2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)

Sequence Coverage Directed Greybox Fuzzing

Hongliang Liang, Yini Zhang, Yue Yu, Zhuosi Xie, Lin Jiang
School of computer science
Beijing University of Posts and Telecommunications
Beijing, China
{hliang,zhangyn2012,revising,xiezhuosi,jianglin}@bupt.edu.cn

***Abstract*—Existing directed fuzzers are not efficient enough. Directed symbolic-execution-based whitebox fuzzers, e.g. BugRedux, spend lots of time on heavyweight program analysis and constraints solving at runtime. Directed greybox fuzzers, such as AFLGo, perform well at runtime, but considerable calculation during instrumentation phase hinders the overall performance.**

In this paper, we propose Sequence-coverage Directed Fuzzing (SCDF), a lightweight directed fuzzing technique which explores towards the user-specified program statements efficiently. Given a set of target statement sequences of a program, SCDF aims to generate inputs that can reach the statements in each sequence in order and trigger bugs in the program. Moreover, we present a novel energy schedule algorithm, which adjusts on demand a seed's energy according to its ability of covering the given statement sequences calculated on demand. We implement the technique in a tool LOLLY in order to achieve efficiency both at instrumentation time and at runtime. Experiments on several real-world software projects demonstrate that LOLLY outperforms two well-established tools on efficiency and effectiveness, i.e., AFLGo–a directed greybox fuzzer and BugRedux–a directed symbolic-execution-based whitebox fuzzer.

***Index Terms*—sequence coverage, greybox fuzzing, directed testing, verifying true positives, crash reproduction**

I. Introduction

Fuzzing is a popular and effective testing technique for automatically discovering bugs in real-world software systems [1]. By mutating a set of provided seed inputs, it generates lots of unexpected or random test cases and feeds them to the program under test (PUT) in order to trigger crashes, assertion violations or other abnormalities. Fuzzing searches a PUT's state space randomly and thus inevitably spends a lot of machine time on code regions unrelated to bugs. Instead, directed fuzzing technique focuses on specific code regions in a program, which can significantly alleviate the randomness of original fuzzing.

Most of the existing directed fuzzers [2], [3], [4], [5], [6], [7], [8], [9], [10], [11] are white box fuzzers based on symbolic execution. They simulate program execution with symbolic values, collect symbolic path constraints, and generate inputs by solving the constraints. For instance, Do et al. [3] employ data dependence analysis and extended chaining approach to build event sequences leading to the target locations, and perform directed dynamic symbolic execution to generate goal-oriented test inputs. To test software patches, KATCH [7] combines symbolic execution with several novel heuristics to reach the patch code quickly. Directed symbolic execution converts the reachability problem to iterative constraint satisfiability problem, and spends most of runtime on heavyweight program analysis and constraint solving, and hence is effective but inefficient [12].

Greybox fuzzers like AFL [13] can generate several orders of magnitude more inputs during the time that symbolic execution generates a single input, so greybox fuzzers can be used to develop lightweight directed fuzzers (e.g. [14], [15], [12], [16]). AFLGo [17] is the state-of-the-art directed greybox fuzzer which focuses on reaching a set of target locations in a program. It takes reachability problem as an optimization problem and aims to minimize the distance of seeds to the target locations by introducing simulated annealing into seed energy schedule. Unfortunately, AFLGo moves most of program analysis to instrumentation phase in exchange of efficiency at runtime. Since it measures seed distance to target locations by calculating the distance between each basic block and a target location in instrumentation phase, AFLGo has to parse the call graph and intra-procedure control flow graph of the PUT. Both parsing graphs and calculating distances in the instrumentation phase is very time consuming. If users are sensitive to time consuming or have limited computing resources, the above overhead will be their serious concern. Therefore, such users will prefer a lightweight analysis approach, which can reduce the computing resource requirements and overall analysis time.

To resolve the problem, we propose a novel lightweight directed fuzzing technique SCDF, which explores towards the user-specified program statements efficiently. Given a set of target statement sequences of a program, our approach aims to generate inputs that can reach the statements in each sequence in order and trigger bugs in the program. To measure a seed's quality, SCDF exploits the seed's ability of covering the target sequences, which we name as *"sequence coverage"*. SCDF gives a seed more opportunity (i.e. energy) to generate new mutations if the seed achieves higher sequence coverage. Our approach can be used in a variety of scenarios, such as bug detection, directed testing and crash reproduction. We implement our approach in LOLLY, a fuzzer built on AFL. Evaluation on several real-world software shows that it is effective and efficient due to its directness and lightweight.

In summary, the main contributions of this paper are:

- A lightweight directed fuzzing technique SCDF which is guided by user-specified statement sequences, and a novel energy schedule algorithm which adjusts a seed's energy according to its ability, calculated on demand, of covering the given statement sequences;

978-1-7281-1519-1/19/$31.00 ©2019 IEEE
DOI 10.1109/ICPC.2019.00044

- A fuzzer named LOLLY which implements the above technique and algorithm to expose/verify bugs;
- An evaluation, which shows LOLLY's better effectiveness and efficiency than two state-of-the-art tools, i.e. AFLGo and BugRedux.

The rest of this paper is structured as follows. In section II, we explain our motivation. In section III, we give an overview and then detail the design of our approach. We describe the implementation of our prototype tool LOLLY in section IV and evaluate the performance of LOLLY in section V. Section VI elicits the threats to validity. The survey of related work in section VII is followed by our conclusion in VIII.

II. MOTIVATION

AFLGo focuses on reaching a given set of target locations in a program and it treats these targets as independent. Therefore, the metric it uses to measure a seed is the distance of the seed to target program points. To reduce runtime overhead, AFLGo chose to statically calculate the distance from each basic block to the target points during instrumentation phase. Specifically, the control flow graph and call graph of the program are parsed and distance information of each node (i.e. basic block) in the graphs is calculated and kept in the node during instrumentation phase. As a result, AFLGo is efficient at runtime as all the distance information required to measure a seed is already calculated.

Though AFLGo is successfully applied in patch testing and crash reproduction, it has two serious issues due to its static distance calculation scheme. First, users must undertake the expensive burden in instrumentation phase and its efficiency in total fuzzing is not obvious. For instance, in our experiments, AFLGo spent nearly *2h* on compiling `Libming` program while AFL spent *40s*; AFLGo spent over *4h* on compiling `Libxml2` program, in comparison, AFL spent *1m44s*. Second, AFLGo is inconsistent to some extent with the dynamic nature of fuzzing because of two facts: 1) it has to statically parse graphs and calculate distances for all nodes in advance, though only part of pre-computed distances are used at runtime; 2) whenever users specify new or changed target statements, which is common case in testing, AFLGo has to re-instrument the program under test, and further hinders its efficiency and exacerbates users' burden.

