AFLTurbo: Speed up Path Discovery for Greybox Fuzzing

Lei Sun*, Xumei Li*, Haipeng Qu*✉, Xiaoshuai Zhang†
*Department of Computer Science and Technology, Ocean University of China, Qingdao, China
†Queen Mary University of London
Email: sunlei7210@stu.ouc.edu.cn, lixumei@stu.ouc.edu.cn, quhaipeng@ouc.edu.cn, xiaoshuai.zhang@qmul.ac.uk

***Abstract*—Coverage-based greybox fuzzing (CGF) is a common method utilizing coverage information to guide fuzzing. American Fuzzy Lop (AFL) is one of the most famous CGF fuzzers and has been used to uncover thousands of vulnerabilities in many software. However, AFL has two major drawbacks, which impedes it from boosting path discovery: (1) aggressively growing mutation overhead; (2) ineffective mutation region selection.**

In this paper, we propose three new approaches to overcome the drawbacks: (1) *Interruptible mutation*, which uses a hang monitor to avoid unnecessary mutation overhead; (2) *Locality-based mutation*, which utilizes mutation information in previous rounds to guide fuzzing useful regions in future rounds; (3) *Hotspot-aware fuzzing*, which exploits a pre-evaluation process to identify metadata and only mutates these regions. We combine these approaches into a tool named AFLTurbo based on AFL 2.52b. Furthermore, the effectiveness of AFLTurbo is evaluated in terms of both path discovery and bug detection on eight programs as well as LAVA-M with state-of-the-art fuzzers. The experimental results manifest that AFLTurbo can find 141%, 101% and 41% more paths, and reveal 14x, 30x and 5x more bugs than AFL, AFLFast and FairFuzz respectively. Additionally, AFLTurbo discovers 20 vulnerabilities, of which 18 are assigned with CVEs.

***Index Terms*—fuzzing, path discovery, vulnerability detection**

I. Introduction

The security vulnerability is consistently identified as one of the major issues of computer programs. Attackers can leverage vulnerabilities to execute malicious code, which may cause denial-of-service attacks, privilege escalation attacks, etc. In contrast to attackers, defenders attempt to discover vulnerabilities as earlier as possible in order to fix them to prevent zero-day attacks.

Therefore, many approaches to discover vulnerabilities have been proposed to help the defenders with many efforts from both academia and industry. In vulnerability discovery, there are three conventional methods discussed by many researchers [27] [16] [22]. Static analysis is performed without executing the program but usually requires lots of manual efforts. Symbolic execution technique analyzes programs with symbolic inputs and produces concrete inputs by utilizing constraint solver (*e.g.*, Z3 [13]) to solve constraints. Dynamic taint analysis labels input data and monitors the program execution to track the propagation of tainted data. However, all of the above methods are infeasible to scale to large programs because they rely on heavy program analysis [29].

Fuzzing is an automated testing technique that aims at finding vulnerabilities in computer programs. It iteratively runs target programs with mutated test cases and expects unintended behaviors (especially crashes) of target programs. Compared with the above mentioned approaches, it requires less manual effort and is with desirable scalability and practicality. In the fuzzing field, coverage-based greybox fuzzing (CGF) has recently become the mainstream of fuzzing because it strikes a good balance between whitebox's effectiveness and blackbox's efficiency. It utilizes lightweight program analysis to guide fuzzing. One state-of-the-art CGF fuzzer is American Fuzzy Lop (AFL) [32], which has been used to find thousands of vulnerabilities in widespread tools and software, including *pdfium*, *tcpdump*, *libtiff*, *QEMU*, *OpenSSH*, *PHP*, etc.

Inspired by the impressive achievements of CGF, numerous studies have devoted to improving it from different angles. Superion [28] tackles the structured input problem in fuzzing by using grammar-aware trimming and mutation strategies. Steelix [18] uses program-state based approach to solve the magic number problem. Notably, more attention has been paid to the performance of CGF, which is also what we focus on in this paper. AFLFast [11] prioritizes test cases which execute low-frequency paths and assigns more energy to them. FairFuzz [17] only selects test cases hitting rare branches. UnTracer [21] avoids needless tracing of test cases by adding a coverage-guided tracing using static binary instrumentation. Xu *et al*. [30] design several system calls to reduce forkserver overhead and achieve scalability on multi-core machines.

However, the aforementioned approaches ignore the mutation overhead of test cases. Mutation overhead means that CGF generates too many useless mutated test cases which cannot find new paths. For example, suffered from the mutation overhead, AFL spends a considerable amount of time on mutating certain test cases, which impedes it from making progress. Besides, we also observe that AFL is ineffective at mutation region selection. Consequently, the fuzzer fails to discover new paths quickly.

In this paper, we take these two challenges into account and propose the following three new approaches to tackle them. To reduce the mutation overhead, we propose an *interruptible mutation* method which utilizes a hang monitor to watch the execution status. With the help of this monitor, AFL is able to skip mutation on test cases which it gets stuck on. To address the second challenge, we propose two approaches: *locality-based mutation* and *hotspot-aware fuzzing*. The former is based on the principle

2332-6549/20/$31.00 ©2020 IEEE
DOI 10.1109/ISSRE5003.2020.00017

of locality, and it leverages information of previous rounds to prioritize mutating useful regions in future rounds. Meanwhile, *hotspot-aware fuzzing* consists of two phases. The first phase mainly relies on a special pre-evaluation process, which dynamically recognizes the metadata of different test cases for various target programs. Having identified the metadata for test cases, AFLTurbo only mutates these regions in the second phase. Throughout the paper, the term "metadata" refers to the data regions of an input, which affect the control flow of target programs.

We implement a prototype named AFLTurbo by extending AFL with the proposed three approaches. We evaluate AFLTurbo against three popular fuzzers AFL, AFLFast and FairFuzz in terms of both path discovery and bug detection. The experimental results demonstrate that our fuzzer discovers paths significantly faster than others and is more effective with regard to bug detection. Moreover, AFLTurbo exposes 20 security vulnerabilities in four popular programs. For example, CVE-2019-19931 is a heap overflow vulnerability in libIEC61850, which can be used to achieve remote code execution by sending carefully crafted packets.

The contributions of this paper are summarized as follows.

- We propose three new approaches *interruptible mutation*, *locality-based mutation* and *hotspot-aware fuzzing* to improve AFL's speed of path discovery.
- We combine these three approaches together and implement a prototype tool named AFLTurbo based on AFL 2.52b. And we open source AFLTurbo at https://github.com/sleicasper/aflturbo for the further research of fuzzing.
- We evaluate its performance on eight programs and LAVA-M, which demonstrates that our fuzzer significantly improves AFL's capability of path discovery and bug detection.
- We find 20 vulnerabilities in four open-source programs, of which 18 are assigned with new CVEs and 2 have been reported previously.

