2020 IEEE 25th Pacific Rim International Symposium on Dependable Computing (PRDC) | 978-1-7281-8003-8/20/$31.00 ©2020 IEEE | DOI: 10.1109/PRDC50213.2020.00027

GTFuzz: Guard Token Directed Grey-Box Fuzzing

Rundong Li[1,3,†], HongLiang Liang[4,‡,§], Liming Liu[1,3,†], Xutong Ma[1,3,†], Rong Qu[1,†], Jun Yan[1,2,3,†,§] and Jian Zhang[1,3,†]
[1]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
[2]Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences
[3]University of Chinese Academy of Sciences
[4]Beijing University of Posts and Telecommunications
Email: †{lird, liulm, maxt, qurong, yanjun, zj}@ios.ac.cn
‡hliang@bupt.edu.cn

***Abstract*—Directed grey-box fuzzing is an effective technique to find bugs in programs with the guidance of user-specified target locations. However, it can hardly reach a target location guarded by certain syntax tokens (*Guard Tokens* for short), which is often seen in programs with string operations or grammar/lexical parsing. Only the test inputs containing *Guard Tokens* are likely to reach the target locations, which challenges the effectiveness of mutation-based fuzzers. In this paper, a *Guard Token* directed grey-box fuzzer called GTFuzz is presented, which extracts *Guard Tokens* according to the target locations first and then exploits them to direct the fuzzing. Specifically, to ensure the new test cases generated from mutations contain *Guard Tokens*, new strategies of seed prioritization, dictionary generation, and seed mutation are also proposed, so as to make them likely to reach the target locations. Experiments on real-world software show that GTFuzz can reach the target locations, reproduce crashes, and expose bugs more efficiently than the state-of-the-art grey-box fuzzers (*i.e.,* AFL, AFLGO and FairFuzz). Moreover, GTFuzz identified 23 previously undiscovered bugs in LibXML2 and MJS.**

***Keywords*-Static Analysis; Fuzz Testing; Crash Reproduction; Verifying True Positives; Exposing Bugs**

I. Introduction

Fuzzing (or fuzz testing) [1] is a significant security testing technique, which generates fuzzy inputs and feeds it to the program under test (PUT) to monitor whether some errors will occur. Since fuzzing has been invented, it has found a variety of weaknesses in a wide array of programs [2]–[5]. Many researchers are committed to making fuzzing smarter and have proposed many augmented fuzzing approaches [6]–[10]. Coverage based grey-box fuzzing (CGF) [10], [11], is a smart fuzzing technique. It guides the generation of new inputs according to the execution path coverage information. It aims to generate inputs that will cover some new paths and continually tests the untested code area's locations and program states. Directed grey-box fuzzing (DGF) [12] is an extension of CGF, which focuses on reaching user-specified code areas (we name such locations as target sites). Compared with CGF, DGF is better at detecting bugs hidden in or near the target sites.

Many DGF tools adopt the mutation-based input generation method (*e.g.,* AFLGO [12], Hawkeye [13], LOLLY [14]). Mutation-based input generation is a method that is good at generating some half-valid inputs. They generate new inputs by randomly modifying some well-formed initial inputs provided by the user. In many cases, the inputs that can trigger bugs are usually such half-valid inputs [15]. On the one hand, the valid part of a mutated input is conducive to pass the validity check in the PUT. On the other hand, the invalid (or abnormal) part of the input helps to trigger deep bugs. However, to make full use of the mutation-based method and expose bugs hidden in or near the target site, DGF needs to solve two critical problems. The first problem is how to exploit the initial inputs to pass the validity check of PUT before reaching the target site as early as possible (directed seed prioritization). And the second is how to ensure that the new inputs generated by the random mutation still contain the specific valid clips (directed seed mutation). The valid clips usually mean specific syntax tokens, magic headers, and checksums, which are difficult to be effectively generated by mutation-based methods.

Some research efforts approach to mitigate the above problems [16]–[22]. For example, Fairfuzz [16] calculates the syntax tokens in the input that will cause the program to reach the rare branch (*i.e.,* branches less covered by inputs) through pre-mutation of the input, and then protects these tokens in the input from being destroyed by the next mutation stage, thereby achieving the purpose of improving the path coverage of the rare branch. TaintScope [17] recognizes the checksum verification in the PUT through dynamic taint analysis and symbolic execution and then transforms the control flow of PUT to bypass these checks. Seededfuzz [18] pre-executes the inputs and filters the initial inputs to cover more critical sites for the subsequent fuzzing process. However, these approaches suffer from low efficiency and thus hardly trigger deep bugs because they adopt heavyweight techniques (*e.g.,* dynamic taint analysis, symbolic execution, or concolic execution).

In this paper, we propose a lightweight directed grey-box fuzzing approach. Specifically, we first identify and extract syntax tokens that guard the target site (we name these tokens as *Guard Tokens* or GTs) by using a backward static analysis technique. Based on the extracted GTs, we design a seeds prioritization method and an improved mutation algorithm. Moreover, we implement these techniques in GTFuzz, a *Guard Token* directed grey-box fuzzer. On the one hand, the GT based seeds prioritization and dictionary generation methods can

[§]Corresponding authors

effectively solve the first problem about making sure that the initial seeds containing specific valid clips are first selected for fuzzing. On the other hand, the GT based improved mutation method can solve the second problem about ensuring that the newly generated inputs can pass the validity check of PUT.

Our contributions are as follows:

- We propose a method to prioritize initial seeds and generate the refined dictionary to optimize the mutation engine.
- We design a directed seeds mutation method based on GTs which leverages different mutation functions on different seeds from two tiers.
- We implement a fuzzer named GTFuzz which combines the above techniques and evaluate its performance in target site covering, crash reproduction, and bugs exposure. The experimental results show that GTFuzz outperforms three state-of-the-art grey-box fuzzers (*i.e.,* AFLGO, FairFuzz, and AFL) on real-world benchmarks.

II. Background

In this section, we first introduce coverage-based grey-box fuzzing and directed grey-box fuzzing techniques and corresponding representative tools. Then we analyze the limitations of directed grey-box fuzzing. Finally, we explain why the limitations inspire us to propose *Guard Token* directed grey-box fuzzing with a motivating example in the LibXML2 project.

A. *Coverage-Based Grey-Box Fuzzing*

Grey-box Fuzzing obtains parts of the information of the program under test (PUT) by performing a lightweight static analysis and uses this information to guide its fuzzing strategies. Coverage-based Grey-box Fuzzing (CGF) prefers those inputs that can increase the path coverage. AFL is a mutation-based CGF fuzzer. It identifies every basic block in the PUT and tracks the execution path of the input. During the mutation, AFL estimates whether a new input should be retained and then evaluates and scores the performance of it. The score is affected by execution time, path coverage, waiting time, and other information, and it decides how many times the new input should be mutated. AFL has an excellent performance in the security field [10] and has found lots of bugs in dozens of real-world software.

B. *Directed Grey-Box Fuzzing and Limitations*

Directed grey-box fuzzing is proposed to guide the fuzzing engine towards the user-specified target sites [12]. For instance, AFLGO expands AFL to enhance the ability to explore the target sites. It aims to generate and schedule the seeds that likely reach the target sites, and outperforms AFL on patch testing, bug report confirmation and crash reproduction.

