DDGF: Dynamic Directed Greybox Fuzzing with Path Profiling

Haoran Fang
Shanghai Jiao Tong University
Shanghai, China
haoran.cs@sjtu.edu.cn

Kaikai Zhang
Shanghai Jiao Tong University
Shanghai, China
lethe_k@sjtu.edu.cn

Donghui Yu
Shanghai Jiao Tong University
Shanghai, China
yudhui@sjtu.edu.cn

Yuanyuan Zhang*
Shanghai Jiao Tong University
Shanghai, China
yyjess@sjtu.edu.cn

Abstract

Coverage-Guided Fuzzing (CGF) has become the most popular and effective method for vulnerability detection. It is usually designed as an automated "black-box" tool. Security auditors start it and then just wait for the results. However, after a period of testing, CGF struggles to find new coverage gradually, thus making it inefficient. It is difficult for users to explain reasons that prevent fuzzing from making further progress and to determine whether the existing coverage is sufficient. In addition, there is no way to interact and guide the fuzzing process.

In this paper, we design the dynamic directed greybox fuzzing (DDGF) to facilitate collaboration between the user and fuzzer. By leveraging Ball-Larus path profiling algorithm, we propose two new techniques: dynamic introspection and dynamic direction. Dynamic introspection reveals the significant imbalance in the distribution of path frequency through encoding and decoding. Based on the insight from introspection, users can dynamically direct the fuzzer to focus testing on the selected paths in real time. We implement DDGF based on AFL++. Experiments on Magma show that DDGF is effective in helping the fuzzer to reproduce vulnerabilities faster, with up to 100x speedup and only 13% performance overhead. DDGF shows the great potential of human-in-the-loop for fuzzing.

CCS Concepts

• **Security and privacy → Software and application security**.

Keywords

Directed Fuzzing, Introspection, Human-in-the-Loop, Profiling

ACM Reference Format:
Haoran Fang, Kaikai Zhang, Donghui Yu, and Yuanyuan Zhang. 2024. DDGF: Dynamic Directed Greybox Fuzzing with Path Profiling. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650212.3680324

*corresponding author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISSTA '24, September 16–20, 2024, Vienna, Austria
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680324

1 Introduction

In recent years, coverage-guided fuzzing (CGF) [1, 38, 47] has become one of the most effective ways to detect vulnerabilities. Test cases that explore new coverage are added to the queue as seeds for subsequent mutations, which helps the fuzzing to evolve towards expanding coverage, gradually exploring the unknown areas of the program. The effectiveness of CGF has been proven in extensive practice. It has attracted widespread academic research[13, 17, 18, 22, 23, 25, 26, 45] and large-scale industrial deployment. In the OSS-Fuzz[10] project, Google conducts 24/7 continuous fuzzing on integrated open source projects. As of August 2023, OSS-Fuzz has helped identify and fix over 10,000 vulnerabilities and 36,000 bugs across 1,000 projects.

However, the limitations of CGF are overshadowed by the tremendous success in the industry. An inevitable problem is that fuzzing will get stuck after a period of time. It means that the fuzzer struggles to discover new coverage, the seed queue stops changing, and the efficiency will drop sharply. Bohme et al.[16] systematically analyzed the relationship between vulnerability discovery time, code coverage, and the number of fuzzing machines. The empirical study reveals that the difficulty of discovering new vulnerabilities increases exponentially after the coverage reaches the plateau. For most projects, coverage bottlenecks are typically reached within 24 hours. However, although CGF is a greybox testing method, it is usually used as the "black-box" tool in practice. Security auditors prepare the initial corpus, start fuzzing, and just wait for results. When the fuzzer gets stuck, it is difficult for users to find the reason why fuzzing is blocked and there is no way to interact and guide fuzzing effectively. As the two key questions raised in the paper *"Fuzzing: Challenges and Reflections"*[15]:

RQ1. How can the fuzzer explain what prevents it from progressing ? RQ2. How can we facilitate a more effective communication between fuzzer and security auditor ?

We interpret the questions as : 1. how to expose more intermediate states from fuzzing, and 2. how security auditors can direct fuzzing process based on these intermediate states. In this paper, we try to answer these two questions. We propose the dynamic directed greybox fuzzing called DDGF. DDGF includes two subsystems: dynamic introspection and dynamic direction, corresponding to the two questions respectively.