II. Background and Related Work

The past few decades have witnessed several efforts to find software vulnerabilities using various fuzzing techniques. In this section, we introduce coverage-based greybox fuzzing as a generic bug finding technique, and discuss AFL and AFL-based approaches that prior works have taken.

A. Coverage-based greybox fuzzing

Fuzzing can be classified into whitebox, blackbox and greybox based on fuzzer's awareness of program state. Whitebox fuzzers (*e.g.*, SAGE [16]) rely on heavy program analysis and excessive constraint solving. Therefore, whitebox fuzzers have an aptitude for exposing vulnerabilities hidden deep in the target program but have slow execution speed. While blackbox fuzzers (*e.g.*, Peach [8]) are unaware of program state, they simply test programs with generated test cases to trigger unintended behaviors. Hence, blackbox fuzzers are capable of testing hundreds of test cases per second. Greybox fuzzing combines advantages of both blackbox fuzzing and whitebox fuzzing via lightweight program analysis.

Coverage-based greybox fuzzing (CGF) is the most common way of implementing greybox fuzzing. It utilizes coverage information obtained through compile-time instrumentation or dynamic binary instrumentation tools [4] [5] [19] to guide fuzzing. CGF preserves the speed of testing and reduces the expense of program analysis. And it has gained great attention of both software developers and the security research community owing to its remarkable performance in vulnerability detection.

B. AFL

AFL is the most famous CGF fuzzer. In this section, we talk about some key concepts and the core algorithm used in AFL.

Bitmap. AFL employs a lightweight compile-time instrumentation to report edge coverage information at runtime. Edge coverage information is stored in a 64KB-shared bitmap *trace_bits*. Every byte in *trace_bits* stores the hit count of each edge (i.e., the i^{th} byte of the *trace_bits* with value x denotes that the edge with hash value i has been hit x times during the current running).

Mutation strategies. During the fuzzing process, the mutation strategies [33] are utilized to determine how to mutate a test case. AFL employs two mutation stages: the deterministic stage and the havoc stage. In the deterministic stage, AFL uses deterministic mutation strategies to mutate test cases. While the havoc stage randomly selects mutation operations to mutate test cases at random locations.

Deterministic mutation. The deterministic mutation mainly consists of four mutation strategies: bit flips (*bitflip 1/1*, *bitflip 2/1*, *bitflip 4/1*), byte flips (*bitflip 8/8*, *bitflip 16/8* and *bitflip 32/8*), arithmetic addition/subtraction (*arith 8/8*, *arith 16/8* and *arith 32/8*) and replacement with interesting values (*interest 8/8*, *interest 16/8* and *interest 32/8*). And *x/y* means applying an operation on test ccaes with length x bits and stepover y bits. For example, bitflip 32/8 represents sequential bit flip with length 32 bits and stepover 8 bits.

Seed schedule. Seed schedule controls the order of selecting seeds from the queue during a cycle, where a cycle means a single iteration of the test case queue. AFL prioritizes test cases that are smaller in size and shorter in execution time.

Core algorithm. Algorithm 1, which starts with a target program P and a set of initial seeds S, gives a general overview of AFL's fuzzing loop: (1) select a test case from the test case queue Q with seed schedule policies, (2) utilize mutation strategies to mutate the selected test case and generate new test cases, (3) run the target program with each generated test case and monitor its execution, (4) append interesting test cases (i.e., test cases leading to new coverage) to Q, add test cases which crash P to the crash set C and discard other test cases, (5) go back to (1) and the loop continues until AFL receives a stop signal.

Algorithm 1 AFL

```
1:  procedure AFL(S, P)
2:      //S(Seeds), P(a Program)
3:      Q ← S
4:      C ← ∅
5:      while true do
6:          t ← SELECTTESTCASE(Q);
7:          for mutation in deterministicMutations do
8:              for i in range(0, |t|) do
9:                  t' ← MUTATE(t)
10:                 RUNANDSAVE(P, t')
11:         HAVOCMUTATESTAGE(P, t)
12: procedure RUNANDSAVE(P, t')
13:     status ← RUNTARGET(P, t')
14:     if ISCRASH(status) then
15:         C ← C ∪ t'
16:         return
17:     if ISINTERESTING(status) then
18:         Q ← Q ∪ t'
19:         return
20:     DISCARD(t)
```

C. Related work

Inspired by the enormous success of AFL, researchers have been committing to improving AFL's performance and usability in heterogeneous approaches and make lots of contributions to the fuzzing community.

AFL performance optimization. Improving the fuzzing speed of path discovery for AFL is necessary, especially running large programs or processing big inputs. Böhme et al. [11] notice that most test cases execute the same high-frequency paths due to AFL's naive schedule policies. Hence, AFLFast implements several innovative schedule algorithms: a power schedule which assigns more energy to those covering low-frequency paths and a seed schedule which prioritizes test cases executing rare paths. Lemieux et al. [17] observe that AFL fails to explore large regions of program. To help AFL achieve a higher code coverage, they implement FairFuzz based on two approaches: (1) identify rare branches (executed by few test cases), (2) create mutation mask for each test case to avoid mutating regions which result in the execution of rare branches. To reduce the overhead of code coverage tracing in CGF, UnTracer introduces coverage-guided tracing. UnTracer works by using two versions of the target binary: (1) an oracle binary instrumented with software interrupts for quickly identifying untouched basic blocks, (2) a fully instrumented binary for calculating code coverage. Xu *et al*. [30] devise several operating system calls (lightweight `snapshot()` to replace `fork()`, specialized file system service for processing small files and shared memory log for syncing) to reduce overheads of forkserver and file operations.

Structure-aware fuzzing. Recently, many techniques have been developed to improve AFL's ability of fuzzing programs which require structured inputs. For example, AFLSmart [23] integrates Peach [8] into AFL so as to take full advantage of Peach's techniques of producing good structured test cases. Superion [28] introduces a grammar-aware trimming strategy and several mutation strategies into AFL to solve this problem.

Magic number aware fuzzing. Magic number detection is also a general challenge of fuzzing. To address this, Steelix [18] uses static analysis and binary instrumentation to provide comparison information feedback to AFL. laf-intel [6] utilizes *LLVM*'s pass to split comparison into smaller ones. Hence, laf-intel can quickly spot path change if some bytes mutated by AFL are just part of the magic number. REDQUEEN [9] solves the magic number challenge by inspecting function calls and comparison instructions.

