method by leveraging the two insights. Section IV describes the implementation details. Section V is the experimental evaluation of each improvement and performance compared result with existing AFL based fuzzers. Finally, we conclude our work and discuss the future work in section VI.

II. Background

AFL based greybox fuzzing is our focus In this paper. So, an introduction of AFL itself and related improvement work based on AFL are presented firstly, followed by the detailed description of our new insights.

A. American Fuzzy Lop (AFL)

AFL based greybox fuzzing employs lightweight instrumentation to collect runtime coverage feedback to advance fuzzers. AFL [16] and its variations like AFLFast [3], FairFuzz [20], AFLGo [**?**], CollAFL [10], Angora [6] are typical state-of-art feedback based greybox fuzzers.

Figure 1 shows the workflow of AFL based grey box fuzzing. It firstly instrumented the target application for sake of collecting feedback later. Then it maintains a queue of testcases and performs a evolutionary fuzzing loop:

1) Select seed that worth fuzzing from the queue with a specific policy (e.g. according to the execution time, seed length, or hit number);
2) Calculate and assign energy for the selected seed based on its attributes (e.g. execute time, bitmap size, etc), where the energy represents the numbers of new test cases that generated by mutating the seed;
3) Mutate the seeds using different mutate operators to generate a batch of new test cases;
4) Feed these new test cases to target application and execute it at a high speed;
5) Track run-time feedback (e.g. coverage, security violation) and compute attributes of the test case.
6) Report vulnerabilities if security violations are detected;
7) Filter good test cases contributing to code coverage and put them into the queue, and go to step 1)

Following the continuous evolutionary loop, AFL could optimize the seeds that have contribution to coverage, and evolve towards a higher code coverage, furthermore improving the probability to trigger crashes.

```
1 cur_location = <COMPILE_TIME_RANDOM>;
2 shared_mem[cur_location ^ pre_locatiom] ++;
3 pre_location = cur_location >> 1;
```

Listing 1: AFL's Instrumentation

AFL's Instrumentation: AFL's instrumentation captures basic block transitions, along with coarse branch-taken hit counts. A sketch of the code that is injected at each branch point in the program is show in listing 1. The variable $cur\_location$ identifies the current basic block. Its random identifier is generated at compile time. Variable $shared\_mem[]$ is a 64 KB share memory region. Every byte that is set in the array marks a hit for a particular tuple (A, B) in the instrumented code where basic block B is executed after basic block A. The shift operation in Line 3 preserves the directionality [(A,B) vs. (B,A)]. A hash over $shared\_mem[]$ is used as the path identifier. AFL uses the coverage information to decide which generated inputs to retain for fuzzing, which input to be the priority choice and how long will the inputs be fuzzed.

Algorithm 1 AFL Greybox Fuzzing

```
   Input: seeds S, program P
 1: Queue ← S ;
 2: while true do
 3:   t ← Select(Queue);
 4:   for i from 0 to Length(t) do
 5:     t' ← DeterministicMutate(t, i);
 6:     runResult ← RunProgram(P, t');
 7:     if runResult is crash then
 8:       SaveCrashInfo(runResult);
 9:     end if
10:     if runResult is Interesting then
11:       AddToQueue(t', Queue);
12:       ComputeAttribute(t');
13:     end if
14:   end for
15:   energy ← AssignEnergy(t);
16:   for i from 0 to energy do
17:     t' ← MutateInput(t, i);
18:     runResult ← RunProgram(P, t');
19:     if runResult is crash then
20:       SaveCrashInfo(runResult);
21:     end if
22:     if runResult is Interesting then
23:       AddToQueue(t', Queue);
24:       ComputeAttribute(t');
25:     end if
26:   end for
27: end while
```

Algorithm 1 illustrates the detailed fuzzing process by means of AFL's implementation. If AFL is provided with seeds S, they are added to the $Queue$, otherwise an empty file is generated as a starting point.

AFL's Energy Distribution: AFL marks a seed as $favorable$ if it is the fastest and smallest input for any of the edges it exercises. The $favorable$ seeds will be selected priority and most of the non-favorite seeds will be ignored with some random probability(line 3). For each seed t, AFL determines the numbers of inputs that are generated by mutating t based on the seed's attributes (line 15). AFL's implementation of $AssignEnergy$ uses the attributes like the execution time, block transition coverage and so on. Then the fuzzer generate $energy$ new inputs by mutating t using various mutation operators (line 17).

AFL's Mutate Operators: AFL's implementation of $MutateInput$ applies some mutate operators(e.g. Flips, Interesting, Arith, Extra and Splice) to the seed, and generate

new inputs. In the havoc stage, AFL would mutate the seed by randomly choosing a sequence of mutation operators and apply them to random locations in the seed file. And in the determine stage, AFL applies a specific set of mutation operators to each input byte of the seed, and assign the similar energy to each seed based on its length. When a new input $t^{'}$ is generated, AFL run the target application using $t^{'}$ and track its feedback.

- **Flips:** simple bitflip, two bitfilp, four bitflip, etc.
- **Interesting:** set "interesting" byte, words, dwords.
- **Arith:** addition or subtraction of bytes, words or dwords.
- **Extra:** block deletion, block insert, overwrite and memset.
- **Splice:** splice two distinct input files at a random location.

AFL's Filter: If the generated input $t^{'}$ is considered to be interesting, AFL will runs a calibration stage to computed attributes of the input and then add the input into the queue. AFL's implementation of $IsInteresting$ return true depending on the numbers of times the basic block transitions, that are executed by $t^{'}$, have been executed by other sees in the queue. Intuitively, AFL retains inputs as new seeds that execute a new block transition or a path where a block transition is exercised twice when it is normally exercised only once. If the generated input $t^{'}$ crash the program, it is added to the set of crashing inputs. A crash input that is also interesting is marked as unique crash.

AFL's Seed Attribution: In the calibration stage, AFL will run the test case multiple times to determine the attributes like average execution time, size of bitmap and whether the associated path identifier changes across executions. These attributes will be used when assigning energy for the test case.

