Energy Distribution Matters in Greybox Fuzzing

***Abstract*—American Fuzzy Lop (AFL) is one of the most effective fuzzing tools to explore vulnerabilities in commercial off-the-shelf software. Existing works improve AFL's abilities to maximize code coverage of tested programs. However, none of them could explore the entire space of real-world applications in practice exhaustively, especially when testing resources (time or computation) are limited. Thus, the strategy of distributing fuzzing energy is essential. Existing energy distribution strategies of AFL and its variants have two limitations. (1) They focus on increasing coverage, but lack guidance to direct the fuzzing tool to approach code regions that are more likely to be vulnerable. (2) Although the mutation number of a seed can be adjusted, existing works randomly select mutators and therefore lack insights regarding which kind of mutators are more helpful at that particular stage.**

We leverage these two new insights to improve AFL's fuzzing energy distribution in a principled way. We direct the fuzzer to strengthen fuzzing toward regions that have higher chance to contain vulnerabilities based on static semantic metrics of the target program. Furthermore, granularity-aware scheduling of energy distribution is proposed, which dynamically assigns ratio to different mutation operators based on their current performance. We implemented these improvements as an extension to AFL. Large-scale experimental evaluations showed the effectiveness of each improvement and performance of integration. The proposed tool has helped us find five unknown bugs and identify one new CVE in Libtasn1-4.13.

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

I. Introduction

Fuzzing [25], as an automatic testing technique, has become one of the most effective and scalable approaches to exploring vulnerabilities or crashes in commercial off-the-shelf software. Nowadays, it has been widely used by mainstream software companies such as Google [20] and Microsoft [21] 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 system under testing (SUT).

State of the art fuzzing approaches [39] [4] could be classified along three dimensions.

- ***Mutation vs. Generation.*** According to ways of generating test cases, fuzzers could be divided into generation-based and mutation-based. Generation-based fuzzers such as Sulley [1] and Peach [8] start without initial seeds. They generate test cases based on specifications of input format and grammar (e.g., protocol session model) that specify message format and session states. They are more suitable for fuzzing programs that process highly-structured inputs. Mutation-based fuzzers such as AFL [16], AutoFuzz [15] and SecFuzz [41]) generate test inputs by mutating pre-provided seeds using various kinds of mutation operators. They could effectively fuzz programs that process with and unstructured data formats.
- ***Stdin vs. File vs. Network.*** According to channels of delivering test cases, fuzzing can be classified into stdin fuzzing, file fuzzing, and network(or protocol) fuzzing. Stdin fuzzing provides test cases by standard input and output, file fuzzing provides test cases by opening and reading files, and protocol fuzzing sends and receives test cases based on network communication.
- ***BlackBox vs. WhiteBox vs. GreyBox.*** Depending on the knowledge to the interal of target programs, fuzzers could be classified into BlackBox, WhiteBox and Grey-Box fuzzers. Black-box fuzzers [1] [8] [12]) have no knowledge about internals of SUT, and thus relatively less effective. White-box fuzzers usually apply heavy-weight program analysis such as taint-analysis [11] [43] or symbolic execution [38] [13] to improve effectiveness, but may suffer from scalability problems. Grey-Box fuzzers such as AFL [16] and its variations (e.g. AFLFast [3], FairFuzz [24], AFLGo [2], CollAFL [10], etc.), LibFuzzer [22] are in between. They apply light-weight program analysis and trivial instrumentation to collect coverage information of the SUT as feedback. Then these feedbacks are used to drive fuzzing. GreyBox fuzzers improve fuzzing efficiency without significantly sacrificing execution speed and scalability.

State-of-the-art feedback-driven and mutation-based grey box fuzzing tools include AFL [16] and its variations (e.g., AFLFast [3], FairFuzz [24], AFLGo [2], CollAFL [10], etc.), LibFuzzer [22], honggfuzz [40], etc. They have proved extremely effective and promising in detecting vulnerabilities because of the improved code coverage.

However, none of existing grey box fuzzing tool could explore entire search space exhaustively for real-world programs in practice, especially when time and computation resources are limited. To make a fuzzer more powerful, many improvement works have been done, which could be categorized into three directions:

- **Effectiveness.** It aims to improve meta-abilities of fuzzers to bypass obstacles and trigger vulnerabilities. It contains aspects such as (1) feedback accuracy [10] and granularity [26]; (2) mutation strategies (i.e. where and what to mutate) [33] [30] [43] [6]; (3) sensitiveness to security violation [35] [37] and so on.
- **Efficiency.** It aims to improve performane on maximizing code coverage, and improve probability of triggering vulnerabilities. It contains aspects like (1) providing high quality and diversity of intial seeds [42] [14] [29] [28];

(2) improving execution speed by prioritizing faster seed, using new primitives [44] and utilizing system fork mechanism and hardware features (e.g Intel-PT) [34] [45]; (3) balancing fuzzing energy distribution like low-frequence and untouched path deserves more energy [3] [33] [10].

- **Guidance.** It aims to make fuzzing be directed effectively for some specific goals. AFLGo [2] directs fuzzing to reach some specific location (i.e., line of code) as soon as possible. SlowFuzz [31] directs fuzzing to trigger specific kinds of bugs (i.e., algorithmic complexity vulnerabilities).

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

Insight 1: One crucial factor of vulnerability discovery is whether regions that vulnerabilities reside are explored. If vulnerability-residing regions are prioritized and strengthened to be fuzzed, then vulnerabilities could be detected faster and more bugs may be found during same period of time. Existing improvements on guidance are trying to direct fuzzing to reach a specific location (i.e., the line of code) given in advance, rather than towards promising regions that are more likely for a vulnerability to reside. There are some understanding that sensitive regions(i.e., regions which contain more memory or string operators), complex regions (i.e. code regions with high complexity), deep regions, and rare-reach regions of a program may have more chance to be vulnerable. Thus, guided fuzzing through optimized energy distribution towards these promising vulnerable regions could improve fuzzers' efficiency and detect more bugs.