Throughout the fuzz testing, AFLGO goes through two phases: *exploration phase* and *exploitation phase*. In the exploration phase, AFLGO randomly mutates the initial seeds into more new inputs. In the exploitation phase, it focuses on generating seeds that have more chances to reach the targets. In the

```
 1 xmlXPtrEvalXPtrPart(name, args...){
 2   if (xmlStrEqual(name, "xpointer")) {
 3     ......
 4     xmlXPathCompStep(op, name, ...);
 5     ......
 6     xmlXPathCompOpEval(op, ...);
 7     ......
 8 }
 9 xmlXPathCompOpEval(op, args...){
10   ......
11   switch (op) {
12   case XPATH_OP_AND://Take false branch.
13   ......
14   case XPATH_OP_RANGETO: //Take true branch.
15     ......
16     obj = valuePop(ctxt);
17     oldset = obj->nodesetval; //obj is NULL.
18     ......
19 }
20 xmlXPathCompStep(op,name,args...){
21   ......
22   rangeto = "range-to"
23   if ((xmlStrEqual(name, rangeto))) {
24     op = XPATH_OP_RANGETO
25     ......
26   }
27 }
```

Fig. 1: Code snippets from LibXML2 V2.9.4

paper of AFLGO [12], the authors pointed out that *'AFLGO slowly transits from the exploration phase to the exploitation phase.'* Also, in the early stage of AFLGO (*i.e.,* deterministic fuzzing steps, which are used to produce compact new inputs by making small mutations to the original seeds), it conducts a non-directed fuzzing, which means that all the initial seeds are given fair chances of mutation.

Such strategies will result in low efficiency because the seeds that can reach the target site are mutated and scheduled in the same manner as other seeds. In other words, although other seeds fail to cover the target sites, AFLGO still spends lots of time on mutating and scheduling these seeds.

Real-world software usually has a built-in checker or parser to validate if its inputs contain some specific data, *e.g.,* syntax tokens or magic headers. Therefore, only the inputs containing these specific data can pass the checks and explore further deep paths. In other words, they usually match some structured fields in input with possible valid values and accept the input if the matching is successful, otherwise reject it. We declare the structured fields related to the target site as GTs. For the seeds that do not contain GTs, it is challenging to generate inputs that can pass the match statement of GTs through random mutation. As shown in the below example, AFLGO wastes much time (81.5% within 12 hours) in exploring paths that are irrelevant to the target site.

C. *Motivating Example*

Fig. 1 shows some code snippets extracted from LibXML2 of version 2.9.4, which contains a null pointer dereference error at line 17. We simplify the actual code for readability. The function `xmlXPtrEvalXPtrPart` is a caller of the function `xmlXPathCompOpEval`[1]. However, the state-of-the-art DGF tool AFLGO fails to reproduce this bug in 12 hours, even if we set the buggy line as the target site and leveraged all 10 test inputs provided by the project as the initial seeds for AFLGO. Through manually tracing the possible execution paths of this bug, we found that the inputs containing syntax tokens `range-to` and `xpointer` can cover the bug location. On the one hand, the matching of `range-to` is directly related to the value of variable `XPATH_OP_RANGETO` (lines 22-23), and the latter value determines whether the program will reach the target line or not. On the other hand, the function `xmlXPathCompOpEval` is called after the program matches `xpointer` successfully (line 2). With these observations, we searched those containing `xpointer` and `range-to` in all test cases and found 4 inputs that can match the above requirements. So we selected them as initial seed files for AFLGO, and it successfully reproduced the bug in 2.22 hours.

After analyzing the seeds generated by AFLGO in the above two experiments, we summarize some attributes of these seed files as follows: (1) some input files contain both `xpointer` and `range-to`, some contain only one, and some contain none of them; (2) all inputs that include both `xpointer` and `range-to` will lead the program to reach the target site while other inputs will not. This result shows that the existence of `xpointer` and `range-to` may be significant to an input to reach the target site with a high possibility; (3) the ratio of seed files containing `xpointer` and `range-to` to the total number of seeds in two experiments is 16.81% and 67.80% respectively. This result indicates that proper seed selection improves the ratio of better seeds in the seed queue. But that is not enough, in the second experiment, 32.20% of the seeds still cannot cover the target site, as they do not contain `xpointer` and `range-to` due to the random (or undirected) mutation.

In the above example, syntax tokens `xpointer` and `range-to` are necessary for test inputs to cover the target site, so we regard them as GTs. The presence or absence of them will influence whether a test input can cover the target site or not. This observation inspires us to develop the *Guard Token* directed grey-box fuzzer, GTFuzz. With a given target site, GTFuzz can extract GTs of the PUT and efficiently fuzz test it under the guide of the GTs. Compared with the state-of-the-art grey-box fuzzers, the advantages of GTFuzz reflect in two aspects. First, it can use GTs for input prioritization and dictionary generation, thereby improving the efficiency of fuzzing. Second, every new test cases generated by the fuzzer include at least one GT, which ensures these new test cases can pass the check of the GT in the program and get closer to the target site.

III. METHODOLOGY

In this section, we describe the key methods used in GTFuzz. Firstly, we design an algorithm to extract GTs based on context-sensitive data flow analysis technique. Secondly, we leverage the extracted GTs for seed prioritization and dictionary generation of DGF. Thirdly, we propose an augmented mutation algorithm based on GTs.

A. GT Extraction

For a target site t and a constant string str in a program, if the match operation of str is executed before t, then the constant string str is regarded as a GT of the target site t. In other words, the match statement of GT and the target site exist in a program path, and the GT is checked before the target site is executed. We extract GTs according to the following two patterns:

1) Strings in string comparison functions (*e.g.,* `strcmp`, `memcmp`) or other functions with similar functionalities (*e.g.,* comparison of consecutive characters). Such functions take at least two arguments, one is the constant string str, and the other is a variable v. If the value of v depends on the input of PUT, the string str is taken as a GT. For example, the function `xmlStrEqual` in Fig. 1 is located in the `if` statement (line 2). The value of the first argument `name` comes from the input of the program. As the target site (*i.e.,* line 17) is executed after the function `xmlStrEqual`, the second argument `xpointer` is regarded as a GT of the target site.
2) Strings in the variable definition. If a variable w is defined as the string str, and w is later directly referred to compare with another variable that has data dependence on the input of PUT, we take the string str as a GT. For instance, in Fig. 1 the variable `rangeto` is defined by a constant string `range-to` (line 22), and the program compares the variable `rangeto` with another variable `name` whose value depends on the input of the PUT (line 23). Therefore we take the string `range-to` as a GT.

Moreover, GTs in different situations have different effects on the target site. After successful matches, some tokens influence whether if the target site can be covered, while others do not. To explain these effects, we consider the example shown in Fig. 2. The left part of the figure is a code snippet that contains multiple token matches. The right part is the control flow graph of this program. The program consists of 6 basic blocks, and for each one, we label its name on the right side of the block. The matching of three tokens appears in the basic blocks b_1, b_2 and b_4, respectively. In the right part, each node in the control flow graph, except a beginning node and an end node, corresponds to the basic block with the same label in the left part. There are three tokens (*i.e.,* `hello`, `world`, `kitty`) and two function calls (*i.e.,* f_1, f_2) in this program. The matching of these three tokens has different effects on the execution of the two functions. Specifically, the function f_1 is called when the execution of the program matches token `hello` in block b_1 *or* token `kitty` in block b_5, while the function f_2 is called when the execution of the program successfully matches tokens `hello` *and* `world`.