Dynamic Introspection aims to expose the intermediate states of the fuzzing process in real time. Based on the Fuzz-Introspector[5], we transform its post-processing way into real-time processing by memory mapping. Different coverage status can be displayed at fuzzing runtime, which will be used to support the dynamic direction subsystem. In addition to basic-block/line coverage and blocking branches, we introduce the Ball-Larus path profiling algorithm[14] to encode paths within functions. With path encoding and decoding, we record and display the path hit frequency for all seeds in the queue dynamically. By analyzing the coverage data, we can see the significant imbalance in distribution of path frequency. Less than 20% of paths account for 80% of hits. Fuzzing wastes too much time on these paths, which prevent vulnerabilities from being triggered.

Dynamic Direction. We use the encoding number of paths as the key for communication between the user and the fuzzer. Based on the coverage information and insight from dynamic introspection, Security auditors are able to add flags for selected paths and push the fuzzer to focus testing on them. The flags will influence the seed selection stage in fuzzing loop and the seeds with flags will be prioritized. Users are able to direct the fuzzer's attention to target areas of code and help to discovery vulnerabilities faster.

Compared with distance-based directed greybox fuzzing(DGF) such as AFLGo[17, 21, 24, 35], "dynamic" indicates that it allows users to dynamically change or add target paths at any time. Traditional DGF "statically" instrument the hard-coded distance into the program during compilation. If the user want to change the target sites, they have to regenerate the distance by heavy static analysis and recompile the program. The instrumentation of path profiling in DDGF is target-independent and it provides a great interface for interaction.

Evaluation. We evaluated the effectiveness of DDGF on Magma benchmark[29] and real programs. The experiment results show that DDGF is able to provide us with more insight on fuzzing and help to reproduce vulnerabilities faster, with up to 100x speedup in some programs. The performance overhead from instrumentation and interaction is only 13%.

At last, We illustrate the role that humans play in DDGF with three case studies which represent three different flagging strategies. For some cases, where AFL++ took about 20 hours, we got 10x speedup by just flagging one path. It demonstrates the value of insight from humans, whose key flags brought significant speedup to fuzzing. Moreover, we detected 4 unknown vulnerabilities with 2 CVE IDs during the experiment.

In summary, the contributions of this paper are as follows:

- We designed and implemented the dynamic directed fuzzing system called DDGF to facilitate collaboration between humans and fuzzers. DDGF includes two subsystems: dynamic introspection and dynamic direction.
- We proposed the dynamic introspection subsystem. It encodes and decodes paths using the Ball-Larus profiling algorithm to show path-level hit frequency in real time, which provides more insight for users to guide the fuzzing process.
- Based on dynamic introspection, we are first to propose the concept of dynamic direction for fuzzing. Dynamic direction supports users to change target paths at any time and direct the fuzzer to focus testing on these critical paths and regions.

In the following sections, we first introduce relevant background on fuzzing. Next, we describe the motivation and challenges through the example. Then we present the dynamic introspection and dynamic direction subsystem in DDGF respectively. Finally, we discuss the implementation and evaluation of DDGF.

2 Background

2.1 Greybox Fuzzing is not a Black Box

The key to the success of greybox fuzzing lies in the evolutionary algorithm [47]. Test cases that discover new coverage will be added to the queue to serve as seeds for subsequent mutations. However, after fuzzing has been running for a period of time, it becomes increasingly difficult to find new coverage[15, 33]. The lack of new seeds leads to a significant drop in testing efficiency. In this context, it is difficult for users to explain the cause of the blockage and determine whether existing coverage is sufficient. This inspires us to think about how to expose more intermediate states of fuzzing, and how users can guide the fuzzing process.

In June 2022, the OpenSSF released Fuzz-Introspector[5], an open source tool that aggregates data collected during fuzzing to generate comprehensive reports. Fuzz-Introspector helps developers better understand the fuzzing process and make improvements. The project is under active development and is currently the official test report display platform for OSS-Fuzz[10, 11]. There has been some research[28, 44] to analyze fuzzing coverage and bottlenecks based on Fuzz-Introspector.