Hybrid fuzzing. Symbolic execution is considerably more effective in solving complicated constraints and is often used to assist fuzzing. Driller [26] combines the strength of fuzzing and symbolic execution. When the fuzzer fails to satisfy constraints, Driller utilizes symbolic solver to produce new test cases to pass checks. However, symbolic execution still drops the scalability of fuzzing. To this end, QSYM [31] designs a fast concolic execution engine to improve the scalability of hybrid fuzzing.

Binary-only fuzzing. Source code is not always available, hence many works have been done to fuzz binary-only programs. Afl-dyninst [2] utilizes static binary instrumentation to inject feedback code into the original binary. Winafl [14], afl-pin [3] and afl-dynamorio [1] leverage dynamic binary instrumentation to gather code coverage information while running. AFL qemu mode and Unicorefuzz [20] both use emulation to gather code coverage information.

III. CHALLENGE

In this section, we encapsulate two performance issues that impede AFL from effectively making progress.

C1: Aggressively growing mutation overhead. With increase in the length of the test cases, the mutation space increases aggressively. However, AFL performs mutation against the test cases without respect to mutation space. In this way, the fuzzer would consume tremendous time in certain test cases after several rounds of fuzzing, which impedes the fuzzer from making progress efficiently.

Figure 1 shows a hang status (cannot find new paths for a long time) of AFL when fuzzing *readelf*. We observe that AFL discovers many paths in the first 6 minutes and the last 3 minutes, while it does not find any new paths in other time. Thus, it's critical to find these test cases and avoid falling into unnecessary aggressive mutation.

C2: Ineffective mutation region selection. When fuzzing different target programs, there are metadata regions dominating the new paths can be found. So mutating on metadata usually promotes the fuzzing process greatly. For example, during testing ELF format parsers (*i.e*., *readelf*), performing mutations on elf header can quickly find lots of new paths; on the contrary, mutating the code section could be fruitless. However, AFL treats the whole test case equally, which is

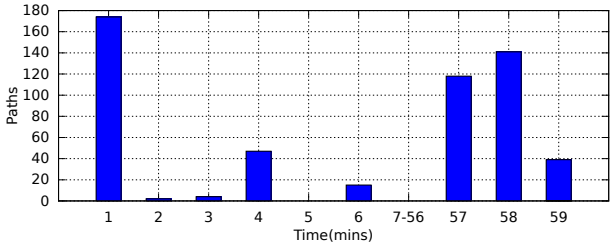

Fig. 1: Number of newly discovered paths of different periods in an hour during the *bitflip 1/1* phase, when fuzzing *readelf*. AFL doesn't find any new paths from the 7th minute to the 56th minute.

ineffective and prevents the fuzzers from quickly finding more paths.

```
1  MmsValue*
2  MmsValue_decodeMmsData(uint8_t* buffer, int bufPos
       , int bufferLength, int* endBufPos)
3  {
4    // get tag from buffer
5    uint8_t tag = buffer[bufPos++];
6    int dataLength;
7    // get data length from buffer
8    bufPos = BerDecoder_decodeLength(buffer, &
       dataLength, bufPos, dataEndBufPos);
9    switch (tag) {
10     ...
11     case 0x86: /* MMS_UNSIGNED */
12       value = MmsValue_newUnsigned(dataLength*8);
13       // heap overflow
14       // copy data with user supplied length
15       memcpy(value->value.integer->octets, buffer+
       bufPos, dataLength);
16       value->value.integer->size = dataLength;
17       bufPos += dataLength;
18       break;
19     ...
20   }
21 }
```

Fig. 2: The motivation example of C2, a real-world heap overflow vulnerability (CVE-2019-19931) in libiec61850 uncovered by AFLTurbo. This vulnerability would be triggered if *tag* is 0x86 and *dataLength* is large enough.

The motivation program shown in Figure 2 contains a heap overflow vulnerability (CVE-2019-19931) at line 15. Suppose that AFL produces a new text case *t* which first passes the check at Line 11 and the value of *dataLength* is legal. Then *t* is retained because a new path is detected. However, when *t* is selected to mutate, AFL mutates it from beginning to end. This is very inefficient because only *tag* and *dataLength* have impacts on the control flow. More importantly, prioritizing mutation on the field related to *dataLength* helps trigger the vulnerability quickly while AFL not. Consequently, AFL fails to discover new paths efficiently as well as expose bugs in time.

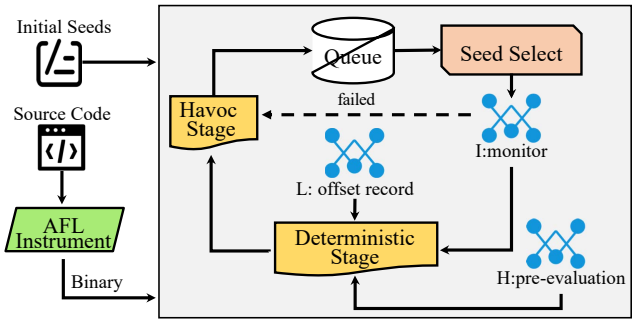

Fig. 3: Overall architecture of AFLTurbo. I, L, H represent *interruptible mutation*, *locality-based mutation* and *hotspot-aware fuzzing* respectively. I monitors slow path discovery and reschedules mutation. H pre-evaluates selected test case. L records mutation offset.

IV. AFLTurbo

We first present our approaches's overview in Section IV-A and then elaborate the detailed design in Section IV-B.

A. Overview

Figure 3 depicts the overall architecture of AFLTurbo, and highlights the three new modules we introduce to AFL: (1) *interruptible mutation* is a monitor component in the deterministic stage to monitor mutation overhead; (2) *locality-based mutation* is a record component for recording locality information (*i.e.*, mutation offset) of test cases; (3) *hotspot-aware fuzzing* is a pre-evaluation process, which is responsible for determining regions that the fuzzer is sensitive to. Equipped with *interruptible mutation*, the fuzzer is able to dynamically make the decision on whether or not to continue with the deterministic mutations. If the further deterministic mutations is considered necessary, *locality-based mutation* and *hotspot-aware fuzzing* take effect.

To sum up, we propose three new approaches to improve the performance of AFL. Besides, we apply each of these approaches to AFL individually and implement three new fuzzers (AFL-int, AFL-loc and AFL-hot).

B. Design

In this section, we give a detailed explanation of each approach.