[1]In fact, there are multiple intermediate callers on this call chain. We simplify it for clarity.

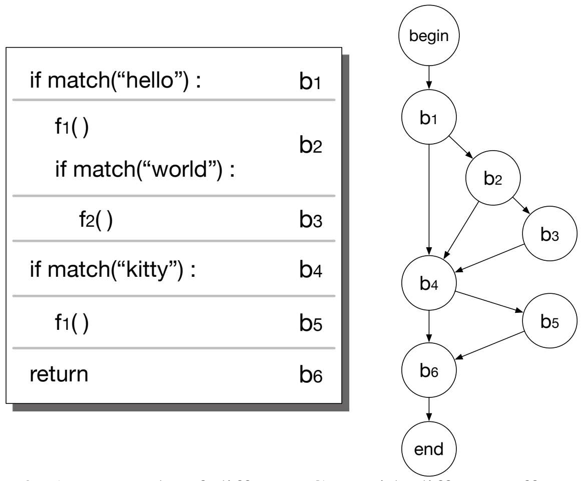

Fig. 2: An example of different GTs with different effects on the target site.

In the above example, some GTs that influence whether the target site would be reached. We define these GTs as Reachability-GTs. Suppose the basic block having the target site (target block for short) is block B_t, the basic block containing the token x is block B_x, the set M is the set of dominators of block B_t, and the set N is the set of strictly post dominators of block B_x. The token x is defined as a *Reachability-GT* when $M \cap N = \emptyset$. For instance, consider the type of token `kitty` in block b_4 for the target block b_5. The dominators set of block b_5 is the set $S_{d5} = \{b_1, b_4, b_5\}$, the strictly post dominators set of block b_4 is the set $S_{spd4} = \{b_6\}$. And $S_{d5} \cap S_{spd4} = \emptyset$, so token `kitty` is a Reachability-GT of block b_5. In contrast, the strictly post dominators set of block b_2 is the set $S_{spd2} = \{b_4, b_6\}$, $S_{d5} \cap S_{spd2} \neq \emptyset$, so token `world` in block b_2 is not a Reachability-GT of block b_5.

Two cases make the identification and usage of Reachability-GT *complex*. One is related to the diversity of string comparisons. Consider the following example: if we invert the condition of block b_4 (*i.e.,* the content of block b_4 is changed to `if !match("kitty")`), then the token `kitty` should not be a Reachability-GT of target block b_5. To resolve the above issue, we use precompiled optimizations to unify all conditional statements before the target block. Specifically, if a condition instruction describes a successful match to a string and the true branch does not contain the target block or the predecessors of the target block, but the corresponding false branch does, we negate the condition expression and swap these two branches.

The other case is that a target site may own multiple independent Reachability-GTs set. For example, if a target site t exists in the function f_1 in Fig. 2, the block b_2 or block b_5 would be regarded as the target blocks. For the case of b_2, the token `hello` is taken as a Reachability-GT. For the case of b_5, the token `kitty` is a Reachability-GT, but the token `hello` is not. Different Reachability-GTs may lead to different execution paths to cover the same target site. Therefore, all the inputs containing different Reachability-GTs for a target site are taken into account to expose the potential bugs in the target site.

We design the GT extraction algorithm to extract GTs of a given target site, as shown in Alg. 1. The inputs of this algorithm are the target site L_t of function F_t in the program P. The output is the set of extracted GTs $GTSet$.

In the beginning, the call graph CG is obtained from program P (line 2). Function GETBASICBLOCK returns the basic block B_t in which the target site L_t resides (line 3). We initialize the GT set $GTSet$ and the set of visited basic blocks $visitedBBSet$ as an empty set (line 4). We then put the binary tuple (F_t, B_t) into a queue $targetQueue$ (line 5). Then the algorithm loops until the queue $targetQueue$ is empty (line 6). In each loop iteration, the binary tuple (F_t, B_t) is first dequeued from the queue $targetQueue$ (line 7). After getting the control flow graph $tCFG$ of function F_t (line 8), we extract GTs that guard the execution of block B_t through the procedure EXTRACTGT and update the set $GTSet$ (line 9). Then we continue to obtain GTs of the predecessors of the target location through a backward analysis. First of all, since the block B_t has been analyzed, we add it to the set $visitedBBSet$ (line 10). Then for each caller F_p of function F_t in call graph CG of the PUT (line 12), we find in function F_p the basic blocks B_us that call the function F_t (line 13) and put them in the set BS_u. Finally, we push every block B_u that is not visited in BS_u and function F_p to $targetQueue$ (lines 14-17). The procedure MAINLOOP return set $GTSet$ when all blocks in $targetQueue$ are visited (line 20).

The procedure EXTRACTGT is responsible for extracting GTs which guard (directly or recursively) the block B_t in the function F_t by traversing $tCFG$. Initially, B_t is put into the set of analyzed basic blocks $BSet$ (line 23), and the set of GTs is set as empty (line 24). We maintain a variable $changed$ that indicates whether the set $BSet$ changes or not, and initialize it to a true value (line 25). The main body of the procedure is a loop (lines 25-34). In its each iteration, we analyze each block B which contains the call to a block B' in set $BSet$ (lines 25-26), and extract GTs as token $tmpGTs$ (line 27). In function EXTRACFROMB, we extract GTs from current basic block B according to the two patterns of GTs. After being analyzed, block B is added to set $BSet$ (line 28). Naturally until the set $BSet$ does not contain block B, it will be updated. Next, we distinguish the Reachability-GTs from the extracted GTs $tmpGTs$ since an input containing the Reachability-GTs has chance to reach the target site. If no basic block is either a dominator of block B_t or a strickly post dominator of block B, we mark token $tmpGTs$ as Reachability-GT (lines 29-31). Then we add the GTs $tmpGTs$ into set $curGTSet$ (line 32). At last the procedure returns the set $curGTSet$ (line 35) to its caller (*i.e.,* MAINLOOP).

B. Seed Prioritization and Dictionary Generation

In this paper, we use the Reachability-GTs to prioritize the initial seeds for DGF. Since the matching result of

Algorithm 1 GTs Extraction Algorithm.

```
 1: procedure MAINLOOP($L_t$, $F_t$, $P$)
 2:     get $CFGs$ and $CG$ from $P$
 3:     $B_t$ = GETBASICBLOCK($L_t$, $CFGs$)
 4:     $GTSet = visitedBBSet = \emptyset$
 5:     $targetQueue$.PUSH($\langle F_t, B_t \rangle$)
 6:     while $targetQueue \neq \emptyset$ do
 7:         $\langle F_t, B_t \rangle = targetQueue$.POP()
 8:         $tCFG$ = GETCFG($F_t, CFGs$)
 9:         $GTSet = GTSet \cup$ EXTRACTGT($tCFG$, $B_t$)
10:         $visitedBBSet = visitedBBSet \cup \{B_t\}$
11:         $tPreds$ = GETPREDECESSORS($F_t, CG$)
12:         for all $F_p \in tPreds$ do
13:             $BS_u$ = {blocks $B_u$s which calls $F_t$}
14:             $BS_u = BS_u / (BS_u \cap visitedBBSet)$
15:             for all $B_u \in BS_u$ do
16:                 $targetQueue$.PUSH($\langle F_p, B_u \rangle$)
17:             end for
18:         end for
19:     end while
20:     return $GTSet$
21: end procedure
22: procedure EXTRACTGT($tCFG$, $B_t$)
23:     $BSet$ = $\{B_t\}$
24:     $curGTSet = \emptyset$
25:     for all $B \in tCFG$ do
26:         if ISPRED($B$, $B'$) $\wedge B' \in BSet$ then
27:             $tmpGTs$ = EXTRACTFROMB($B$)
28:             $BSet = BSet \cup B$
29:             if DOMS($B_t$) $\cap$ SPDOMS($B$) $= \emptyset$ then
30:                 mark $tmpGTs$ as REACHABILITY-GT
31:             end if
32:             $curGTSet = curGTSet \cup tmpGTs$
33:         end if
34:     end for
35:     return $curGTSet$
36: end procedure
```

Reachability-GTs influences whether an input can cover the target site or not, we assign a high priority to those seeds containing the Reachability-GTs.

