2016 10th International Symposium on Theoretical Aspects of Software Engineering

SeededFuzz: Selecting and Generating Seeds for Directed Fuzzing

Weiguang Wang, Hao Sun, Qingkai Zeng
State Key Laboratory for Novel Software Technology
Department of Computer Science and Technology, Nanjing University
Nanjing 210046, China
wang.wguang@gmail.com,shqking@gmail.com,zqk@nju.edu.cn

***Abstract*—As an improvement on traditional random fuzzing, directed fuzzing utilizes dynamic taint analysis to locate regions of seed inputs which can influence security-sensitive program points, and focuses on mutating these identified regions to generate error-revealing test cases. The seed inputs are of great importance to directed fuzzing, because they essentially determine the number of security-sensitive program points we can test. In this paper, we present a seed selection method complementing with a seed generation method for directed fuzzing. Using static analysis, dynamic monitoring and symbolic execution, our approach can provide directed fuzzing with seeds that can cover more security-sensitive program points in a cost-effective way. We implemented a prototype called SeededFuzz, and applied it to five real-world applications. Experimental results show that starting directed fuzzing with our carefully selected and generated seeds, SeededFuzz can test more critical program sites and detect more bugs.**

***Keywords*—directed fuzzing; seed selection; seed generation.**

I. Introduction

Fuzzing [1][2] is a well-known software testing technique. The idea behind it is simple: randomly modifying an initial input (*seed*) to generate new inputs and feeding them to the target application; if the application crashes, a potential vulnerability is detected. Although fuzzing can be remarkably effective, its limitation is also well-known. It has been ineffective at triggering errors deep inside applications [3][4], for most new inputs from random modifications fail to satisfy the syntactic constraints that characterize well-formed inputs, and hence fail to make them execute the remaining code.

Motivated by the need to expose deep errors, researchers have proposed *directed fuzzing* [3][5][6]. It utilizes dynamic taint analysis to identify the bytes of seeds which can influence values at security-sensitive program sites (e.g., memory allocation function `malloc`, string manipulation function `strcpy`). In contrast with traditional fuzzing that blindly modifies the entire seeds, directed fuzzing focuses on mutating only these identified bytes. In this paper, we refer to such security-sensitive operations in target program as *critical sites*, and the input bytes which can pollute these critical sites as *relative bytes*. Directed fuzzing has two advantages: 1) it can dramatically reduce the cardinality of the mutation space because only a small part of the seed is modified; 2) minor modification conducted by directed fuzzing usually does not break the syntactic structure in the seed and the new generated inputs are more likely to trigger deep vulnerabilities.

We argue that in directed fuzzing 1) one seed covers a number of critical sites and contains a set of relative bytes; 2) modifications on these relative bytes determine whether error-revealing test cases can be constructed. Hence, it is of great importance to start directed fuzzing with appropriate seeds. However, existing directed fuzzing techniques usually randomly select seeds from a corpus of well-formed test cases [3][5][6]. Given a set of candidate seeds S_{cand}, and a subset of seeds $S_{\mathrm{rand}} \subseteq S_{\mathrm{cand}}$ selected randomly for directed fuzzing to fuzz the target program, we use CS_{all} to denote all the critical sites in the target program, CS_{cand} to denote the critical sites that S_{cand} can cover, and CS_{rand} to denote the critical sites that S_{rand} can cover. From the vulnerability detection perspective, random seed selection strategy is prone to producing false negatives. On the one hand, it is usually not easy for randomly selected seeds in S_{rand} to make CS_{rand} equal CS_{cand}. That is, it is possible that some test cases in S_{cand} which can cover unique critical sites are missed in S_{rand}. Moreover, even if we make directed fuzzing with all candidate seeds (i.e. using S_{cand} as S_{rand}), we cannot guarantee that all critical sites in the target program will be tested either, because it is usually difficult to find a set of inputs which can cover all interested program points. That is, CS_{cand} is usually a subset of CS_{all}. In a word, randomly selected seeds are inadequate for covering critical sites, which will bring about negative effect to directed fuzzing on vulnerability detection. Thus, a strategy to help directed fuzzing obtain high-quality seeds is necessary.

In this paper, we present an approach to select and generate seeds for directed fuzzing on the basis of S_{cand}, and implement the approach in a new directed fuzzing tool called *SeededFuzz*. The goal we set ourselves is an effective seed supplying approach to improve directed fuzzing to test as many critical sites as possible.

To make the approach work, we need to address two main challenges. The first one is how to *select* appropriate seeds from the candidate seeds (i.e. S_{cand}). We utilize static analysis and dynamic monitoring to select the minimal subset of S_{cand}, which features the capacity of covering all the critical sites that S_{cand} can cover (i.e. CS_{cand}). We use S_{sel} to denote these carefully selected seeds and CS_{sel} to denote the critical sites that S_{sel} can cover[1]. The intuition behind our seed selection method is twofold: 1) Critical site is the key factor in directed fuzzing. If the seeds can cover more critical sites, they are likely to produce more bugs; 2) Some seeds cover the same

[1]Note that CS_{sel} is equal to CS_{cand}

978-1-5090-1764-5/16 $31.00 © 2016 IEEE
DOI 10.1109/TASE.2016.15

critical sites, and such seeds are likely to produce the same bugs during directed fuzzing.