1) Interruptible mutation: To prevent AFL from getting stuck with some test cases as aforementioned in Section III, we introduce a new hang monitor. The monitor is responsible for monitoring the execution status of AFL, and guides AFL towards skipping mutation on test cases which make AFL hang. To implement the monitor, we introduce two thresholds: the number of newly generated paths and the running time of each mutation stage respectively. If the monitor observes that mutation on a test case can't lead to enough new paths or it spends too much time, it prevents AFL from further mutation at the current stage.

Algorithm 2 shows the general idea of the first approach with a given test case `T`. First, AFL-int performs three types of byte flips (*bitflip 8/8*, *bitflip 16/8* and *bitflip 32/8*) on `T` and stores each newly discovered path count in `cbyte8`, `cbyte16` and `cbyte32` respectively (Line 4-6). We choose these byte filps as reference based on two reasons: (1) bit flip has larger impact on testcases than other mutation types, i.e., it affects all bits when mutating; (2) *bitflip 1/1*, *bitflip 2/1* and *bitflip 4/1* generate a large number of testcases (about 40% in the deterministic stage), so it's very time-consuming and inefficient. Next, AFL-int checks whether the sum of `cbyte8`, `cbyte16` and `cbyte32` is larger than a predefined threshold. If the sum is larger than the threshold, `T` is considered to be worthy of continuous mutation in the deterministic stage (Line 7-11). Otherwise, AFL-int jumps to the havoc stage directly. More specifically, the threshold is configurable and we set it to 2 in our implementation. We believe if a test case couldn't generate more than 2 new paths in all of the three mutation types, it is hard to generate more paths in other mutation stages.

Algorithm 2 *Interruptible mutation*-Approach1

```
1:  procedure AFL_INTERRUPTIBLEAPPROACH1(T, P)
2:      //T(a Testcase), P(a Program)
3:      threshold ← 2
4:      cbyte8 ← MUTATEBYTE8STAGE(T, P)
5:      cbyte16 ← MUTATEBYTE16STAGE(T, P)
6:      cbyte32 ← MUTATEBYTE32STAGE(T, P)
7:      if (cbyte8 + cbyte16 + cbyte32) > threshold then
8:          MUTATEBITFLIPSTAGE(T, P)
9:          MUTATEARITHSTAGE(T, P)
10:         MUTATEINTERESTINGSTAGE(T, P)
11:         MUTATEDICTSTAGE(T, P)
12:     MUTATEHAVOCSTAGE(T, P)
```

Algorithm 3 demonstrates the overview of the second approach for a given test case `T`. First, AFL-int records the start time at the beginning of the current mutation stage (Line 4). Then it mutates `T` from scratch with the current mutation strategy and runs generated test cases with the target program (Line 6-8). When AFL-int reaches a maximum running time in any mutation stage (*e.g.*, bit flip), AFL-int jumps to the havoc stage directly in order to mitigate the excessive overhead of the deterministic mutation (Line 10-11). In our implementation, we set the maximum time to be 4 minutes, because 4 minutes is long enough for a single type of mutation to generate new paths in general based on our early experiments.

2) Locality-based mutation: To tackle C2, we propose *locality-based mutation*, a technique which labels each test case with an offset calculated dynamically and focuses on mutating bytes around it. Motivation behind this technique is the principle of locality [7]. Similar to the principle of locality, if a mutated test case that is generated by mutating at a specific offset exercises a new path, it's more likely to produce more interesting test cases if the fuzzer pays more attention to mutate bytes around this offset. The reason is that

Algorithm 3 *Interruptible mutation*-Approach2

```
1:  procedure AFL_INTERRUPTIBLEAPPROACH2(T, P)
2:      //T(a Testcase), P(a Program)
3:      for mutation in deterministicMutations do
4:          starttime ← GETCURRENTTIME()
5:          for i in range(0, |T|) do
6:              T' ← MUTATE(T, mutation)
7:              RUNANDSAVE(P, T')
8:              SAVEIFINTERESTING(Queue, T')
9:              curtime = GETCURRENTTIME()
10:             if curtime - starttime > 4mins then
11:                 goto havoc
```

bytes around this offset have a higher probability of containing data which affects control flow of programs.

Algorithm 4 describes how *locality-based mutation* works for a given test case `T`. AFL-loc maintains a variable `offset` for each test case, which records an interesting offset of the corresponding test case. `offset` is set according to the following rules: (1) For each test case provided by user, its `offset` is initialized with a special value `0xffffffff` which marks the seed inputs provided by the user. (2) For each mutated test case leading to a new path, its `offset` value is the mutation position of its parent (Line 13-14). During the deterministic stage, AFL-loc mutates every byte if the test case is the seed. Otherwise, it only mutates the first 0x100 bytes and content around `offset` (*i.e.*, 0x100 bytes above and below `offset`).

Algorithm 4 *Locality-based mutation*

```
1:  procedure AFL_LOCALITY(T, P)
2:      //T(a Testcase), P(a Program)
3:      //T.offset has been initialized to 0xffffffff if it is seed.
4:      lowbound ← T.offset − 0x100
5:      highbound ← T.offset + 0x100
6:      for mutation in deterministicMutations do
7:          for i in range(0, |T|) do
8:              if T.offset != 0xffffffff AND i > 0x100 then
9:                  if i < lowbound OR i > highbound then
10:                     continue
11:             T' ← MUTATE(T, i)
12:             RUNTARGET(P, T')
13:             if ISINTERESTING(T') then
14:                 T'.offset ← i
15:                 ADDTOQUEUE(T')
16:     MUTATEHAVOCSTAGE(T, P)
```

We decide to mutate 0x100 bytes above `offset` as well as below `offset` is based on the observation that empirically the size of a struct or class in C/C++ is no larger than 0x100 bytes. Besides, it isn't too big, which would cause big overhead and it isn't too small, which may miss the opportunity to mutate useful fields within the same struct or object. Keeping mutating the first 0x100 bytes is because this region is most likely to

Fig. 4: Ratio diagram of linux command *ls*'s binary. Code section occupies the largest proportion with 82.6%. It's reduntant to mutate code section when fuzzing *readelf*

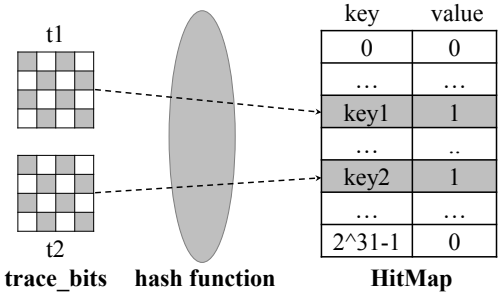

Fig. 5: Mapping between *trace_bits*'s hash value and *HitMap*. Use the hash value of *trace_bits* as the key and set 1 in *HitMap*

be interesting data for most test cases (*i.e.*, many file formats utilize the first few bytes to store critical data).