Above observations motivate us to research on a novel directed fuzzing technique with better efficiency, which is named SCDF. SCDF allows that users specify a set of target statement sequences which they are interested. SCDF has two important features: sequence coverage and energy schedule. First, for each target statement sequence, SCDF considers the order of the statements which are explored. For each seed, SCDF evaluates the seed's ability of covering given statement sequences at runtime, which is called sequence coverage, and leverages it as the measure of distance (i.e., seed to target statement sequences). Second, using this novel measure of distance, we propose a new energy schedule algorithm in order to control the fuzzing process by adjusting the seeds' energy. The energy of a seed is defined as the number of new inputs generated from the seed. SCDF prefers the seeds with higher sequence coverage, and assigns more energy to them.

III. DESIGN

A. Overview

Fig.1 shows the high-level architecture of SCDF. It mainly consists of two components: SCDF Instrumentor and SCDF Fuzzer. The inputs of SCDF are: a) source code files of a software project under test, b) a set of statement sequences and c) initial seeds for fuzzing. Its output is a set of test cases that trigger bugs in the project.

A statement sequence (SS for short) is a set of statements that SCDF tries to generate test cases to arrive in order, in other words, an SS is considered as a path fragment that seeds are expected to execute at runtime. An SS can be specified by users or provided by other analysis tools (e.g., method calls in a crash dump). Each statement in an SS is identified by its source file and line in source code. Instead of treating each target location independently as AFLGo does, in SCDF, each given sequence is considered to be independent while statements in each sequence are expected to be reached in order.

Next we describe the workflow of SCDF shown in the Fig.1. Given the source files of a project under test (a), at the time of compiling the project, SCDF Instrumentor instruments the files to facilitate collecting information at runtime, such as branch coverage in AFL, and produces an instrumented binary (d). Meanwhile, SCDF Sequences Transformer uses the debug information to map each statement in the sequences to the corresponding basic block, and each block is identified by a location (i.e. a unique integer generated randomly). As a consequence, each SS is converted to a basic block location sequence (BLS for short) (c).

Our seed energy schedule algorithm named SCDF Strategy is integrated in SCDF Fuzzer. Beginning with the initial seeds (g), SCDF Fuzzer analyzes the instrumented binary (e), calculates the sequence coverage of each seed using the BLSs (f), and schedules the energy of all seeds. The fuzzer finally generates test cases which can trigger crashes or expose bugs in the project under test.

B. Instrumentation

To calculate the sequence coverage of a seed on a BLS, SCDF needs to record the trace of basic blocks in the BLS that the seed executes during a run, called *target basic block execution trace* (TBBET for short). Therefore, SCDF Instrumentor only instruments the basic blocks each of which contains at least a target statement. Moreover, SCDF uses a shared memory to sequentially record *location identifiers* of the blocks in TBBET following the order in which they are executed. More precisely, we denote the shared memory as an integer array $M[n]$, each of which records the location identifier of a block in a TBBET. M_0 stores the number of executed basic blocks, initially 0. M_i is used to record the location identifier of the *i*th basic block in the TBBET, $i \in \{1, ..., n-1\}$.

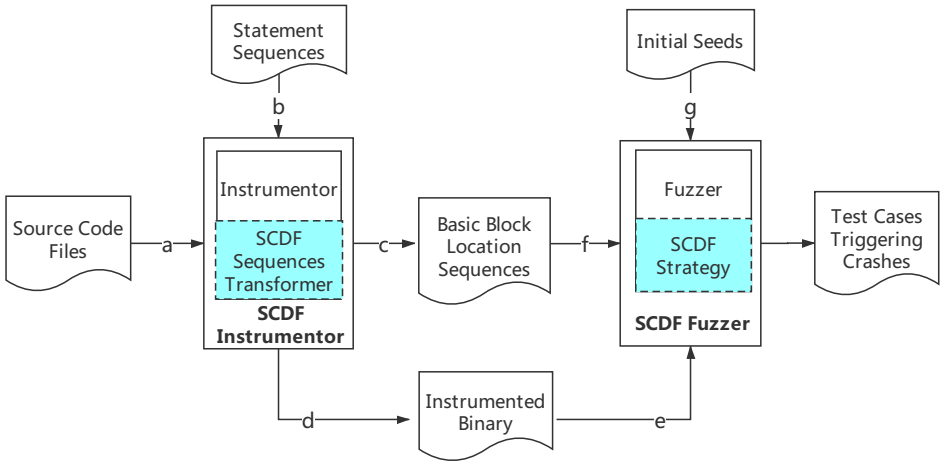

Fig. 1. SCDF's architecture

Specifically, while compiling a project under test, SCDF Instrumentor traverses each statement in a basic block, and obtains the source files and lines of the statements with the help of debug information, recording them in a set *StmtSet*. If a statement in a SS is in *StmtSet*, the basic block is labeled as a target basic block. At the entry of each target basic block, SCDF Instrumentor adds the following instructions:

1) Get the shared memory pointer, i.e., M;
2) Read M_0's value, e.g., i, and add 1 to it, i.e., $i = i + 1$;
3) Write the location identifier of current basic block to M_i;
4) Store the value of i back to M_0.

During the process of fuzzing, when a seed explores the target basic blocks in a BLS, the instrumented code is executed, and thus the trace of basic blocks in the BLS (i.e., TBBET) is sequentially recorded in the shared memory.

C. Fuzzing

Algorithm 1 shows the process of the directed fuzzing in SCDF approach. The inputs of the algorithm are: a) A set of seed inputs S; b) The given BLSs $BBSeqs$. The output is a set of test cases TC that can trigger crashes or expose bugs in the project under test. SCDF fuzzer first selects a seed s from S via SelectSeed (line 2). RunSeed (line 3) executes the instrumented binary with seed s, and with the help of instrumented code, the fuzzer obtains the target basic block execution trace $exeTrace$ from the shared memory. Then for each BLS $BBSeq_i$ in $BBSeqs$, the fuzzer compares $exeTrace$ and $BBSeq_i$ to calculate the sequence coverage of seed s by invoking CalSeqCov (line 7), whose algorithm is shown in Algorithm 2. After calculating the sequence coverage of $exeTrace$ on each BLS, the fuzzer takes the average value as the coverage capability cov of seed s (lines 8-12). The seed energy schedule algorithm of SCDF assigns energy to seed s based on its sequence coverage (line 14). Seed energy determines the number of new inputs generated by mutating the seed. The mutation strategy is implemented in MutateSeed (line 16), where the fuzzer generates new inputs by performing mutations, e.g., bit/byte flips, addition or subtraction, splicing two distinct inputs at a random location.

