Sequence Directed Hybrid Fuzzing

Hongliang Liang, Lin Jiang, Lu Ai, Jinyi Wei
School of computer science
Beijing University of Posts and Telecommunications
Beijing, China
{hliang,jianglin}@bupt.edu.cn; im.ailu@outlook.com; lrwei@bupt.edu.cn

***Abstract*—Existing directed grey-box fuzzers are effective compared with coverage-based fuzzers. However, they fail to achieve a balance between effectiveness and efficiency, and it is difficult to cover complex paths due to random mutation. To mitigate the issue, we propose a novel approach, sequence directed hybrid fuzzing (SDHF), which leverages a sequence-directed strategy and concolic execution technique to enhance the effectiveness of fuzzing. Given a set of target statement sequences of a program, SDHF aims to generate inputs that can reach the statements in each sequence in order and trigger potential bugs in the program. We implement the proposed approach in a tool called Berry and evaluate its capability on crash reproduction, true positive verification, and vulnerability detection. Experimental results demonstrate that Berry outperforms four state-of-the-art fuzzers, including directed fuzzers BugRedux, AFLGo and Lolly, and undirected hybrid fuzzer QSYM. Moreover, Berry found 7 new vulnerabilities in real-world programs such as UPX and GNU Libextractor, and 3 new CVEs were assigned.**

***Index Terms*—sequence guidance, concolic execution, crash reproduction, true positive verification, vulnerability detection**

I. Introduction

Fuzzing is an effective technology to automatically discover vulnerabilities in real-world software systems, by generating various inputs to execute a program and monitoring its abnormal behaviors (e.g., stack or buffer overflow, invalid read/write, assertion failures, or memory leaks) [1]. Fuzzers are usually classified as white-box [2], [3], grey-box [4] and black-box [5] according to their awareness of the internal structure of the program.

In recent years, two kinds of fuzzing techniques are proposed and demonstrated nice effectiveness: *Hybrid fuzzing* [6]–[8] that combines grey-box and white-box fuzzing, and *Directed grey-box fuzzing* that focuses on specific code regions in a program.

Hybrid fuzzers, such as QSYM [8], are built on the observation that grey-box fuzzing and concolic execution (white-box fuzzing) are complementary. With the help of concolic execution, they can explore more branches and obtain better path coverage. The current hybrid fuzzers feed all seeds retained by the fuzzers to concolic execution for exploring all missed paths. Concolic execution is then overwhelmed by the massive number of missed paths, and might generate a helping input for a specific path after a long time. However, there exist several testing scenarios in which only particular program states are concerned and required to be sufficiently tested. In other words, not all seeds are equal, and not all missed paths are worthy of attention. Therefore, the fuzzer and the concolic executor should be directed for exploring concerned program states: The seeds fed to the concolic executor are likely to guide the concolic execution to reach the targets and the concolic execution should focus on the paths reachable to the targets.

Directed grey-box fuzzing and fuzzers, e.g., AFLGo [9], Hawkeye [10] and Lolly [11], pay more attention to the statements of interest of a program without wasting resources on unrelated program code, and thus are effective and/or efficient than undirected fuzzers in several application cases, e.g., vulnerability detection, crash reproduction. For instance, AFLGo aims to generate test inputs which can reach a set of independent target statements in a program. Taking into account the dependencies between target statements, Lolly can generate test inputs which target the statement sequences, and thus discover the bugs resulted from the sequential execution of multiple statements. However, due to their dependence on random mutation to generate test inputs, grey-box fuzzers fail to reach deep targets and find deep bugs along complex paths [12]–[15]. Moreover, their strategies to schedule seed energy are coarse-grained because they only consider the coverage of statements in the target sequences while ignoring the context of these target statements.

To solve the above problems, we propose a sequence directed hybrid fuzzing (SDHF) technique. On one hand, SDHF is sequence directed because it directs both the fuzzing process and the concolic execution process with the enhanced target statement sequences of a program. On the other hand, SDHF is hybrid as it combines fuzzing process and concolic execution process through seed synchronization. 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. First, we enhance the target sequence with statements which must be explored before the target ones (necessary nodes for short). Next, in the fuzzing process, we propose a new seed energy scheduling algorithm, which assigns energy to a seed according to the similarity between the seed's execution trace and the enhanced target sequence. Moreover, we introduce a priority mechanism to manage the test cases (i.e., seeds) which are assigned different similarities with the enhanced target sequences. Finally, in the concolic execution process, we match the seed's execution trace with the enhanced target sequence and call the last matched statement/node as the switch point. In other words,

978-1-7281-5143-4/20/$31.00 © 2020 IEEE

```
104 template <class T>
105 typename T::Shdr const *PackVmlinuxBase<T>::
        getElfSections()
106 {
      ...
112   for (p = shdri, j = ehdri.e_shnum; --j>=0; ++p)
        {
113     if (Shdr::SHT_STRTAB == p->sh_type
114      && (p->sh_size + p->sh_offset) <= (unsigned
        long)file_size
115      && p->sh_name < p->sh_size
116      && (10+ p->sh_name) <= p->sh_size
117      ) {
118         delete [] shstrtab;
119         shstrtab = new char[1+ p->sh_size];
       ...
```

Fig. 1. A motivating example CVE-2019-14295

the concrete execution switches to symbolic execution after the switch point, which can not only help to cover complex paths but also generate more interested test cases which may reach the target sequences.

We implement the SDHF approach in a tool called Berry. Experiments show that, Berry is 1.28X faster by average than the hybrid fuzzer QSYM on vulnerability reproduction with LAVA-M dataset [16]. With the call stacks of crashes as targets, Berry can detect several deep bugs in LAVA-M programs while two directed grey-box/white-box fuzzers, AFLGo and BugRedux, cannot. Furthermore, Berry is 1.12X∼7.24X faster than Lolly when applied in true positive verification in Libming software [17]. In terms of the vulnerability detection, Berry found 7 new vulnerabilities in the latest versions of GNU Libextractor [18] and UPX [19], and 3 CVEs were assigned.

The main contributions of this paper are as follows:

- A sequence directed hybrid fuzzing (SDHF) technique which combines directed grey-box fuzzing and concolic execution, and guides them with user-specified statement sequences.
- A novel energy scheduling algorithm based on sequence similarity and a seed priority mechanism, in order to improve the guidance of the fuzzing process.
- A customized concolic execution method in which the execution of the program under test switches from concrete execution to symbolic execution when meeting a switch point, which can not only help to cover complex paths but also generate more interested test cases to reach the target sequences.
- A tool called Berry which implements the above techniques, and an evaluation which shows Berry's better effectiveness and efficiency than four state-of-the-art tools, i.e., QSYM, BugRedux, AFLGo and Lolly.