Alg. 2 outlines the seed prioritization algorithm. We split the seed queue into two tiers, *i.e.,* Q_1 and Q_2, and the fuzzer performs different mutation operations on the inputs in different queues. The input of this algorithm is the candidate input set $candidateInputs$, and the outputs are the seed queues Q_1 and Q_2. In the beginning, the seed queues Q_1 and Q_2 are initialized to an empty set (line 2). For each seed s in set $candidateInputs$ (line 3), if it contains a Reachability-GT, *i.e.,* rGT, we push it to the queue Q_1 (lines 5-6). Otherwise, we push it to the queue Q_2 (lines 7-8). Finally, the algorithm returns the seed queues Q_1 and Q_2

The dictionary in a fuzzer consists of language keywords, magic headers, or other tokens used in the PUT. It can be used to optimize the mutation engine and improve the

Algorithm 2 Seed Prioritization Algorithm.

```
 1: procedure SEEDPRIORITIZATION($candidateInputs$)
 2:     $Q_1 \leftarrow Q_2 \leftarrow \emptyset$
 3:     for all $s \in candidateInputs$ do
 4:         for all $rGT \in$ Reachability-GTs do
 5:             if $rGT \in s$ then
 6:                 $Q_1$.PUSH($s$)
 7:                 break
 8:             end if
 9:         end for
10:         if $s \notin Q_1$ then
11:             $Q_2$.PUSH($s$)
12:         end if
13:     end for
14:     return $Q_1$ and $Q_2$
15: end procedure
```

fuzzer's syntax-awareness ability [10]. The GTs extracted by our approach can be used to generate a dictionary. For instance, AFL provides an interface to accept the dictionary. The extracted GTs can be directly supplied to the fuzzer through this interface. Besides, compared to a complete dictionary which is composed of most of the syntax tokens appear in the PUT), the dictionary consisting of GTs is more conducive to the fuzzer's exploration towards the target sites.

C. GT-based Mutation

AFLGO follows the seed mutation strategies of AFL. We improve these strategies with *Guard Tokens* on two aspects: (1) Narrowing down the search space for mutation functions by using GTs; (2) Using only the dictionary related mutation functions, instead of all mutation functions, on the low-priority inputs (*i.e.,* inputs in the queue Q_2). For example, suppose that an input inp (*e.g.,* `a-1==a(0)`) is a low-priority input because it does not contain any Reachability-GTs, we can overwrite certain fields (*e.g.,* all the characters `a` in input inp) with a specific token (*e.g.,* token `isNaN`), through the mutation function HAVOC15, to generate a new input (*e.g.,* `isNaN-1==isNaN(0)`).

Based on the above considerations, we propose a GT-based Mutation Algorithm. We mainly modify the three mutation stages of AFL's algorithm, *i.e.,* deterministic stage, havoc stage, and splice stage. Alg. 3 outlines the details.

In procedure DETERMINISTICSTAGE, the function array $functions$ is initialized by multiple mutation functions (such as `bitflip`, `arithmetic`, `dictionary`, *etc.*) (line 2). We add special treatment for each seed s from the queue Q_2, *i.e.,* we only use the functions in $dictionaryFunctions$ to mutate these seeds (lines 3-4). These functions are charge of inserting a random Reachability-GT into s, or overwriting a certain segment of seed s by this Reachability-GT, to make the seed s likely cover the target site. In line 6, the procedure tries to mutate each bit or byte of the seed s in turn by the mutation function $func$ in array $functions$. During the mutation, the function MUTATEEXCEPTGTS skips the positions where GTs

Algorithm 3 GT-based Mutation Algorithm.

```
procedure DETERMINISTICSTAGE($s$, $GTs$)
    $functions \leftarrow$ deterministicFunctions
    if $s$ is from $Q_2$ then
        $functions \leftarrow$ dictionaryFunctions
    end if
    for all $func \in functions$ do
        $\bar{S}$.PUSH(MUTATEEXCEPTGTS($s$, $func$, $GTs$))
    end for
    return $\bar{S}$
end procedure
procedure HAVOCSTAGE($s$, $score$, $GTs$)
    for all $i \in [0, score]$ do
        if $s$ is from $Q_2$ then
            $func \leftarrow$ HAVOC15() or HAVOC16()
        else
            $func \leftarrow$ RANDOMHAVOCFUCTION()
        end if
        $\bar{S}$.PUSH(MUTATEEXCEPTGTS($s$, $func$, $GTs$))
        return $\bar{S}$
    end for
end procedure
procedure SPLICESTAGE($s$, $score$, $GTs$)
    if $s$ is from $Q_2$ then
        $s' \leftarrow$ SELECTSEED($Q_1$)
    else
        $s' \leftarrow$ SELECTSEED($Q_1 \cup Q_2$)
    end if
    $newInput \leftarrow$ CONCATEXCEPTGTS($s$, $s'$, $GTs$)
    HAVOCSTAGE($newInput$, $score$, $GTs$)
end procedure
```

appear in seed s to protect these critical fields from being modified, and returns the mutated seeds to update the queue $\bar{S}$ (line 7). Then the procedure returns the queue $\bar{S}$ (line 9).

In procedure HAVOCSTAGE, the score of the seed s determines how many times it will be mutated (line 12). During the mutation, only havoc functions HAVOC15 (which overwrites a seed's some bytes of with a token) and HAVOC16 (which inserts a token into a seed) participate in the mutation of seed s that comes from the queue Q_2 (lines 13-14). Similar to the functions in function queue $dictionaryFunctions$, the purpose of these two functions is to add Reachability-GTs to the seed s. If the seed s comes from the queue Q_1, it will be mutated by a random havoc function (line 16). Then the procedure mutates the seed s by masking the GTs in seed s (line 18) and returns the mutated seeds queue $\bar{S}$ (line 19).

In procedure SPLICESTAGE, we first select another seed file s' to join the mutation of seed s. To ensure that the new input $newInput$ contains GTs, if seed s comes from the queue Q_2, we select the seed s' from the queue Q_1 (lines 23-24). Otherwise, we select it from the merged queue $Q_1 \cup Q_2$ (lines 25-26). Next, the function CONCATEXCEPTGTS concatenates these two seeds while protecting the GTs in them from being modified, and returns a new input $newInput$ (line 28). Finally, the new input is transferred to the procedure HAVOCSTAGE (line 29).