For each new input s' generated by mutation, if it causes the program to crash or exposes a bug, it is added to TC (lines

Algorithm 1 SCDF's directed fuzzing

Input:
a. Seed inputs S
b. Basic block location sequences $BBSeqs$

```
1:  repeat
2:    s = SelectSeed(S)
3:    exeTrace = RunSeed(s)
4:    cov = 0    //sequence coverage of s on BBSeqs
5:    bnum = 0 //number of sequences on each of which s
      gains coverage greater than 0
6:    for each BBSeq_i in BBSeqs do
7:      bcov = CalSeqCov(BBSeq_i, exeTrace)
8:      if bcov > 0 then
9:        bnum = bnum + 1
10:       cov = cov + bcov
11:     end if
12:     cov = cov / bnum
13:   end for
14:   e = AssignEnergy(s, cov)    // energy schedule for s
15:   for i from 1 to e do
16:     s' = MutateSeed(s)
17:     if s' crashes or exposes a bug then
18:       add s' to TC
19:     else if IsInterested(s') then
20:       add s' to S
21:     end if
22:   end for
23: until time budget exhausted or abort-signal
```

Output: Test cases TC that trigger crashes or expose bugs in the project

Algorithm 2 CalSeqCov: Calculating sequence coverage

Input:

a. A basic block location sequence (BLS) $BBSeq_i$, e.g. $< B_0, B_1, ..., B_n >$

b. The target basic block execution trace $exeTrace$, e.g. $< t_1, t_2, ..., t_m >$

```
1:  maxSClen = 0
2:  curGoal = B_0
3:  curSClen = 0
4:  for each t_i in exeTrace do
5:    if curGoal == t_i then
6:      curSClen = curSClen+1
7:      curGoal = the block that just follows curGoal in BBSeq_i
8:    else if t_i is prior to curGoal in BBSeq_i then
9:      maxSClen = max(maxSClen, curSClen)
10:     curSClen = 1
11:     curGoal = the block that just follows t_i in BBSeq_i
12:   else if t_i is posterior to curGoal in BBSeq_i then
13:     curSClen = curSClen+1
14:     curGoal = the block that just follows t_i in BBSeq_i
15:   end if
16: end for
17: maxSClen = max(maxSClen, curSClen)
18: cov_i = maxSClen / len(BBSeq_i)
```

Output: Sequence coverage cov_i for the ith sequence in BBSeqs (i.e. $BBSeq_i$)

17-18). If it covers a new branch, it is added to S as a seed input for subsequent fuzzing process (line 19-20). If neither of the above is true, the input is discarded. The fuzzer repeats the process until time budget is exhausted or an abort signal is received (line 23).

D. Sequence Coverage Calculation

Below we describe in detail the algorithm for calculating the sequence coverage of a seed s on a BLS. As shown in Algorithm 2, CalSeqCov() function is given a BLS (i.e., $BBSeq_i < B_0, B_1, ..., B_n >$) and the target basic block execution trace of current seed s (i.e., $exeTrace < t_1, t_2, ..., t_m >$). The algorithm traverses each basic block location recorded in $exeTrace$, and compares it with the locations of target basic blocks in $BBSeq_i$. It aims to find in the two sequences, i.e. $exeTrace$ and $BBSeq_i$, the longest common sub-sequence, which is called *maximum sequential coverage*.

At the beginning of the algorithm, $maxSClen$ which records the length of the maximum sequential coverage is initialized as 0 (line 1). The $curGoal$ denotes the target block that $exeTrace$ is expected to reach, and is initialized to be the first target in $BBSeq_i$ (line 2). The $curSClen$ is the current sequential coverage length of $exeTrace$, initially 0 (line 3). In a loop of traversing each basic block location t_i in $exeTrace$ (line 4), the algorithm handles the following three situations:

1) If t_i is the same block with the current goal $curGoal$, the algorithm updates $curSClen$ to $curSClen$+1 and $curGoal$ to the next goal[1] following $curGoal$ in $BBSeq_i$ (lines 5-7);
2) If t_i is a goal before $curGoal$ in $BBSeq_i$, which means that $exeTrace$ does not reach in order the target blocks in $BBSeq_i$, the algorithm updates $maxSClen$ to max($maxSClen$, $curSClen$), resets $curSClen$ to 1 and updates $curGoal$ to the next target following t_i in $BBSeq_i$ (lines 8-11);
3) If t_i is a goal after $curGoal$ in $BBSeq_i$, $exeTrace$ is still considered to reach the target blocks in $BBSeq_i$ in order, but those target blocks between $curGoal$ and t_i are missed. So the algorithm updates $curSClen$ to $curSClen + 1$ and updates $curGoal$ to the next target following t_i in $BBSeq_i$ (lines 12-14).

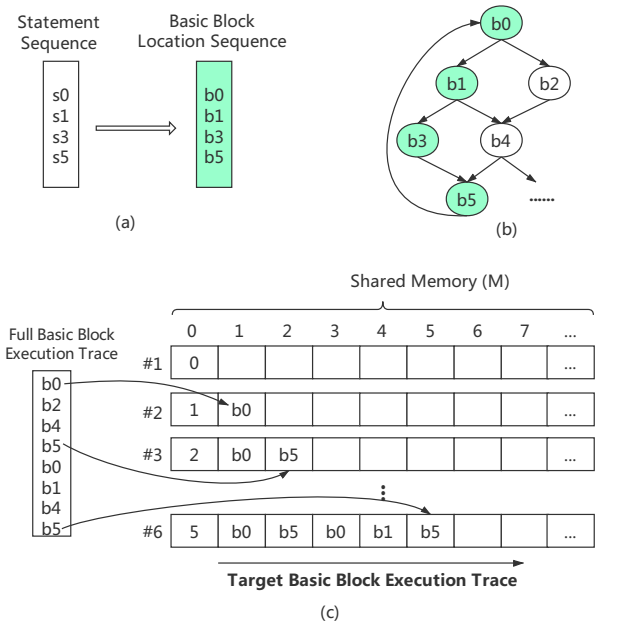

Fig. 2. An example of SS, BLS and basic block execution trace. (a) denotes the conversion from statement sequence to basic block location sequence; (b) depicts the nodes in the control flow graph; (c) describes how SCDF records the target basic block execution trace in the shared memory.

After exeTrace is traversed, the algorithm performs the last update (line 17) and calculates the *coverage of exeTrace on* $BBSeq_i$ (line 18) as:

$$cov_i = maxSClen/len(BBSeq_i) \quad (1)$$

where $len(BBSeq_i)$ is the number of blocks in $BBSeq_i$.