II. Motivation

As an example, we use the CVE-2019-14295 detected by Berry to discuss the limitations of directed grey-box fuzzing (DGF) and introduce the sequence directed hybrid fuzzing (SDHF) approach.

CVE-2019-14295 is an integer overflow in the *getElfSections* function in *p_vmlinx.cpp* in UPX[1] v3.95 [19], as shown in Fig.1. As there is an integer overflow in line 114, remote attackers can cause a denial of service (crash) via a skewed offset (i.e., $p \rightarrow sh\_offset$) larger than the size of the PE section (i.e., $file\_size$) in a UPX packed executable, which triggers the allocation of excessive memory in line 119.

Taking line 119 as the target statement as it contains a memory operation, i.e., *new* function, we use two directed fuzzers, AFLGo [9] and Lolly [11], to analyze the target program. Unfortunately, after their running for 20 hours respectively, both AFLGo and Lolly didn't find the vulnerability.

AFLGo and Lolly could not detect this vulnerability for two reasons. First, their guiding strategies are not effective enough. Suppose that three seeds s_1, s_2 and s_3 do not cover the target statement, s_2 covers line 112, and s_3 covers lines 112∼113. In Lolly, the target coverage of the three seeds is considered to be zero, although lines 112∼113 are necessary statements before reaching line 119. Intuitively, compared with s_1 and s_2, s_3 has a greater chance of reaching the target statement. As for AFLGo, if it detects two seeds that can reach the target statement, the seed with shorter trace will be favored [10]. Given two seeds s_4 and s_5, if s_4 performs a loop once, covering lines 112∼114, s_5 performs a loop twice, covering lines 112∼113 and lines 112∼114, then AFLGo will prefer s_4. In fact, both seeds have an equal opportunity to generate seeds that reach the target statement. Second, AFLGo and Lolly have difficulty passing complex path conditions (such as magic bytes) because they rely on random mutations to generate test cases. In fact, even if the seed is close to the targets, the new seed produced through random mutation is not necessarily close to the targets.

To improve the guiding strategies of directed greybox fuzzers, we propose and implement in Berry 1) A novel energy scheduling algorithm which adjusts on demand a seed's energy according to its similarity corresponding to the target sequences. 2) A seed priority queue into which those seeds close to the targets are put. Berry first selects the seeds in the high priority queue, and then the ones in the low priority queue. Usually, fuzzers try to generate illegal test inputs to trigger the crashes in a program. In the sample program, it is very common to modify the data in the header of the ELF file. However, most of the constructed illegal seeds can reach line 114 but difficult to pass the condition. Different from Lolly, Berry not only considers the target statement (line 119), but also considers some of the statements necessary for a path to reach the targets (lines 112∼116 in this case). Thus, Berry believes that the seed covering lines 112∼114 but failing to pass the condition in line 114 is of high quality and will give the seed more energy in the following mutation compared with the seeds covering no necessary statement.

What's else, Berry measures a seed based on the max matched prefix between seed execution trace and the enhanced

[1] UPX is a portable, scalable and high-performance executable packager for a variety of different executable formats.

target sequence (lines 112∼115, 119 in this example). The max matched prefix between s_3, s_4 and the enhanced target sequence are the same: Lines 112∼114. Therefore, contrary to AFLGo, Berry does not bias to shorter traces. Besides, seeds of high-quality are placed in high-priority seed queues and be processed by the fuzzer first.

To mitigate the limitations caused by random mutation, Berry leverages concolic execution to generate seeds that can pass complex path conditions and thus have more chances to reach the targets. Therefore, the above seeds covering lines 112∼114 are more likely to be selected by the concolic executor, where the execution will follow the trace as the original seeds until line 114 and take its true branch. Then the concolic executor will collect the condition of the true branch in line 114 and seek a solution with the constraint solver. As the concolic executor considers the boundary values and checks the integer overflow when processing an *add* instruction, then it generates a test case triggering an integer overflow.

III. Design

An overview of SDHF is shown in Fig. 2. SDHF includes static analysis and dynamic analysis. In the static analysis phase, given the target sequence and the source code of the program under test (PUT), SDHF first maps the statements in the target sequence to basic blocks[2] and calculates the enhanced target sequence (ETS) based on dominate tree analysis. The ETS is the combination of the target sequence and the necessary nodes which are basic blocks required to reach the target sequence for all paths. Then, the instrumentor instruments the PUT to facilitate collecting information at runtime, such as branch coverage and the target execution trace (TET) which contains basic blocks in the ETS executed by a seed during a run, and produces an instrumented binary.

In the dynamic analysis phase, the Fuzzer and Concolic executor communicate with each other by test case synchronization. The fuzzer takes as inputs the instrumented binary and the ETS, while the concolic executor executes the PUT. Moreover, we construct three seed queues with different levels, i.e., L1, L2 and L3 with high, normal and low priority respectively. The fuzzer puts the generated seeds into three queues and schedules them in the order of L1, L2 and L3 according to the similarities between the seeds and ETS as well as the coverage information. The concolic executor takes as inputs the seeds from the L1 queue and sends the newly generated seeds back to L2 queue. Finally, the seeds which cause the PUT crash are preserved in the Crashes set.

A. Static Analysis

1) Enhanced Target Sequence: To support a fine-grained strategy to schedule seeds, Berry considers the coverage of nodes in the target sequences as well as their execution context, e.g., necessary nodes. Therefore, Berry enhances the target sequences with necessary nodes which are basic blocks required to reach the nodes in the target sequences for all paths. Algorithm 1 shows how to get the necessary nodes based on the Dominate Tree [21].

Algorithm 1: Construct dom tree and calculate NNs

```
   Data: Target Sequence: $TS$, Control Flow Graph:
         $CFG$
   Result: Necessary Nodes: $NNs$
 1 $NNs = \phi$;
 2 $Dom(n_0) = \{n_0\}$;
 3 foreach $n \in N - \{n_0\}$ do
 4     $Dom(n) = N$;
 5     while changes in any Dom(n) do
 6         foreach $n \in N - \{n_0\}$ do
 7             foreach $p \in pred(n)$ do
 8                 $Dom(n) = Dom(n) \cap Dom(p)$ ;
 9                 $Dom(n) = Dom(n) \cup \{n\}$ ;
10             end
11         end
12     end
13     foreach $BB \in TS$ do
14         $NNs = NNs \cup Dom(BB)$ ;
15     end
16 end
```

In a control flow graph, if every path reaching node n must pass through node d, we call the node d dominates the node n, denoted as $d\ dom\ n$. By definition, each node dominates itself. A tree is called a dominate tree if each node in the tree only dominates itself and its descendants.