2.2 Human-in-the-Loop for Program Analysis

Computers have better speed and accuracy, while humans have deeper insight and global perspective. Automated analysis method such as data-flow analysis and fuzzing are not able to understand the semantics of programs. The analyzer takes a general and abstract view to explore the program, such as transfers on the control flow graph. Vulnerabilities are reasoned and detected by computing data dependency or interval on these abstract data. This fundamentally limits the capabilities of analyzers. For example, an automated analyzer is unable to understand the logic of the game and just tries to shift on the control flow. However, humans are able to easily trigger entirely new areas of code based on game logic and semantics[4]. Therefore, if the program analysis tool can expose more intermediate states, integrate human insights into the vulnerability detection process, and fill in semantic information gaps with human-computer interactions, it will greatly enhance the capabilities and efficiency of analysis tools.

After the CGC[3] project, DARPA recognized the importance of human-in-the-loop for vulnerability discovery and proposed the CHESS[4] (Computers and Humans Exploring Software Security) research project. CHESS aims to develop human-computer collaborative systems to detect zero-day vulnerabilities by combining human insights and computers. So far, the implemented subprojects include the static analyzer MATE[7] based on the code property graph and the integrated symbolic execution engine Manticore. Users can inspect static analysis results through an interactive interface, set source-sink for taint analysis, and provide additional constraints for symbolic variables, etc.

2.3 Directed Greybox Fuzzing

Directed Greybox Fuzzing (DGF) [17, 31, 32, 35] can guide the fuzzing direction towards target sites, which is used for patch testing, vulnerability reproduction, etc. The pioneering work of DGF is AFLGo[17]. It computes the distance from the execution path to the target sites by static analysis and assign more energy to seeds closer to the target based on the simulated annealing algorithm. Hawkeye[21] proposes corresponding optimization methods for AFLGo, such as considering indirect calls, optimizing distance metrics, etc. By lightweight static analysis, Beacon[30] instruments the weakest preconditions to terminate unreachable paths in advance. Basically, these methods inherit the overall workflow of AFLGo, hard-coding the constant distance statically in basic blocks during compilation. For this approach, it is difficult to dynamically change or set targets to guide the fuzzing at runtime. However, we believe that directed greybox fuzzing is a kind of prototype of human-computer collaboration, which has great potential for improvement.

3 Motivation & Challenge

In this section, we illustrate our research motivation and corresponding challenges through an example.

3.1 Motivating Example

Switch-case statements often match the program states or key options. Security auditors can easily understand the local program logic through the case name.

Listing 1: Motivating Example

```
1  int wavlike_read_fmt_chunk (SF_PRIVATE *psf, int fmtsize ) {
2      ...
3      switch (wav_fmt->format) {
4          case WAVE_FORMAT_IMA_ADPCM:
5              ...
6              break;
7          case WAVE_FORMAT_GSM610:
8              ...
9              break;
10         case WAVE_FORMAT_EXTENSIBLE:
11             // heap overflow (commit a8ab5b3)
12             // how to focus this case in fuzzing?
13     }
14 }
```

The above code Listing 1 shows a snippet of a heap overflow vulnerability (Commit a8ab5b3) in the *libsndfile* library, which is an encoding/decoding library for common audio formats. We can easily see that the vulnerability is only triggered when the parsed WAVE format is *EXTENSIBLE.* If we want to reproduce this vulnerability faster, it's natural that we want to exclude other branches and have as many test cases as possible that meet the *EXTENSIBLE* format. **When we have preferences for testing some paths, how to tell the fuzzer? For this example, how can we direct fuzzing to focus on a specific case in the switch statement?**

This inspires us that the security auditors need a communication way to tell fuzzers that this case is the most important path to trigger the vulnerability. Once a seed is *EXTENSIBLE* format, it should take priority over other seeds, because the mutation results of this seed are more likely to be the same format. Imagining that there are 100 seeds in the queue, and only 5 of them are the *EXTENSIBLE* format. We should prioritize mutating these seeds to accelerate vulnerability triggering. Traditional DGF such as AFLGo allocates different energy to seeds based on the distance to the target point. However, for the above example, the path distance is nearly the same for different cases, making it difficult to direct fuzzing to a specific case statement.

3.2 Challenges

The motivating example above inspires us to think about how to guide fuzzing to test target paths as much as possible. It includes two parts: 1. how to describe paths and show coverage status; 2. how users can use the status to direct fuzzing. We need to address the following three challenges.