To further illustrate the Algorithm 2, suppose that the BLS is <b0, b1, b3, b5> and the TBBET of a seed s is <b0, b5, b0, b1, b5> as shown in Fig.2. At the beginning, the algorithm performs initialization of three variables as $\{curGoal = b0, curSClen = 0, maxSClen = 0\}$. The algorithm then traverses the trace and meets the first block b0, which is the same with $curGoal$, so three variables are updated

[1]If curGoal is the last block in $BBSeq_i$, we define its next one as itself.

as $\{curGoal = b1, curSClen = 1, maxSClen = 0\}$. The algorithm continues to handle the next block b5, a target block after $curGoal$. Since b5 is the last one in the BLS, $curGoal$ is maintained to be b5, so current result of three variables is $\{curGoal = b5, curSClen = 2, maxSClen = 0\}$. Next, the algorithm meets b0 again in the trace, since b0 is a target before $curGoal$ in the BLS, $maxSClen$ is updated as the maximum of $maxSClen$ and $curSClen$ (Line 9) and we get $\{curGoal = b1, curSClen = 1, maxSClen = 2\}$. The next b1 makes the result be $\{curGoal = b3, curSClen = 2, maxSClen = 2\}$. Finally, the last block in the trace b5 is met, and it is a target after $curGoal$ in the BLS, so three variables are updated as $\{curGoal = b5, curSClen = 3, maxSClen = 2\}$ and the for loop ends. Then the algorithm update $maxSClen$ to 3. Since the length of the BLS is 4, the sequence coverage of the seed s on the BLS is 3/4=0.75.

E. Seed energy schedule

Before fuzzing or mutating a seed s, the fuzzer calculates the sequence coverage of s and leverages it to determine the number of mutated inputs from s, i.e. the seed energy. This strategy is implemented in SCDF's seed energy schedule algorithm. We first define the sequence coverage $seqcov$ of a seed s. Suppose the target basic block execution trace of s is $exeTrace$, SCDF calculates the coverage cov_i of $exeTrace$ on each BLS $BBSeq_i$ and takes their average value as the *sequence coverage seqcov of s*, which is:

$$seqcov = avg(cov_i) \tag{2}$$

The coverage of $exeTrace$ on a BLS is calculated according to the algorithm 2. The sequence coverage of an input reflects the capability of the input to cover the given sequences. Therefore, the higher sequence coverage an input achieves, the more energy SCDF assigns to it.

In addition, AFLGo presented simulated annealing-based power schedule for greybox fuzzing. It regards greybox fuzzing as a Markov chain, and thus fuzzing process can be optimized with Simulated Annealing (SA for short). Classical random walk schedule always accepts better solutions which may be trapped in local optimum. Unlike random walk, SA accepts a solution that is worse than the current solution with a certain probability, so it is possible to jump out of the local optimum and reach the global optimal solution. This probability gradually decreases as the control parameter *temperature* decreases. An initial temperature is given at the beginning of SA, and as the temperature continues to decrease, the algorithm will find an approximate optimal solution to the problem in polynomial time. The temperature reduction follows a specified cooling schedule algorithm.

SCDF borrows the above idea from AFLGo and applies simulated annealing to our seed energy schedule algorithm for global optimum. For the directed fuzzing in SCDF, an optimal solution is a test case that can achieve maximum sequence coverage. In our approach, temperature T with an initial value $T_0 = 1$ decreases following an *exponential cooling schedule* as:

$$T = T_0 \times \alpha^k \tag{3}$$

where α is a constant satisfying $0.8 \leq \alpha \leq 0.99$ and k is the temperature cycle. The threshold of temperature T_k is set to 0.05, and SCDF Fuzzer will not accept worse solutions when the temperature is lower than T_k. Specifically, if $T_k > 0.05$, the cooling schedule is in the exploration stage during which SCDF randomly mutates the provided seeds to generate many new inputs. Otherwise, it enters the exploitation stage during which SCDF generates more new inputs from seeds that have higher sequence coverage. In this case, the simulated annealing process is comparable to a classic gradient descent algorithm.

Since the common limitation in fuzzing is the time budget, time t is used as a factor to adjust temperature cycle k as:

$$k/k_x = t/t_x \tag{4}$$

where k_x and t_x are the temperature cycle and time when the temperature reaches T_k, respectively. Therefore, we can establish the *relationship between time t and temperature T by means of k*:

$$T_k = 0.05 = \alpha^{k_x} \tag{5}$$

$$T = \alpha^k = \alpha^{\frac{t}{t_x} \times \frac{log(0.05)}{log(\alpha)}} = 20^{-\frac{t}{t_x}} \tag{6}$$

Similar with the annealing-based energy schedule in AFLGo, given a seed s whose sequence coverage is $seqcov$, we define the *capability of s to cover the given statement sequences* is:

$$capcov = seqcov * (1 - T) + 0.5 * T \tag{7}$$

At the beginning of fuzzing, the initial value of temperature T is 1, that is, $capcov$ is independent on the sequence coverage. As time goes by, the temperature T gradually decreases and the sequence coverage becomes increasingly important.

To integrate SCDF Strategy into the existing seed energy schedule algorithm of a fuzzer (e.g. AFL), SCDF integrates the capability of covering the sequences as an impact factor into the seed energy calculation formula. SCDF calculates the *integrated energy for a seed* as:

$$Lenergy = energy * 2.0^{(capcov - 0.2) * 10} \tag{8}$$

where $energy$ is the original energy given by the original fuzzer and $Lenergy$ is the energy given by SCDF Fuzzer which integrates SCDF Strategy in the original fuzzer.

IV. IMPLEMENTATION

In our current implementation, we integrate our approach into American Fuzzing Lop (AFL) 2.52b and implement a prototype system LOLLY. In theory, the instrumentor and fuzzer in our approach can be any tool that can accomplish the tasks described in section III.

AFL instrumentor uses LLVM pass to instrument a software project under test (PUT) during compilation. First, the front

end of LLVM compiler converts the PUT's source code to LLVM IR. Then the instrumentor traverses all basic blocks of each function in IR and produces a random integer for each basic block as its location identifier. LOLLY utilizes these location identifiers to convert a statement sequence (SS) to a basic block location sequence (BLS) during the instrumentation phase. Specifically, LOLLY's instrumentor first gets the source file name of currently compiled module and then reads the related SS if they exist in the file. With the help of debug information, LOLLY's instrumentor maps the statements in each SS to location identifiers of the basic blocks that contain the statements. Once all source files are handled, all SS are converted to BLS.

AFL uses a 64KB-shared memory to save branch execution information at runtime. LOLLY uses additional 2M bytes as an array to record the target basic block execution traces (TBBET), where on a 64-bit architecture, the first element of the array records the number of exercised basic blocks and the remaining space is used to sequentially record the location identifiers of the exercised basic blocks.

AFL fuzzer mutates seeds to generate new inputs. It considers each seed as a byte sequence and performs two stages of mutations: deterministic stage and havoc stage. At deterministic stage, AFL traverses every byte of a seed and makes several kinds of mutations on consecutive positions in the byte sequence. The number of generated new inputs in this stage depends on the length of the seed. At havoc stage, AFL makes a series of random mutations on the seed and the number of produced new inputs depends on the energy that AFL assigns to the seed. AFL assigns energy to each seed to be fuzzed based on its execution time, branch coverage, and how late and deep it is discovered, etc. In our approach, after executing a seed input, LOLLY obtains the TBBET from the extended space in the shared memory and computes the sequence coverage of the seed according to the given BLSs. LOLLY integrates SCDF Strategy into AFL's energy schedule algorithm and leverages sequence coverage of a seed as main energy measure.