Algorithm 1 shows how to acquire the necessary nodes corresponding to a target sequence by constructing the dominant tree of the PUT. The inputs are the target sequence(TS) and the control flow graph (CFG) of the PUT. $n0$ is the entry node and N is the set of all basic blocks. The dominator of the entry node is itself (line 2). The dominator set of non-entry node n is the intersection of the dominator set of all precursor nodes of n. First, the dominator of non-entry node n is assigned N (line 4). Then the nodes that do not belong to the dominator of node n are iteratively removed (lines 5-9). Finally, we extract the union set of the dominators of each node in the target sequence (lines 13-14) to obtain the necessary nodes.

The necessary nodes and the target sequence are combined to form the enhanced target sequence (ETS).

We give an example in Fig. 3. The left part is a CFG of the program, whose yellow nodes, i.e., "c,g", represent the target sequence. The dominant tree constructed by algorithm 1 is shown in the upper right in the figure, where $b\ dom\ c$, $a\ dom\ g$, and $a\ dom\ b$. So, the blue nodes, "a, b", are the necessary nodes. Finally, ETS, "abcg", is composed of the necessary nodes, "a,b" and target sequence, "c,g".

2) Static Instrumentation: In addition to instrument each basic blocks to get the branch coverage information like AFL, SDHF also instruments the basic blocks in the ETS to get the target execution trace (TET) of a seed during a run. The

[2] During compilation, the statements in target sequence are mapped to basic blocks (or nodes) based on the debugging information of LLVM IR [20]

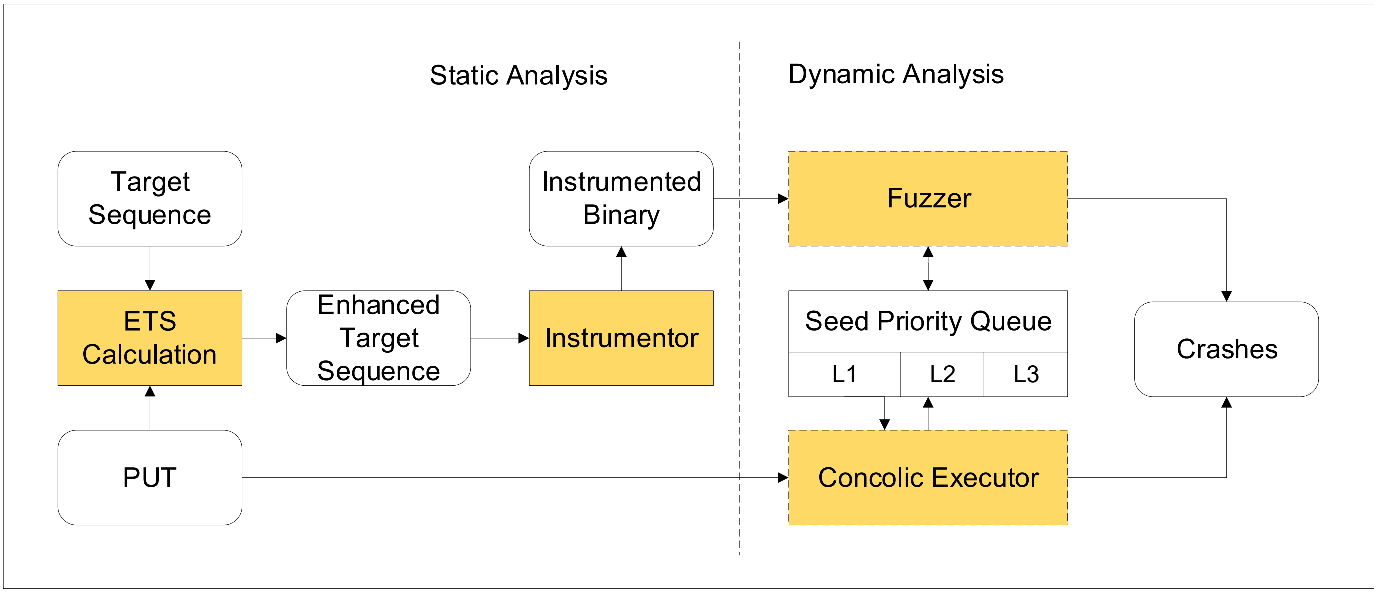

Fig. 2. An overview of SDHF

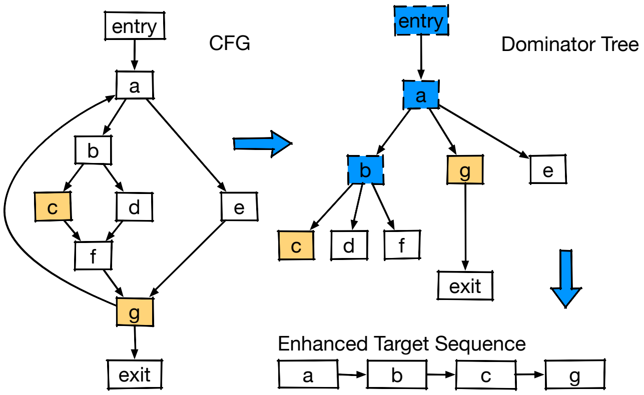

Fig. 3. Example: Construct ETS based on dominator tree

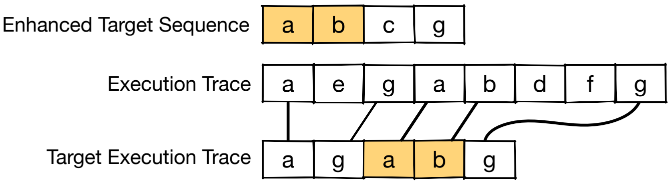

Fig. 4. Example: Map Execution Trace to TET

coverage information and the similarity between TET and ETS is used to decide the priority of a seed, as discussed in section III-B.

Specifically, SDHF uses a shared memory to record the IDs of the basic blocks in ETS following the order in which they are executed during a seed's run. So, we can obtain the seed's coverage information and TET.

Taking Fig. 4 as an example, the execution trace of a test input is “aegabdfg” and the ETS is “abcg”. Then the TET of the test input is “agabg”.