After the seed selection, there may exist a number of critical sites in set CS_{all}-CS_{sel} that the selected seeds cannot reach. Our second challenge is how to *generate* seeds for directed fuzzing to test these remaining uncovered critical sites. Synthesizing inputs to cover a target is an essential problem in automated test generation and debugging [7][8][9][10]. While we borrow ideas from the state of the art in these areas, our approach combines static analysis, dynamic monitoring and symbolic execution to make the input generation process more effective. First, we collect the conditional branch instructions from program entry to the uncovered critical site and record their values. Then we use this trace information to steer symbolic execution to explore paths that are more likely to reach the uncovered critical site. When the target critical site is executed, an input is generated by solving the current path constraints. We use S_{gen} to denote the new generated seeds in this process and CS_{gen} to denote the critical sites that S_{gen} can cover.

Finally, the selected seeds together with the newly generated seeds (i.e. $S_{\text{sel}} \cup S_{\text{gen}}$) are provided to directed fuzzing as its starting point. Compared with random seed selection strategy, our approach is more selective and focused, covering the critical sites efficiently and aggressively.

In summary, our work makes the following contributions:

1) We propose a novel approach to select and generate seeds for directed fuzzing to test as many critical sites as possible.
2) We implement this approach in a new directed fuzzing tool called SeededFuzz, and conduct experiments with five real-world applications. The experimental results show that SeededFuzz can test more critical sites and detect more bugs.

The rest of this paper is organized as follows: In Section II, we start with a motivating example and an overview of our system. Then we detail our approach and the system implementation in Section III. Section IV presents the experimental evaluation. Section V discusses the related work. Finally, Section VI concludes our work.

II. Motivating Example and System Overview

As a motivating example, Figure 1 shows an example code that parses the input `file`. Function `decode_input` reads the input `file` and allocates memories according to the type field of the input (i.e., `T`). If `T` is equal to 0 (line 3), the code checks the variables `X` and `Y`, and allocates memory of size `X*Y*4`. Similarly, if `T` is equal to 1 (line 9), the code reads the width and height fields of the input (i.e., `W` and `H`), and allocates memory of size `W*H*2`. However, crafted inputs can cause integer overflows in the above expressions at line 7 and line 12, further leading to insufficient memory allocations. Heap overflows will eventually occur when the code reads data into these memories. Function `df` at line 8 and line 13 is used to illustrate the seed generation process of SeededFuzz, which should be ignored for now and will be discussed in Section III-C.

```
1  void decode_input(FILE* file) {
2      int T=get_type(file);
3      if (T == 0) {
4          int X = read_16bits(file);
5          int Y = read_16bits (file);
6          if(X>0&&Y>0)
7              int * p=malloc(X*Y*4);//
                   overflow
8          df(0);
9      } else if (T == 1) {
10         int W = get_width(file);
11         int H = get_height(file);
12         int * p=malloc(W*H*2);//overflow
13         df(1);
14     } else printf("no critical sites here"
          );
15 }
16 void df ( int argu) {
17     if ( argu ==0)
18         another critical site here ;
19 }
```

Fig. 1. Example code

In Figure 1, we can see that only several bytes in the input `file` (i.e. `X`, `Y`, `W` and `H` fields) are related to the integer overflows. It is relatively expensive (time-consuming) for traditional random fuzzing methods to detect such overflow errors, because the probability of exactly modifying these key bytes is low. As for directed fuzzing, its effectiveness highly depends on the quality of seeds. In this example, we assume that the seed is noted as `I` in which `T` is equal to 1. Starting with `I`, directed fuzzing can identify the `malloc` at line 12 as critical site, because it is security-sensitive and can be influenced by the width and height fields of the input. It then focuses on modifying these relative bytes (`W` and `H`), and the overflow at line 12 can be triggered. However, using `I` as seed, directed fuzzing cannot identify `X` or `Y` as relative bytes, let alone change their values during the testing process. Thus, directed fuzzing fails to detect the integer overflow at line 7 only with seed `I`.

To solve this problem, SeededFuzz leverages static analysis, dynamic monitoring and symbolic execution to select and generate appropriate seeds for directed fuzzing. Figure 2 depicts its overall architecture. At a high level, SeededFuzz is comprised of four phases: *critical site analysis*, *seed selection*, *seed generation*, and *directed fuzzing*. In the following, we will use the example in Figure 1 to illustrate how SeededFuzz works.

Critical site analysis. SeededFuzz performs static analysis on the target program to locate critical sites. In Figure 1, as `malloc` at line 7 and line 12 can be affected by user inputs and are closely related to overflows [11], we identify them as critical sites (i.e. CS_{all} is {line 7, line 12}). In addition, conditional branch instructions from the program entry to critical sites are also collected and their values are recorded in this phase. For example, in order to reach the critical site at line 7, the conditions at line 3 and line 6 should be both evaluated as *true*. In this paper, we refer to such values as "*valid jumps*".