V. EVALUATION

A. Infrastructure

We executed all experiments on a laptop computer with an Intel Core CPU i7-6500U processor that has 4 logical cores running at 2.5GHz with access to 12GB of main memory and Ubuntu 16.04 (64 bit) as operating system. To evaluate the effectiveness and efficiency of LOLLY, we compare LOLLY with two state-of-the-art directed fuzzing tools, i.e., AFLGo and BugRedux. We evaluated them with the same programs under test, initial input corpus, time budget, computing resources, and target locations as LOLLY.

B. True positives verification

In the software development cycle, developers and testers usually apply analysis tools to discover bugs or vulnerabilities in the programs before release. Static analysis is a popular analysis technique to find structural errors and security vulnerabilities in programs. Since it does not execute the programs actually, static analysis usually has high false positive, and thus requires a lot of manual efforts to verify the analysis results. Due its directed execution feature, directed fuzzing technique can be used for automatic verification of bugs. Specifically, the analysis results of static analyzers, e.g. Clang static analyzer [18], which contain potential bugs or vulnerabilities in a program, can be converted to the target statements of directed fuzzing, which can generate test cases exposing real bugs in the program.

We use Libming 0.4.8 [19], a library for generating and reading with Macromedia Flash files (.swf) written in C, as our subject program. We leverage Clang static analyzer to analyze Libming and obtain some statements, which witness the found potential bugs by the way of vulnerable paths. We provide these statements as targets to LOLLY and AFLGo, a state-of-the-art directed greybox fuzzer. In the experiment, the analysis results of Clang analyzer are not purposely filtered, and thus may contain false positives and infeasible paths.

In order to evaluate the efficiency of two fuzzers, we use them to trigger the same CVE vulnerabilities of Libming guided by the above target statements, and compare the time cost that two fuzzers take in the instrumentation phase and runtime phase. We repeated all experiments 20 times and used the average values.

The CVE vulnerabilities used in experiments are listed in Table I, which shows their CVE-ID and vulnerability type. Both AFLGo and LOLLY successfully generated crash inputs triggering the vulnerabilities in Table I, in other words, for each given CVE, the generated crash inputs by two fuzzers can produce exactly the same stack traces as that in the CVE's description.

In the experiments, LOLLY and AFLGo spent 41.568s and 119m12.512s respectively in instrumentation phase. As a baseline, the original AFL took 39.718s. Obviously, Calculating the distances between each basic block and targets in AFLGo is significantly expensive.

In runtime phase, we ran LOLLY and AFLGo to fuzz Libming in order to trigger the CVE vulnerabilities listed in Table I. We set the time budget as 5 hours, where the exploration phase of fuzzing is 1 hour.

The experimental results are shown in Table II. The first column of Table II is the CVE ID of each vulnerability and the second column indicates the fuzzing tools. The third column shows the number of runs that successfully trigger a particular vulnerability. Time-to-Exposure (TTE) measures the length of the fuzzing campaign until the first test input is generated that exposes a given vulnerability. And the forth column shows the mean of TTE values in all 20 experiments. In particular, if a tool fails to trigger a vulnerability in a run within the time limit, its TTE is uniformly recorded as the time budget. The factor improvement (Factor) measures the performance gain as the mean TTE of AFLGo divided by the mean TTE of LOLLY. Values of Factor greater than one mean that LOLLY outperforms AFLGo. The Vargha-Delaney statistic (A_{12}) is

TABLE I
CVE VULNERABILITIES OF LIBMING USED IN EXPERIMENTS

CVE-ID	Type of vulnerability
2017-16883	NULL pointer dereference
2018-7874	Invalid memory address dereference
2018-7876	Memory exhaustion
2018-8807	Use after free
2018-8961	Use after free
2018-8962	Use after free
2018-8963	Use after free

a recommended standard measure for evaluating randomized algorithms [20]. Given a performance measure M (e.g., TTE here) seen in m measures of LOLLY and n measures of AFLGo, the A_{12} statistic measures the probability that running LOLLY yields higher M values than running AFLGo, which indicates the confidence that LOLLY performs better than AFLGo.

As shown in Table II, LOLLY is `2.44X` to `12.85X` faster in most cases than AFLGo at runtime. A single exception occurs for CVE-2018-8807, LOLLY triggered the vulnerability in 50% of runs while AFLGo succeeded in 60% of runs, and LOLLY spent a little longer time than AFLGo. But the A_{12} value (i.e., 0.5) means that they had similar performance statistically. For CVE-2018-8961, LOLLY outperforms AFLGo on the Factor value, though the A_{12} value is 0.35. That's because LOLLY successfully exposed the vulnerability in *all* runs and TTE of most runs is 14 minutes, while AFLGo succeeded in 18 runs and most runs completed in 11 minutes, but its mean TTE greatly increases due to the other 2 failed runs. LOLLY performs best on CVE-2018-7874 since it is 12.85 times faster than AFLGo and LOLLY outperforms AFLGo with 85% confidence.

In summary, experimental results show that LOLLY's directed fuzzing is effective in exposing bugs or vulnerabilities in programs and more efficient than AFLGo.

C. Crash reproduction

Today software systems are typically released with potential bugs or vulnerabilities due to various factors. When a user encounters a crash in a software product, he can submit a crash report to the developers through the built-in crash reporting mechanism in the software. A crash report usually contains information about the crash, such as memory dumps or call stacks. Based on it, developers need to generate test cases that trigger the crash, i.e. reproduce the crash. Directed fuzzing technique can also be applied to crash reproduction.

We evaluate the effectiveness of LOLLY in crash reproduction and compare it with AFLGo and BugRedux [21]. BugRedux is a directed whitebox fuzzer that is built on the symbolic execution engine KLEE [22]. Given a sequence of target locations of a program, BugRedux aims to generate test cases that can exercise the targets in the sequence in order to crash the program.