Next, we discuss why *locality-based mutation* is effective when fuzzing the code fragment in Figure 2. Powered by *locality-based mutation*, AFL primarily mutates regions around `tag`, which include field `dataLength`. As a result, AFL-loc can quickly spot this vulnerability by simply flipping the most significant bit of `dataLength`.

3) Hotspot-aware fuzzing with HitMap: Intuitively, mutating metadata is most likely leading to new coverage because it contains interesting fields like *type*, *offset* and *number*, where *type* may be used as a *switch* case while *offset* may be used to dereference memory which may cause out of bound access. Consider the ELF header structure from ELF specification [25], *e_machine* is a *type* field in ELF file header that specifies the required architecture. Different values in *e_machine* would lead to different paths when fuzzing *readelf*.

In contrast, mutating data regions which do not belong to the metadata impedes the target program from exploring new paths. Figure 4 depicts an ELF file structure, where most of the data in this ELF file is composed of *code* and *data* (about 90% in total), and there is no point mutating these data when fuzzing *readelf* because these data are not processed by it. Unfortunately, AFL is data-insensitive and has no ability to distinguish metadata and non-metadata.

A straightforward idea is to specify metadata and focus on it. After some attempts, we reject this approach because metadata is program-related and it's hard to establish a clear boundary between metadata and non-metadata. Instead, we propose a new technique *hotspot-aware fuzzing* with a new

Algorithm 5 *Hotspot-aware fuzzing*

```
1:  global variables
2:      Hitmap
3:      Bitmap
4:      Queue
5:  end global variables
6:  procedure AFL_HotSpot(T, P)
7:      //T(a Testcase), P(a Program)
8:      step ← 16
9:      unchangearr ← Malloc(|T|/step)
10:     SetFalse(unchangearr)
11:     for i in range(0, |T|, step) do
12:         T' ← BitFlip(T, i, step)
13:         currhash ← RunAndUpdate(P, T')
14:         if Contains(Hitmap, currhash) then
15:             unchangearr[i] ← True
16:     for mutation in deterministicMutations do
17:         for i in range(0, |T|) do
18:             if unchangearr[i/step] then
19:                 continue
20:             T' ← Mutate(T, mutation)
21:             RunAndUpdate(P, T')
22:     MutateHavocStage(T, P)
23:     Free(unchangearr)
24: procedure RunAndUpdate(P, T')
25:     RunAndSave(P, T')
26:     hash ← CalBitmapHash(Bitmap)
27:     SetTrue(Hitmap, hash)
```

data structure *HitMap* to address C2 and incorporate this technique into AFL.

Before presenting our *hotspot-aware fuzzing* algorithm, we first introduce a new data structure named *HitMap* to AFL. Each bit of *HitMap* represents the status of related code coverage, where 0 indicates that this code coverage status hasn't been achieved by any test cases and 1 otherwise. The *trace_bits* is used to record the coverage information that a test case achieves after each running, so we take its 32-bit hash value, which is calculated by AFL, as the key of *HitMap*. To hold all different values, we set the size of *HitMap* to be 512MB ((2^32)/8 bytes). Due to the low price of memory cards, it's easy to configure a fuzzing machine with large memory (*e.g.*, 128GB) for software testing or academic research.

At the beginning of fuzzing, every bit of *HitMap* is initialized to 0. Then AFL-hot continuously updates *HitMap* after each running of the test case. A bit in *HitMap* is set to 1, if and only if, there exists a running getting a coverage which its key represents. As shown in Figure 5, *trace_bits* is updated to `t1` after a running, then it is mapped to `key1` by a hash function. AFL-hot sets 1 in *HitMap* using `key1` as the key afterwards.

There are two phases in the *hotspot-aware fuzzing* as shown in Algorithm 5. In the first phase, *hotspot-aware fuzzing*

performs a pre-evaluation process for each test case (Line 11-15). First, it allocates an array `unchangearr` storing the pre-evaluation result of the current test case `T`, whose size is determined by the length of `T` as well as the variable `step`. And every element in `unchangearr` represents whether control flow is sensitive to the related region in `T`. Then *hotspot-aware fuzzing* iteratively mutates `step` bytes in `T` with the stepover `step`, *i.e.*, flips all bits of a specific region which contains `step` bytes at a time (Line 12). After each mutation, *hotspot-aware fuzzing* checks the running result of the mutated test case. If hash value of *trace_bits* has previously appeared (*i.e.*, corresponding bit is 1 in *HitMap* using calculated hash value as the key), it means that the mutated region in `T` contributes nothing to new code coverage. So *hotspot-aware fuzzing* marks true (Line 15) in `unchangearr` indicating AFL shouldn't mutate this region later.

In the second phase, *hotspot-aware fuzzing* checks whether current mutation offset is control flow sensitive by inspecting `unchangearr`. More specifically, *hotspot-aware fuzzing* divides current mutation offset by `step`, and uses the division result as the key to acquire control flow sensitive information in `unchangearr` (Line 18). If current offset is control flow sensitive, *hotspot-aware fuzzing* mutates at current offset as normal deterministic mutation does (Line 20-21). If not, skip current mutation offset and continue checking next offset (Line 19). Finally, *hotspot-aware fuzzing* frees `unchangearr` to avoid memory leakage.

V. Evaluation

We build a prototype named AFLTurbo by extending AFL 2.52b. To evaluate its effectiveness in terms of both path discovery and vulnerability detection, we conduct a series of experiments with various benchmarks.

A. Experiments setup

1) Baseline Fuzzers: We compare AFLTurbo against three representative state-of-the-art fuzzers, including AFL, AFLFast and FairFuzz. These baseline fuzzers are chosen based on the following considerations. AFL is one of the most popular coverage-based greybox fuzzers and many researches ([10], [11], [12], [15], [17], [21], [24], etc.) take it as baseline fuzzer. AFLFast improves AFL's performance by developing a new power schedule to prioritize test cases which hit low-frequency paths. FairFuzz is also based on AFL and it accelerates AFL by prioritizing test cases which hit rare branches. Note that we don't select UnTracer and [30], because UnTracer is targeted to the instrumentation and [30] is used to achieve scalability under multi-core machines.