Seed selection. SeededFuzz evaluates each candidate seed on its ability in covering critical sites, and selects a smallest set

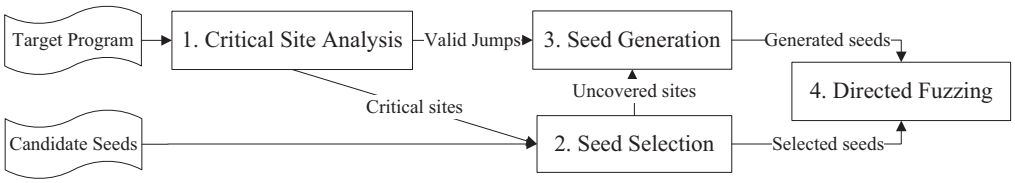

Fig. 2. Seededfuzz system overview

of seeds that can cover the maximum number of critical sites. In our example, suppose that the candidate seed set S_{cand} has three elements `I`, $\texttt{I}_1$, $\texttt{I}_2$, and their type fields (i.e., `T`) hold the values 1, 1, 2 respectively. In this phase, `I` is selected because it can cover the critical site at line 12 (i.e. S_{sel} is $\{\texttt{I}\}$ and CS_{sel} is {line 12}). $\texttt{I}_1$ is discarded because it is duplicate with `I` in covering critical site, while $\texttt{I}_2$ is discarded because it makes no contribution to either critical site coverage or vulnerability detection.

Seed generation. This phase is responsible for generating seeds for directed fuzzing to test the uncovered critical sites. Specifically, SeededFuzz starts this phase with an initial input. It then utilizes the valid jumps to guide symbolic execution, modifying the initial input to explore paths towards the uncovered critical sites. Continuing with our example in Figure 1, the critical site at line 7 cannot be covered by the selected seed `I`, and we aim to use `I` as initial input to generate new seed to reach this uncovered critical site. According to the collected valid jumps of this critical site, SeededFuzz flips the conditions at line 3 and 6 to construct a set of path constraints (satisfying `T=0`, `X>0` and `Y>0`) which characterizes a new program path, and then a new input `I'` which can cover the target (critical site at line 7) can be generated with the help of constraint solver.

Directed fuzzing. Finally, the selected and generated seeds are fed to directed fuzzing to test the target program. In this example, `I` and `I'` are used as seeds to conduct directed fuzzing. `X`, `Y`, `W` and `H` fields are then identified as relative bytes and mutated to generate error-revealing test cases. The overflows will be detected when the code reads data outside the boundaries.

III. System Design and Implementation

In this section, we detail the design and implementation of SeededFuzz. First, we introduce our critical site analysis technique in Section III-A. Next, we present the approach to select and generate seeds for directed fuzzing in Section III-B and Section III-C, respectively. Then, we discuss the directed fuzzing in Section III-D. Finally, we introduce the implementation of the SeededFuzz system in Section III-E.

A. Critical Site Analysis

Critical sites are program locations that may exhibit errors if the program is presented with an appropriate error-revealing input. In SeededFuzz we choose memory allocation functions (e.g., `malloc`, `realloc`), division operations and string functions (e.g., `strcpy`, `strcmp`) as critical sites. Valid jumps refer to the values of conditional branch instructions that can lead to these critical sites. The critical site analysis phase is responsible for locating critical sites and recording valid jumps for them.

Input:
$CFGs$: All the control flow graphs of the target program;
CS_{target}: The set of target critical sites;
Output:
$valJumps$: The set of valid jumps;
1: **for each** $cs \in CS_{\text{target}}$ **do**
2: Search $CFGs$ to find the CFG CFG_{vul} of function fn which contains cs;
3: Search CFG_{vul} to find basic block bb which contains cs;
4: $BB_{\text{pre}} =$ getParentBB(bb, CFG_{vul});
5: **for each** basic block $bb_{\text{p}} \in BB_{\text{pre}}$ **do**
6: Get the last branch instruction $inst$ in bb_{p};
7: $choice =$ getValidCon($inst$);
8: $valJumps.insert(< fn, bb_{\text{p}}, choice >)$;
9: $BB_{\text{pp}} =$ getParentBB($bb_{\text{p}}, CFG_{\text{vul}}$);
10: $BB_{\text{pre}} = BB_{\text{pre}} \cup BB_{\text{pp}}$;
11: **return** $valJumps$;

Fig. 3. Algorithm of valid jump collection

This part of our work is built on top of the LLVM [12] compiler infrastructure. Utilities provided by LLVM can generate call graphs and control flow graphs (CFG) from bytecode files directly [13]. Based on pre-defined information, critical sites can be easily located by traversing each statement in CFGs.

For each identified critical site, we then collect its valid jumps, which will be used at the following seed generation phase. Figure 3 illustrates the intra-procedural valid jump collection process. At the first step, we search all the control flow graphs to find the CFG of the function `fn` and the basic block `bb` which contain the target critical site `cs` (line 1-3). Next we traverse the preceding basic block $\texttt{bb}_{\text{p}}$ of `bb`, and save the value of the conditional branch instruction from $\texttt{bb}_{\text{p}}$ to `bb` as a valid jump (line 4-10). We use the tuple <`fn`, $\texttt{bb}_{\text{p}}$, `choice`> to store the valid jump of `cs`, meaning that the basic block $\texttt{bb}_{\text{p}}$ in function `fn` may lead to critical site `cs` when the condition `choice` is satisfied.