Path Description. We want to establish the connection between users and the fuzzer, so it is crucial that both the fuzzer and the user share the same or similar view of the code. However, the design of coverage instrumentation in popular fuzzers makes it difficult to support this. For example, an edge is represented by the XOR of the random IDs of two basic blocks in AFL, and the hit count is stored in the bitmap with the edge ID. This code-bitmap mapping is one-way and cannot be interpreted by users. It is not easy to map the ID from the bitmap back to the corresponding edge. AFLFast[18] uses the hash of the entire trace_bit array to distinguish different paths and count the hit frequency. This is also irreversible and unexplainable. We need a way to describe a path, which is both understandable and efficient for users and fuzzer.

Dynamic Directing Method. To enable interactions, it should allow users to view and select paths in real time. In coverage-guided fuzzing, the goal of evolutionary algorithm is to expand coverage as much as possible. The fitness function is set to reward test cases that discover new edge coverage. Once we can describe paths and communicate with the fuzzer, we need a method to interact with the existing coverage-driven evolutionary algorithm. This would guide the fuzzer to intensively test the paths selected by user at runtime. Moreover, this user-involved directing approach should be transparent to fuzzing. If there is no interaction or direction from users, the fuzzer can still run in the way of increasing coverage.

Performance Overhead. In order to support users to check fuzzing coverage status and dynamically guide fuzzing, we need to expose these intermediate states in real time. The Fuzz-Introspector uses the post-processing way, and the instrumentation brings the additional overhead. If the overhead is too high, it will slow down the fuzzing speed. This could potentially lead to directed fuzzing being less capable than general coverage-guided fuzzing. Balancing performance and additional features is a significant challenge. There is always a trade-off.

4 Approach

In this chapter, we first introduce the overall architecture of Dynamic Directed Greybox Fuzzing (DDGF). DDGF consists of two subsystems: dynamic introspection and dynamic direction, which allows users to inspect intermediate states and direct fuzzing in real time. When users discover low-frequency or potential vulnerable paths by introspection, they can dynamically add flags for these

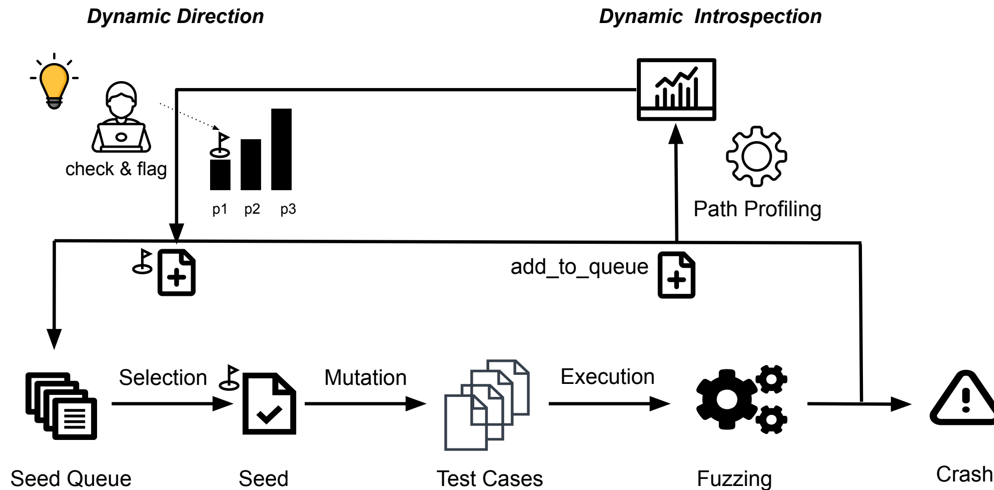


Figure 1: Design Overview of DDGF

paths. This will direct the fuzzer to test these paths as much as possible, thus to detect vulnerabilities more efficiently. We will explain the details of two subsystems respectively.

4.1 Design Overview

Algorithm 1 Fuzzing Loop in DDGF

Input: Seed Inputs S, PathProfiling Instrumented binary bin, External Shared PathProfiling File pp_file
Output: Crash Outputs C

```
1:  for each s ∈ S do
2:      s.flag ← PathProfiling(bin, s, pp_file)
3:  end for
4:  repeat
5:      s ← chooseNext(S) // prioritize seeds with flag
6:      p ← assignEnergy(s)
7:      for i = 1 to p do
8:          s′ ← mutateInput(s)
9:          if s′ crashes then
10:             add s′ to C
11:         else if s′ isInteresting then
12:             s′.flag ← PathProfiling(bin, s′, pp_file)
13:             add s′ to S
14:         end if
15:     end for
16: until timeout reached or abort signal
```