B. Improvements of AFL

In order to make AFL more powerful, researchers have done a lot of improment work, which could be categorized into three directions:

(1) **Effectiveness:** It is aimed to improve the meta-abilityies of fuzzers to bypass obstacles and trigger vulnerabilities. It contains aspects such as (1) feedback accuracy [10] and granularity [22]; (2) smart mutate strategies (i.e. where and what to mutate) [**?**] [6]; (3) sensitive to security violation [32] [34] and so on.

(2) **Efficiency:** It is aimed to improve performane to maximize the code coverage in order to improve the probability of trigger more vulnerabilities. It contains aspects like (1) providing high quality and diversity of intial seeds [39] [14] [26] [24]; (2) Improving the execution speed by prioritize faster and smaller seed, using new primitives [41] and utilizing system fork mechanism and hardware features (e.g Intel-PT) [31] [42]; (3) Balance the fuzzing energy distribution like low frequnce path, untouched path deserve more enery [3] [10].

(3) **Directiveness:** It is aimed to make fuzzer coule be directed effectively for some specific goals. Typical, AFLGo [2] directed fuzzing to reach some specific location as soon as possible.

Improvement of effectiveness, detailed improvement could be concluded as some aspects as follows:

- *Feedback granularity and accuracy:* Steelix [22] add instrumentation to collection the mutate progress for magic byte compassion, and continue to mutate on input has progress for each bit. CollAFL [10] demonstrated the inaccuracy of feedback (i.e. hash collision issue) in AFL will limit its efficiency of path discovery. They designed an algorithm to resove the has collision issue in AFL, improving the edge coverage accuracy with a low-overhead instrumentation scheme.
- *Smart mutate strategy:* The mutate strategy need to answer where to mutate and what to mutate. To answer where, [23] propsed a solution to identify sensitive bytes to mutate using static data lineage analysis. [29] proposes a deep neural network soulution to predicate which bytes to mutate, showing promising improvements. To answer What to mutate, Vuzzer [30] use dynamic analysis to infer interesting values (e.g., magic numbers to use for mutating.)
- *Sensitive to security violation:* Fuzzers often use program crashes as an indicator of vulnerabilities, because they are easy to detect even without instrumentation. However, programs do not always crash when a vulnerability is triggered, e.g., when a padding byte following an array is overwritten. Researchers have proposed several solutions to detect kinds of security violations. For example, the widely used AddressSanitizer [32] and MemorySanitizer [34] could detect buffer overflow and use-after-free vulnerabilities. There are many other sanitizers available, including UBSan [19], DataFlowsanitizer [7], ThreadSanitizer [33] and so on.

Improvement of efficiency, detailed improvement could be concluded as some aspects as follows:

- *Quality and diversity of initial seed:* Skyfire [39] learns a probabilistic context sensitive grammar from abundant inputs to guide seed generation. Learn & input [14] proposes a RNN (Recurrent Neural Network) solution to generate valide seed files, and could help generate inputs to pass format checks, improving the code coverage. Nicole Nichols et.al propose a GAN (Generative Adversarial Network) [24] solution to argument the seed pool with extra seeds, showing another promising solution.
- *Execution speed:* AFL utilizes the fork mechanism provided by Linux to accelerate, and further adopts the forkserver mode and persistent mode to reduce the burden of fork. In addition, AFL prioritizes seeds that are smaller and executed faster,and thus it is likely that more test cases could be tested in a given time. Moreover, AFL also supports a parallel mode, which enabling multiple fuzzer instances to collaborate with each other. kAFL [31] and PTfuzzer [42] use the hareware features to accelerate the execute speed. Wen Xu et.al. proposes several new primitives [41], speeding up AFL by 6.1 to 28.9 times.
- *Balance of fuzzing energy distribution*: AFLFast [3]prioritizes seeds exercising less-frequent paths and thus it is likely that cold paths could be tested thoroughly and

fewer energy will be wasted on hot paths in order to balance the energy on cold and hot path. FairFuzz [20] spend more energy on those rare reach branch. CollAFL [10] prioritizes seed that hit more untouched neighbors to improve the possibility to cover more new paths.

Improvement of directiveness, directed greybox fuzzing is modeled as a object optimizating problem, detailed improvement work are including:

- *Directiveness for specific location:* AFLGo [**?**] using distance metrics to direct fuzzing trigger or reproduce vulnerabilities in specific location.
- *Directiveness for specific kinds bug:* SlowFuzz [28] prioritizes seeds that use more resources (e.g., CPU, memory and energy), and try to increase the probability of triggering algorithmic complexity vulnerabilities.

C. New Insights

During the process of using AFL based fuzzers, we have some two new insights about existing AFL's fuzzing energy distribution, which motivated our work. In this section, detailed description about the two insights are illustrated.