Insight 2: Existing energy distribution strategies only tune the mutation number, and randomly select mutation operators. Empirical studies have shown that coarse-grained mutators (e.g., extra mutators) have better ability to generate diversity test cases, which is helpful for path growth, while fine-grained mutators tend to perform better in exploring nearby execution paths. Therefore, a better strategy is to adjust the proportion of different granularity mutation operators over time. If certain kinds of mutators perform better, we increase the ratio of these mutators, and vice versa.

Key Observation: Both insight 1 and insight 2 imply that energy distribution matters in greybox fuzzing.

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

Our Work: We leverage the two new insights to improve existing AFL's fuzzing energy distribution in a principled way. We direct fuzzers to stress fuzzing toward regions that are more likely for a vulnerability to reside based on static semantic metrics of target program. More specifically, four kinds of promising vulnerable regions (i.e., sensitive, cpmplex, deep and rare-reach regions) are evaluated. And a granularity-aware scheduling of different mutation operators is proposed. The ratio of certain kinds of mutation operators is increased gradually, if they have better ability to trigger new paths. All improvements are integrated and implemented into an newly developed open source fuzzing tool named TAFL. Large-scale experimental evaluations have shown effectiveness of each improvement and the integration. Furthermore, TAFL helps us find five unknown bugs and one new CVE in Libtasn1-4.13 [19].

In summary, contributions of our work are as follows:

- **Insights.** We provide two new insights into the energy distribution of AFL based greybox fuzzing.
- **Improvements.** We improve the energy distribution strategy in a principled way by leveraging two new insights. More specifically, we improve guidance by directing fuzzing toward four kinds of promising vulnerable regions, and scheduling of different kinds of mutation operators.
- **Tool.** We implement and integrate our improvements into a new fuzzing tool named TAFL, which could be accessible from: https://sites.google.com/view/tafl/tool.
- **Vulnerabilities.** We perform large-scale experimental evaluations to show effectiveness of each improvement and performance of integration. Furthermore, TAFL helps us to find five unknown bugs and one new CVE in Libtasn1-4.13.

The remainder of this article is structured as follows: Section II is the introduction of American Fuzz Lop (AFL). Section III explains our new insights of existing AFL's energy distribution. Section IV gives details of the improvements method by leveraging two insights. Section V describes implementation details. Section VI is the experimental evaluation of each improvement and performance of integration. Section VII states related works. Finally, we conclude our work in Section VIII.

II. American Fuzzy Lop

AFL based greybox fuzzing is our focus in this paper. AFL based fuzzing tools employ lightweight instrumentation to collect runtime coverage feedback to facilities fuzzing. AFL [16] and its variations like AFLFast [3], FairFuzz [24], AFLGo [2], CollAFL [10], Angora [6] are state-of-art AFL based greybox fuzzers.

After instrumenting target program, AFL executes it with some initial seeds. Then, it maintains a queue of test cases and performs an evolutionary fuzzing loop as below:

1) **Selecting Seed:** select seed that worth fuzzing from testcase queue with a specific policy (e.g., according to execution time, seed length, and hit number);
2) **Assigning Energy:** calculate and assign energy for selected seed based on its attributes (e.g., execute time, bitmap size, etc.), where the energy of a seed represents number of new test cases that generated from the seed by mutation;
3) **Mutating Input:** mutate seeds using different kinds of mutation operators to generate a batch of new test cases;
4) **Executing Target:** feed these new test cases to target application and execute them at a high speed as possible;

5) **Tracking Feedback:** track run-time feedback including code coverage, security violation and so on. Meanwhile, vulnerabilities are reported if security violations are detected.
6) **Filtering Testcase**: compute and update attributions of testcases, put those good testcases that contributing to code coverage into seed queue, and go to step 1).

Following above continuous evolutionary loop, AFL could generate optimized seeds that explore new execution paths, and thus evolve towards a higher code coverage. As a result, the probability to trigger crashes is increased. The main concepts of AFL fuzzing is described in the following:

A. Instrumentation

AFL's instrumentation captures basic block transitions, along with coarse branch-taken hit counts. A sketch of the code that shows in listing 1 is injected at each branch point in the program.

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

Listing 1: AFL's Instrumentation

The variable $cur\_location$ identifies current basic block. Its random identifier is generated at compile time. Variable $shared\_mem[]$ is a 64 KB shared 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 path identifier. AFL uses coverage information to decide which generated input to retain for fuzzing. Also decides which input to be the priority choice and how long the input will be fuzzed.

Algorithm 1 illustrates detailed fuzzing process through 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.

B. Selecting Seed

AFL determines a seed if it is $favorable$ based on its length and execution time. A seed will be marked as $favorable$ if it is the fastest and smallest input for any of the edges it exercises. In process of selecting seed, these $favorable$ seeds will be selected with priority. Non-favorable seeds will be ignored with random probability.

C. Assigning Energy

The energy of a seed t refers to number of new test cases that generated from t after applying various mutation operators. In deterministic stage, AFL determines a seed's energy according to its length. And in havoc stage, AFL firstly determines the basis energy based on its execution time and average execution time. Then it updates total energy based on others attributes like block transition coverage(i.e., bitmap_size), handicap, depth and so on.