TABLE II
PERFORMANCE OF LOLLY OVER AFLGO ON LIBMING

CVE-ID	Tool	Runs	μTTE(m)	Factor	A_{12}
2017-16883	LOLLY	20	10.64	1.02	0.65
	AFLGo	20	10.83	-	-
2018-7874	LOLLY	20	9.62	12.85	0.85
	AFLGo	12	123.59	-	-
2018-7876	LOLLY	19	24.31	5.70	0.775
	AFLGo	11	138.59	-	-
2018-8807	LOLLY	10	157.24	0.80	0.5
	AFLGo	12	126.36	-	-
2018-8961	LOLLY	20	14.38	2.81	0.35
	AFLGo	18	40.39	-	-
2018-8962	LOLLY	20	4.54	4.38	0.8
	AFLGo	19	19.91	-	-
2018-8963	LOLLY	20	9.16	2.44	0.75
	AFLGo	19	22.34	-	-
			Average A_{12}		Median A_{12}
	LOLLY		0.668		0.75

In our experiments, the subjects and bugs come from the dataset of BugRedux, which were also used by AFLGo in its evaluation. The dataset of BugRedux contains some test cases which can crash the programs under test, hence we can run a subject with its test cases and obtain the stack dump corresponding to a crash. The time budget for fuzzing is set to 24 hours. We provide the stack dump of each bug to LOLLY, where each stack dump is represented as a sequence of statements. LOLLY performs directed fuzzing with the sequences as targets and generates test cases to reproduce the bugs, i.e. the generated test cases can lead to the same stack dumps. The experiment results are shown in Table III. The first column in Table III is the subjects and bugs to be reproduced, and the remaining columns shows whether three tools successfully reproduce the crashes. If a tool succeeds within the time budget, its result is recorded as "$\surd$". Otherwise marked as "$\times$".

As shown in table III, LOLLY and AFLGo are more effective than BugRedux. They reproduced three times more crashes than BugRedux. AFLGo and LOLLY failed to reproduce the sed.fault1 crash because it requires a fuzzer to fuzz two input files at the same time, for which AFLGo and LOLLY are incapable. The experimental results show that LOLLY can effectively reproduce a crash given its stack dump.

Note that, like other fuzzing techniques, our approach also depends on the given set of initial seeds. If the initial seeds cannot exercise any statements in the given sequences, which is the worst case, LOLLY will get no information about the sequence coverage, and thus acts same as the original AFL.

Moreover, to compare the performance of AFLGo and LOLLY on triggering the crashes in the dataset of BugRedux, we repeated each crash reproduction experiment 5 times and

TABLE III
SUBJECTS AND RESULTS OF CRASH REPRODUCTION

Subjects	BugRedux	AFLGo	LOLLY
sed.fault1	×	×	×
sed.fault2	×	√	√
grep	×	√	√
gzip.fault1	×	√	√
gzip.fault2	×	√	√
ncompress	√	√	√
polymorph	√	√	√

TABLE IV
PERFORMANCE OF LOLLY AND AFLGO ON BUGREDUX DATASET

Subjects	AFLGo		LOLLY	
	Instru. time	Runtime	Instru. time	Runtime
sed.fault1	-	-	-	-
sed.fault2	1m13s	0.07s	2.79s	0.08s
grep	56.61s	0.05s	2.75s	0.11s
gzip.fault1	2.64s	13.27m	0.28s	11.18m
gzip.fault2	45s	16.4s	1.77s	18s
ncompress	1.5s	12.46m	0.09s	11.12m
polymorph	13.34s	10.77m	0.23s	10.86m

use the average values, as shown in Table IV. The first column shows the subjects and bugs to be reproduced. The second and third columns indicate the time AFLGo took at instrumentation time and runtime respectively. The fourth and fifth columns show the time LOLLY took at the two phases respectively. The results show that the performance of LOLLY and AFLGo is comparable statistically at runtime phase. However, LOLLY is superior in terms of overall performance in a whole run which contains both the instrumentation phase and runtime phase. For instance, LOLLY is 25X faster than AFLGo on the sed.fault2 subject.

D. Bugs exposure

In order to verify LOLLY's ability to detect bugs or vulnerabilities as a fuzzer, we evaluated LOLLY on Binutils [23] 2.26, 2.30 and 2.32 versions, Libxml2 [24] 2.9.4, and Libming [19] 0.4.8. Binutils are a set of programming tools for creating and managing binary programs, object file, libraries, profile data, and assembly source code. Libxml2 is a XML C parser and toolkit developed for the Gnome project. We chose them because they are widely used and well tested software. The target statement sequences in this experiment come from the results of Clang analyzer, i.e., these given sequences may contain false positives and unreachable paths. LOLLY aims to explore towards the potentially buggy code, so the reachability of a given statement sequence is not strictly required.

In the experiments, LOLLY successfully triggers eight (8) distinct bugs shown in Table IV. The first column shows the names of subjects, and their versions are shown in the second column. The third and fourth columns indicate the type of the vulnerabilities and the buggy function where LOLLY exposes respectively. In the fifth column, if the bug has been publicly reported, we give its report ID, such as CVE ID or Bugzilla ID. Otherwise, we label it as "Previously Undiscovered".

```
1595 if(OpCode(actions, n-1, maxn) == SWFACTION_PUSH
       &&
1596   OpCode(actions, n+1, maxn) ==
         SWFACTION_STOREREGISTER &&
1597   regs[actions[n+1].SWF_ACTIONSTOREREGISTER.
         Register]->Type == PUSH_VARIABLE)
1598 {
1599   var = newVar2(dblop,getString(var));
1600   if((OpCode(actions, n+2, maxn) == SWFACTION_POP
1601     && actions[n-1].SWF_ACTIONPUSH.NumParam==1)
1602     || OpCode(actions, n+3, maxn) ==
          SWFACTION_POP)
1603   {
1604     var->Type=11;  // later print inc/dec
1605   }
1606   else
1607   {
1608     ...
```

Fig. 3. Code snippet of decompile.c in Libming project

As shown in Table V, four (4) of them have already been reported in CVE and Bugzilla, and LOLLY also exposes them successfully. The other four (4) are previously undiscovered bugs found by LOLLY. We have reported them to the developers and are waiting their confirmation. Below we take a bug found in function "decompileINCR_DECR" in Libming 0.4.8 as an example. When testing the "swftocxx" utility in Libming 0.4.8, LOLLY triggered a crash in decompile.c. The stack dump shows the specific crash location is line 1597 in the decompile.c file whose related code snippet is shown in Fig.3 .

We debugged the Libming program using the crash input and found that there is a null pointer dereference bug on line 1597, where the program uses the pointer *regs[actions[n+1].SWF_ACTIONSTOREREGISTER.Register]* without checking beforehand whether it is a null pointer or not. Hence, line 1597 may cause a null pointer dereference, and result in a program crash.

VI. THREATS TO VALIDITY

In terms of external validity, our results may not hold for other programs that we did not test, though our experiments are all conducted on real-world open-source projects, which are widely used by the literature. As a part of future work, we will enhance our evaluation on a larger range of real-world software.