To extend the valid jump collection process above to inter-procedural level, we conduct a bottom-up traversal on the call graph of the program, in order to identify the callers of `fn` and the corresponding call sites. Finally, we add the basic blocks containing the call statements as new targets, and conduct the valid jump collection process repeatedly.

Input:
S_{cand}: The set of candidate seeds;
CS_{all}: The set of identified critical sites;
Output:
S_{sel}: The set of selected seeds;
CS_{uncov}: The set of uncovered critical sites;
1: $\{S_{\text{ranked}}, CS_{\text{cand}}\}$ = Check&Rank($S_{\text{cand}}, CS_{\text{all}}$);
2: **for each** critical site $cs \in CS_{\text{cand}}$ **do**
3: **for each** candidate seed $candidate \in S_{\text{ranked}}$ **do**
4: **if** cs is covered by $candidate$ **then**
5: Update($S_{\text{sel}}, candidate$);
6: break;
7: $CS_{\text{uncov}} = CS_{\text{all}} - CS_{\text{cand}}$;
8: **return** $\{S_{\text{sel}}, CS_{\text{uncov}}\}$;

Fig. 4. Algorithm of seed selection

B. Seed Selection

At the seed selection phase, we focus on identifying and selecting the smallest set of inputs (i.e. S_{sel}) which can cover the same critical sites as all the candidate seeds (i.e. S_{cand}) do. Our approach is motivated by the following two insights: 1) S_{sel} is equivalent to S_{cand} in terms of the vulnerability detection ability as they cover the same critical sites; 2) Directed fuzzing starting with S_{sel} as seeds would be more efficient than with S_{cand}, as mutation on each seed is usually time-consuming [14].

Figure 4 describes the algorithm of the seed selection. Given a set of candidate seeds S_{cand}, we monitor their execution to record the critical sites they can cover, and rank them according to the number of critical sites they can cover (line 1). Then, for each critical site `cs` in CS_{cand}, we repeatedly pick the high-ranking candidates, verify whether the input can cover the critical site `cs` (line 2-4). If so, we update the selected seed set S_{sel} with this input, and continue the next iteration (line 5-6). Finally, the seed selection phase outputs two sets: 1) the selected seed set S_{sel} which can cover the same critical sites as S_{cand} do; 2) the uncovered critical site set CS_{uncov} (line 7), which will be processed in the next seed generation phase.

C. Seed Generation

At this phase, we aim to generate new seeds for directed fuzzing to test the critical sites in set CS_{uncov} which are not covered by the selected seeds S_{sel}. As a baseline, we use *concolic execution* [15][16][17][18]: a combination of concrete execution and symbolic execution. Concolic execution runs the program on a concrete initial input, while gathering symbolic path constraints from conditional statements encountered along the execution. To test alternative paths, it systematically negates the collected path constraints, and checks whether the new expression is satisfiable. If so, it generates a new input with the help of constraint solver [19].

To reach the uncovered critical site, the seed generation phase iteratively modifies an initial input by selectively exploring paths towards the target critical site. Figure 5 presents the core steps of the seed generation phase. In this subsection, we will first explain the seed generation algorithm, and present an example to illustrate this procedure later.

Input:
S_{cand}: The set of candidate seeds;
CS_{uncov}: The set of uncovered critical sites;
$valJumps$: The set of valid jumps;
Output:
S_{gen}: The set of generated seeds;
1: **for each** uncovered critical site $ucs \in CS_{\text{uncov}}$ **do**
2: $input$ = PickInitialInput($S_{\text{cand}}, ucs, valJumps$);
3: **while** time limit does not expire **do**
4: PC = ComputePathConstraint($input$);
5: i = SearchConflict($PC, valJumps$);
6: **if** $(PC[0, ...(i-1)] \wedge \neg PC[i])$ has a solution $newInput$ **then**
7: $input = newInput$;
8: **if** ucs can be covered by $newInput$ **then**
9: Update($S_{\text{gen}}, newInput$);
10: break;
11: **else**
12: $input$ = BackTrace($PC, valJumps$);
13: **return** S_{gen};

Fig. 5. Algorithm of seed generation

For each uncovered critical site `ucs` in set CS_{uncov}, the algorithm first selects an *initial input* for concolic execution with the function `PickInitialInput` (line 1-2). The ideal initial input should execute branches close to `ucs` (the executed branch which has minimum distance in the CFG to ucs), so that we may quickly steer the concolic execution to reach the target by switching few path conditions. This function runs each input in set S_{cand} and records the values of branch conditions it takes. It then makes a comparison between these recorded condition values and the valid jumps of `ucs` to identify the first divergence point where the execution begins to deviate from the possible paths to `ucs`. Finally, the number of branches between the identified divergence point and `ucs` in CFG is counted, and the input with the minimum distance is selected as the initial input of the following concolic execution (line 2).

Next, we execute the program using the initial input, and gather path constraints `PC` along the execution (line 4). When we reach the branching point where current path conflicts with the valid jumps of `ucs`, we negate its branch condition to construct a new set of path constraints which characterizes a new path. If the new path constraints are satisfiable, a new input `newInput` is generated (line 5-7). Furthermore, if `ucs` can be covered by `newInput`, our algorithm successfully generates a new seed for directed fuzzing and stores it into set S_{gen} (line 8-10). If the new path constraints are unsatisfiable or there are no new test cases can be generated, we search for a new path to cover `ucs` with the function `BackTrace`. This function traverses the execution path backwards in order to find a branch point whose values in the valid jump set of `ucs` are *true* and *false*, which means that the executions following either side of this branch are possible to reach `ucs`. Then the function takes the unexplored side of this branch to approach `ucs`.