2) Benchmarks: We measure our fuzzer on two datasets. One consists of eight real-world open-source programs detailed in Table I. These programs are popular in reality and commonly tested by other researchers, including tools for image processing (e.g., *libtiff*, *libpng*) and linux utilities (e.g., *nm*, *readelf*). Particularly, *readelf*, *nm* and *c++filt* are tested by both AFLFast and FairFuzz. The other is the LAVA-M dataset, which has been extensively used to evaluate fuzzer's performance by [11], [12], [15], [17], [24], [31], etc.

TABLE I: Programs used in real-world evaluation

Program	Description	Command line	version
libtiff	TIFF library and tool	`tiff2pdf @@`	4.1.0
libpng	PNG library and tool	`readpng`	1.6.37
libsixel	SIXEL encoder/decoder	`img2sixel @@`	1.8.6
exiv2	Image metadata library and tool	`exifprint @@`	0.27.2
mozjpeg	JPEG encoder	`djpeg-static @@`	3.3.1
nm	Linux utils	`nm @@`	2.34
c++filt	Linux utils	`c++filt`	2.34
readelf	Linux utils	`readelf -a @@`	2.34

3) Experiment Infrastructure: The experiments are conducted on two servers. One is a 40-core machine configured with two Xeon CPU E5-2687W v3 processors and 96 GB RAM, and the other is a 56-core machine configured with two Xeon E5-2680 v4 processor and 32 GB RAM. Both machines are installed ubuntu 18.04. To eliminate experiments inaccuracy caused by environment, we only test one specific software on one machine. For instance, we run *c++filt* only on the first machine by all fuzzers.

B. Performance of path discovery

In this section, we mainly evaluate our tool's effectiveness in terms of path discovery.

1) Setup: We run four different configurations of our fuzzer (AFLTurbo, AFL-int, AFL-loc and AFL-hot) and three baseline fuzzers on 8 real-world programs shown in Table I. To mitigate the randomness of fuzzing, we run each fuzzer on each program with a single core for 24 hours and repeat each experiment 10 times. In addition, initial seeds play a critical role in fuzzing. We use randomly collected test cases which are small in order to increase execution speed and valid in order to increase the speed of path discovery in 24 hours. Otherwise, fuzzers would need more time (e.g., 7 days or more) to find the same number of paths if using invalid and big seeds, which is time-consuming for the purpose of fuzzer comparison.

2) Path Coverage: Figure 6 depicts the average number of paths discovered by each fuzzer for each program among 10 runs. From the results, we make two major observations. First, unnecessary mutation overhead limits the path discovery ability of AFL seriously. At the beginning, AFL can cover many paths because initial seeds are small and some of branch constraints are easy to satisfy. After that, AFL continues to mutate blindly and spends most time on useless mutation of test cases. So it's hard to make considerable progress in the future. For example, AFL gets stuck within 2 hours when fuzzing *readelf*, and it finds 1600 paths in the first 2 hours while only 269 paths in the next 22 hours. However, AFLTurbo always keeps a steady speed and finally discovers 3.4 times more paths than AFL.

Second, AFLTurbo achieves the best performance on all programs except *exiv2* where AFL-int discovers the most number of paths in 24 hours. But the trend for *exiv2* is that AFLTurbo will supersede AFL-int if we continue to fuzz it. Compared with AFL, AFLTurbo covers 141% more paths, and

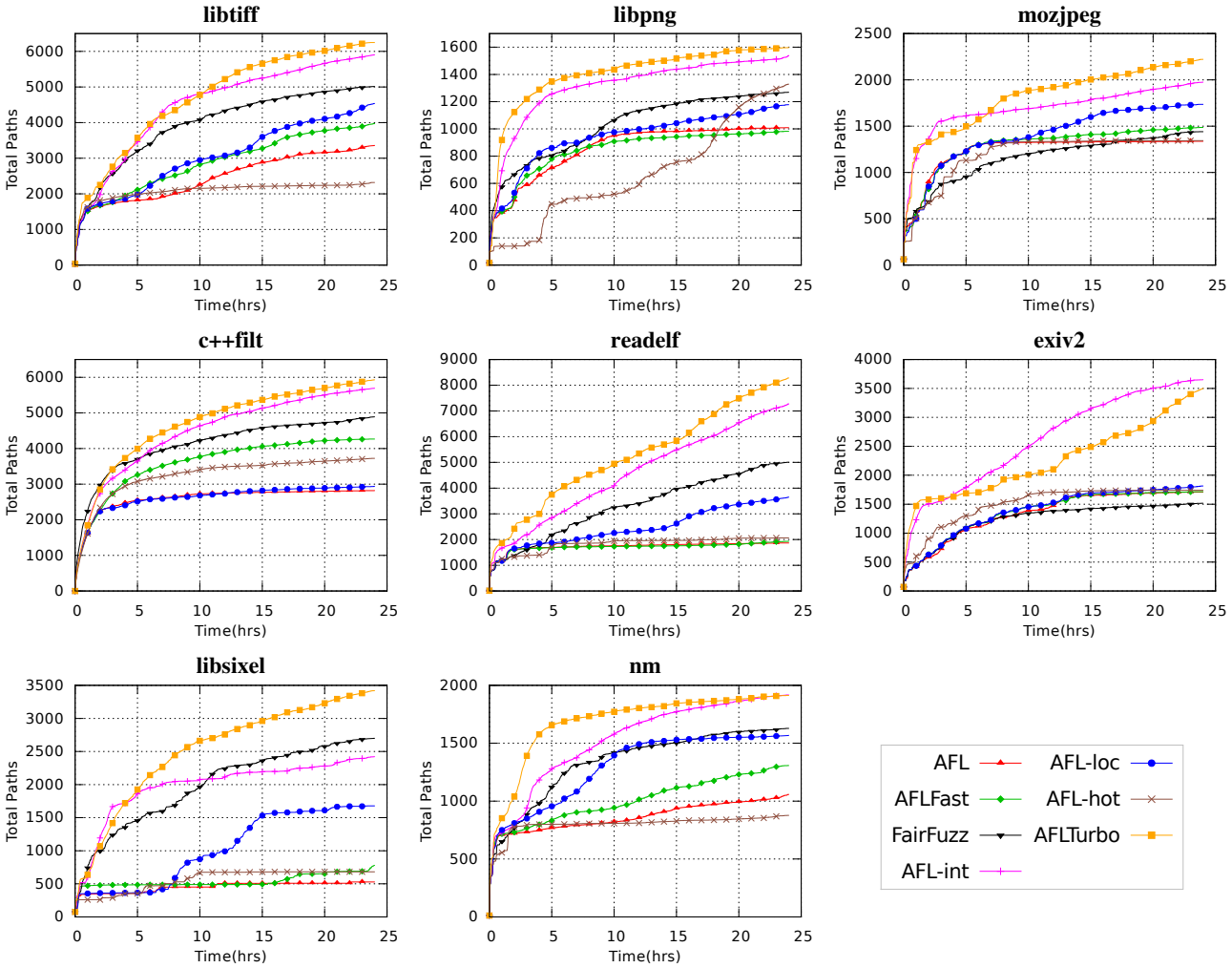

Fig. 6: Average number of unique paths explored by each fuzzers for each real-world program among 10 24-hour runs. Every subgraph presents the evaluation on each program. AFLTurbo has the best path discovery performance in general.