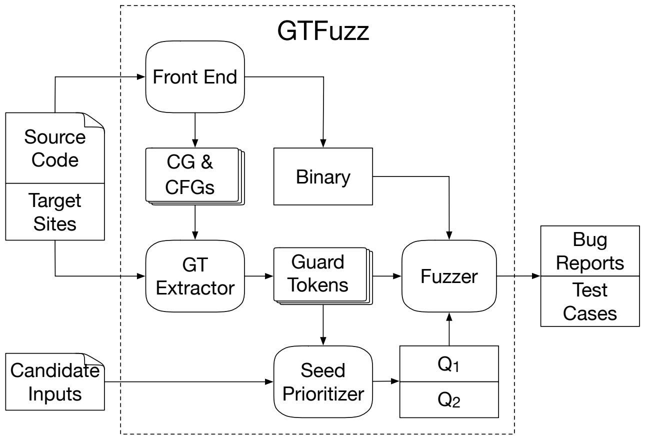

Fig. 3: Overview of GTFuzz

IV. IMPLEMENTATION

By combining several techniques introduced in the previous sections, we develop the *Guard Token* directed greybox fuzzer, GTFuzz for short, on the basis of AFLGO. The overview of GTFuzz is shown in Fig. 3.

GTFuzz consists of four components: Front End, GT Extractor, Seed Prioritizer, and Fuzzer. The input of GTFuzz is the PUT source code, target locations, and candidate inputs. Firstly, the source code and target sites are processed by the Front End which is used to generate the call graph, control flow graph of each function, and the compiled binary file of PUT. Then, the target location, call graph, and control flow graphs are passed to GT Extractor. By applying our proposed GT extraction algorithm (Alg. 1), this component outputs *GTs*. And these *GTs* are sent to the Seed Prioritizer and Fuzzer. By searching for the inclusion of *GTs* in the candidate inputs, the Seed Prioritizer prioritizes the initial inputs into two seed queues Q_1 and Q_2 according to Alg. 3. The Fuzzer receives *GTs* and constructs a dictionary with them. When the seed queue is not empty, the Fuzzer starts a directed fuzzing and uses *GTs* to mutate the seeds. Finally, the Fuzzer outputs bug reports and corresponding test cases. In the following, we introduce how these components are implemented.

1) Front End (FE). FE is implemented on top of AFLGO's static analyzer. It constructs call graph (CG) and the relevant control flow graphs (CFGs) based on the inclusion-based pointer analysis technique [23]. Since a statically generated CG may not always be complete, FE also supports manually adding caller-callee relationship in the potential incomplete case. The output of FE is the compiled binary file of the PUT and the files (.dot) which describe the CG and CFGs.
2) GT Extractor (GTE). GTE is the implementation of GTs extraction algorithm (*i.e.,* Alg. 1). It is responsible to extract *GTs* for the target sites from the source code of the PUT, by using CG and CFGs obtained from TE,

so the inputs of GTE are the target sites, source code of the PUT, CG, and CFGs. The main body of GTE is implemented in Python, and the graphs are parsed by the *networkx* package. The outputs of GTE are GTs, each of them is marked by its type (*i.e.,* Reachability-GT or not). For the recognition of GTs (the core implementation of function EXTRACTFROMB in Alg. 1), we calculate all possible syntax tokens by traversing the abstract syntax trees (AST) of the PUT. Clang [24] is adopted as our front end to generate the ASTs.

3) Seeds Prioritizer (SP). SP implements the seed prioritization algorithm (*i.e.,* Alg. 2). It is implemented as a shell script file.
4) Fuzzer. We built the Fuzzer on top of AFLGO (commit c2888e), and improve mainly its fuzzing module (`afl-fuzz.c`). The Fuzzer supports directly reading GTs, stores them as a dictionary (`user extra` in `afl-fuzz.c`), and implements the GT-based mutation algorithm (*i.e.,* Alg. 3).

V. Evaluation

We aim to answer the following questions through experiments:

RQ1 What is the directedness ability of GTFuzz? *i.e.,* are the inputs generated by GTFuzz effective to reach target sites?

RQ2 What is the performance of GTFuzz in reproducing bugs at or near the target sites?

RQ3 Can GTFuzz expose (known or previously undiscovered) bugs in real-world software?

We select four real-world open-source projects as benchmarks, and their profiles are as follows:

1) **LibXML2** is an XML document parsing library.
2) **MJS** is a javascript engine for embedded systems.
3) **cJSON** is a library for parsing JSON data.
4) **LibPNG** is a library for parsing image files in PNG format.

These projects are chosen because they are widely used in practice and also used in the evaluation of other fuzzers (*e.g.,* AFLGO [12], Hawkeye [13], FairFuzz [16] and *etc.*).

We compare GTFuzz with a state-of-the-art directed grey-box fuzzer AFLGO [12] for answering RQ1. Though other directed grey-box fuzzers (*e.g.,* Hawkeye [13], FuzzGuard [25], ParmeSan [26], *etc.*) also compared with AFLGO, they are not publicly available. Besides, for the evaluation of bug reproduction (*i.e.,* RQ2), we add the comparative experiment with other grey-box fuzzers FairFuzz [16] and AFL [10], because FairFuzz is good at reproducing bugs requiring that the inputs must contain some special tokens, and AFL is a state-of-the-art grey-box fuzzer and also the foundation of GTFuzz, AFLGO, and FairFuzz.

The target sites selected in the experiments are mainly from three sources: CVE lists, potential bug reports, and the issue lists of open-source repositories. The potential bug reports come from Canalyze [27], a static bug-finding tool for C programs.

Our experiments are conducted on an Intel® Core™ i7-4712HQ @ 2.30GHz CPU with 4 threads and 8GB memory, running on a 64-bit Ubuntu 16.04 virtual machine. For each comparison experiment, we set the same PUT, inputs, time budget, and fuzzer parameters.

A. Target Site Covering

The capability of covering the target sites is an important factor for measuring DGFs [13]. In this experiment, we evaluate this capability of GTFuzz by measuring the time it consumes to generate an input that can cover the target site for the first time and compare its time with AFLGO's. The target sites in the experiment come from MJS, LibXML2, cJSON, and LibPNG. Specifically, as shown in Table I, cases #1, #3, and #5 come from the static bug reports of Canalyzer, and case #2 comes from Issue-114 in MJS repository [28]. The target site in case #4 is a patch location. Cases #6 and #7 come from Fuzzer Test Suite [29] (LibPNG 1.2.56) that evaluates fuzzers' abilities to explore the locations which are usually hard to reach. These subjects are also used in the evaluation of Hawkeye [13] (a DGF tool).

Table I shows the results of this experiment. The first column of the table is the ID of each experiment. The file and corresponding line number of the target sites are shown in the second column. The third and fourth columns describe the tools and the number of runs that successfully trigger a particular vulnerability. The *Time-to-Exposure* (TTE) measures the length of the fuzzing process until the first test input that reaches the target site is generated. We calculate the mean of TTE and show it in the fifth column. The *factor improvement* (Factor) measures the performance gain as the mean TTE of AFLGO divided by the mean TTE of GTFuzz. Values of Factor greater than one mean that GTFuzz outperforms AFLGo. The Vargha-Delaney statistic ($\hat{A}12$) is a recommended standard measure for evaluating randomized algorithms [30]. It measures the probability that running GTFuzz yields higher TTE values than running AFLGO, and the higher $\hat{A}12$ means the higher confidence that GTFuzz performs better than AFLGO.