Algorithm 1 AFL Greybox Fuzzing

Input: seeds S, program P
Output: crashes Tx

```
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, Tx);
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, Tx);
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
```

D. Mutating Input

In the act of mutating input, AFL utilizes some typical mutation operators to modify the initial seed, and generate a batch of new test cases. These mutation operators include Flips, Interesting, Arith, Extra and Splice in the following. In determined stage of mutation, these mutation operators will be used separately and sequentially to generate new test cases; And in 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.

These mutation operators in AFL are:

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

E. Tracking Feedback

When a new test case is generated, it is fed to target program and executed by AFL. At run-time, feedback including code coverage and security violation are tracked. AFL determines an input to be interesting only if that input has contribution to code coverage. Intuitively, AFL retains inputs that invoke a new block transition or a path where a block transition is executed twice while it normally executes only once. If the

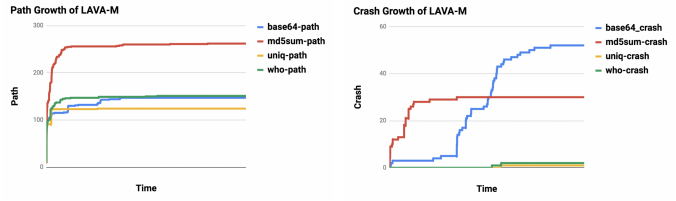

Fig. 1: path and crash growth over time

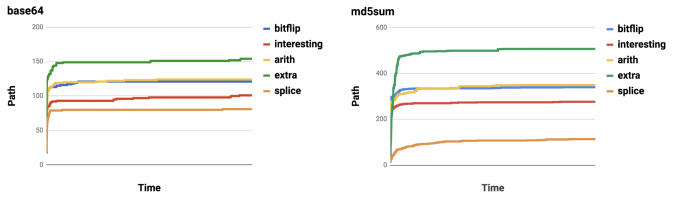

Fig. 2: path growth using different mutators

generated input $t^{'}$ crashes the program, it is added to a set of crashing inputs. A crash input that is also interesting will be marked as a unique crash.

F. Filtering Testcase

If the generated input $t^{'}$ is considered to be interesting, AFL will run a calibration stage to compute and update its attributes, then add it into the queue. These attributes are important because they are the basis for energy assignment. More specifically, these attributes include execution, bitmap, hit numbers and so on.

III. New Insights

During the process of using AFL based greybox fuzzers (e.g., AFL and AFLFast), we have some new insights about existing AFL's fuzzing energy distribution strategy. In this section, a detailed description of two insights is illustrated.

Insight 1: Growth of paths follows the same pattern, while growth of crashes has no definite rules. The Fig.1 is a statistical result of path growth and crash growth of LAVA-M [18] benchamrk fuzzed by AFL. As shown in Fig.1, number of total path increases quickly at the beginning, and growth rate gradually becomes flat over time. Finally, it will become harder to find new path. And slop of growth curve is approximating zero. A natural reason is that discovery 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 number of total paths. From this formula, we can see that the discovery of new paths is increasingly difficult over time. Compared with path growth pattern, occurrence and increment of crashes are completely different. There is no universal model for the emergence and growth of crashes. As can be seen from above Fig. 1, crash may appear very early. It also may occur very late. Crash may grow faster at the beginning and then not grow later. It also may not grow at first, and then grow faster later. The reason behind this phenomenon is that direct cause of crash discovery is not increment of path coverage, but whether regions where vulnerabilities resided are explored.

If those regions that are more likely for a vulnerability to reside are prioritized and strengthened to fuzz, related vulnerabilities buried in these regions will be detected faster. Furthermore, more bugs could be found during the same period of time.

There are some common intuitions that sensitive region(i.e. region containing much memory or string related operators), complex region, deep region, and rare-reach region are more likely to be vulnerable due to developer negligence and inadequate testing. Thus, guide fuzzing toward these promising vulnerable regions and spend more fuzzing energy on them will improve the efficiency of fuzzing tool.

Insight 2: Existing energy distribution strategy only considers calculation of mutations numbers. It does not consider scheduling of distribution ratio of different kinds of mutation operators. The energy refers to the number of new test cases generated by mutating seed using various granularity mutation operators. It is calculated based on seed's attributes such as execution time, bitmap size, depth, hit numbers and so on. However, scheduling of distribution proportion for different kinds of mutation operations in assigned number does not get enough attention it deserves. In determined stage of mutation, AFL applies mutation operators (e.g., Flips, Interesting, Arith, Extra, and Splice) separately and sequentially to generate new test case; And in havoc stage, AFL would mutate the seed by randomly choosing a sequence of different mutation operators and apply them to random locations in the seed files.

Experimental evaluation of each kind of mutation operators are performed on LAVA-M [18], and the results are listed in Fig. 2. It indicates that different mutators have different abilities to trigger new path. Extra mutations have better ability to generate diversity test case, which is more helpful to the path growth. On the other hand, splice mutators perform poor. (2) The effect of combining multiple kinds of mutators is better than the effect of using single kind of mutator.However, randomly choosing mutators without scheduling may cause overuse of some low effective mutation operators, and lack of diversity.

Thus, the energy distribution should consider not only the number of mutations but also the proportion of different kinds of mutation operators in assigned number. Despite the fact that finding new path will become more and more difficult, the proportion of mutation operations who has better ability to trigger new paths should gradually increase over time.

IV. Improvements

Two new insights were leveraged to improve the fuzzing energy distribution strategy in a principled way. The improvements contain two aspects: (1) improvement of directing fuzzing toward promising regions that are more likely to have vulnerabilities; and (2) improvement of scheduling of the distribution ratio of different mutation operators. Finally, these improvements are organically integrated and implemented into a new tool.