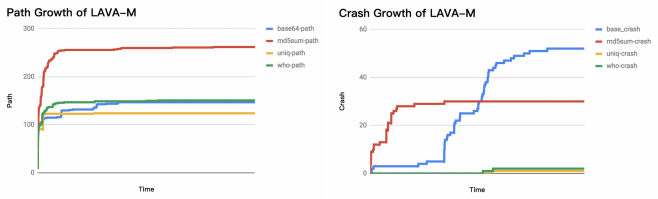

Fig. 2: path and crash growth over time

Insight 1: As shown as Fig.2, the growth of the path follows nearly the same pattern. It grows rapidly at the beginning, then growth rate gradually becomes flat over time, and finally it will become harder to grows. The natural reason is that the discovery problem of new path is a Coupon Collectors Problem(CCP) [9]. After finding $i-1$ new paths, the probability of finding the ith new path is $P_i = (N-i+1)/N$, where N is the total paths. Which means the difficult to find new path will become higher and higher. Meanwhile, the occur and increment pattern of crashes is different from the path growth, it may occur at any time, and the increment has no direct rules. The reason behind the phenomenon is that the direct cause of vulnerability discovery is not the increment of path coverage, but whether the vulnerability-related areas are explored sufficiently. If the vulnerability-related areas are prioritized to be fuzzed, then related vulnerabilities buried in these areas will be detected faster, furthermore more bugs will be found in same time. There are some common institutions that sensitive area, complex area, deeper area, and rare-to-reach area have more chance to be vulnerable. For institution, take following code fragment listed in the Listing 2 as an example to show the rationality of promising area deserve more energy. Consider the Path A and Path B in Listing 2, Path A contains more sensitive instruments like memcpy(), and more complex (it contains more instruments), and the bug is

```
#include "stdio.h"
int fun (const uint8_t *Data, size_t Size, bool e){
   if(e = true && Size ==22){                    // Path A
     uint64_t x = 0;
   int64_t  y = 0;
   int32_t z = 0;
     uint16_t a = 0;
     memcpy(&x, Data, 8);
     memcpy(&y, Data + 8, 8);
     memcpy(&z, Data + 16, sizeof(z));
     memcpy(&a, Data + 20, sizeof(a));

     if(x = 10 && y == 0xbaddcafedeadbeefUL)
             abort();

  }else if (e == false && Size < 22){     // Path B
     printf("%d\n", *Data);
      uint64_t x = 0;
      memcpy(&x, Data, 8);
      if(x>10)
          abort();
  }
}
```

Listing 2: Sample code of energy distribution

located deeper and rare-to-reach. Then we should spend more energy on Path A, no matter if there is bugs or not.

Thus, directed fuzzers prioritize fuzzing these vulnerability-promising areas and spend more fuzzing energy on these areas will improve the efficiency to detect vulnerabilities.

Insight 2: Existing energy distribution strategy do not consider the schedule about the distribution of different granularity of mutation operators used in assigned energy numbers. The energy refers to the numbers of new test cases generated by mutating the seed. Existing energy assignment in AFL based greybox fuzzers only consider the calculation of the numbers based on different attributes, such as the execute time, bitmap size, depth, hit numbers and so on. But they do not take into account the schedule about distribution of different granularity mutation operations in assigned amount. In determined stage, AFL will use various mutators sequentially. In havoc stage, AFL will randomly choose a certain mutation operators (i.e. bitflip, interesting, arith, extra, splice), and apply them on a seed to generate a new test case.

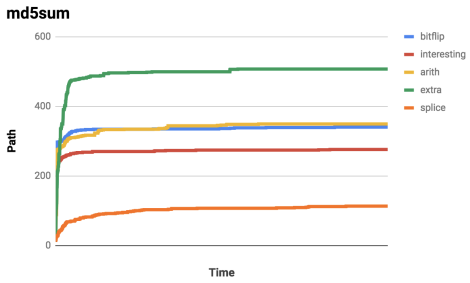

Fig. 3: path growth using different mutators

Experimental studies listed in Fig. 3 have shown that extra mutations has better ability to generate diversity testcase, which help the path to grow quickly, while splice mutators perform pool.

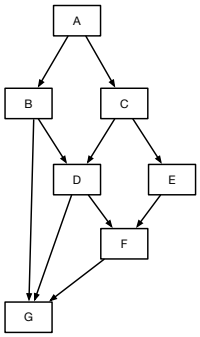

Fig. 4: Abstract CFG

Thus, the energy assignment strategies should consider not only the number of mutation, but also the proportion of different granularity mutation operators in the assigned number. the proportion of extra mutation operations should be gradually increased over time.

III. Improvements

Two new insights were leveraged to improve the fuzzing energy distribution in a principled way. The improvements contains two aspects: (1) Improvement of directing toward vulnerability promising areas. And (2) Improvement of schedule on distribution of different mutation operators. Finally, these improvements are integrated into a new tool in an organic way.

A. Improvement of Directing Toward Promising Areas

In order to improve the directiveness of AFL toward vulnerability promising areas, we borrow the idea of AFLGo, which directs AFL to reach some specific locations. The core ideas is extracting semantic metrics of basic blocks of target program using light-weight static analysis. Then instrumented these qualitative weight into the original program by compiler-time instrumentation. At run-time, the path reward (i.e. sum of the weight) will be traced. If the test case is interesting, we will assign the energy on the seed based on its path reward. The more reward, the more energy. By assigning more energy on those seed that gained more reward, fuzzing could be directed to stress fuzzing those more sensitive, complex, deeper and rare-to-reach areas. For the sake of institution, we illustrated the energy assignment by an abstract example as below Fig.4.

Let the weight of a basic block BB for a specific $metric$ be $Weight_{metric}(BB)$. Then the reward of a path that excised by a input t is denoted as follows:

$$Reward(t) = \frac{\sum_{BB_i \in Path(t)} Weight_{metric}(BB_i)}{BB_{num}} \quad (1)$$

It count the sum of every basic block's weight value along the path. Note that we will continue count the basic blocks in loops. For example, if test case t execute the path be $p = A \rightarrow B \rightarrow D \rightarrow G$, then the $Reward(t)$=($Weight_{metric}(A)$ + $Weight_{metric}(B)$ + $Weight_{metric}(D)$ + $Weight_{metric}(G)$)/4. When the execution is finished, we will update the maximize, minimize reward as well as the average reward so far at the same time. The average $AvgReward$ is calculated as flow:

$$AvgReward = \frac{MaxReward + MinReward}{2} \quad (2)$$

Furthermore, based on the path reward and average reward, we compute the factor $Factor$ used for energy assignment.

$$Factor(t_i) = \frac{Reward(t_i)}{AvgReward} \quad (3)$$

The $Factor$ is bigger, the more energy is assigned. In other words, the energy is positive related to the $Factor$. More specific, we design an exponential energy assignment strategy using below equation.

$$P(t_i) = P_{afl} * 2^{10*Factor(t_i)} \quad (4)$$

Furthermore, based on the institution that more sensitive, more complex, deeper, and rare to reach areas have more change to be vulnerabilities. Four specific metrics are designed and used to driven fuzzing toward four kinds of vulnerability promising areas. They are sensitive area, complex area, deeper area, rarer to reach areas

1) Towards Sensitive Areas: Based on the intuition that if a path is more sensitive operators like memory and string related instruments, then it is very likely to have more change to have memory corruption vulnerabilities. Thus, sensitive metric is designed to measure the sensitive degree of each basic block. More specifically, the measurement of sensitive metric is defined as follows:

$$SensitiveDegree(BB) = MemoryOP + StringOP \quad (5)$$

where $SensitiveDegree$ represent the numbers of memory and string related instruments in the basic block BB.