Cases #1 to #5 indicate that GTFuzz outperforms AFLGO on quickly reaching the target sites. In these experiments, GTFuzz extracts *Guard Tokens* of the target sites successfully. All these tokens are deep in the program logic and guard the key predecessors of target blocks. With the help of these tokens, GTFuzz effectively prioritizes the initial seeds as two tiers, first mutates and schedules those containing Reachability-GT, and thus reach the target sites more quickly than AFLGO. The Factors in these cases show that GTFuzz is $2.73\times$ to $10.7\times$ faster than AFLGO in covering the target sites. And the $\hat{A}12$ values in these cases show that GTFuzz owns a stable advantage of reaching the target sites than AFLGO.

In cases #6 and #7, GTFuzz performs comparably with AFLGO since most of the input files provided by the developers contain GTs, (*e.g.,* the PNG header and `IDAT`).

TABLE I: Results of GTFuzz and AFLGo on Covering Target Sites

ID	Target site	Tool	Runs	μTTE	Factor	$\hat{A}12$
#1	mjs.c:12168	GTFuzz	20	23.10s	-	-
		AFLGO	20	165.38s	7.16	1.00
#2	mjs.c:9729	GTFuzz	20	86.05s	-	-
		AFLGO	20	496.00s	5.47	1.00
#3	xpath.c:11367	GTFuzz	20	14.89s	-	-
		AFLGO	20	160.31s	10.70	1.00
#4	xpath.c:14111	GTFuzz	20	25.04s	-	-
		AFLGO	20	251.88s	10.06	1.00
#5	cJSON.c:2688	GTFuzz	20	2.08s	-	-
		AFLGO	20	5.67s	2.73	0.95
#6	pngread.c:738	GTFuzz	20	12.72s	-	-
		AFLGO	20	13.16s	1.03	0.83
#7	pngrutil.c:3182	GTFuzz	20	15.80s	-	-
		AFLGO	20	15.94s	1.01	0.47
				Average $\hat{A}12$		Median $\hat{A}12$
		GTFuzz		0.89		0.83

TABLE II: Subjects for Crash Reproduction

Project	Version	Bug ID	Source
LibXML2	2.9.4	#1	issue 129-1
		#2	issue 129-2
		#3	issue 129-3
	2.9.2	#4	CVE-2017-5969
MJS	2.17.0	#5	issue 150-1
		#6	issue 111
LibPNG	1.2.44	#7	CVE-2011-2501
	1.0.65	#8	CVE-2015-8540

B. Crash Reproduction

To evaluate the crash reproduction capability of GTFuzz, we measure the average time to reproduce crashes for four fuzzers, *i.e.,* GTFuzz, AFLGO, FairFuzz, and AFL. The subjects of this experiment are shown in Table II. The last column describes the source of each crash. For each experiment, four fuzzers are configured with the same initial inputs provided by the developers. The time budget for each run is set to 4 hours and the time-to-exploitation is set to 40 minutes.

Table III shows the results. In reproducing the crashes in LibXML2, GTFuzz is $1.15\times$ to $5.21\times$ faster than AFLGO, $1.04\times$ to $21.01\times$ faster than FairFuzz, and $1.23\times$ to $6.19\times$ faster than AFL. Especially in case #4, GTFuzz performs the best as shown by the Factor values. The crash is caused by a null pointer error and also a typical example in AFLGO's repository [31]. In this case, GTFuzz extracts one Reachability-GT `<!DOCTYPE` and some other GTs (*e.g.,* `<!ELEMENT`, `save`, *etc.*). These GTs helps GTFuzz generate more inputs that can pass the checks of GTs in the PUT and reach the target site. So GTFuzz triggers the error in the target site quickly. In contrast, FairFuzz captures 49 tokens in this experiment, most of which are incomplete or incorrect, *e.g.,* `doc` and `<!ENTIn?Do`. Computing these tokens and exploring the unrelated branches took FairFuzz lots of time. Moreover, the completeness of the extracted tokens shows that GTFuzz using backward static analysis has a significant advantage over FairFuzz using forward dynamic taint analysis.

In reproducing the crashes in MJS, GTFuzz is $3.27\times$ to $5.52\times$ faster than AFLGO, $88.89\times$ to $144.60\times$ faster than FairFuzz, and $4.05\times$ to $88.89\times$ faster than AFL. The $\hat{A}12$ values of cases #5 and #6 are both 1.00, which means GTFuzz outperforms the other three fuzzers with almost 100% confidence. Through manual analysis, we find that the Reachability-GTs extracted from the two cases by GTFuzz have strong relevances with the target functions, which means that the target sites could be covered only if these GTs appear in the input files.

In reproducing the crashes in LibPNG, GTFuzz is $1.00\times$ to $1.1\times$ faster than AFLGO, and $1.00\times$ to $2.07\times$ faster than FairFuzz. AFL performs best in case #7 because 1) this bug is easily triggered with some simple mutations; 2) AFL does less or no extra work (*e.g.,* calculate and maintain tokens/distances) than other three fuzzers, and thus is the first to start fuzzing. However, GTFuzz is $5.68\times$ faster than AFL in case #8. In both cases, the input types that LibPNG can accept are PNG image files, and its developers provide few seed files. The Reachability-GTs extracted by GTFuzz in these cases are PNG header, `tEXt`, `zTXt`, *etc.*. Additionally, a lot of other GTs are extracted, including most of the keywords in general PNG files. Although these GTs provide GTFuzz little help to effectively prioritize the initial seeds, GTFuzz still performs better than the other fuzzers due to the GT-based mutation algorithm, as shown from the $\hat{A}12$ values in these cases.

C. Bugs Exposure

To evaluate the bug exposure capability of GTFuzz against real-world software, we use it to conduct directed fuzzing on four open-source software, *i.e.,* LibXML2, MJS, cJSON, and LibPNG, since they contain rich syntax tokens. We adopt the bug locations reported by Canalyze [27] as the target sites to detect potential bugs near these locations. GTFuzz consumes an average of 3.50 minutes to extract GTs for a target site.

Table IV shows the bugs exposed by GTFuzz in this experiment. Since the fuzzer may find multiple crashes near a target site, here we distinguish unique bugs according to the bug type and the location where the bug occurred. The first two columns show the name and version of the software projects, followed by the numbers of bugs exposed by GTFuzz. The fourth column describes whether these bugs are reported before or not. Specifically, if a bug was reported, the CVE-ID or issue number of the report would be listed. Among the previously unreported bugs, some have the same bug types and causes as some reported bugs, but different locations where crashes occur.