B. Dynamic Analysis

The dynamic analysis phase is mainly the interaction between the Fuzzer and the Concolic Executor, as shown in Fig. 2. They share three seed queues with different priorities and add new seeds to them. The fuzzer takes the seeds from the seed queues and sends back a newly generated seed to one of the queues according to the similarity between the TET and ETS as well as its branch coverage information. By contrast, the concolic executor executes the PUT with the guide of the seeds from L1 queue (with high priority) and puts newly generated seeds into L2 queue (with normal priority).

1) Fuzzing Process: The workflow of the fuzzing process is shown in Algorithm 2. S is a seed queue, which initially contains a few test inputs from the test suite of the PUT or built manually. The fuzzer selects a seed (line 2) from the seed queue S every time and assigns an energy to the seed (line 3). The seed's energy denotes its mutation time (line 4). Then, the fuzzer mutates the seed and executes P with the newly generated seed s' (line 5). If s' triggers a crash in the program, the seed is put in the crash seeds set Sx (lines 6-7). For each newly generated seed, its branch coverage and TET are recorded during the execution of the instrumented binary. So, for a seed s' that does not crash P, the fuzzer calculates the similarity between the seed's TET and the ETS (line 9) and gets the coverage information of the seed (line 10). Then, the fuzzer puts the seed into one of the three queues based on the similarity and coverage information.

In the following, we present the similarity calculation, energy assignment scheduling and seed priority mechanism in detail.

a) Similarity Calculation: We use an approach based on Max Matched Prefix (MMP) to calculate the similarity between execution traces of seeds and the enhanced target sequence (ETS). As the static analysis phase has instrumented

```
Algorithm 2: Directed Grey-box Fuzzing
Data: Instrumented PUT $P$, Seed queue $S$, Enhanced
      target sequence $ETS$
Result: Crash seeds $Sx$
do
    $s = chooseNext(S)$ ;
    $p = assignEnergy(s)$ ;
    for $i \leftarrow 0$ to $p$ do
        $s' = mutate(s)$;
        if $s'$ crashes $P$ then
            add $s'$ to $Sx$ ;
        else
            $sim = Similarity(s', ETS)$;
            $cov = IsNewCov(s')$;
            if $sim > d \wedge cov$ then
                add $s'$ to $L1$;
            else
                if $sim > d \vee cov$ then
                    add $s'$ to $L2$;
                else
                    add $s'$ to $L3$;
                end
            end
        end
    end
until timeout or abort;
```

the basic blocks in the ETS, we can get a TET corresponding to an ETS during execution and calculate MMP between the TET and the ETS.

We transform the matching problem of basic block sequence into the matching problem of strings, where the basic block ID is regarded as the character in the string. To calculate the MMP between sequences, we introduce the KMP algorithm [22], which is commonly used in string matching. The algorithm makes use of heuristic methods and shifts the characters as much as possible in the matching process, so as to improve the matching efficiency.

After obtaining the MMP, we calculate the similarity between seed S and ETS.

$$Similarity(S, ETS) = {}^{len(MMP)}/_{len(ETS)} \tag{1}$$

For seed S and an ETS, if $Similarity(S, ETS) > d$, where d is a threshold, the seed is likely to be sent to L1 queue and then be the input of concolic executor for generating seeds with higher similarity. In the implementation, we set d as the average similarity of the previous seeds. What's else, we record MMP and a switch point for a seed corresponding to a ETS, which is the last matched node between TET and ETS and is also the last node in MMP. The MMP and SP is used in concolic execution to control the execution mode. (Description of concolic executor is in SectionIII-B2).

In the example of Fig. 4, the MMP of ETS “abcg” and TET “agabg” can be obtained by KMP algorithm, which is “ab”, and is identified in yellow in the figure. According to the similarity calculation method above, we can get that the similarity between the seed s and ETS:

$$Similarity(s, ETS) = {}^{2}/_{4}$$

If the similarity of seed s is greater than threshold d, the seed with a MMP of “ab” corresponding to ETS “abcg” is likely to be put into seed queue with high priority.

Since multiple target sequences may be given, we obtained a weighted average of the similarity between the seed and the ETSs to obtain the global similarity (GS) between the seed and the ETSs.

$$GS(s, ETSs) = \sum_{ETS \in ETSs} {}^{Similarity(s,ETS)}/_{|ETS|} \tag{2}$$

b) Seed Energy Scheduling: Seed energy scheduling refers to the number of times that a seed is mutated by a fuzzer. In this paper, the seed with a higher similarity has a higher energy.

AFLGo and Lolly use simulated annealing based energy scheduling schemes in grey-box testing. It regards grey-box ambiguity as a Markov chain, and optimizes the process of fuzzing by simulated annealing algorithm (SA). Unlike the traditional random walk scheduling, which always accepts the best solution and leads to local optimum, simulated annealing accepts the solution which is not as good as the current one with a certain probability, so it is possible to jump out of the local optimum to achieve the global optimum. This probability decreases with the decrease of temperature. The algorithm can find the approximate optimal solution of the corresponding problem in polynomial time.

Berry also draws on the idea of the existing directed grey-box fuzzing techniques and applies simulated annealing to the seed energy scheduling scheme to achieve global optimization. For sequence matching-based directed testing, the optimal solution means that the test input can obtain the maximum similarity. In our method, the initial value of temperature T is $T_0 = 1$, which is cooled exponentially.

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

In the above formula, α is a constant, which satisfies $0.8 \leq \alpha \leq 0.99$, k is a temperature cycle. The threshold of temperature T_k is set to 0.05. If the temperature is lower than T_k, the fuzzer will not accept worse solutions. In particular, when $T_k > 0.05$, the power scheduling is in the exploratory stage, and the fuzzer generates many new inputs by random mutation of the existing seeds. Otherwise, it enters the exploitation stage, where the fuzzer selects seeds with a higher similarity to mutate. In the exploitation stage, the simulated annealing process is similar to the traditional gradient descent algorithm.

Since the common limitation of fuzzing is time budget, we use time t as a factor to adjust the temperature cycle k:

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

Among them, k_x is the temperature cycle and t_x is the time when the temperature drops to T_k. So we can get the relationship between time t and temperature T by 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 to AFLGo's simulated annealing based energy scheduling, given the global similarity GS, we define the capability for matching of the seed (CapMatch,CM) as:

$$CM = GS \times (1 - T) + 0.5 \times T \tag{7}$$

At the beginning of fuzzing, the initial value of temperature T is 1, that is to say, the matching ability of seeds is independent of the similarity. As time goes on, the temperature T decreases and the similarity becomes more and more important.