For example, considering function `df` in Figure 1 which contains a critical site (line 18) we intend to reach, `T` is used to determine the argument of this function (line 8 and line 13).

When using `I` (in which `T` equals 1) as initial input, it is clear that we cannot negate the condition at line 17 to reach this critical site, because `argu` is concrete and control dependent on `T`. In this case, `BackTrace` travels back along the trace, and finds that the branch at line 3 has two values in the valid jump set of the target critical site (line 18). It then generates input to take the unexecuted side of this branch (the *true* side), and finally a new input is generated to reach the target critical site.

After the seed generation phase, some of the uncovered critical sites in CS_{uncov} may still cannot be reached due to the time limitation (line 3) or some other reasons, such as lack of valid jumps. Specifically, in the seed generation phase, valid jumps are used to guide the exploration towards the uncovered critical sites. As described in Figure 3, we make the valid jump collection based on CFG and static analysis. Because of the inherent limitations of CFG generation and static analysis, we currently cannot collect valid jumps for the critical sites which are accessible only through function pointer calls. For the critical sites we fail to generate new inputs to cover in this phase, the following directed fuzzing fails to test them, either.

D. Directed Fuzzing

The directed fuzzing module starts with the selected and generated seeds (i.e. $S_{\text{sel}} \cup S_{\text{gen}}$) to generate error-revealing test cases. For each seed, it first uses fine-grained dynamic taint analysis to identify relative bytes. Each relative byte is then set to small, large, zero or negative values to generate new inputs. Finally, it runs the program under test on these new inputs to see if the inputs can reveal any errors.

E. System Implementation

We have implemented a prototype of SeededFuzz. It consists of four modules: the critical site analysis module, seed selection module, seed generation module and directed fuzzing module. As mentioned before, the critical site analysis module is built on top of the LLVM [12]. Using the `opt` utilities provided by LLVM, this module conducts static analysis on the call graph and CFGs to record critical sites and their corresponding valid jumps. The seed selection module utilizes the LLVM passes provided by KATCH [9] to track the execution of candidate seeds, obtains the critical site coverage information and determines which seeds to select. The seed generation module is implemented on the basis of the `zesti` version of KLEE [16], which starts symbolic execution from concrete inputs. The directed fuzzing module implements fine-grained dynamic taint analysis based on instruction instrumentation to identify relative bytes, generates new inputs by mutating the relative bytes and feeds them to target programs to trigger errors.

IV. Evaluation

A. Experiment Setup

We evaluate SeededFuzz on five open source applications: `mpeg3dump` (a utility from libmpeg3-1.8 which extracts data from MPEG files), `png2swf` (an SWF generation utility from swftools 0.9.0), `gif2swf` (another SWF generation utility from swftools 0.9.0), `cjpeg` (a JPEG manipulation application from libjpeg 7) and `speexenc 1.2` (a compressor for speex format). For each application, we first collected a corpus of test cases (including MPEG, PNG, GIF, BMP, JPEG and WAV formats) from its existing test suites. Using each application and its corresponding test cases as inputs, SeededFuzz then selected and generated a number of seeds for directed fuzzing. At last, the directed fuzzing module constructed new test cases by modifying these seeds to test the target programs, and logged the inputs which can cause crashes.

Note that we set a timeout of 30 minutes for each uncovered critical site in the seed generation phase and we fuzzed each seed for 2 hours in the directed fuzzing phase. Our test platform is a machine with Intel Core i5-750 2.67GHz, 4GB memory.

B. Critical Site Coverage Evaluation

Table I presents the results of the first three phases of SeededFuzz, including critical site analysis, seed selection and seed generation. This table contains a row for each analyzed application, and the `Total` row shows the sum of all the numbers in each column. The $|CS_{\text{all}}|$ column lists the total number of critical sites in each analyzed application. The $|S_{\text{cand}}|$ column presents the number of the candidate seeds and the $|CS_{\text{cand}}|$ column lists the number of critical sites that these candidate seeds can cover. In the experiments, we provided each application with 20 test cases as candidate seeds. The $|S_{\text{sel}}|$ column presents the number of selected seeds after the seed selection phase and the $|CS_{\text{sel}}|$ column lists the number of critical sites covered by these selected seeds. It can be seen that *our selected seeds can cover the same number of critical sites as all candidate seeds do for each applications, while the numbers of these selected seeds are not more than 3*. The reason is that candidate seed files are often duplicative in the critical site coverage. On the other hand, more than one seeds are usually needed. For example, in `png2swf`, the function `getPNG` reads files and allocates memories of different sizes according to the color type of the inputs. If the input is a grayscale PNG file with an alpha channel, the code calls memory management function and allocates memory of size `header.width*2`. If the input is a truecolor PNG file, the code calls another critical memory management function and allocates memory of size `header.width*4`. Thus, we need to select at least two files with different color types to cover these two critical sites.