TABLE III: Performance of GTFuzz, AFLGO, FairFuzz and AFL on Crash reproduction

Bug ID	Tool	Runs	μTTE	Factor	$\hat{A}12$
#1	GTFuzz	6	3.26h	-	-
	AFLGO	2	3.76h	1.15	0.69
	FairFuzz	2	3.40h	1.04	0.56
	AFL	0	4.00h	1.23	0.75
#2	GTFuzz	8	0.19h	-	-
	AFLGO	8	0.34h	1.80	1.00
	FairFuzz	0	4.00h	21.01	1.00
	AFL	8	0.48h	2.53	1.00
#3	GTFuzz	8	0.48h	-	-
	AFLGO	8	1.55h	3.18	0.94
	FairFuzz	4	2.35h	4.80	0.81
	AFL	6	3.02h	6.19	1.00
#4	GTFuzz	8	0.23h	-	-
	AFLGO	6	1.20h	5.21	0.81
	FairFuzz	4	2.10h	9.15	0.62
	AFL	8	0.74h	3.21	0.53
#5	GTFuzz	8	2.70m	-	-
	AFLGO	8	8.84m	3.27	1.00
	FairFuzz	0	240.00m	88.89	1.00
	AFL	0	240.00m	88.89	1.00
#6	GTFuzz	8	18.72s	-	-
	AFLGO	8	103.35s	5.52	1.00
	FairFuzz	8	2706.41s	144.60	1.00
	AFL	8	75.73s	4.05	1.00
#7	GTFuzz	8	5.41s	-	-
	AFLGO	8	5.38s	1.00	0.69
	FairFuzz	8	5.39s	1.00	0.62
	AFL	8	4.30s	0.79	0.34
#8	GTFuzz	8	1.10m	-	-
	AFLGO	8	1.21m	1.1	0.94
	FairFuzz	8	2.27m	2.07	1.00
	AFL	8	6.25m	5.68	0.69

In this experiment, GTFuzz detected 46 bugs, 23 bugs of which are previously undiscovered. We have reported these 23 bugs to their developers, 5 of them have been fixed and others are waiting for confirmation. Among the previously undiscovered bugs, 8 bugs found in LibXML2 affect version 2.9.4 and previous versions, and 15 bugs in MJS still exist in the latest version. The inputs for reproduction (also called proof-of-concepts, PoCs) of all bugs include the Reachability-GTs extracted by GTFuzz at the corresponding target sites. For example, GTFuzz found a buffer over-read bug in `mjs.c:12168` in project MJS. For this target site, GTFuzz extracted a Reachability-GT `stringify` and many other GTs (*e.g.*, −, ++, == and *etc.*). In this experiment, only one seed file (*i.e.*, `mjs/tests/test_11.js`) provided by developers contains the Reachability-GT `stringify`, so GTFuzz puts this seed into the queue Q_1 and others to the queue Q_2. Additionally, all the GTs are also transmitted to the fuzzer. During the fuzzing stage, GTFuzz first mutates the inputs in queue Q_1. Directed with the Reachability-GT `stringify`, the inputs mutated from the input in queue Q_1 reach the target site quickly. For instance, a generated POC of this bug is: `let a = JSON.stringify -;JSON.stringify(a);`, and it is obtained by a "mutant" of the input file in queue Q_1.

TABLE IV: Bugs Exposed by GTFuzz

Project	Version	Bugs	Reported Before
LibXML2	2.9.4	10	CVE-2018-14404 (1)
			CVE-2017-5969 (1)
			Previously Undiscovered (8)
MJS	2.17.0	27	issue 113 (4)
			issue 107 (2)
			issue 114 (2)
			issue 55 (1)
			issue 64 (1)
			issue 66 (1)
			issue 111 (1)
			Previously Undiscovered (15)
cJSON	1.7.10	7	CVE-2019-11834 (7)
LibPNG	1.2.44	2	CVE-2011-2501 (1)
			CVE-2015-8540 (1)

VI. Discussion and Future Work

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

A threat to internal validity is the correctness of our implementation of GTFuzz. To mitigate the threat, we built GTFuzz on AFL, a state-of-the-art grey-box fuzzer. And the experimental results show that GTFuzz gains both effectiveness and efficiency. Moreover, when identifying potential GTs, we mainly considered two kinds of token patterns and many valid GTs are recognized from the benchmarks, although there may be other kinds of GTs that should be considered. We leave it as future work to consider more patterns and extract more GTs.

VII. Related Works

Coverage-based Grey-box Fuzzing. CGF generally uses and improves the input path coverage of the PUT to guide the generation of new inputs [10], [11], [16], [32]–[34]. A new path may lead to a new program execution state or new code coverage, and thus facilitate the exposure of potential vulnerabilities in the PUT. The coverage-based grey-box fuzzer, AFL [10], provides a feature for guessing the syntax tokens in PUT during the *deterministic stage* of seeds mutation. A sequence of consecutive characters is recognized as a syntax token if a flip operation of these characters still covers the same path. The syntax tokens extracted by AFL are often different from than *Guard Tokens* extracted by GTFuzz because it does not consider the target sites. As a result, many tokens that AFL extracts from the seed files are unrelated to the target sites. In contrast, GTFuzz extracts *Guard Tokens* of target sites from the PUT's source code and seed files.

Fairfuzz [16] automatically identifies branches exercised by few AFL-produced inputs (rare branches) and dynamically calculates the mutation masks during fuzzing to help the generated inputs likely cover these branches. Fairfuzz uses the mutation masks to protect key pieces of new inputs from being broken. The *Guard Tokens* extracted by GTFuzz play a similar role in the mutation of inputs, while the extraction methods are different. The mutation masks in Fairfuzz are dynamically computed during the fuzzing process through forward dynamic taint analysis while GTFuzz uses a lightweight static analysis to extract *Guard Tokens*, which is much more efficient because it does not take up extra time during fuzzing, as demonstrated in section V-B. Moreover, when few or no seed inputs can cover the target site, Fairfuzz has trouble in extracting mutation masks related to the target. In contrast, GTFuzz can still extract the *Guard Tokens* of the target site from the PUT's source code.

Orthrus [21] extracts syntax tokens from the heuristic taint sinks as a dictionary of fuzzer by using static analysis. Different from Orthrus, GTFuzz measures the control flow information of the PUT when extracting syntax tokens, so it can further extract the Reachability-GT of the target site.

Directed Grey-box Fuzzing. DGF focuses on detecting bugs at or near the user-specified target location of the PUT [12]–[14]. AFLGO [12] is the state-of-the-art directed grey-box fuzzer. It casts the reachability of target locations as an optimization problem and uses a heuristic method to generate the inputs that are close to the target.

Inspired by AFLGO, an improved directed grey-box fuzzer Hawkeye [13] was proposed. It collectd from the PUT the information (*e.g.,* call graph, function, and basic block level distances), and designs seed prioritization method, augmented annealing-based power schedule and adaptive mutation algorithm. These improvements enhance its directedness and are complementary to the proposed method in this paper, including seed prioritization and mutation.

SCDF (Sequence-coverage Directed Fuzzing) [14] considers a more specific application scenario for DGF – covering a user-specified target sequence in consecutive order. It focuses on generating inputs that can reach the target sites in each sequence in order and triggers bugs in the PUT. In the new scenario, SCDF designs a novel energy schedule algorithm to measure the mutation time based on the distance between the target execution traces and the given statement sequence.

Taint Analysis Aided Fuzzing. There are many fuzzers [17], [18], [35], [36] aided with taint analysis which is used to obtain the relationship between a portion of an input and certain branches of the PUT. GTFuzz is a light weight directed fuzzing approach compared with these fuzzers which require heavy run-time taint analysis.