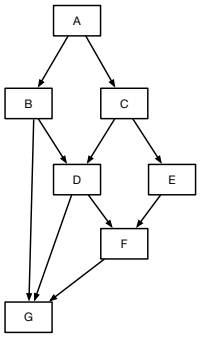

Fig. 3: Abstract CFG

A. Improvement of Directing Toward Promising Regions

Inspired by AFLGo, which directs fuzzing to reach some specific code lines. We improve the guidance of AFL toward promising vulnerable code regions. The core idea is to extract semantic metrics of basic blocks of target program using light-weight Intermediate representation (IR) level static analysis. Then we instrument these qualitative weights into original program through compiler-time instrumentation. At run-time, path reward (i.e., sum of weight) is traced. If a test case is interesting, we assign energy based on its path reward. The seeds who win more rewards will be assigned with more energy. In this way, based on specific semantic metrics of code region, fuzzing could be directed to stress fuzzing those promising vulnerable regions, like sensitive regions, complex regions, deep region and rare-reach regions.

We demonstrate an Intuitive example to explain energy assignment using the abstract control flow graph (CFG) shown in Fig.3.

Let weight of a basic block BB for a specific $metric$ be $weight_{metric}(BB)$. Then, reward of a path that executed with input t could be denoted as:

$$Reward(t) = \frac{\sum_{i=0}^{n} weight(BB_i)}{n}, BB_i \in path(t) \quad (1)$$

Sum of block weights in path $path(t)$ exercised by the input t is counted. Note that we count the basic blocks in loop for multiple times. Intuitively, if test case t executes the path of $p = A \rightarrow B \rightarrow D \rightarrow G$, then $Reward(t)$=($weight_{metric}(A)$ + $weight_{metric}(B)$ + $weight_{metric}(D)$ + $weight_{metric}(G)$)/4. After an execution is finished, the maximum, minimum reward and average reward are updated. Average reward $AvgReward$ is calculated using equation as shown below:

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

Furthermore, $Factor$ used for energy assignment is computed based on a seed's reward and average reward value.

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

$Factor$ affects the amount of energy assignment. The bigger the $Factor$ is, the more energy is assigned. More specifically, an exponential energy assignment formula in the following is used to assign energy based on the $Factor$. Let $P_{afl}(t)$ be the energy assigned by AFL for input t, and energy $P(t)$ for test case t assigned after improvement could be donated as following:

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

Thus, based on institutions that the more sensitive, complex, deeper and more rarely reachable regions, the more chance to be vulnerable. Four kinds of related semantic metric are designed and used to drive fuzzing. The four kinds of semantic metric represents four kinds of promising vulnerable regions. They are sensitive region, complex region, deep region, and rarer-reach region. Note that our metrics are one possible representation of four kinds of regions, the four regions could also be represented using other metrics.

1) Towards Sensitive Region: According to the intuition that if a path contains more sensitive operators like memory and string related instruments, it is more likely to occur memory corruption vulnerabilities. Thus, a sensitive metric is designed to measure the sensitive degree of the code region (i.e., basic block). More specifically, measurement of sensitive metric is defined as follows:

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

Where, $SensitiveDegree$ represents the total numbers of memory and string related instruments in the basic block BB.

2) Toward Complexity Region: Based on intuition that more complex areas are more likely to have vulnerabilities, complexity metric are designed to measure complexity of each basic block. The number of total instruments in a basic block is a simple and direct indicator to show the complexity of a code region. Thus, we used instrument number as the complex metric, and the measurement is as follows:

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

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

3) Toward Deep Region: Based on intuition that it has more chance to detect vulnerability by exploring the deep path, we define depth metric to show the deep degree of a path. It represents the distance from the function's entry block to the basic block itself. More specifically, the measurement of depth metric of basic block BB as follows:

For a given basic block BB, all the paths P that from entry block E within function to the basic block B are traversed, which is donated by $P = path(E, BB)$. Assume the depth of path $p_i \in P$ is d_i, then depth of BB is as follows equation:

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

Intuitively, take Fig. 3 as an example, to calculate depth of basic block G, the abstract CFG is traversed using deep first search (DFS) algorithm. And paths from entry block A to G are obtained firstly. They are $p_1 = A \rightarrow B \rightarrow G$, $p_2 = A \rightarrow B \rightarrow D \rightarrow G$, $p_3 = A \rightarrow C \rightarrow D \rightarrow G$, $p_4 = A \rightarrow$

$B \rightarrow D \rightarrow F \rightarrow G$, $p_5 = A \rightarrow C \rightarrow D \rightarrow F \rightarrow G$, $p_5 = A \rightarrow C \rightarrow E \rightarrow F \rightarrow G$. 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, depth of basic block G is computed as equation (7), and result is $1/(1/2 + 1/3 + 1/3 + 1/4 + 1/4) = 3/5$. Note that computation of depth is based on intra-procedure static analysis without considering function-level distance. It may sacrifice certain accuracy for conveniences of usage.

4) Toward Rare-Reach Region: Based on intuition that if a code region has lower reachable probability, the region may have higher chance to have issues. The reason behind this intuition is that rare-reach areas are usually not suffering from sufficient testing, especially in integration testing phase. Thus, metric called $RareReachDegree$ is defined to represent the rarely reachable degree of a region (i.e., basic block of program). More specifically, measurement of rarely reachable degree is calculated as follows.

For a given basic block BB, the probability to all its outgoing edges are assumed to be equal. Hence, if $succ(BB)$ denotes number of all successor basic blocks of BB, then $\forall b \in succ(BB)$, $TPro(B, b)$ = 1 / $succ(B).size$, where $succ(B).size > 0$. All paths P that from entry block E of function to basic block BB itself are traversed, and denoted by $P = path(E, BB)$.