2) Toward Complexity Areas: Based on the intuition that if a path is more complex, then it is very likely to have more change to be vulnerable. Complex metrics is designed to measure the complexity of each basic block. For simply, we use numbers of total instruments in a basic block to define the complex metrics. Instrument numbers is the directly indicators of program complexity, if a basic block has more instruments, then it has more change to have issues. The measurement is as follows:

$$ComplexityDegree(BB) = InstNum \quad (6)$$

where $ComplexityDegree$ represent the total numbers of instruments in the basic block BB.

3) Toward Deep Areas: Based on the intuition that if a the path is deeper, then it has more chance to have vulnerability by exploring the deeper path. Depth metric are used to define the path deep degree. It represent the distance from the main function's entry block to the basic block itself. More specifically, the measurement of depth metric of basic block BB is in the following.

For a given basic block BB, we traverse all the paths P that from the entry block E of the function to the basic block B,

donated by $P = path(E, BB)$. Assume the distance of path $p_i \in P$ is d_i, then the depth of BB is as follows equation:

$$Depth(BB) = \frac{1}{\sum_{p_i \in P} \frac{1}{d_i}} \tag{7}$$

where $EntryBlock$ represents the entry block of a function that basic block BB located. Taking the Figure. 4 as example. In order to compute depth of basic block G, the CFG was traversed using DFS algorithm, and paths from entry block A to G are obtained. They are $p_1 = A \rightarrow B \rightarrow G$, $p_2 = A \rightarrow B \rightarrow D \rightarrow G$, $p_3 = A \rightarrow C \rightarrow D \rightarrow G$, $p_4 = A \rightarrow B \rightarrow D \rightarrow FrightarrowG$, $p_5 = A \rightarrow C \rightarrow D \rightarrow FrightarrowG$, $p_5 = A \rightarrow C \rightarrow E \rightarrow F \rightarrow G$. The length of above paths are 2, 3, 3, 4, 4. (i.e. $l_{p_1} = 2$, $l_{p_2} = 3$, $l_{p_3} = 3$, $l_{p_4} = 4$, $l_{p_5} = 4$. Then the depth of basic block G is $1/(1/2 + 1/3 + 1/3 + 1/4 + 1/4) = 3/5$. Note that the the distance computation based on intra-procedure static analysis without considering the function-level distance. It may sacrifices a certain accuracy for the conveniences of usage.

4) Toward Rare Reach Areas: Based on the intuition that if an area is rarer to be reached, then these area may has more chance to have issues, because the testing for these area may not sufficient. The metric called $RareReachDegree$ was defined to represent the rare reach degree of a basic block. If the rare reach degree is higher, the more energy should be spent on it. More specific, the measurement of rare reach degree is calculated as follows:

For a given basic block BB, we assign equal probability to all its outgoing edges. Hence, if $succ(BB)$ denotes the numbers of all succor basic blocks of BB, then $\forall b \in succ(BB), TPro(B, b) = 1/succ(B).size$, where $succ(B).size > 0$. We traverse all the paths P that from the entry block E of the function to the basic block BB, denoted by $P = path(E, BB)$.

For a path $p_i \in P$, the reachable probability of BB in path p_i is calculated as follows:

$$RPro(BB, p_i) = \frac{1}{\prod_{j=0}^{p_i.size-1} TPro(B_j, B_{j+1})} \tag{8}$$

Furthermore, we will calculate all the paths and finally get the rare reach degree of basic block BB using below equation:

$$RareDegree(BB) = \frac{1}{\sum_{i=0}^{P.size-1} RPro(BB, p_i)} \tag{9}$$

Taking the Figure.4 as an example. In order to compute depth of basic block G, we traverse the CFG using DFS algorithm, and get the paths from entry block A to D. They are $p_1 = A \rightarrow B \rightarrow D$, $p_2 = A \rightarrow C \rightarrow D$. Then the rare reach degree of B is $1/(1 * 1/2 + 1 * 1/2) = 4$.

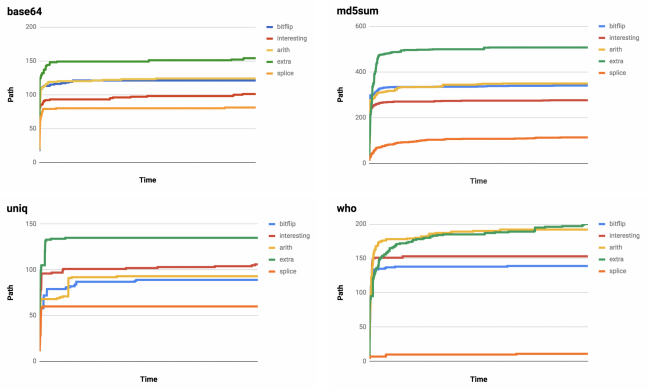

Fig. 5: Performance of Different Mutators

B. Improvement of Schedule on Mutators

Existing energy assignment strategies only consider the numbers of mutation, it does not take into account the distribution of different granularity mutation operations in assignment numbers. In order to add a reasonable schedule among the various granularity of mutation operations, we need to have an understanding of the ability of different granularity mutation operations to promote path growth firstly.

Empirical evaluation of AFL's five kinds of mutators was performed to answer above question. The result on LAVA-M data set is shown in Figure 5. It indicates that extra (i.e. block level deletion, insert and overwrite) mutation operators have better performance to help the path growth, while splice mutation operators perform pool.