TaintScope [17] utilizes dynamic taint analysis to identify which bytes in a well-formed input are used in security-sensitive operations, and such bytes are given more attention by the mutation operation. Additionally it recognizes the checksum fields in the inputs and enables the fuzzer to test deeper logic of the PUT by helping the new inputs bypass such checks via control flow alteration. GTFuzz has the same intuition as TaintScope: it facilitates the inputs pass some specific checks (*i.e., Guard Tokens*) in the PUT by identifying and protecting the key fields in the inputs. Unlike TaintScope, GTFuzz helps the inputs pass the checks by protecting the Reachability-GTs in the inputs, and TaintScope bypasses these checks by modifying the PUT and thus may break the code logic of PUT at the checkpoints.

SeededFuzz [18] locates the key fields of inputs that are related to security-sensitive code via dynamic taint analysis. It directly selects the seeds based on their reachability to the target site. When all seed inputs fail to reach the target, SeededFuzz cannot make a valid selection though GTFuzz can still generate appropriate seed inputs that may reach the target with the help of *Guard Tokens*. In fact, the selection strategy of SeededFuzz does not conflict with GTFuzz. When there are a lot of candidate inputs, the combination of the two strategies is conducive to obtain a small and precise set of seeds.

VIII. Conclusion

In this paper, we present the *Guard Token* directed grey-box fuzzing. By introducing GTs of target site, we design a GT extraction algorithm. We also propose a seed input prioritization and dictionary generation method based on these tokens. By using GTs in the mutation process, we enhance the seed mutation algorithm. As a result, the proposed methods can efficiently prioritize inputs for directed fuzzing, and generate new inputs containing GTs by protecting GT from being destroyed. For those inputs that cannot cover the target site, our method is good at mutating them into intended inputs by inserting or overwriting some GTs to these inputs.

We implement these methods in the fuzzer GTFuzz. The experimental results show that GTFuzz is more efficient in target site covering than AFLGO. GTFuzz can reproduce the crashes faster than AFL, AFLGO and FairFuzz. Moreover, GTFuzz exposed 23 bugs that have never been discovered before.

Acknowledgment

This work is supported by the Key Research Program of Frontier Sciences, Chinese Academy of Sciences (Grant No. QYZDJ-SSWJSC036).

The authors would like to thank the anonymous reviewers for their helpful comments, and Ke Yang, Pei huang and Xiaoni Meng for their valuable advice on this project.

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] J. Viide, A. Helin, M. Laakso, P. Pietikäinen, M. Seppänen, K. Halunen, R. Puuperä, and J. Röning, "Experiences with model inference assisted fuzzing," in *2nd USENIX Workshop on Offensive Technologies, WOOT'08, San Jose, CA, USA, July 28, 2008, Proceedings*, D. Boneh, T. Garfinkel, and D. Song, Eds. USENIX Association, 2008.

[3] H. Yang, Y. Zhang, Y. Hu, and Q. Liu, "IKE vulnerability discovery based on fuzzing," *Security and Communication Networks*, vol. 6, no. 7, pp. 889–901, 2013.

[4] J. Yan, Y. Zhang, and D. Yang, "Structurized grammar-based fuzz testing for programs with highly structured inputs," *Security and Communication Networks*, vol. 6, no. 11, pp. 1319–1330, 2013.

[5] N. Palsetia, G. Deepa, F. A. Khan, P. S. Thilagam, and A. R. Pais, "Securing native XML database-driven web applications from xquery injection vulnerabilities," *J. Syst. Softw.*, vol. 122, pp. 93–109, 2016.

[6] P. Godefroid, M. Y. Levin, and D. A. Molnar, "SAGE: whitebox fuzzing for security testing," *ACM Queue*, vol. 10, no. 1, p. 20, 2012.

[7] S. K. Fayaz, T. Yu, Y. Tobioka, S. Chaki, and V. Sekar, "BUZZ: testing context-dependent policies in stateful networks," in *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*, K. J. Argyraki and R. Isaacs, Eds. USENIX Association, 2016, pp. 275–289.

[8] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017.* IEEE Computer Society, 2017, pp. 579–594.

[9] P. D. Marinescu and C. Cadar, "Katch: high-coverage testing of software patches," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 235–245.

[10] Lcamtuf, "american fuzzy lop (2.52b)," Website, 2019, http://lcamtuf.coredump.cx/afl/.

[11] "Libfuzzer: A library for coverage-guided fuzz testing," http://llvm.org/docs/LibFuzzer.html, dec 15, 2019.

[12] M. Böhme, V. Pham, M. 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.

[13] 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 2018, Toronto, ON, Canada, October 15-19, 2018*, 2018, pp. 2095–2108.

[14] 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.

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

[16] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 475–485.

[17] 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, 16-19 May 2010, Berleley/Oakland, California, USA.* IEEE Computer Society, 2010, pp. 497–512.

[20] B. Mathis, R. Gopinath, M. Mera, A. Kampmann, M. Höschele, and A. Zeller, "Parser-directed fuzzing," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, 2019, pp. 548–560.

[18] W. Wang, H. Sun, and Q. Zeng, "Seededfuzz: Selecting and generating seeds for directed fuzzing," in *2016 10th International Symposium on Theoretical Aspects of Software Engineering (TASE).* IEEE, 2016, pp. 49–56.

[19] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing." in *NDSS*, vol. 17, 2017, pp. 1–14.

[21] B. Shastry, M. Leutner, T. Fiebig, K. Thimmaraju, F. Yamaguchi, K. Rieck, S. Schmid, J. Seifert, and A. Feldmann, "Static program analysis as a fuzzing aid," in *Research in Attacks, Intrusions, and Defenses - 20th International Symposium, RAID 2017, Atlanta, GA, USA, September 18-20, 2017, Proceedings*, 2017, pp. 26–47.

[22] laf intel, "Circumventing fuzzing roadblocks with compiler transformations," https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/, aug 15, 2016.

[23] Y. Sui and J. Xue, "Svf: interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th international conference on compiler construction.* ACM, 2016, pp. 265–266.

[24] "Clang: a c language family frontend for llvm," https://clang.llvm.org/.

[25] P. Zong, T. Lv, D. Wang, Z. Deng, R. Liang, and K. Chen, "Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning," 2019.

[26] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida, "Parmesan: Sanitizer-guided greybox fuzzing," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[27] Z. Xu, J. Zhang, Z. Xu, and J. Wang, "Canalyze: a static bug-finding tool for C programs," in *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, 2014, pp. 425–428.

[28] "mjs: Restricted javascript engine," https://github.com/cesanta/mjs/.

[29] Google, "Fuzzer test suite," https://github.com/google/fuzzer-test-suite, 2020.

[30] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[31] "Aflgo: Directed greybox fuzzing," https://github.com/aflgo/aflgo/.

[32] W. You, X. Liu, S. Ma, D. M. Perry, X. Zhang, and B. Liang, "SLF: fuzzing without valid seed inputs," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, J. M. Atlee, T. Bultan, and J. Whittle, Eds. IEEE / ACM, 2019, pp. 712–723.

[33] M. Böhme, V. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM, 2016, pp. 1032–1043.

[34] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017.* The Internet Society, 2017.

[35] V. Ganesh, T. Leek, and M. C. Rinard, "Taint-based directed whitebox fuzzing," in *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings.* IEEE, 2009, pp. 474–484.

[36] 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 Computer Society, 2018, pp. 711–725.