To mitigate internal validity, our initial seeds for fuzzing come from the regression test suites in projects under test, or the seed corpus of common important file-formats provided by AFL. And when comparing two fuzzers, they are started with the same seed corpus. Another threat to internal validity is the correctness of our implementation of LOLLY. To mitigate

TABLE V
BUGS EXPOSED BY LOLLY

Subject	Version	Vuln. type	Buggy function	Reported before
Binutils	2.26	Integer Overflow	d_unqualified_name	CVE-2016-4490
Binutils	2.26	Out of bounds reference	scan_unit_for_symbols	CVE-2017-15022
Binutils	2.30	Null pointer dereference	print_symbol	Previously Undiscovered
Binutils	2.30	Assertion failure	find_section	Sourceware Bugzilla 22793
Binutils	2.32	Memory leak	bfd_malloc	Previously Undiscovered
Libxml2	2.9.4	Null pointer dereference	xmlDumpElementContent	GNOME Bugzilla 773707
Libming	0.4.8	Heap buffer overflow	getName	Previously Undiscovered
Libming	0.4.8	Null pointer dereference	decompileINCR_DECR	Previously Undiscovered

the threat, we built LOLLY on AFL, a state-of-art greybox fuzzer. And the experimental results show that LOLLY gains both effectiveness and efficiency.

VII. RELATED WORK

A. Directed fuzzing based on greybox fuzzing

Böhme et al. [15] observe that most test inputs exercise the same few "high-frequency" paths. They model greybox fuzzing as a Markov chain and aim to cover low-frequency paths. Their fuzzer AFLFast can expose an order of magnitude more unique crashes than AFL in the same time budget and expose bugs that AFL cannot find. Furthermore, AFLGo [17] introduces simulated annealing into seed energy schedule of fuzzing and measures a seed by the distance from it to a target location. To improve the runtime efficiency, it moves most of program analysis and distance calculation from runtime to instrumentation phase. SCDF follows the idea of modeling greybox fuzzing as a Markov chain and leverages simulated annealing algorithm in seed energy schedule. Different from AFLGo, SCDF introduces the ideas of *sequence coverage* and calculates on demand the sequence coverage ability of a seed, in order to make directed greybox fuzzing more effective and efficient.

A program expecting complex structured inputs often handles with its inputs in two broad stages: a syntax parser parses the raw input into an internal data structure and then a semantic parser checks the data structure and performs the core logic of the program. Some studies attempt to generate valid inputs based on the input domains, which helps greybox fuzzers perform deep states of programs and improve the coverage. Zest [25], whose two key ideas are validity fuzzing and parametric generator, can generate valid inputs that exercise the code deep in semantic analysis stages of programs with high coverage. Pham et al. [26] propose smart greybox fuzzing (SGF) to generate test files for applications which expect and process complex file formats. The mutation operators of SGF work on the virtual file structure which allows SGF to explore completely new input domains while maintaining file validity. With the help of its validity-based power schedule, SGF attempts to generate files that are more likely to expose vulnerabilities deep in the processing logic.

Some research efforts introduce taint analysis into greybox fuzzing. ANGORA [14] aims to generate high quality inputs to execute unexplored branches. It selects an uncovered branch each time during fuzzing and utilizes taint tracking and search algorithm based on gradient descent to solve path constraints. In addition, it employs context-sensitive branch count, shape and type inference and input length exploration to improve its performance. In the future work, we plan to employ Angora's gradient-descent based constraints solving method to enhance SCDF's branch breakthrough capability. BuzzFuzz [27] uses dynamic taint tracing to identify those regions in a seed that influence values used at attack points, e.g., library and system calls. Then it mutates the identified regions to produce new inputs which preserve the underlying syntactic structure of the original seed. So these inputs can pass the input check/validation and reach locations deep in a program. FAIRFUZZ [16] aims to explore rare parts of the program under test. It first prioritizes inputs that exercise rare parts of the program and then increases the probability that mutated inputs can exercise the same rare parts while still exploring different paths. SAFL [28] is an efficient fuzzing tool augmented with qualified seed generation based on symbolic execution and efficient coverage-directed mutation, since the performance of mutation-based fuzzing is greatly affected by the quality of initial seeds and the effectiveness of mutation strategy. Its approach helps the fuzzing process to exercise rare and deep paths with higher probability.

In the fuzzing process, in addition to the seed energy schedule algorithm, the mutation strategy is also important in determining the performance of a fuzzer. Another future work for SCDF is to leverage taint analysis technique for seed mutation in a targeted manner, which will help further improve the detection efficiency.

B. Directed fuzzing based on symbolic execution

KATCH [7] presents a technique based on symbolic execution to automatically test software patches. It employs several synergistic heuristics based on static and dynamic program analysis to guide the exploration process. Ma et al. [2] propose two directed symbolic execution strategies, which are shortest-distance symbolic execution (SDSE) and call-chain-backward symbolic execution (CCBSE). SDSE uses a distance

metric in an inter-procedural control flow graph (ICFG) to guide symbolic execution towards targets, that is, it prioritizes the program branches associated with the shortest path to the target in ICFG. CCBSE first performs forward symbolic execution in the function containing the target line and then iteratively searches the calling function backward along the call chain until it finds a feasible path from the program entry to the target line.

Do et al. [6] propose a goal-oriented test method to effectively and efficiently explore security vulnerability errors, where a test goal is a potential security violation. They use type inference analysis to diagnose potential security violations and dynamic symbolic execution for test input generation using chaining approach. The main idea of chaining approach is to find the statements that must be executed before the execution of a given test target, and use these statements to form a sequence of events which guide the exploration process of symbolic execution. Dinges et al. [9] present a method that focuses on the security-sensitive functions and performs symbolic execution on the vulnerability-related execution paths, which effectively mitigates the path explosion problem in symbolic execution technique. First, it analyzes the source code of a program through pattern matching technology to find all hotspots that may cause vulnerabilities and build security constraints (SC) for each hotspot. Then, it analyzes the program backwards, builds a data flow tree for each hotspot and obtains all paths that can reach the hotspot. Finally, symbolic execution is performed along the paths, and path constraints (PC) is collected. If PC$\wedge\neg$SC is satisfied, the hotspot is reported to be vulnerable.

WOODPECKER [11] directs symbolic execution toward the program paths relevant to system rules and soundly prunes redundant paths. Dowser [29] pinpoints the instructions that access arrays in loops and ranks them according to an estimation of how likely they are to contain interesting vulnerabilities (buffer overflow and underflow). It determines which input bytes influence the array index with taint analysis and perform symbolic execution on the program, making only this set of inputs symbolic. BugRedux [21] is a directed whitebox fuzzer built on the symbolic execution engine KLEE, and aims to help developers to reproduce field failures in house. It takes a sequence of target locations and attempts to generate test cases which exercise the locations in the sequence in order. BugRedux considers four types of increasingly rich execution data for reproduction, i.e. points of failure, stack traces, call sequences, and complete program traces, where call sequences show best performance.