Given a path $p_i \in P$, the reachable probability of BB in path p_i is calculated using below equation:

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

Furthermore, we will calculate all paths and finally get the rarely reachable degree of basic block BB as follows:

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

Similarly, take Fig.4 as an example for sake of clarity. In order to compute depth of basic block G, we will traverse the control flow graph using DFS algorithm and get all the paths from entry block A to basic block D. They are $p_1 = A \rightarrow B \rightarrow D$, $p_2 = A \rightarrow C \rightarrow D$. Then, rare-reach degree of B is calculated using equation (8) and (9), and the result is $1/(1 * 1/2 + 1 * 1/2) = 4$.

B. Improvement of Schedule of Mutators

Existing energy assignment strategies only consider number of mutations. But they do not take into account the distribution of different granularity mutation operations in assignment number. In order to add flexible granularity-aware scheduling for different kinds of mutation operations, an understanding of different granularity mutation operators' abilities to promote path growth is needed firstly.

Thus, an empirical evaluation of AFL's five mainstream kinds of mutators is performed to answer above question. The statistical result on LAVA-M data set is shown in Fig.4. It indicates that (1) different kinds of mutation operators have different power for path growth; (2) extra (i.e., block-level deletion, insert and overwrite) mutation operators have

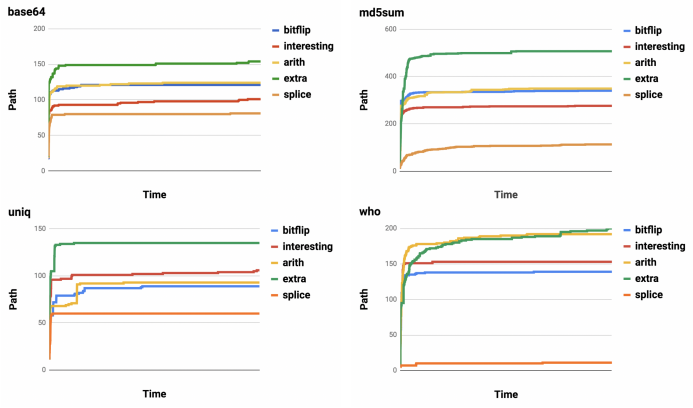

Fig. 4: Performance of Different Mutators

better performance to help path growth, while splice mutation operators perform poor in general.

Motivated by above observation, granularity-aware scheduling of mutators is proposed. The proportion of mutation operators, which has better ability to trigger new paths (e.g., extra mutators) will be increased gradually. And over time, number of mutation operators that are chosen to applied on the seed is increased too. More specific, the scheduling algorithm for mutation operators is illustrated in algorithm 2.

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

Where p represents number of selecting and applying different granularity mutation operators. The $process\_time$ refers to the execution time of fuzzing. And α is a constant ratio to conduct extra mutation. Over time, the β will become smaller, and proportion of extra mutators will become higher. At the same time, p becomes bigger, and modification will become more full. In practice implementation, the α is set to 0.3.

V. Implementation

We incorporate the aforementioned improvements into AFL's energy distribution, and developed a new fuzzing tool named TAFL on top of afl-2.52b.

An overview of TAFL is illustrated in Fig. 5. Firstly, four kinds of promising vulnerable regions related metrics are extracted from a target program by static analysis. Then extracted weights of metrics are instrumented into target binary using compiler instrumentation. The reward of path exercised by a input was traced at run-time. These path rewards are used to distribute fuzzing energy, which includes phrases of selecting seed and assigning energy. TAFL prefers to select test cases that win higher rewards and assign more energy for

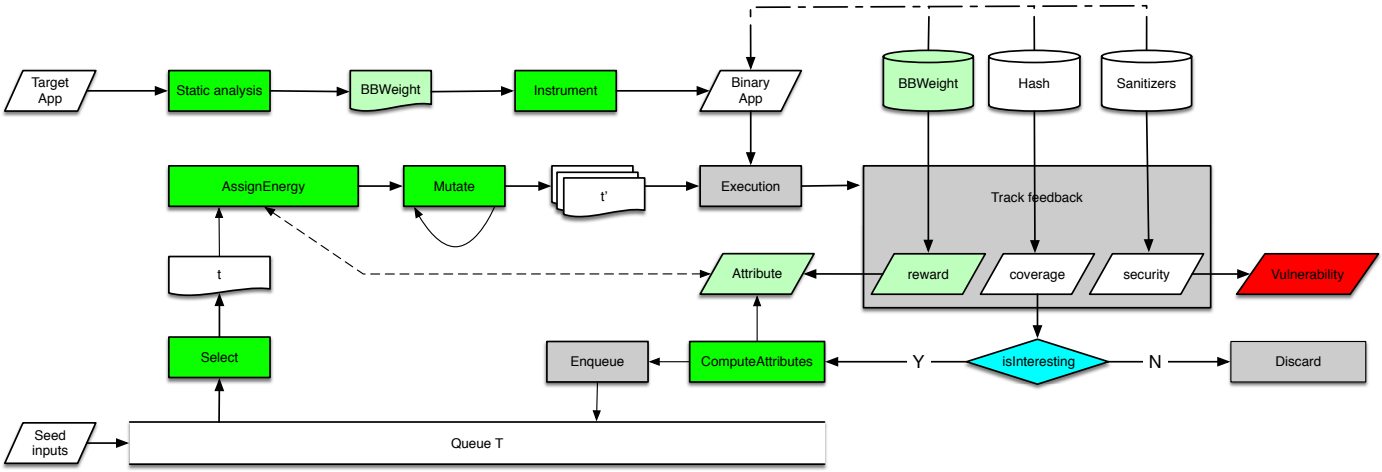

Fig. 5: Overview of TAFL

them. The directed fuzzer strengthens fuzzing these promising vulnerable regions (i.e. sensitive regions, complex regions, deep regions and rare-reach regions). Furthermore, scheduling of mutation operators will increase the proportion of extra mutators over time, which have better ability to trigger new paths. The details of implementation are described in the following:

1) Metric Extraction: Weight extraction for each metric is implemented as one LLVM pass [17]. They are used to obtain weight value of region's feature related metrics including sensitive degree, complexity, depth and rare-reach degree. These passes are used and activated by compiler afl-clang-preprocess. Each preprocess pass has a corresponding environment variable, i.e., GET_MEM_DENSITY, GET_INST_NUM, GET_DEPTH, and GET_ENTRY_DEGREE. When environment variable of compiler CC is set to afl-clang-preprocess, and any of above LLVM passes' environment variable is set, related weight information of each basic block is generated and stored into a text file after the project is built.