Comparing the $|CS_{\text{all}}|$ column with the $|CS_{\text{sel}}|$ column, we can find that there still exist 43% critical sites (got from `Total` row) that cannot be covered by the selected seeds. We try to generate inputs to cover these critical sites in the seed generation phase. The $|S_{\text{gen}}|$ column lists the number of the generated seeds. The $|CS_{\text{sel}}|$+$|CS_{\text{gen}}|$ column presents the number of critical sites covered by the selected and generated seeds. In the `Total` row, it can be seen that *the generated seeds can successfully increase the overall critical site coverage from 57% to 65%.*

As we can see in Table I, there are 35% (99 out of 281) critical sites that our seed generation method fails to reach. After manual analysis, we found three main reasons for that. 1) Long control flow: During seed generation, when the new constructed path constraints are unsatisfiable, we use valid jumps to find a specified branch point and take its unexplored

TABLE I. CRITICAL SITE ANALYSIS, SEED SELECTION AND GENERATION RESULTS

Applications	$\lvert CS_{\text{all}}\rvert$	$\lvert S_{\text{cand}}\rvert$	$\lvert CS_{\text{cand}}\rvert$	$\lvert S_{\text{sel}}\rvert$	$\lvert CS_{\text{sel}}\rvert$	$\lvert S_{\text{gen}}\rvert$	$\lvert CS_{\text{sel}}\rvert$+$\lvert CS_{\text{gen}}\rvert$
mpeg3dump	138	20	59 (43%)	3	59 (43%)	10	69 (50%)
png2swf	60	20	48 (80%)	3	48 (80%)	3	51 (85%)
gif2swf	33	20	21 (64%)	3	21 (64%)	4	25 (76%)
cjpeg	23	20	13 (57%)	2	13 (57%)	2	15 (65%)
speexenc	27	20	20 (74%)	1	20 (74%)	2	22 (81%)
Total	281	100	161 (57%)	12	161 (57%)	21	182 (65%)

side with the aim of covering the target critical site through a new path. In the experiments, many critical sites cannot be reached because the distance between the target critical site and "right" branch point is too long for us to find the proper path in the given time budget. 2) Function pointer: In `png2swf` and `gif2swf`, several functions are accessible only through function pointer calls. Such indirect calls pose problems both during valid jump collection and during the seed generation. 3) Special input structure: In `mpeg3dump`, several critical sites in function `mpeg3_delete` can only be reached when the input has special structures (such as `subtitle` section). Inserting one or more sections in the input will significantly increase the complexity of symbolic execution in seed generation phase, because it possibly affects other parts of the input.

C. Vulnerability Detection Evaluation

Table II presents the number of bugs detected by directed fuzzing with different sets of seeds. After manual analysis, we found that all these bugs have been reported previously [11][20]. In this table, the first row lists the applications we analyzed. The second row shows the number of the detected bugs when we started directed fuzzing with all candidate seeds (i.e. S_{cand}), and the third row shows the result of directed fuzzing starting with seeds selected by the seed selection module (i.e. S_{sel}). It can be seen that the numbers of bugs detected by directed fuzzing with these two seed sets are the same for all the applications. The reason is that the seeds from the two sets can cover the same critical sites. The last row presents the number of the detected bugs when we started directed fuzzing with both selected seeds and generated seeds (i.e., $S_{\text{sel}} \cup S_{\text{gen}}$). It is worth noting that the generated seeds can help directed fuzzing detect one more bug in `mpeg3dump` because of the improvement on the critical site coverage. In short, in this table, we can see that:

1) *our seed selection method can largely reduce the size of seeds for directed fuzzing, while preserving the bug detection ability well*;
2) *our seed generation process can successfully improve the bug detection ability of directed fuzzing*.

D. Case Study

Advantage of seed selection. Figure 6 shows a heap overflow in `cjpeg` detected by SeededFuzz. It shows in boldface the flow of dynamic taint from `sptr→rows_in_array` (line 2) to the parameter of function `alloc_small` (line 10). In this function, `malloc` is called with the third parameter (i.e., `size`) to allocate memory of size `numrows*sizeof(JSAMPROW)`. Finally, the code reads data into the allocated buffer (line 12). A specially crafted image resulting in large `sptr→rows_in_array` value can

```
1  void realize_virt_arrays(j_common_ptr
       cinfo){
2      maximum_space += sptr->rows_in_array *
           sptr->samplesperrow * SIZEOF(
           JSAMPLE);
3      minheights = (sptr->rows_in_array -1)
           / sptr->maxaccess+1;
4      if (minheights <= max_minheights)
5          sptr->rows_in_mem =
               sptr->rows_in_array;
6          sptr->mem_buffer = alloc_sarray (
               cinfo, JPOOL_IMAGE, sptr->
               samplesperrow,
               sptr->rows_in_mem);
7  }
8  alloc_sarray (j_common_ptr cinfo, int
       pool_id, JDIMENSION samplesperrow,
       JDIMENSION numrows) {
9      size = numrows*SIZEOF(JSAMPROW);//
           integer overflow
10     result =alloc_small(cinfo, pool_id,
           size); // call malloc here
11     while (currow< numrows){  .....
12         result[currow++] = workspace; //
               heap overflow }
13 }
```

Fig. 6. A heap overflow in cjpeg

cause an integer overflow in the above expression (line 9) and further lead to an insufficient memory allocation (line 10). A heap overflow will eventually occur at line 12 when the code reads data into memory. To detect this vulnerability, directed fuzzing needs to starting with a seed which can cover the `malloc` called by function `alloc_small` at line 10. In the candidate seeds, there are only several PNG files (4 out of 20) can access this critical site. SeededFuzz can easily select the right inputs as seeds according to their execution traces, and further make directed fuzzing to detect this vulnerability. In comparison, there exists a certain probability for traditional directed fuzzing to select the appropriate seeds randomly.