The above work utilizes symbolic execution technique to explore specific target codes of a program directionally. They collect the path constraints to the target code, and generate test cases that satisfy the constraints with the help of constraint solvers. By contrast, as a greybox fuzzing technique, SCDF performs lightweight instrumentation on a program under test and does not require expensive program analysis and constraints solving at runtime.

VIII. CONCLUSION

In this paper, we present a novel lightweight directed fuzzing technique SCDF, which is guided by a set of user-specified statement sequences. We also propose a novel energy schedule algorithm which adjusts a seed's energy according to its ability of covering the given statement sequences. Instead of statically calculating all distance information in instrumentation phase, SCDF evaluates the sequence coverage of seeds on demand using a lightweight calculation scheme at runtime, which improves runtime efficiency without introducing excessive overhead in instrumentation phase.

We implemented this technique in a tool called LOLLY, and evaluated LOLLY to three application scenarios, i.e. true positive verification, crash reproduction, and bugs exposure, in comparison with two state-of-the-art tools, i.e. AFLGo and BugRedux on several real-world software. The experimental results demonstrate that LOLLY outperforms AFLGo and BugRedux both in terms of effectiveness and efficiency.

The performance of our technique is affected by the given statement sequences and set of initial seeds. In the future work, we plan to investigate how to optimally set target statement sequences. For example, what is the optimal length of a sequence to guide the directed fuzzing process, and how the independence of statements in a sequence may affect the performance.

ACKNOWLEDGMENT

This work was supported by the National Natural Science Foundation of China (NSFC) under Grant No. U1713212.

REFERENCES

[1] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the Art," *IEEE Trans. Reliability*, vol. 67, no. 3, pp. 1199–1218, 2018.

[2] K.-K. Ma, Y. P. Khoo, J. S. Foster, and M. Hicks, "Directed Symbolic Execution," in *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*, ser. Lecture Notes in Computer Science, E. Yahav, Ed., vol. 6887. Springer, 2011, pp. 95–111.

[3] T. Do, A. C. M. Fong, and R. Pears, "Dynamic Symbolic Execution Guided by Data Dependency Analysis for High Structural Coverage," in *Evaluation of Novel Approaches to Software Engineering - 7th International Conference, ENASE 2012, Warsaw, Poland, June 29-30, 2012, Revised Selected Papers*, ser. Communications in Computer and Information Science, L. A. Maciaszek and J. Filipe, Eds., vol. 410. Springer, 2012, pp. 3–15.

[4] X. Ge, K. Taneja, T. Xie, and N. Tillmann, "DyTa: Dynamic symbolic execution guided with static verification results," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, R. N. Taylor, H. C. Gall, and N. Medvidovic, Eds. ACM, 2011, pp. 992–994.

[5] P. McMinn and M. Holcombe, "Evolutionary Testing Using an Extended Chaining Approach," *Evolutionary Computation*, vol. 14, no. 1, pp. 41–64, 2006.

[6] T. Do, S.-C. Khoo, A. C. M. Fong, R. Pears, and T. T. Quan, "Goal-oriented dynamic test generation," *Information & Software Technology*, vol. 66, pp. 40–57, 2015.

[7] P. D. Marinescu and C. Cadar, "KATCH: High-coverage testing of software patches," in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, B. Meyer, L. Baresi, and M. Mezini, Eds. ACM, 2013, pp. 235–245.

[8] H. Li, T. Kim, M. Bat-Erdene, and H. Lee, "Software Vulnerability Detection Using Backward Trace Analysis and Symbolic Execution," in *2013 Eighth International Conference on Availability, Reliability and Security (ARES)*, Sep. 2013, pp. 446–454.

[9] P. Dinges and G. Agha, "Targeted Test Input Generation Using Symbolic-concrete Backward Execution," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 31–36.

[10] S. Arzt, S. Rasthofer, R. Hahn, and E. Bodden, "Using targeted symbolic execution for reducing false-positives in dataflow analysis," in *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2015, Portland, OR, USA, June 15 - 17, 2015*, A. Møller and M. Naik, Eds. ACM, 2015, pp. 1–6.

[11] H. Cui, G. Hu, J. Wu, and J. Yang, "Verifying systems rules using rule-directed symbolic execution," in *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, V. Sarkar and R. Bodik, Eds. ACM, 2013, pp. 329–342.

[12] M. Böhme and S. Paul, "A Probabilistic Analysis of the Efficiency of Automated Software Testing," *IEEE Trans. Software Eng.*, vol. 42, no. 4, pp. 345–360, 2016.

[13] M. Zalewski, "American fuzzy lop," http://lcamtuf.coredump.cx/afl/.

[14] P. Chen and H. Chen, "Angora: Efficient Fuzzing by Principled Search," in *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE, 2018, pp. 711–725.

[15] 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, Vienna, Austria, October 24-28, 2016*, 2016, pp. 1032–1043.

[16] C. Lemieux and K. 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, ASE 2018, Montpellier, France, September 3-7, 2018*, M. Huchard, C. Kästner, and G. Fraser, Eds. ACM, 2018, pp. 475–485.

[17] 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, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, B. M. Thuraisingham, D. Evans,

[28] M. Wang, J. Liang, Y. Chen, Y. Jiang, X. Jiao, H. Liu, X. Zhao, and J. Sun, "SAFL: Increasing and accelerating testing coverage with symbolic execution and guided fuzzing," in *Proceedings of the 40th* T. Malkin, and D. Xu, Eds. ACM, 2017, pp. 2329–2344.

[18] Clang analyzer, "The clang static analyzer is a source code analysis tool that finds bugs in c, c++, and objective-c programs," http://clang-analyzer.llvm.org/.

[19] Libming, "Libming is a library for generating macromedia flash files," http://www.libming.org/.

[20] A. Vargha and H. D. Delaney, "A Critique and Improvement of the "CL" Common Language Effect Size Statistics of McGraw and Wong," *Journal of Educational & Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[21] W. Jin and A. Orso, "BugRedux: Reproducing field failures for in-house debugging," in *2012 34th International Conference on Software Engineering (ICSE)*, Jun. 2012, pp. 474–484.

[22] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, R. Draves and R. van Renesse, Eds. USENIX Association, 2008, pp. 209–224.

[23] GNU, "The GNU Binutils are a collection of binary tools," http://www.gnu.org/software/binutils/.

[24] Libxml2, "Libxml2 is the xml c parser and toolkit developed for the gnome project." http://xmlsoft.org/.

[25] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. L. Traon, "Zest: Validity fuzzing and parametric generators for effective random testing," *CoRR*, vol. abs/1812.00078, 2018.

[26] V. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *CoRR*, vol. abs/1811.09447, 2018.

[27] V. Ganesh, T. Leek, and M. C. Rinard, "Taint-based directed whitebox fuzzing," in *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, 2009, pp. 474–484.
International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 61–64.

[29] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations," in *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, S. T. King, Ed. USENIX Association, 2013, pp. 49–64.