2) Compiler Instrumentation: The BB weight file for each metric consists of the BB's name and its metric weight value. It is taken by compiler to instrument the weight value into target executable binary. More specifically, an extended trampoline is injected for each basic block. The trampoline is a piece of assembly code that is executed after the jump instruction, in order to keep track of the coverage control-flow edges. An edge is represented by a byte in a 64KB shared memory. On a 64-bit architecture, we extend 16 additional bytes of shared memory to record the reward feedback. 8 bytes are used to accumulate weight value, and another 8 bytes are used to record number of executed basic blocks. The instrumentation is implemented as an extension of AFL LLVM pass. When performing instrumentation, set compiler to afl-clang-fast, and compiler's flags to reference related BB weight file, then build target project.

3) Energy Distribution: TAFL fuzzes the instrumented executable binary with integrated energy distribution strategy. It selects seed and assigns energy based on run-time reward of test case. The current test case's reward is computed by dividing accumulated BB weight for different metrics by the number of exercised basic blocks. TAFL selects those test cases that wins higher reward and assign energy based on seed's reward factor. Besides, its implementation of MutateInput in havoc stage is modified to improve the proportion of extra mutators over time.

We have open sourced our tool which is available for download at https://sites.google.com/view/tafl/tool.

VI. Evaluation

In this section, we evaluate the performance improvements brought by TAFL using well-known benchmarks. The evaluation setup environments are firstly described, including the used benchmarks, evaluation tools, experimental infrastructure and research questions we are trying to answer. We also represent and explain our evaluation results.

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 specific vulnerabilities, and hence form a ground-truth corpus for evaluating fuzzing tools.

- **LAVA-M Dataset**. LAVA-M consists of four buggy versions of Linux utilities, i.e., `base64`, `md5sum`, `uniq` and `who`. It was generated by automatically injecting known vulnerabilities into hard-to-reach regions of the source code [18]. It is commonly used as a benchmark for evaluating the capability of various fuzzing tools. The authors of recent fuzzing tools (e.g. VUzzer [33], CollAFL [10], and Angora [5] all used this benchmark.
- **Real-World Projects**. Some popular open source Linux projects are selected, including document process libraries (e.g., libxml2), image processing libraries (e.g., libtiff) and so on. They are chosen based on the following criteria: popularity in the community, development activeness, and diversity of categories.

App	AFL			TAFL-sen			TAFL-com			TAFL-deep			TAFL-rare		
	first(min)	crash	path	first(min)	crash	path	first(min)	crash	path	first(min)	crash	path	first(min)	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%**

TABLE I: Results of Directing Towards Four Kinds of Promising Regions

Evaluation Tools: AFL based greybox fuzzing is our focus. Thus AFL and its variants based on source level instrumentation are our comparison targets. Furthermore, we only select tools that are available for download online. These tools include:

- **AFL-2.52b**: It is the latest version of official AFL.
- **AFLFast**: It is an AFL variant that tries to balance the energy distribution by spending more energy on low-frequency path.
- **FairFuzz**: It is an AFL variant that guides the fuzzer to approach rarely reached branches.
- **TAFL**: It is our extension of AFL, which is integrated with improvements of directing towards four kinds of promising vulnerable regions and granularity-aware scheduling of different kinds of mutation operators.

Experimental Infrastructure: Fuzzing tools are evaluated on collected benchmarks with the same configuration, i.e., a virtual machine configured eight core of 2GHz Intel CPU and 8 GB RAM, running Ubuntu 16.04.

Research Questions: We evaluated each improvement and integrated performance of TAFL, and try to answer the following research questions:

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

B. Evaluation of Directing Towards Promising Regions

The effectiveness of directing fuzzers toward four kinds of promising regions that are more likely for a vulnerability to reside are evaluated. Furthermore, we compare the results with the original AFL. Note that TAFL used in this evaluation is only integrated with improvement of guidance, without scheduling of mutators. Each project is executed ten times for 24 hours in order to reduce the random noise of fuzzing. Results are collected and illustrated in Table I. In the table, the four directed fuzzing strategies are donated as *TAFL-sen* (sensitive region), *TAFL-com* (complex region), *TAFL-deep* (deep region) and *TAFL-rare* (rare-reach region) respectively. Three measurement metrics, including time to trigger the first crash, total unique crashes and total paths are used. They are labeled as *first*, *crash* and *path* respectively in the table.

1) First Blood : Time to trigger the first crash is an important factor to evaluate a directed fuzzing tool. As we could see from Table I, all the four directed fuzzing strategies advance the time to find the first crash in most cases. The average advance ratio for detecting first blood are 36.94%, 32.21%, 20.91%, 28.27% for four kinds of area directed strategies respectively. Generally speaking, sensitive regions guided fuzzing has better performance of triggering the first crash on selected benchmarks. In some specific cases like bison-3.0.4, sensitive area directed strategy brings 95.43% performance improvement. This result proves that if a code region contains more memory and string related instruments, it has more chances to occur memory corruption problems.