Combining the strategy based on similarity with the seed energy scheduling algorithm of existing fuzzer, AFL, berry takes the seed's CM as an influence factor to calculate the seed energy:

$$Energy = Energy \times 2.0^{(CM-0.2)\times 10} \tag{8}$$

Where $Energy$ is the energy calculated by the origin fuzzer, which both consider the information of seed coverage and running time. Energy is obtained by considering both the original energy and the seed's capability for matching.

c) Seed Priority Queue Mechanism: Not all the seeds have equal or similar priorities, ideally, the queue that stores the seeds to be mutated should be a priority queue. To prioritize the seeds which are close to the target sequence, we set up seed queues of three ranks according to the priority. The seeds in the level-1 priority queue will be picked firstly, then the level-2 priority queue, and finally the level-3 priority queue.

The attributes of the seeds in the three seed priority queues are as follows:

- Level-1 seed queue: The newly generated seed should satisfy all the following requirements: With the similarity greater than the specific threshold, brings new coverage.
- Level-2 seed queue: The newly generated seed not belonging to Level-1 queue should satisfy at least one of the following requirements: With a similarity greater than the specific threshold, generated by the concolic executor or brings new coverage.
- Level-3 seed queue: Other seeds.

2) Concolic Execution: Concolic execution is an aid to the fuzzing process. It takes partially matched seeds as input, hoping to generate seeds with a higher similarity. Because the fuzzer generates seeds based on random mutation without considering the context of the program, even if the seeds with a high similarity are mutated, it is not sure that the newly generated seeds are closer to the target sequences. Concolic execution engine obtains the information of the seed execution trace and generates test inputs executing specific paths by constraint solving.

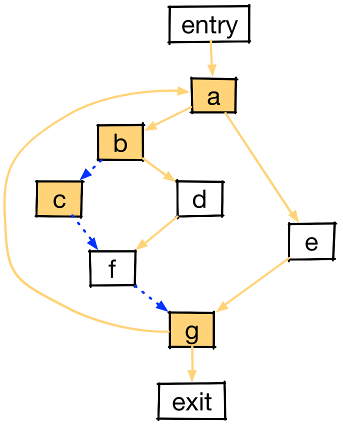

Fig. 5. An example showing the workflow of concolic execution. The orange edges represent the execution path of the original test input, *b* is the switch point and *abcg* is the enhanced target sequence. The blue edges represent the execution path of the newly generated seed after the switch point.

We use the concolic executor of QSYM [8] for concolic execution, which is lighter than traditional symbolic execution [3], [2].

Unlike QSYM's goal of improving program coverage, Berry leverages hybrid testing to improve similarities between execution traces of seeds and the ETS. Given some seeds for concolic execution, Berry tries to generate seeds with higher similarity. The new seed and the original seed follow the same path to the last matched point of MMP, which is called a switch point. After the switch point, the new seed takes a different branch.

To improve the efficiency of generating seeds with high similarities, we propose a concolic execution method guided by specific test inputs, as shown in Algorithm3. The inputs are original seed S from Level-1 seed queue, switch point SP, and max matched prefix MMP. With S as input, the program under test (line 1) is executed. If the current basic block BB is in the MMP (line 3) and is the same as $target$ (line 5), it will be judged whether BB is SP (line 4). In the case of SP, the concolic executor collects the constraints of the reverse branch of the branch to be executed (lines 6) and sends the constraints to the constraint solver for generating new seeds (line 7). If BB isn't SP, the concolic executor assigns the $target$ to the next basic block of the MMP (line 10). If the BB is in the MMP, but not the $target$, the target is reassigned to the first basic block of the MMP (line 13).

In Fig. 4, the execution path of original seed S is "aegabdfg", represented by orange edges. Corresponding to the ETS, "abcg", represented by yellow nodes, the TET is "agabg", and the MMP is "ab". Then, Berry passes the seed to the concolic executor with 'b' as switch point. The execution follows the original seed's path until reaching "b" and collects the constraint of branch $b \to c$. The constraint solver then generates a new seed S'. The execution path of S' is "aegabcfg". The TET is "agabcg" and the MMP is "abcg". As a result, $Similarity(S', ETS) = 4/4$, which is larger than the similarity of the original seed.

Algorithm 3: Concolic execution Guided by Test Input

```
Data: original seed $S$, switch point $SP$, max matched
      prefix $MMP$
Result: seed set $NewSeeds$
$Trace = execute(S)$;
foreach $BB \in Trace$ do
    if $BB \in MMP$ then
        if $BB == target$ then
            if $BB = SP$ then
                $Cons =$
                  $Trace.getNext(BB).getNeighbor()$;
                $NewSeeds = Solve(Cons)$;
                $returnNewSeeds$;
            else
                $target = MMP.next()$;
            end
        else
            $target = MMP.first()$;
        end
    end
end
```

With the seed from L1 seed queue as input, the execution is under the guidance of the seed until reaching the switch point. After reaching the switch point, the concolic executor begins to explore other branches and collect path constraints. The new seeds generated by the constraint solver will be sent back to the L2 seed queue.

IV. Implementation

A. Static Phase

Dominator Tree Analysis: We wrote an LLVM pass [23] to generate the dominating tree for every function of the program under test, and then obtain the necessary nodes of the target sequence by parsing the dominating tree.

Instrumentor: Based on the AFL's instrumentation pass [4], Berry's instrumentor adds additional instructions into the basic blocks in the enhanced target sequence. These instructions are in charge of writing the basic block ID to a shared memory in order to record the target execution trace.

B. Dynamic Phase

Fuzzer: On top of AFL, we implemented an energy scheduling algorithm, which relys on seeds' similarities with the target sequences, a simulated annealing algorithm, and a seed priority mechanism with three-levels queues. The fuzzer puts (selects) a seed into (from) a specific queue according to its priority.

Concolic Executor: We extended the concolic executor of QSYM [8] to provide the guidance ability by test inputs, in order to generate test inputs with high sequence similarities. It uses PIN [24] as the dynamic binary translator, and Z3 [25] to solve the path constraints.

V. Evaluation

In this section, we first conduct a comprehensive evaluation on the effectiveness and the performance of Berry by comparing it with the state-of-the-art directed fuzzers, AFLGo, BugRedux and Lolly and hybrid fuzzer, QSYM. Then we evaluated Berry's capability of detecting vulnerabilities in real-world programs.

A. Experiment Setup

We executed all experiments on a virtual machine with an Intel Core CPU i7-6500U, 8GB RAM and Ubuntu 16.04 (64 bit) as operating system. We evaluated Berry and other tools with the same programs, initial input corpus, time budget, computing resources, and target sites.