the number of paths discovered by AFLTurbo in the first 3 hours even exceeds the number of paths executed by AFL in 24 hours for almost all programs. Compared with AFLFast and FairFuzz, AFLTurbo still keeps ahead and discovers 101% and 41% more paths respectively, which proves that our improvements are superior to others.

Overall, AFLTurbo significantly outperforms other baseline fuzzers in all cases with respect to path discovery. On average, AFLTurbo explores 141%, 101% and 41% more paths than AFL, AFLFast and FairFuzz respectively

3) Contribution of Each Approach: In this section, we discuss the contribution of each approach to AFLTurbo. Following the controlling variable method, this measurement mainly based on each approach's improvement on AFL.

First, *interruptible mutation* contributes greatly to the performance of AFLTurbo. On the whole, AFL-int explores 120% more paths than AFL as well as performs better than other baseline fuzzers. Take *readelf* as an example, the number of paths discovered by AFL-int maintains a stable growth and AFL-int totally covers 2.89 times more paths than AFL. On the contrary, AFL and AFLFast get stuck after the first two hours, and the performance of FairFuzz is also inferior to AFL-int. Furthermore, as Figure 6 shows, curves of AFLTurbo and AFL-int are very similar and the distance between them is small.

Second, *locality-based mutation* makes considerable contribution to AFLTurbo. Compared with AFL, this approach only has notable improvements on 5 software (*libtiff*, *mozjpeg*, *readelf*, *libsixel* and *nm*) and has almost no improvement on other 3 software (*c++filt*, *exiv2* and *libpng*). It finds more paths than AFLFast. However, it is not as effective as FairFuzz. Overall AFL-loc discovers 39% more paths than AFL.

TABLE II: Total unique crashes and paths found by each fuzzer on each program in LAVA-M among 10 times. AFLTurbo has the best bug detection performance.

Fuzzer	base64		md5sum		uniq		who		total	
	crash	path	crash	path	crash	path	crash	path	crash	path
AFL	0	1432	0	4350	0	1221	8	1183	8	8186
AFLFast	0	1480	0	4341	0	1279	4	1377	4	8477
FairFuzz	15	1811	0	4699	1	1255	4	1506	20	9271
AFL-int	11	1811	3	5910	15	1261	14	1845	43	10827
AFL-loc	0	1377	0	4259	0	1272	6	1295	6	8203
AFL-hot	0	1493	0	4851	3	1375	18	1842	21	9561
AFLTurbo	87	1847	3	6586	17	1321	17	1845	124	11599

Third, *hotspot-aware fuzzing* only has little impact on AFLTurbo. This is because we only use smaller seeds as input to those fuzzers, *hotspot-aware fuzzing* doesn't perform well on small test cases. We further discuss the impact of test cases' size in Section V-D.

C. Bug detection using LAVA-M

We restrict our attention to measuring AFLTurbo's capability of bug detection in this section.

1) LAVA-M: Vulnerability corpora with high quality is vital to bug detection measurement. LAVA proposed by Dolan-Gavitt *et.al.* is a technique which generates ground-truth corpora by automatically and quickly injecting a lot of bugs into source code using taint analysis. LAVA-M is produced by the LAVA technique and has four coreutils programs (*base64*, *uniq*, *md5sum* and *who*), where each program contains many injected bugs. Besides, LAVA-M has been tested extensively by others. So we decide to choose LAVA-M to test AFLTurbo's bug detection capability. To improve the reliability of results, we also fuzz these four programs for 24 hours, and each experiment is repeated 10 times.

2) Conclusion: Table II shows the statistics result of LAVA-M evaluation, including the number of total paths and unique crashes found by AFLTurbo and other fuzzers. We use unique crashes reported by fuzzers instead of bug ID because LAVA-M contains unintended bugs which affect the accuracy of bug detection (*e.g.*, *who* crashes before triggering injected bugs).

First, as we can see, AFLTurbo significantly outperforms all baseline fuzzers in terms of bug detection. AFLTurbo finds 124 unique bugs in total while AFL, AFL-Fast and FairFuzz only find 8, 4 and 20 respectively. Note that AFLTurbo shows the best bug detection capability when fuzzing *uniq* and *base64*. Furthermore, AFLTurbo finds 41% more paths than AFL, 36% more paths than AFL-Fast and 25% more paths than FairFuzz.

Second, each approach makes different contribution to AFLTurbo. Similar to previous section, this comparison is also based on AFL. *Interruptible mutation* benefits AFLTurbo most, and AFL-int finds 40 unique crashes as well as executes 32% more paths than AFL. Note that although none of baseline fuzzers find any bugs in *md5sum* in 24 hours, *interruptible mutation* helps AFLTurbo to find 3 bugs in it. On the other hand, *hotspot-aware fuzzing* boosts AFLTurbo a lot and AFL-hot triggers 13 more bugs than AFL. Moreover, *locality-based mutation* has no remarkable impact on AFLTurbo when fuzzing LAVA-M.

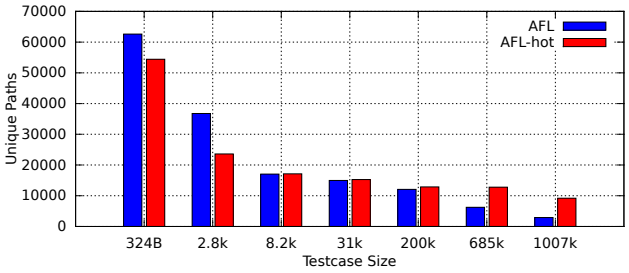

Fig. 7: Unique paths covered by AFL and AFL-hot of different test case size when fuzzing *readelf*. AFL-hot has much better performance than AFL using bigger test case as the seed.

Third, the better path coverage tends to cause better capability in detecting bugs. As shown in the last two columns of Table II, AFLTurbo achieves the maximum number of paths as well as exposes most bugs; in contrast, AFL covers fewer paths and hence triggers fewer bugs.