Necessity of seed generation. The code snippet in Figure 7 shows the details of the bug in `mpeg3dump` which can only be detected by SeededFuzz with the help of seed generation process. Function `mpeg3_read_ifo` has many critical sites inside, while this function can only be invoked when the condition at line 4 is satisfied. Even if starting with all the candidate seeds, directed fuzzing cannot test this function mainly because: 1) there are no files in the candidate seeds that can satisfy the condition at line 4; 2) the variable `bits` at line 4 is not related to any critical sites and will never be modified during directed fuzzing.

TABLE II. BUGS DETECTED BY DIRECTED FUZZING WITH DIFFERENT SETS OF SEEDS

Source of Seeds	png2swf	cjpeg	speexenc	gif2swf	mpeg3dump
all candidates	1	3	1	1	1
selected	1	3	1	1	1
selected+generated	1	3	1	1	2

```
1  // in libmpeg3.c, mpeg3_get_file_type ()
2  uint32_t bits = mpeg3io_read_int32(file->
       fs);
3  ......
4  if(is_ifo(bits)) {
5      file->is_program_stream = 1;
6      mpeg3_read_ifo(file, 0);
7  }
8  // in mpeg3ifo.c   mpeg3_read_ifo ()
9  if(!strncmp((char*)ifo->data[ID_MAT], "
       DVDVIDEO-VTS", 12)) {
10     ......
11     ifo_table(ifo, OFF_TITLE_PGCI,
           ID_TITLE_PGCI);
12 }
13 // in the function cellplayinfo()
14 ......
15 ifo_hdr_t *hdr = (ifo_hdr_t*)ifo->data[
       ID_TITLE_PGCI];
```

Fig. 7. A null pointer dereference in mpeg3dump

Using SeededFuzz to test `mpeg3dump`, after the seed selection phase, the uncovered critical sites in function `mpeg3_read_ifo` are taken as targets in seed generation phase. SeededFuzz constructs sets of path constraints which can take the *true* side of the branch at line 4 and generates inputs to satisfy them. Using the generated inputs as seeds, directed fuzzing module can successfully identify the relative bytes, i.e. the bytes related to `ifo→data[ID_MAT]` at line 9. Modifications on these relative bytes can make the branch (line 9) take the *false* side. As a result, the statement at line 11 which is responsible for allocating memories for `ifo→data[ID_TITLE_PGCI]` is not executed. Eventually, the null pointer dereference can be triggered when the application tries to visit `ifo→data[ID_TITLE_PGCI]` at line 15.

V. RELATED WORK

Traditional fuzzing and directed fuzzing. Fuzzing refers to techniques for randomly generating or mutating seeds to get new test cases, which has been shown to be effective in uncovering errors [1][2]. Fuzzing is relatively cheap and easy to apply. However, it suffers from the problem that most new generated inputs are prematurely dropped [4].

To improve the effectiveness of fuzzing tools, researchers have proposed directed fuzzing techniques [3][5][6]. BuzzFuzz [3] uses dynamic taint analysis to locate regions of seeds that influence values used at library calls. Similarly, TaintScope [5] uses dynamic taint analysis to select fields of the seeds which influence memory management and string manipulation functions. They then focus on only fuzzing the identified input bytes to reduce the mutation space and improve the quality of the new generated inputs.

The quality of the traditional fuzzing and directed fuzzing heavily depends on the quality of seeds [21]. In traditional fuzzing, it is usually suggested to use executable code coverage as the seed selection strategy [14][22]. Directed fuzzing is different from traditional fuzzing in that it only makes modification on the relative bytes, and the directed fuzzing tools mentioned above all start their work flow with randomly selected seeds. In this paper, we present a seed selection method for directed fuzzing and a seed generation method to complement it. As shown in Section IV, our work can successfully increase the error detection ability of directed fuzzing.

Fuzzing interleaving with symbolic and concolic approaches. Similar to our work, DIODE [4] and Dowser [23] also combine symbolic execution, dynamic taint analysis and static analysis together to detect overflow vulnerabilities. Dowser employs dynamic taint analysis to identify the bytes that influence the "interesting" array accesses, and proposes a new path selection method to guide symbolic execution to explore paths that are more likely to lead to overflows. DIODE proposes a specific input format parser based on dynamic taint analysis and symbolic execution to reconstruct test input files with the aim of detecting integer overflows. Both Dowser and DIODE require vendors to provide specified inputs which can exercise the "interesting" program points to start their work as seeds, while SeededFuzz can select and generate seeds automatically for the test process.