Figure 1 shows the overall design architecture of DDGF. We can see there are two loops. The lower loop is a typical coverage-guided fuzzing process. Seeds are taken from the queue and mutated to generate test cases, which are fed to the target program for execution. If the test case discovers new coverage, it will be added to the queue as a seed for subsequent mutations. The upper loop is our dynamic introspection and direction system. When a new seed is added to the queue, another target binary program instrumented with path profiling will be executed once with that seed as input. Path profiling is used to analyze path-level hit counts of all seeds, and combined with fuzz-introspector to display line/block/function-level coverage and blocking branches. Path profiling establishes a way for the fuzzer to communicate with the user, which builds the dynamic direction subsystem. More specifically, users can analyze the hit frequency distribution of different paths using introspection, and dynamically add flags for specific paths such as low frequency but critical paths. Seeds that go through these selected paths will get the flag. The flags will affect the seed selection stage in fuzzing loop and the seeds with the flag will be preferentially selected. **In evolutionary algorithms, if seeds carrying certain characteristics are preferentially selected each time, their mutations are also more likely to carry the certain characteristics. That is, fuzzing can evolve towards these flagged paths.** In addition, the flags added by users are transparent to the fuzzer. If no flags are found, fuzzer will still proceed in the status of expanding coverage. Users can switch fuzzing modes at any time.

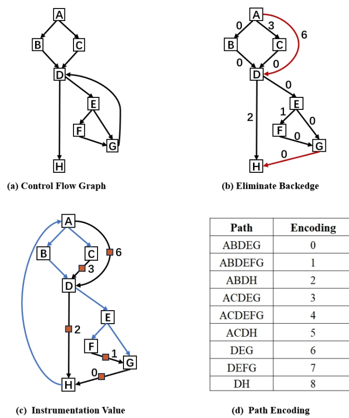
Algorithm 1 illustrates the fuzzing loop in DDGF. First, we need to compile the program with path profiling instrumentation and get a special binary program *bin*. On the one hand, this program is used for introspection to count path frequency. On the other hand, we use it as an interface between fuzzer and user. Users can add flags for the selected paths before fuzzing starts or gradually add flags at runtime, both of which can be done by simply modifying *pp_file* with python scripts. During the whole period of fuzzing, *bin* will be executed in two different stages: ① Before fuzzing starts. ② Finding a new seed. The seeds with the flag will be selected with priority. We use the shared memory-mapped file *pp_file* for the

path counters and flag reading/writing. More details are provided in Section 4.3.

4.2 Dynamic Introspection

Path Profiling. In related fuzzing research work such as JMPScare[36] and Fuzz-Introspector[5], introspection refers to reporting various states of the program under test, such as coverage information and blocking branches. As mentioned in the challenges above, in order to support directed testing of selected paths, we first need a way to describe the path. Therefore, we introduce the Ball-Larus algorithm[14] for path profiling.

Path profiling describes the dynamic control flow behavior for a program. It records the sequence of instructions executed within functions for each test case, which can help to analyze the behavior of the function. Ball and Larus proposed an efficient path profiling algorithm. Their approach provides an encoding and decoding strategy that allows us to map any path in a directed acyclic graph to a unique integer.



Path	Encoding
ABDEG	0
ABDEFG	1
ABDH	2
ACDEG	3
ACDEFG	4
ACDH	5
DEG	6
DEFG	7
DH	8

Figure 2: Ball-Larus Path Profiling

The Figure 2 demonstrates the simplified workflow of Ball-Larus algorithm. For the CFG with loops, we convert it into the directed acyclic graph (DAG) by eliminating the backedge. For instance, for a backedge v -> w, the steps are as follows: ① Add a dummy edge *ENTRY* -> w. ② Add a dummy edge v -> *EXIT*. ③ Eliminate the backedge. At the end of the algorithm, we can calculate a unique encoding value for each path in the CFG, which is used as the path ID. Moreover, we can easily map the path ID back to the corresponding path in the CFG with decoding algorithm. We instrument a special function at the end of the path to count the hits during the runtime. Besides, path ID is not only used for introspection, but also provides an important interface to implement the direction subsystem in DDGF. We will discuss the details in the next section.

Two-Files Fuzzing Strategy. To reduce the performance overhead of path profiling instrumentation, we adopt the two-files fuzzing strategy. The path profiling instrumented binary is compiled separately. By sampling paths hit by all seeds, we can get the distribution of hit frequency. As shown in the Algorithm 1, just after fuzzing starts and before new seeds are added to the queue, we run the path profiling instrumented binary for them. Thus we maintain the set of all executed paths and their hit frequency during the runtime of fuzzing. The overhead of path profiling instrumentation is amortized over the mutation of each seed, with small impact on fuzzing speed.