B. Crash Reproduction

Due to insufficient testing of software systems, there are often undetected bugs or vulnerabilities after release and users may encounter program crashes when using them. Many software systems have built-in crash reporting mechanisms that return crash reports to developers when a crash occurs. Crash reports usually contain information such as call stacks, but no input information that triggers crashes for protecting the privacy of users. This requires developers to reproduce crashes and fix bugs in software based on the reported call stacks.

First, we evaluate Berry's guiding capability by comparing it with the hybrid fuzzer QSYM in terms of efficiency. Next we evaluate Berry's effectiveness on crash reproduction with two state-of-the-art fuzzers, BugRedux and AFLGo.

1) Is SDHF Efficient Compared with Hybrid Fuzzing?: To evaluate Berry's efficiency, we measure and compare the average time of Berry and QSYM to reproduce crashes occurred in LAVA-M programs [16]. We choose LAVA-M benchmark which includes *uniq, base64, md5sum* and *who* programs as subjects because it contains many complex vulnerabilities injected manually and several fuzzers are also evaluated against it.

For each program under test, we first ran QSYM on the program and chose 2 crashes/vulnerabilities triggered by QSYM randomly[3]. Then their call stacks and test inputs which triggered the crashes were recorded. For each crash, we use the method calls in its stack trace as target sites of Berry.

To reproduce these crashes, we ran Berry and QSYM on the subject programs for 2 hours with the same test inputs, respectively. We ran 20 times for each subject and used the mean value of cost time.

The experimental results are shown in Table I. The first column is the program and the fault ID. 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 second and third columns show the mean of TTE values in all 20 experiments by QSYM and Berry respectively. Factor measures the performance gain as the mean TTE of

[3] The time budget for fuzzing is set to 2 hours.

TABLE I
COMPARISON OF QSYM AND BERRY ON LAVA-M

Subject	**TTE-QSYM (min)**	**TTE-Berry (min)**	**Factor**
uniq.fault1	49.34	27.45	1.80
uniq.fault2	38.03	30.40	1.25
base64.fault1	18.23	15.34	1.19
base64.fault2	22.24	17.09	1.30
md5sum.fault1	9.83	8.02	1.23
md5sum.fault2	15.39	10.23	1.50
who.fault1	73.39	78.20	0.96
who.fault2	20.01	19.42	1.03

TABLE II
THE RESULT OF CRASH REPRODUCTION

Subjects	BugRedux	AFLGo	Berry
sed.fault1	×	×	×
sed.fault2	×	√	√
grep	×	√	√
gzip.fault1	×	√	√
gzip.fault2	×	√	√
ncompress	√	√	√
polymorph	√	√	√
uniq.fault1	×	×	√
uniq.fault2	×	×	√
base64.fault1	×	√	√
base64.fault2	×	√	√
md5sum.fault1	×	×	√
md5sum.fault2	×	×	√
who.fault1	×	×	√
who.fault2	×	×	√

TABLE III
PERFORMANCE COMPARISON.

	AFLGo		Berry	
	Instu.	Run	Instru.	Run
Average	32.02s	367.75s	5.19s	299.89s
Median	29.17s	331.29s	3.69s	284.90s
Maximum	73.00s	796.19s	12.34s	621.59s

QSYM divided by the mean TTE of Berry. We can see that Berry detected all target vulnerabilities faster than QSYM, except who.fault1, and the average performance improvement is 1.28. In other words, Berry can generate test inputs to trigger a given vulnerability/crash faster than QSYM, a state-of-the-art hybrid fuzzer.

2) Is SDHF Effective Compared with Directed Fuzzing?: To evaluate SDHF's effectiveness, we compare Berry with two stat-of-the-art directed fuzzers, BugRedux and AFLGo on crash reproduction. BugRedux is an open-source white-box fuzzer, and AFLGo is an open-source grey-box fuzzer. Based on the KLEE symbolic execution engine, BugRedux generates test inputs that arrive at the target statement in sequence. AFLGo generates test inputs reaching the targets by reducing the distance between seeds and targets continuously.

In this experiment, the target vulnerabilities are shown in the first column of Table II. The first 7 vulnerabilities are from BugRedux and lay in simple paths in the programs. The others are from section V-B1 and are relatively complex. The time budget is set to 2 hours. We ran the three tools with the crash call stacks as the targets.

Table II shows the experimental results. For vulnerabilities from BugRedux, Berry and AFLGo reproduced 6 crashes, while BugRedux only reproduced 2. As sed.fault1 needs two files as input to trigger the crash, Berry and AFLGo cannot reproduce the crash for only mutating a single file at a time. However, for complex bugs, BugRedux and AFLGo did not perform well. For the 8 crashes from LAVA-M programs, BugRedux reproduced no crash, AFLGo only reproduced 2 crashes, and Berry reproduced all crashes successfully.

In a word, Berry can detect vulnerabilities under complex path conditions and is substantially more effective than BugRedux and AFLGo.

Moreover, for the BugRedux's benchmark, where Berry and AFLGo can reproduce the crashes, we further compare the time cost by both tools to trigger the crashes. Table III shows the Average, Median, and Maximum values of the instrumentation time and running time used by both tools. It can be seen that Berry's instrument time is much less than AFLGo because Berry has a more lightweight static analysis than AFLGo. The running time of Berry is a little less than AFLGo, which shows the advantage of Berry's concolic execution, although these vulnerabilities are shallow (i.e., in simple paths) and easily found by random mutation.

C. True Positives Verification

In the process of software development, program testing is of great importance. To detect bugs or vulnerabilities in programs as early as possible, developers are likely to test the software with static analysis tools. But it is known that static analysis is quickly but usually has high false positive. Directed fuzzers, e.g., AFLGo and Lolly, have been applied to the automatic verification of vulnerabilities reported by static analysis. Moreover, Lolly outperformed over AFLGo due to its sequence coverage approach.

To justify Berry's design decisions, i.e., fine-grained strategy to schedule seed energy and conclic execution towards ETS, we measured and compared the performance of Berry, Berry- which represents Berry without the concolic executor, and Lolly on true positive verification.

We used the same subject program, i.e., Libming[4] 0.4.8 [17] as Lolly and the CVE vulnerabilities are listed in Table IV. In addition, we ran Clang Analyzer [26] on the subject program and used its analysis results as target sequences, i.e., the paths which result in the potential bugs.

Specifically, for Berry- and Lolly, we ran a master and two slaves with the parallelization mechanism as AFL. For Berry, we ran a master and a slave of fuzzer as well as the concolic executor in parallel. We repeated each CVE experiment 20

[4] Libming is a library written in C language for generating and reading Macromedia Flash files.