D. Impact of test case's size

As aforementioned in Section V-B, AFL-hot makes little improvement for AFL on almost all 8 programs. We think this is related to the size of test cases as we only use small seeds. So we decide to conduct an experiment about the impact of the test case's size on the speed of path discovery. We only use two fuzzers (AFL and AFL-hot) to study the relationship between test case's size and the number of newly generated paths in AFL. As for the target program and initial seeds, we choose *readelf* and valid ELF binaries of different sizes (324B, 2.8KB, 8.2KB, 31KB, 200KB, 685KB and 1007KB). We run each fuzzer with provided seed for 24 hours and repeat ten times to reduce the randomness of fuzzing. Figure 7 shows the result of this evaluation.

1) Prevalence of File Size: As we can see, with the increase of the size of test case, the number of generated paths for both fuzzers decreases. This is rational because bigger ELF files contain more data that needs to be processed by *readelf*, which causes execution speed to slow down. As a result, the number of generated paths becomes smaller within the same amount of evaluation time when fuzzing larger test cases. This indicates that prioritizing small seeds has better performance of path discovery.

2) Effectiveness of AFL-hot: After applying the approach *hotspot-aware fuzzing*, we can see that for small test cases, *hotspot-aware fuzzing* doesn't increase path discovery speed and AFL even performs better than AFL-hot. However, as the test case's size becomes bigger, AFL-hot performs much well in terms of path discovery. For example, AFL-hot generates 217% more paths than AFL when the size is 1007k. The reason behind this phenomenon is that only a small proportion of big test cases is metadata and *hotspot-aware fuzzing* biases the fuzzer towards mutating metadata. On the contrary, traditional fuzzer spends lots of time mutating non-metadata, which is

TABLE III: Vulnerabilities detected by AFLTurbo, including name of target programs, CVE IDs, vulnerability description and vulnerability severity.

Project	CVE-ID	Description	Severity
ffjpeg	CVE-2019-19887	NULL pointer dereference	Medium
ffjpeg	CVE-2019-19888	dividing by zero	Medium
libIEC61850	CVE-2019-19930	integer overflow	Medium
libIEC61850	CVE-2019-19931	heap overflow	High
libIEC61850	CVE-2019-19944	out of bound access	Medium
libIEC61850	CVE-2019-19957	out of bound access	Medium
libIEC61850	CVE-2019-19958	integer overflow	Medium
libIEC61850	CVE-2020-7054	heap overflow	High
stb	CVE-2020-6617	assertion failure	High
stb	CVE-2020-6618	heap overflow	High
stb	CVE-2020-6619	assertion failure	High
stb	CVE-2020-6620	heap overflow	High
stb	CVE-2020-6621	heap overflow	High
stb	CVE-2020-6622	heap overflow	High
stb	CVE-2020-6623	assertion failure	High
libsixel	CVE-2019-20056	assertion failure	Medium
libsixel	CVE-2019-20205	integer overflow	High
libsixel	CVE-2020-11721	uninitialized variable	Medium

ineffective. In general, AFL-hot works better with large test cases.

E. Finding zero vulnerabilities

We use AFLTurbo to continuously fuzz open-source projects (*ffjpeg*, *libIEC61850*, *stb* and *libsixel*) acquired from github. Encouragingly, AFLTurbo exposes 20 security vulnerabilities in these four software. And all vulnerabilities are found within 2 days. We report these vulnerabilities to corresponding vendors and are finally rewarded with 18 CVEs listed in Table III. Among these CVEs, 10 of them are rated as high severity and 8 of them are rated as medium severity. At the time of writing, 10 of them have been fixed and 8 of them are still under investigation. This shows that AFLTurbo has an excellent ability of uncovering unknown vulnerabilities in real world.

VI. Conclusion

This paper presents three new approaches to accelerate the path discovery of AFL. We implement a prototype tool on top of AFL and name it as AFLTurbo. Our evaluation results show that AFLTurbo significantly performs better than other state-of-the-art fuzzers in terms of both path discovery and bug detection. Besides, AFLTurbo finds 20 security vulnerabilities in *libsixel*, *libIEC61850*, *ffjpeg* and *stb*, of which 18 are assigned with new CVEs.

References

[1] afl-dynamorio. https://github.com/vanhauser-thc/afl-dynamorio. [Online; accessed 2020-03-01].

[2] afl-dyninst. https://github.com/talos-vulndev/afl-dyninst. [Online; accessed 2020-03-01].

[3] afl-pin. https://github.com/vanhauser-thc/afl-pin. [Online; accessed 2020-03-01].

[4] Dynamorio. https://dynamorio.org/. [Online; accessed 2020-03-01].

[5] Frida. https://frida.re/. [Online; accessed 2020-03-01].

[6] laf-intel. https://lafintel.wordpress.com/. [Online; accessed 2020-03-01].

[7] Locality of reference. https://en.wikipedia.org/wiki/Locality_of_reference. [Online; accessed 2020-03-01].

[8] Peach. https://www.peach.tech/. [Online; accessed 2020-03-01].

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

[10] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, 2017.

[11] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2017.

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

[13] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[14] Ivan Fratric. winafl. https://github.com/googleprojectzero/winafl. [Online; accessed 2020-03-01].

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

[16] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

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

[18] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 627–637, 2017.

[19] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6):190–200, 2005.

[20] Dominik Maier, Benedikt Radtke, and Bastian Harren. Unicorefuzz: On the viability of emulation for kernelspace fuzzing. In *WOOT @ USENIX Security Symposium*, 2019.

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

[22] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

[23] Van-Thuan Pham, Marcel Böhme, Andrew Edward Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 2019.

[24] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[25] Tool Interface Standard. Executable and linking format (elf) specification. *Web: http://refspecs.linuxbase.org/elf/elf.pdf*, 1995.

[26] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[27] David A. Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS*, 2000.

[28] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 724–735. IEEE, 2019.

[29] Y. Wang, Z. L. Chua, Y. Liu, P. Su, and Z. Liang. Fuzzing program logic deeply hidden in binary program stages. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 105–116, 2019.

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

[31] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. {QSYM}: A practical concolic execution engine tailored for hybrid

fuzzing. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 745–761, 2018.

[32] Michał Zalewski. american fuzzy lop. http://lcamtuf.coredump.cx/afl/. [Online; accessed 2020-03-01].

[33] Michał Zalewski. Binary fuzzing strategies: what works, what doesn't. https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html. [Online; accessed 2020-03-01].