2) Unique Crashes: Another important characteristic of a good fuzzers is its ability to find unique crashes in limited time. Although the same root crash may cause multiple unique crashes, and some are even not exploitable, in general, more unique crashes indicate better chances to find vulnerabilities. As can be seen from Table I, four regions directed strategies get more unique crashes in most cases, and average improvements are 15.05%, 17.09%, 25.12%, 31.26% respectively. The results prove the effectiveness of improvement in directing fuzzing toward promising areas. Moreover, among four kinds of region directed strategies, rare-reach region directed strategy performs the best in general. Especially, its improvement reaches 216.66% in the case of libxml2-2.9.2. It proves that regions without sufficient testing may have higher probabilities for a vulnerability to reside.

3) Total Paths: Another important factor is the number of total paths a fuzzer can traverse in limited time. For four vulnerable region directed strategies, the average improvements are 2.29%, 1.91%, 3.17% and 17.39% respectively. The result shows that rarely reachable region guided strategy is much more helpful to path growth. Furthermore, we could conclude from the above results that the strategy of directing fuzzing towards rarely reachable regions outperform all the other on selected benchmarks.

C. Evaluation of Scheduling of Different Mutators

Code coverage is one of the critical factors for fuzzers' success. We use two coverage related metrics (i.e., path growth over time and total reachable paths) to validate the effectiveness of our improvement to mutator scheduling. The evaluation was performed on collected benchmarks, and each project ran ten times for 24 hours for the sake of reducing

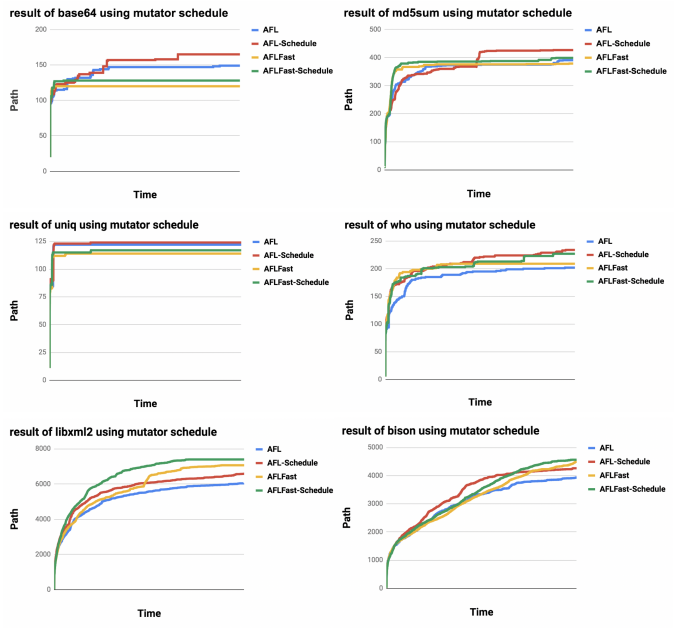

Fig. 6: Results of Scheduling Mutators

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%**

TABLE II: Paths Improvement of Scheduling Mutators

fuzzing's random noise. The result is counted and illustrated in Table II.

1) Path Growth Over Time: Path growth over time is a direct indicator of fuzzer's ability to explore new paths. The original AFL and AFLFast are modified to add improvement of scheduling for different mutation operators. Due to limited space of the article, we selected partial results to illustrate the effectiveness of scheduling of mutation operators in Fig. 6. For more empirical results, please refer to our website (https://sites.google.com/view/tafl).

As can be seen from above Fig.6, fuzzers with scheduling of mutators could trigger new paths more quickly than than those without scheduling in most cases. Scheduling of mutators is helpful to maximize code coverage in limited time. The results prove the effectiveness of scheduling mutators. Besides, we observe that AFLFast is better than AFL in most cases within 24 hours. However, in some case such as base64, AFLFast's performance is worse than AFL. In practice, AFLFast's will slow down to zero after a specific time. The reason is that low-frequency paths in base64 are time-consuming, which stuck AFLFast.

2) Improvement of Total Path: Furthermore, we show the total executed paths by AFL, AFLFast and their improved versions in Table II. Scheduling of mutators is useful to improve code coverage in most cases. For AFL and AFLast, the average increased percentage on selected benchmarks are 7.03% and 4.82% respectively. In certain cases such as libxml2-2.9.2, the promotion percentage could be as high as 15.8% and 14.7%. In some lousy cases such as libtiff-3.7.0, the ratio could drop to -0.9%. We observe that the the difference of different fuzzing strategies' efficiency are obvious when limited time is given. If enough time is given, the eventual discovered paths will be very close for different fuzzing strategies. This is because all the easily reachable regions are already explored.

D. Evaluation of Integrated Performance

In this section, we evaluate the overall performance of TAFL compared with AFL, AFLFast and FairFuzz, which means TAFL is integrated with the capability to direct towards promising regions and to schedule of different mutators. Similar to Section VI-B, the same set of metrics are used to measure the performance. Each project is fuzzed ten times for 24 hours in order to reduce the random noise of fuzzing, and the results are collected and illustrated in Table III.

1) First Blood: As is shown in Table III, the integration of improvements is effective to advance the time to trigger the first crash. In general, the time to trigger the first crash is advanced 26.37% after using mutator schedule. This is better than AFLFast and FairFuzz.

2) Unique Crashes: As can be seen from the Table III, compared to AFL and its extension AFLFast, TAFL-rare detects more crashes within 24 hours on selected benchmark. The improvements in total crashes of AFLFast and TAFL-rare are 11.39% and 23.55%. TAFL-rare is twice as many as AFLFast. Meanwhile, FairFuzz performs poor, especially on LAVA-M benchmark, it cannot detect crashes in base64, uniq, and who programs, but it could get a better result on the cflow-1.5 project.