Hybrid concolic testing [24] combines random testing and concolic testing together. The technique starts from random testing to quickly reach a deep state of the target program by executing a large number of random inputs. When the random testing stops improving coverage, it switches to concolic testing to exhaustively search the state space from the current program state. However, as the authors mentioned, hybrid concolic testing works best for reactive programs that receive inputs periodically while SeededFuzz has no specifical requests on the target programs.

Directed symbolic execution. Much research [9][10][25] has been done to guide path search towards a specific program point in symbolic execution, which is related to our seed generation phase. Xu et al. [10] introduce directed test suite augmentation to execute uncovered branches in a patch. Given a block of uncovered codes and an initial input that can reach the block's entry, they use symbolic execution to generate test cases that execute the uncovered branches. Similarly, Bloem et al. [25] combine symbolic execution and model checking to modify existing test cases in such a way that full branch coverage in specified target functions can be achieved. KATCH [9] is a tool to increase the coverage on new patches committed to software repositories. It exploits the existing test suite to find a good starting input and uses symbolic execution with several heuristics to generate inputs to test the patches. Different for these works, our approach focuses on improving directed

fuzzing's error detection ability rather than focuses on testing patches. In addition, compared with the related works which depend on the availability of initial inputs reaching the source node of the uncovered blocks or the entry of target function, our approach can automatically select initial inputs to start the seed generation process and has no requirements on the candidate inputs.

VI. Conclusion

In this paper, we present a seed selection and seed generation method to improve directed fuzzing. We integrate this method in SeededFuzz, a new directed fuzzing tool. Based on static analysis, dynamic monitoring and symbolic execution, SeededFuzz is able to start directed fuzzing with seeds which can cover more critical sites in the programs (points where the programs are prone to errors). We applied SeededFuzz to five open source applications. Experimental results show that SeededFuzz can successfully improve the bug detection ability of directed fuzzing.

Acknowledgment

This work has been partly supported by National NSF of China under Grant No. 61170070, 61572248, 61431008, 61321491; National Key Technology R&D Program of China under Grant No. 2012BAK26B01.

References

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

[2] S. Michael, G. Adam, and A. Pedram, *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

[3] V. Ganesh, T. Leek, and M. C. Rinard, "Taint-based directed whitebox fuzzing," in *31st International Conference on Software Engineering, ICSE*, 2009, pp. 474–484.

[4] S. Sidiroglou-Douskos, E. Lahtinen, N. Rittenhouse, P. Piselli, F. Long, D. Kim, and M. C. Rinard, "Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2015, pp. 473–486.

[5] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *31st IEEE Symposium on Security and Privacy, S&P*, 2010, pp. 497–512.

[6] M. Vuagnoux, "Autodafe: An act of software torture," in *22nd Chaos Communications Congress, Berlin, Germany*, 2005.

[7] P. Dinges and G. A. Agha, "Targeted test input generation using symbolic-concrete backward execution," in *ACM/IEEE International Conference on Automated Software Engineering, ASE*, 2014, pp. 31–36.

[8] K. Ma, Y. P. Khoo, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *Static Analysis - 18th International Symposium, SAS*, 2011, pp. 95–111.

[9] 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*, 2013, pp. 235–245.

[10] Z. Xu and G. Rothermel, "Directed test suite augmentation," in *16th Asia-Pacific Software Engineering Conference, APSEC*, 2009, pp. 406–413.

[11] F. Long, S. Sidiroglou-Douskos, D. Kim, and M. C. Rinard, "Sound input filter generation for integer overflow errors," in *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, 2014, pp. 439–452.

[12] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *2nd IEEE / ACM International Symposium on Code Generation and Optimization CGO*, 2004, pp. 75–88.

[13] Y. Wang, H. Sun, and Q. Zeng, "Statically-guided fork-based symbolic execution for vulnerability detection," in *The 27th International Conference on Software Engineering and Knowledge Engineering, SEKE*, 2015, pp. 536–539.

[14] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *Proceedings of the 23rd USENIX Security Symposium*, 2014, pp. 861–875.

[15] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation PLDI*, 2005, pp. 213–223.

[16] P. D. Marinescu and C. Cadar, "make test-zesti: A symbolic execution solution for improving regression testing," in *34th International Conference on Software Engineering, ICSE*, 2012, pp. 716–726.

[17] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering ESEC/FSE*, 2005, pp. 263–272.

[18] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Proceedings of the Network and Distributed System Security Symposium, NDSS*, 2008.

[19] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Computer Aided Verification, 19th International Conference, CAV*, 2007, pp. 519–531.

[20] I. K. Isaev and D. V. Sidorov, "The use of dynamic analysis for generation of input data that demonstrates critical bugs and vulnerabilities in programs," *Programming and Computer Software*, vol. 36, no. 4, pp. 225–236, 2010.

[21] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling black-box mutational fuzzing," in *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2013, pp. 511–522.

[22] M. Eddington, "Peach fuzzer." http://peachfuzzer.com/.

[23] 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*, 2013, pp. 49–64.

[24] R. Majumdar and K. Sen, "Hybrid concolic testing," in *29th International Conference on Software Engineering (ICSE*, 2007, pp. 416–426.

[25] R. Bloem, R. Könighofer, F. Röck, and M. Tautschnig, "Automating test-suite augmentation," in *2014 14th International Conference on Quality Software, QSIC*, 2014, pp. 67–72.