Motivated by above observation, we proposed a schedule that improving the proportion of coarse mutation operators (i.e. extra mutators) over time, which has better ability to trigger new paths. More specific, the schedule algorithm on mutation operators is illustrated in algorithm 2.

Algorithm 2 MutateInput(): Schedule on Mutators

Input: s, the seed to be mutated.
Input: i, the number of generated new test case.
Output: t, the generated new test case.

1: $p \leftarrow iteratorNum(i) * 0.7$
2: β= 1/ $3^{process\_time}$;
3: $s' \leftarrow$ `RandomMutate`(s, $p * (1 - \alpha + \beta)$);
4: $t \leftarrow$ `ExtraMutate`(s', $p * (\alpha - \beta)$);
5: return t

where p represent the times of apply various mutators for generating a new test case. $process\_time$ refer to the execution time. α is a constant ratio to do the extra mutations. Overtime, the β become smaller and smaller, then proportion of extra mutators will become higher and higher. In practice, we set α equal to 0.3.

IV. Implementation

The two improvements were integrated into the AFL's existing energy distribution method, and a new fuzzing tool

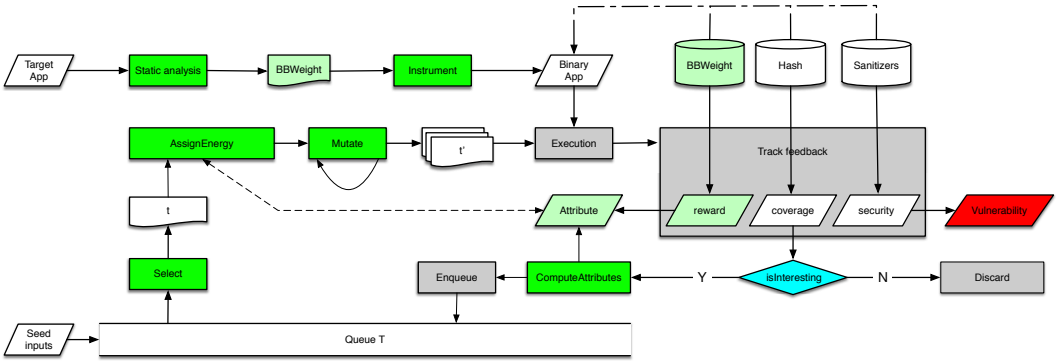

Fig. 6: Overview of TAFL

named TAFL with the integrated approach were implemented on top of the latest version of AFL. The overview of TAFL is illustrated as Fig. 6. Firstly, four kinds of area related semantic metrics of target program are extracted and instrumented. Then metric reward of path will be traced at runtime, and these reward feedback are used to distribute fuzzing energy. TAFL will prefer to select highly reward test case and assignment more energy on them, in order to direct stress fuzzing toward these vulnerability-promising areas represented by defined metrics. Furthermore, the schedule on mutators will increase the proportion of extra mutation operations overtime. The tool are available from https://github.com/stuartly/TargetFuzz, and the details of implementation was introduced in the following:

1) Metric Extraction: Each metric weight extraction is implemented as one llvm pass, they are used to get the weight of different metrics including sensitive degree, complex degree, depth and rare reach degree. These passes are is used and activated by the compiler afl-clang-preprocess. Each preprocess pass has a corresponding environment variable, they are GET_MEM_DENSITY, GET_INST_NUM, GET_DEPTH, and GET_ENTRY_DEGREE. When the environment variable CC is set to afl-clang-preprocess, and any of above llvm pass's environment is set, the related weight information of each basic block is generated and stored as text files after the project is built.

2) Compiler Instrumentation: The BB metric weight file contains the BB name and its weight information, it will be taken by the instrumentor to instruments the weight content into the target binary. Specifically, for each BB, it will injects an extended trampoline. The trampoline is a piece of assembly code that is executed after each jump instruction to keep track of the coverage control-flow edges. An edge is identified by a byte in a 64KB shared memory. On a 64-bit architecture, 16 additional bytes of shared memory are extended and used to record the feedback: 8 bytes to accumulate the weight, and 8 bytes to record the numbers of exercised basic blocks. The instrumentation is implemented as an extension of the AFL LLVM passs. When performing instrumentation, just set the compiler to afl-clang-fast and the compiler flags to reference the BB weight file, then build the target project.

3) Energy Distribution: TAFL fuzzes the instrumented binary according the integrate energy distribution strategy. The additional 16 bytes in the shared memory informs the fuzzer about the reward of current test case. The current test case's reward is computed by dividing the accumulated BB weight for different metrics by the numbers of exercised BBs. We will select those test cast that run faster and get higher reward and assignment energy based on seed's runtime reward. In addition, we modify the implementation of MutateInput and improve the proportion of extra mutators over time.

V. Evaluation

In this section, experimental evaluation of our improvements and integration was performed on well known benchmarks. We will firstly state the evaluation setup environments including the benchmark, compared tools and research questions. And then describe and explain the evaluation results of each improvement and integration performance.

A. Evaluation Setup

Evaluation Datasets: A widely-used benchmark (i.e. LAVA-M) and some popular open source projects are selected for the evaluation. The programs in selected benchmarks are known to have certain vulnerabilities, and hence form a ground-truth corpora for evaluation of fuzzing tools.

- **LAVA-M Dataset**. LAVA-M consist of 4 buggy version of Linux utilities, i.e., base64, md5sum, uniq and who. It was generated by automatically injecting hard-to-reach bugs into existing program source code [17]. And it was designed as a benchmark for evaluating the bug detection capability of fuzzers. Recend fuzzers (e.g. VUzzer [30], CollAFL [10], and Angora [5] all used this benchmark.
- **Real-World Projects**. Some popular open source Linux applications, including well-known tools (e.g., GUN libtasn1), document process libraries(e.g. libxml2), image processing libraries(e.g. libtiff) and so on. They are chosen based on the following features: popularity in the community, development activeness, and diversity of categories.