3) Total Paths: As can be seen from the Table III, the promising regions directed and mutator schedule strategy are helpful to improvement on the entire path. On the selected benchmarks, TAFL-rare brings an average improvement of 18.57%, while the average improvements to AFLFast and FairFuzz are 8.35% and -9.62% respectively. Furthermore, TAFL helps us find five unknown bugs and identify one new CVE [19].

VII. Related Work

We review related work that improves AFL's capabilities from three dimensions.

(1) **Effectiveness:** Effectiveness concerns about meta-abilities of fuzzers to bypass obstacles and trigger vulnerabilities. Generally speaking, there are three approaches to improving effectiveness.

- *Feedback granularity and accuracy:* Steelix [26] instruments AFL to collect mutate progress for magic bytes compassion, and continue to mutate on input that has progressed for each bit. CollAFL [10] demonstrates that inaccuracy of feedback (i.e., hash collision issue) in AFL would limit the effectiveness of discovering path. The

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%**

TABLE III: Comparison Results of TAFL and AFL based Greybox Fuzzers

authors design an algorithm to resolve the hash collision problem, and improve the edge coverage accuracy with a low-overhead instrumentation scheme.

- *Smart mutation strategy:* The mutate strategy answers the questions about where to mutate and what to mutate. To answer the former, Lin et al. [27] propose a solution to identify raw bytes to mutate using static data lineage analysis. A deep neural network solution is proposed in [32] to predicate which bytes to mutate. To answer the latter, Vuzzer [33] uses dynamic analysis to infer exceptional values (e.g., magic numbers to use for mutating.)
- *Sensitive to security violation:* Fuzzers usually use program crashes as an indicator of vulnerabilities, because they are easy to be detected 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 various kinds of security violations. For example, the widely used AddressSanitizer [35] and MemorySanitizer [37] could detect buffer overflow and use-after-free vulnerabilities. There are many other sanitizers available, including UBSan [23], DataFlowsanitizer [7], ThreadSanitizer [36] and so on.

(2) **Efficiency:** Efficiency concerns about improving code coverage in order to improve the probability of trigger more vulnerabilities. Previous efforts were conducted following three approaches.

- *Quality and diversity of initial seeds:* Skyfire [42] learns a probabilistic context-sensitive grammar from abundant inputs to guide seed generation. Learn & input [14] utilizes recurrent neural network(RNN) solution to generate valid seed files and could help generate inputs to pass format checks. Nicole Nichols et al. [28] proposed a generative adversarial network(GAN) solution to argument the seed pool with extra seeds.
- *Execution speed:* AFL utilizes fork mechanism of Linux to accelerate the execution speed. It further uses fork-server mode and persistent mode to reduce the overhead of the fork operations. Besides, AFL prioritizes seeds that are executed faster, and thus it is likely that more test cases could be tested in a given time. Moreover, AFL also supports parallel mode, which enables multiple fuzzer instances to collaborate with each other. kAFL [34] and PTfuzzer [45] use hardware features (i.e., Intel PT) to accelerate the execution speed. Wen Xu et al. proposes several new primitives [44] which speed up AFL by 6.1x to 28.9x.
- *Balance of fuzzing energy distribution:* AFLFast [3] prioritizes seeds that were exercising less-frequent paths. Thus it is more likely that cold paths could be tested thoroughly and less energy will be wasted on hot paths. FairFuzz [24] spends more energy on less reachable branches. CollAFL [10] prioritizes seeds that hit more untouched neighbors to improve the possibility to cover more new paths.

(3) **Guidance:** The guidance criteria directs a fuzzer to focus on a specific goal. Along this direction, several works have been proposed.

- *Directing for specific locations:* AFLGo [2] uses distance metrics to direct fuzzing to trigger and reproduce vulnerabilities in the specific location by giving the target locations at priority.
- *Directing for specific kinds bugs:* SlowFuzz [31] prioritizes seeds that use more resources (e.g., CPU, memory, and energy), and try to increase the probability of triggering algorithmic complexity vulnerabilities.

VIII. Conclusion

We leverage two new insights to improve AFL's fuzzing energy distribution in a principled way. First, intuition suggests that four kinds code regions (i.e., sensitive, complex, deep and rare-to-reach regions) are more likely to contain vulnerabilities. Therefore, we use static analysis to obtain semantic metrics of the target program, and then direct the fuzzing tool to stress on such regions. Second, existing energy distribution strategies randomly select mutators. Since fine-grained mutators and coarse-grained mutators have their own advantages and disadvantages, we can dynamically adjust the ratio of a kind of mutators if they perform better. This results in a new granularity-aware energy distribution scheduling strategy. These improvements have been incorporated into AFL, leading to a new fuzzing tool named TAFL. To show the power of TAFL, we have conducted large-scale experiments on popular benchmarks and real-world programs. The result is promising. Not only does TAFL outperform existing AFL and AFL variants regarding effectiveness, but also it helps us locate five unknown bugs and identify a new CEV in Libtasn1-4.13.

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://llvm.org/docs/WritingAnLLVMPass.html, "Writing an llvm pass."

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

[19] https://sites.google.com/view/tafl/vulnerabilities, "bugs."

[20] https://testing.googleblog.com/2016/12/announcing-oss-fuzz-continuous fuzzing.html, "Google oss fuzzing," 2016.

[21] https://www.microsoft.com/en-us/security-risk detection/, "Microsoft security risk detection," 2016.

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

[23] 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.

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

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

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.

[31] 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.

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

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

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

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

[41] 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.

[42] 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.

[43] 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.

[44] 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.

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