TABLE IV
The CVE list used in verification

CVE-ID	Type of vulnerability
2019-9114	Out of bounds write
2019-9113	NULL pointer dereference
2019-12982	Heap buffer overflow
2019-12980	Integer overflow
2018-8963	Use after free
2018-7874	Invalid memory address dereference
2018-7876	Memory exhaustion

TABLE V
The results on true positive verification

CVE-ID	Tool	Runs	μTTE(m)	Factor
2019-9114	Berry	20	8.74	-
	Berry-	20	12.91	1.48
	Lolly	20	21.38	2.45
2019-9113	Berry	20	9.74	-
	Berry-	20	11.28	1.16
	Lolly	20	12.20	1.25
2019-12982	Berry	19	14.40	-
	Berry-	17	50.52	1.51
	Lolly	12	104.23	7.24
2019-12980	Berry	10	181.52	-
	Berry-	10	191.20	1.05
	Lolly	9	202.84	1.12
2018-8963	Berry	20	29.24	-
	Berry-	16	56.01	1.12
	Lolly	17	56.72	1.94
2018-7874	Berry	20	12.51	-
	Berry-	20	13.75	1.10
	Lolly	20	15.92	1.43
2018-7876	Berry	20	17.54	-
	Berry-	19	23.31	1.33
	Lolly	19	25.20	1.59

times and used the average value. We set 5 hours as the time budget for each experiment.

The experimental results are shown in Table V. The first column is the CVE-ID, the second column lists the fuzzer name, the third one shows the number of runs that successfully trigger the vulnerability, and the forth column is the mean of TTE values in all 20 experiments. In particular, if a tool fails to trigger the target vulnerability in a run within the time limit, its TTE is uniformly recorded as the time budget. The last column measures the performance gain as the mean TTE of Berry- divided by the mean TTE of Berry, and the mean TTE of Lolly divided by the mean TTE of Berry. Values of Factor greater than one mean that Berry performs better than Lolly or Berry-.

As shown in Table V, Berry is 1.12 to 7.24 times faster than Lolly in all cases. For easy-to-detect vulnerabilities, e.g., CVE 2019-12980, Berry's performance improvement over Lolly is not obvious. For complex vulnerabilities, Berry outperforms Lolly significantly. For example, Berry is 7.24X faster than Lolly on CVE-2019-12982. Moreover, the results show that Berry is 1.05 to 1.51 times faster than Berry- as it introduces concolic execution.

D. Vulnerabilities Exposure

To evaluate Berry's vulnerability exposure ability on real world programs, we chose two widely used software, i.e., GNU Libextractor and UPX, and tested their latest versions. GNU Libextractor [18] is a library for extracting metadata from files. UPX [19] is a high-performance packer for a lot of executable formats.

We leveraged Clang static analyzer to generate the statement sequences of potential vulnerabilities. With the above sequences, Berry tested each program for 10 hours and attempted to expose potential vulnerabilities. As a result, Berry found 7 previously unreported vulnerabilities, which are shown in Table VI. The table shows the subject software and version, the vulnerability type, the buggy method and confirmation to the vulnerabilities. All 7 vulnerabilities have been confirmed by the developers of two software, and 3 vulnerabilities with high threat scores were assigned with CVE-IDs, which we discuss them in detail as bellows.

CVE-2019-15531 is a heap-buffer-overflow vulnerability in function EXTRACTOR_dvi_extract_method of dvi_extractor.c in Libextractor 1.9, which can result in a crash. The developers recommend users to upgrade the Libextractor packages in Debian LTS security advisories.

CVE-2019-14296 is a buffer overflow bug which lies in function canUnpack of p_vmlinx.cpp in UPX 3.95. It allows remote attackers to cause a denial of service or possibly have unspecified other impacts via a crafted packed file.

CVE-2019-14295, as described in Section II, is an integer overflow in the getElfSections function of p_vmlinx.cpp in UPX 3.95. It allows remote attackers to cause a denial of service via a skewed offset larger than the size of the PE section in a UPX packed executable.

VI. Related Work

Hybrid fuzzing. In hybrid fuzzing, a concolic executor's role is to assist a fuzzer to get over narrow-ranged constraints and go deeper in the program's logic. Driller [6] adaptively switches between concolic execution and fuzzing depending on the increase rate in program coverage. DigFuzz [27] employs the Monte Carlo model to estimate probabilities and prioritize paths. TaintScope [28] deploys dynamic taint analysis to identify the checksum checkpoints and then applies symbolic execution to generate valid inputs. QSYM [8] integrates the symbolic emulation with the native execution using dynamic binary translation. It achieves better performance and is scalable to real-world programs.Such hybrid fuzzers is coverage based and can explore more branches and obtain better path coverage than grey-box fuzzers. However, in testing scenarios where only particular program states are concerned, exploring all missed paths with the concolic executor is a wast. Berry is a directed hybrid fuzzer, in which the fuzzer and the concolic executor are directed for exploring concerned program states: the seeds fed to the concolic executor are more likely to generate seeds that reach the target, and the concolic execution focuses on the paths that can reach the targets.

TABLE VI
UNREPORTED VULNERABILITIES FOUND BY BERRY

Program	Version	Vuln. type	Buggy method	Reported
GNU Libextractor	1.9	Heap-Buffer-Overflow	canUnpack	CVE-2019-15531
		Memory Leak	print_symbol	GNUnet Bug Tracker-0005847
UPX	3.95	Integer Overflow	getElfSections	CVE-2019-14295
		Buffer Overflow	canUnpack	CVE-2019-14296
		Heap-Buffer-Overflow	getElfSections	Github Issue #291
		Heap-Buffer-Overflow	acc_ua_get_be32	Github Issue #292
		NULL pointer dereference	get32	Github Issue #293

Directed symbolic execution. To implement a directed fuzzer, symbolic execution has always been the technique of choice due to its systematic path exploration. Directed symbolic execution casts the reachability problem as iterative constraint satisfaction problems. The patch testing tool Katch [29] uses the symbolic execution engine Klee [2] to reach a changed statement. BugRedux [30] takes as input a sequence of program statements and generates as output a test case that exercises that sequence and crashes the program. However, DSE is effective but spending considerable time with heavy-weight program analysis and constraint solving. In contrast, Berry uses symbolic execution as an aid to grey-box fuzzing, which alleviates the inefficiency of symbolic execution.