Evaluation Tools: We will compared TAFL with AFL based source level instrumented fuzzers with the features of energy distribution. More specifically, these tools are as following:

TABLE I: Effectiveness of Directing Towards Promising Areas

App	AFL			TAFL -sen			TAFL -com			TAFL -dep			TAFL -rare		
	first	crash	path	first	crash	path	first	crash	path	first	crash	path	first	crash	path
base64	23.15	53	155	-34.64%	+30.18%	-15.48%	-38.83%	+28.30%	-4.51%	-92.74%	+15%	-2.5%	-16%	+66%	+87.4%
md5sum	1.95	32	370	-18.22%	+21.87%	+2.43%	-35.44%	-12.5%	-2.43%	-52.91%	+31.25%	+4.32%	-28.86%	+15.62%	+5.13%
uniq	1072.65	1	126	-29.95%	0%	0%	-7.07%	0%	+1.28%	-84.81%	0%	+4.76%	-13.51%	0%	+6.34%
who	937.31	2	202	-6.18	0 %	+7.9%	-29.66%	0%	-7.4%	-15.37%	0%	+10.89%	+4.36%	0%	+1.98%
libxml2-2.9.2	534.61	12	6080	-60.84%	+41.66%	+5.50%	+20.54%	-33.33%	+3.79%	-35.90%	+133.33%	+7.36%	-56.82%	+216.66%	+12.61%
libtiff-3.7.0	0.08	52	469	0%	+21.53%	+15.35%	0%	+17.30%	+10.87%	0%	+36.53%	+11.08%	0%	+23.07%	+15.99%
bison-3.0.4	52.14	161	3591	-95.43%	+18.01%	+8.71%	-90.64%	+4.96%	+4.42%	-58.38%	+31.67%	+3.70%	-72.13%	+19.87%	+29.15%
cflow-1.5	22.65	166	1331	-68.78%	+6.02%	-0.52%	-55.05%	+44.57%	+2.55%	+51.74%	-3.61%	-0.75%	-58.41%	+5.42%	+0.75%
libjpeg-tubo-1.2.0	117.25	22	2908	-18.43%	-18.18%	-3.25%	-53.72%	+104.54%	+8.70%	-108.31%	-18.18%	-10.24%	+10.53%	-45.45%	-3.12%
Average	-	-	-	-36.94%	+15.05%	2.29%	-32.21%	+17.09%	1.91%	-20.01%	+25.12%	+3.17%	-28.27%	+ 31.26%	+17.39%

- **AFL-2.52b**: it is the latest official version of AFL.
- **AFLFast:** it is an variant of AFL with the feature of spending more energy on the low frequent path.
- **FairFuzz:** it is an extension of AFL with the feature of targeting rare reach branch.
- **TAFL:** It is an modification of AFL integrated the improvements of directing towards four kinds of promising areas and scheduling mutation operators of different granularity.

Experimental Infrastructure: We evaluated these fuzzers on collected benchmarks with a same configuration, i.e,. a virtual machine configured 8 core of 2GH Intel CPU and 8 GB RAM, running ubuntu 16.04.

Research Questions: we will evaluate our improvements and the performance of TAFL. And trying to answer following research questions.

- **RQ1:** The effectiveness of directing towards four kinds of vulnerability promising areas.
- **RQ2:** The effectiveness of schedule on multi-granularity mutation operators.
- **RQ3:** The performance of integrated TAFL compared with existing mainstream AFL based fuzzers.

B. Evaluation of Directing Towards Promising Areas

The effectiveness of directing fuzzing toward four kinds of vulnerability promising areas are evaluated. Furthermore, we compared the four kinds of areas based guidance strategy with original AFL. Note that the TAFL used in this evaluation is only integrated into the improvement for directivenss, without scheduling on mutators. Every project was run 10 times for 24 hours, in order to reduce the random factors of fuzzing. The results are illustrated in Table I. Three measure metrics (i.e. Time of triggering the first blood, Improvement of total unique crashes, Improvement of total paths) are collected to state the effectiveness and performance of four kinds of vulnerability promising areas directed search strategies.

1) First Blood : Time to trigger the first crash is one of the important factor to the success and efficiency of directed fuzzing tools. As we could see from Table I, direct fuzzing towards promising areas can effectively improve the time to find the first crash in most cases. The average improvement time for detecting first blood is 36.94%, 32.21%, 20.91%, 28.27% for four kinds of directed areas. Generally speaking, sensitive areas guided fuzzing has better performance to trigger first crash on selected benchmarks. Especially in some specific case like bison-3.0.4, sensitive area directed strategy could reach 95.43% improvement.

2) Unique Crashes: One of the most important factor to fuzzers is the numbers of unique crashes it founded in limit time. Some crashes may be caused by a same root cause, and some are even not security vulnerabilities. But in general, there is a better probability that more vulnerabilities could be identified if the more crashes are detected. In AFL, different paths to same crash point are marked as different unique crash. Thus unique crashes metric also show the explored degree of vulnerable areas. The metric to indirectly prove the effectiveness of the improvement of directiveness. As we can see from Table I. Four kinds of areas directed strategies obtain more unique crashes in moset cases, the average improvement are 15.05%,17.09%, 25.12%, 31.26% for sensitive areas, complex area, deep area and rare-to-reach areas. The result prove the effectiveness of our improvement to directing fuzzing toward promising areas. Among four kinds of target areas directed strategies, rare-to-reach areas directed strategy perform best in average, more specifically in case libxml2, the improvement of reach 216.66%.

3) Total Path: Total path is another important factor to show the fuzzers' ablility in limit time. We static the total path of four kinds of directiveness. The average improvement are 2.29%, 1.91%, 3.17% and 17.39%. The result show that rare-to-reach area guide strategies are more helpful to path growth. And conclude from above results, directing fuzzing towards rare-to-reach areas seems has the best performance in general cases.