Directed Grey-box Fuzzing. As the state-of-the-art directed grey-box fuzzer, AFLGo [9] cast the reachability of target sites as an optimization problem and adopts a meta-heuristic to promote the test seeds with shorter distances. Hawkeye [10] extended AFLGo by adding attributes such as an indirect function call to alleviate the partiality of AFLGo's distance calculation. Lolly [11] is a directed grey-box fuzzer based on sequence coverage. It is lightweight than AFLGo but the seed scheduling mechanism is coarse-grained. Berry takes a lightweight and fine-grained seed energy scheduling mechanism, which trys to get a balance between effectiveness and efficiency. The directed grey-box fuzzing above is based on random mutation. As a result, they suffer from inefficient seed generation. Berry combines grey-box fuzzing and concolic execution with a customized schedule for directed test input generation. Enfuzz [31] is a framework that can integrate different fuzzers via its seed synchronization mechanism. Integrating Berry and the existing directed grey-box fuzzers with the seed synchronization mechanism of Enfuzz is one of our future work.

VII. THREATS TO VALIDITY

The first concern is external validity and notably generality. First, our results may not hold for subjects that we did not test. Though we conducted the experiments on real-world open-source projects, which included security-critical vulnerabilities. In the future, we will enhance our evaluation on a larger range of real-world software. Second, a comparison with a directed greybox fuzzer other than AFLGo and Lolly or a directed whitebox fuzzer other than bugredux might turn out differently. However,AFLGo and Lolly are state-of-the-art directed greybox fuzzers based in AFL. Bugredux is an advanced directed whitebox fuzzer.

The second concern is internal validity. A common threat to internal validity for fuzzer experiments is the selection of initial seeds. However, 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. Moreover, two fuzzers under comparing are always started with the same seed corpus. Second, Berry may not faithfully implement the technique presented above like implementations of other techniques. However, as shown in the comparison with QSYM, Berry is effectively directed.

VIII. CONCLUSION

We present a sequence directed hybrid fuzzing (SDHF) approach, which leverages a sequence-directed strategy and concolic execution technique to enhance the effectiveness of fuzzing. Given a set of target statement sequences of a program, SDHF aims to generate inputs that can reach the statements in each sequence in order and trigger potential bugs in the program. We implement the proposed approach in a tool called Berry. Experimental results show that Berry is more effective than three state-of-the-art directed fuzzers, i.e., BugRedux, AFLGo and Lolly, in both crash reproduction and true positive verification. Moreover, Berry found 7 previously unreported vulnerabilities in the latest versions of two real-world programs, and 3 CVE were assigned.

REFERENCES

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

[2] 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.

[3] J. Springer and W.-c. Feng, "Teaching with angr: A Symbolic Execution Curriculum and CTF," in *2018 USENIX Workshop on Advances in Security Education, ASE 2018, Baltimore, MD, USA, August 13, 2018.*, 2018.

[4] M. Zalewski, "american fuzzy lop." [Online]. Available: http://lcamtuf.coredump.cx/afl/

[5] Peach, "Peach Fuzzer: Discover unknown vulnerabilities." *Peach Fuzzer*, peach. [Online]. Available: http://www.peachfuzzer.com/

[6] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, “Driller: Augmenting Fuzzing Through Selective Symbolic Execution.” in *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.

[7] 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 International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE ’18. New York, NY, USA: ACM, 2018, pp. 61–64.

[8] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, “{QSYM} : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing,” 2018, pp. 745–761.

[9] 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*, 2017, pp. 2329–2344.

[10] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, “Hawkeye: Towards a Desired Directed Grey-box Fuzzer,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security - CCS ’18*. Toronto, Canada: ACM Press, 2018, pp. 2095–2108.

[11] H. Liang, Y. Zhang, Y. Yu, Z. Xie, and L. Jiang, “Sequence coverage directed greybox fuzzing,” in *Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019, Montreal, QC, Canada, May 25-31, 2019*, 2019, pp. 249–259.

[12] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, “VUzzer: Application-aware Evolutionary Fuzzing,” in *The Network and Distributed System Security Symposium (NDSS)*, 2017.

[13] 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.

[14] M. Bohme, V.-T. Pham, and A. Roychoudhury, “Coverage-Based Greybox Fuzzing as Markov Chain,” *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, May 2019.

[15] C. Lemieux and K. Sen, “FairFuzz: Targeting Rare Branches to Rapidly Increase Greybox Fuzz Testing Coverage,” *arXiv:1709.07101 [cs]*, Sep. 2017.

[16] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robert-

[28] T. Wang, T. Wei, G. Gu, and W. Zou, “TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection,”

son, F. Ulrich, and R. Whelan, “LAVA: Large-Scale Automated Vulnerability Addition,” in *2016 IEEE Symposium on Security and Privacy (SP)*, May 2016, pp. 110–121.

[17] “SWF output library. Contribute to libming/libming development by creating an account on GitHub,” Jul. 2019, original-date: 2010-11-27T22:22:20Z. [Online]. Available: https://github.com/libming/libming

[18] V. S. a. C. Grothoff, “GNU Libextractor.” [Online]. Available: https://www.gnu.org/software/libextractor/

[19] “UPX: the Ultimate Packer for eXecutables - Homepage.” [Online]. Available: https://upx.github.io/

[20] “LLVM Language Reference Manual — LLVM 10 documentation.” [Online]. Available: http://llvm.org/docs/LangRef.html

[21] R. E. Tarjan, “Finding Dominators in Directed Graphs,” *SIAM J. Comput.*, vol. 3, no. 1, pp. 62–89, 1974.

[22] D. E. Knuth, J. H. M. Jr, and V. R. Pratt, “Fast Pattern Matching in Strings,” *SIAM J. Comput.*, vol. 6, no. 2, pp. 323–350, 1977.

[23] “Writing an LLVM Pass — LLVM 10 documentation.” [Online]. Available: http://llvm.org/docs/WritingAnLLVMPass.html

[24] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood, “Pin: building customized program analysis tools with dynamic instrumentation,” in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, 2005, pp. 190–200.

[25] A. Dutcher, “z3-solver: an efficient SMT solver library.” [Online]. Available: https://github.com/Z3Prover/z3

[26] “Clang Static Analyzer.” [Online]. Available: http://clang-analyzer.llvm.org/

[27] L. Zhao, Y. Duan, H. Yin, and J. Xuan, “Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing,” in *NDSS*, 2019.
in *2010 IEEE Symposium on Security and Privacy*, 2010, pp. 497–512, v.

[29] P. D. Marinescu and C. Cadar, “KATCH: high-coverage testing of software patches,” in *ESEC/SIGSOFT FSE*, 2013.

[30] 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.

[31] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su, “EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers,” in *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019.*, 2019, pp. 1967–1983.