C. Evaluation of Schedule on Mutators

Code coverage is one of the important factors to the success of fuzzers, it is also the most important metric to show the performance of different distribution strategies of various mutation operators. Thus, we use two coverage related metrics (i.e. path growth over time, improvement of total path) to validate the effectiveness of our schedule on mutators. Evaluation was performed on our benchmark, every project was run 10 times for 24 hours for sake of reducing the random factors of fuzzing. The result is collected and illustrated in Table III.

1) Path Growth Over Time: The path growth is a directly indicator of fuzzer's ability to explore new paths. Due to the limited space of the article, we selected the results on the LAVA-M data set to illustrate the effectiveness of schedule on the mutation operation for path growth. For more evaluation

TABLE II: Performance of TAFL

App	AFL			AFLFast			FairFuzz			TAFL -rare		
	first blood	crashes	paths	first blood	crashes	paths	first blood	crashes	paths	first blood	crashes	paths
base64	23.15	53	155	59.46	52	139	0	0	134	20.3	83	288
md5sum	3.95	32	370	2.13	37	378	5.81	30	301	3.23	43	412
uniq	1072.65	1	126	901.01	1	132	0	0	134	717.19	1	144
who	937.31	2	202	890.33	2	209	0	0	206	915.54	2	216
libxml2-2.9.2	534.61	12	6080	560.23	17	6402	826.7	22	6410	330.81	41	7331
libtiff-3.7.0	0.08	52	469	0.08	52	512	0.08	62	490	0.08	68	571
bison-3.0.4	52.14	161	3591	42.23	208	4324	5.68	119	3214	19.53	196	4709
cflow-1.5	22.65	166	1331	5.81	177	1321	33.52	204	1197	7.42	173	1412
libjpeg-tubo-1.2.0	117.25	22	2908	58.5	12	3087	786.03	4	1681	85.91	12	2978
Average	337.08	55.66	1692.44	279.97	62	1833.77	276.34	49	1529.66	248.18	68.77	2006.78
Improvement	-	-	-	-16.94%	+11.39%	+8.35%	-18.01%	-11.97%	-9.62%	-26.37%	+23.55	+18.57%

results, please refer to the website (https://sites.google.com/view/tafl). We modify original AFL and AFLFast to add the improvement of schedule on mutation operators, and evaluate them on LAVA-M Benchmark. The result is shown as Figure 7 in the following.

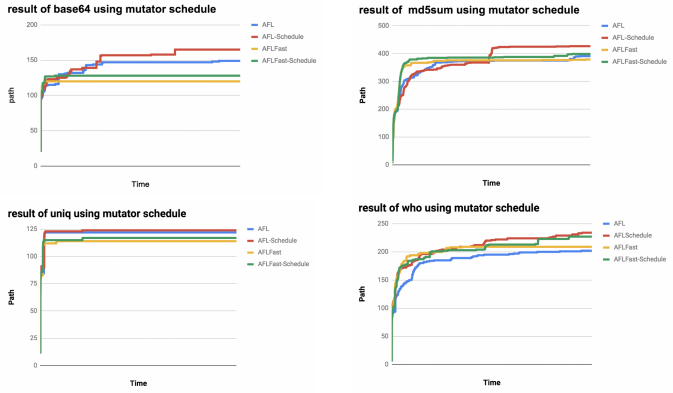

Fig. 7: Result of LAVA-M using mutator schedule

As we can seed from Figure 7, improving the extra mutatos over time is helpful to maximize the code coverage. It indicates the effectiveness of scheduling different granularity mutation operators. In addition, we could conclude that AFLFast is not always better than AFL, AFLFast will has quickly path growth at the beginning, but AFL will catch up later. And in some case such as based64, AFLFast's execute speed will slow down to 0 after a certain time.

2) Improvement of Total Path: Furthermore, the proportion of path growth after using the mutation operation schedule was counted, and the result is illustrated in Table III.

TABLE III: Effectiveness of Schedule on Mutators

Application	AFL	AFL-schedule	AFLFast	AFLFast-Schedule
base64	155	+10.7%	120	+6.6%
md5sum	370	+5.2%	378	+8.9%
uniq	126	+3.2%	114	+2.6%
who	201	+15.8%	219	+8.6%
libxml2-2.9.2	6080	+14.7%	6402	+13.9 %
libtiff-3.7.0	469	-0.9%	514	+2.5%
bison-3.0.4	3591	+9.4%	4324	+4.7%
cflow-1.5	1331	+1.6%	1294	+2.7%
libjpeg-tubo-1.52	2908	+3.6%	3087	+1.3%
Average	-	+7.03%	-	+4.82%

As can be seen from theTable III, the schedule on mutator is effective to improve the code coverage in most cases. The average increase ratio is 7.03% and 4.82% for AFL and AFLast. In some good examples, the promotion ratio has reached 15.8% and 14.7%, but also in some bad case the reduce ratio reach -0.9%.

D. Evaluation of Integrated Performance

Finally, we evaluate the performance of improvements integration by comparing with existing source level AFL based greybox fuzzing tools(i.e. AFLFast and FairFuzz). Similarly, three metrics are used measure the performance of TAFL, they are (1) Time to trigger first crash; (2) Improvement of total crashes; And (3) Improvement of total paths. Each project was run 10 times for 24 hours, in order to reduce the random factors of fuzzing, result is collected and illustrated in Table II.

1) First Blood: As we can see from the Table II, the integration of improvements has effect to improve the time to trigger the first crash. In general, the time to trigger the first crash was improved 26.37% after using mutator schedule, which is better thatn AFLFast and FairFuzz.

2) Unique Crashes: As we can see from the Table II, compared with original AFL, AFLFast and TAFL-rare detect more crashes in 2 4 hours on selected benchmarks. The improvement are 11.39% and 23.55%. TAFL is twice as more as AFLFast. Meanwhile, FairFuzz perform not well, especially on LAVA-M benchmark, it can not detect crashes in most of the programs, but it get better result on cflow-1.5.

3) Total Paths: As we can see from the Table II, The target area guidance and mutator schedule strategy is helpful to improvement on the total path. On selected benchmark, the average improvement of TAFL-rare is 18.57%. While the average improvements fo AFLFast and FairFuzz are 8.35% and -9.62%.

VI. Conclusion and Future Work

We leverage two new insights to improve existing AFL's fuzzing energy distribution in a principled way. We directed AFL to stress fuzzing toward vulnerability-promising areas based on program static metrics. More specific, four kinds of vulnerability-promising areas (sensitive areas, complexity areas, deep areas and rare-to-reach areas) directed fuzzing are evaluated; And we schedule the distribution proportion of different granularity mutation operators. The proportion of mutations operations which has better ability to trigger new path will be increased gradually over time; Furthermore, all the improvements are integrated and implemented into an open source fuzzing tool named TAFL. Large scale experimental

evaluation have show the effectiveness of each improvement and the performance of TAFL.

REFERENCES

[1] P. Amini and A. Portnoy, "Sulley-pure python fully automated and unattended fuzzing framework," *May*, 2013.

[2] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2329–2344.

[3] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 1032–1043.

[4] C. Chen, B. Cui, J. Ma, R. Wu, J. Guo, and W. Liu, "A systematic review of fuzzing techniques," *Computers & Security*, 2018.

[5] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," *arXiv preprint arXiv:1803.01307*, 2018.

[6] P. Chen, H. Chen, Y. Zhang, J. Dai, X. Zhang, S. Huang, Z. Yang, M. Yang, H. Chen, W. Han *et al.*, "Angora: efficient fuzzing by principled search," in *IEEE Symposium on Security\& Privacy*, vol. 14. Springer-Verlag New York, 2013, pp. 117–149.

[7] DataFlowSanitizer, "https://clang.llvm.org/docs/dataflowsanitizer.html," 2018.

[8] M. Eddington, "Peach fuzzing platform," *Peach Fuzzer*, p. 34, 2011.

[9] M. Ferrante and M. Saltalamacchia, "The coupon collectors problem," *Materials matemàtics*, pp. 0001–35, 2014.

[10] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *CollAFL: Path Sensitive Fuzzing*. IEEE, p. 0.

[11] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 474–484.

[12] H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck, "Pulsar: stateful black-box fuzzing of proprietary network protocols," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2015, pp. 330–347.

[13] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: whitebox fuzzing for security testing," *Queue*, vol. 10, no. 1, p. 20, 2012.

[14] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 50–59.

[15] S. Gorbunov and A. Rosenbloom, "Autofuzz: Automated network protocol fuzzing framework," *IJCSNS*, vol. 10, no. 8, p. 239, 2010.

[16] http://lcamtuf.coredump.cx/afl/, "Afl."

[17] http://moyix.blogspot.com/2016/10/the-lava-synthetic-bug corpora.html, "Lava-m."

[18] L. C. Infrastructure, "libfuzzer: a library for coverage-guided fuzz testing," 2017.

[19] B. Lee, C. Song, T. Kim, and W. Lee, "Type casting verification: Stopping an emerging attack vector." in *USENIX Security Symposium*, 2015, pp. 81–96.

[20] C. Lemieux and K. Sen, "Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage," *arXiv preprint arXiv:1709.07101*, 2017.

[21] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a survey," *Cybersecurity*, vol. 1, no. 1, p. 6, 2018.

[22] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: program-state based binary fuzzing," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 627–637.

[23] Z. Lin, X. Zhang, and D. Xu, "Convicting exploitable software vulnerabilities: An efficient input provenance based approach," in *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*. IEEE, 2008, pp. 247–256.

[24] C. Lv, S. Ji, Y. Li, J. Zhou, J. Chen, P. Zhou, and J. Chen, "Smartseed: Smart seed generation for efficient fuzzing," *arXiv preprint arXiv:1807.02606*, 2018.

[25] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[26] N. Nichols, M. Raugas, R. Jasper, and N. Hilliard, "Faster fuzzing: Reinitialization with deep neural models," *arXiv preprint arXiv:1711.02807*, 2017.

[27] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: fuzzing by program transformation," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 697–710.

[28] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2155–2168.

[29] M. Rajpal, W. Blum, and R. Singh, "Not all bytes are equal: Neural byte sieve for fuzzing," *arXiv preprint arXiv:1711.04596*, 2017.

[30] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[31] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kafl: Hardware-assisted feedback fuzzing for os kernels," in *Adresse: https://www. usenix. org/system/files/conference/usenixsecurity17/sec17-schumilo. pdf (besucht am 10. 08. 2017)*, 2017.

[32] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-sanitizer: A fast address sanity checker." in *USENIX Annual Technical Conference*, 2012, pp. 309–318.

[33] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: data race detection in practice," in *Proceedings of the workshop on binary instrumentation and applications*. ACM, 2009, pp. 62–71.

[34] E. Stepanov and K. Serebryany, "Memorysanitizer: fast detector of uninitialized memory use in c++," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2015, pp. 46–55.

[35] 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 *NDSS*, vol. 16, 2016, pp. 1–16.

[36] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[37] R. Swiecki, "Honggfuzz," *Available online at: http://code. google. com/p/honggfuzz*, 2016.

[38] P. Tsankov, M. T. Dashti, and D. Basin, "Secfuzz: Fuzz-testing security protocols," in *Automation of Software Test (AST), 2012 7th International Workshop on*. IEEE, 2012, pp. 1–7.

[39] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 2017, pp. 579–594.

[40] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *Security and privacy (SP), 2010 IEEE symposium on*. IEEE, 2010, pp. 497–512.

[41] W. Xu, S. Kashyap, C. Min, and T. Kim, "Designing new operating primitives to improve fuzzing performance," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2313–2328.

[42] G. Zhang, X. Zhou, Y. Luo, X. Wu, and E. Min, "Ptfuzz: Guided fuzzing with processor trace feedback," *IEEE Access*, 2018.