From Post-Processing to Dynamic Introspector. Through post-processing, Fuzz-Introspector can extract information such as line coverage and blocked branches of the target program. However, it does not support dynamic (real-time) display. Inspired by the continuous mode in LLVM profile-guided optimization[6] (PGO), we constructed dynamic introspector by mapping the different coverage information from memory to files in real time through *mmap*. This is an engineering challenge, but it is very necessary. Dynamic introspector allows us to view different coverage information, including path-profiling, in real time. It provides a platform for human-fuzzer interaction and direction.

4.3 Dynamic Direction

In this section, we will introduce the dynamic direction subsystem. For the upper loop in Figure 1, users can check the path frequency distribution in real time with introspection, and map the path ID back to the path in the source code with decoding. If users have specific requirements, such as increasing the test probability of some low-frequency paths or testing a certain case in switch statement as much as possible, they can flag the relevant paths and then direct fuzzer to test these areas. Next, we will introduce the details of path flagging and seed selection in dynamic direction subsystem of DDGF.

Path Flagging. Path flagging plays a very important role in the DDGF. It provides a key interface for users to interact with fuzzer, because the path ID is understandable for both of them. Users can add flags to the path based on the introspection results. The flags will be written into an external shared profiling file. When the program executed into the instrumented function, it checks whether the current path is flagged or not and then return it to fuzzer. Total flags are accumulated in different functions. Fuzzer will set the flag for current seed. The seeds with the flag will be prioritized in the fuzzing loop. It is worth noting that the runtime overhead from instrumentation is relatively low due to the two-files fuzzing strategy mentioned before.

Algorithm 2 Instrumented Function: update_counters_flags

Input: *current_func*, *path_id*, *pp_file*, Global *total_flags*
Output: *total_flags*

```
1: flag = get_flag_from_file(pp_file, current_func, path_id)
2: if flag == 1 then
3:     total_flags++ // accumulated in different functions
4: end if
5: update_hit_counters( pp_file, current_func, path_id)
```

In Algorithm 2, we describe the function *update_counters_flags* , which is instrumented at the end of paths in profiling algorithm. This function has two purposes: ① To count the frequency of the current path, and ② To check if the path has been flagged by the user. The flags are accumulated in the global variable *total_flags*, which represents the total number of flags paths encountered across

all functions for the current seed. We use shared container in Boost library to build the *pp_file*. It is implemented by memory mapping, which is very fast and helps to reduce the runtime overhead. Users can also read and write the *pp_file*, and view or add the flag for paths with python script or web GUI.

Seed Selection. Our directing approach is based on a natural but effective observation in evolutionary algorithms: if the seeds selected for mutation always have some certain characteristics, the mutated results are more likely with these characteristics. It means that if we preferentially select seeds with the flag for mutation, we will increase the probability of testing these flagged paths.

The existence of flags also gives the other stages of fuzzing loop a chance to have an indirect effect on the seed selection results. However, if we change the feedback strategy, e.g., by adding only flagged seeds to the queue, it will seriously affect the queue size. This makes fuzzing difficult to recover to the original coverage-guided mode and may fall into a local optimum. If we want to recover from directed mode to the coverage-guided mode, we just need to make it ignore the flag. Changing the seed selection method based on flags will not do damage to the original seed queue.

Algorithm 3 Constraints for update_bitmap_score

Input: *top_rated_arrays*, current_seed s
Output: *top_rated_arrays*
1: **if** $directed_mode$ && $s.flag == 0$ **then**
2: return;
3: **end if**
4: update_bitmap_score(s)

We prioritize seeds with the flag, pushing fuzzing to focus on these flagged paths. Once fuzzer has explored the flags, the directed mode is turned on. Specifically, we change the seed selection process through the *update_bitmap_score* function in AFL++. *top_rated[bitmap_size]* is an array used in seed selection algorithm of AFL. It records the winner of each edge, i.e., the seed with the shorter length and faster execution time. As shown in Algorithm 3, in directed-mode, we add a constraint that seeds without the flag will not be eligible to compete. Eventually, there are more winners with the flag in *top_rated* array. These winners will be marked as favored in the cull_queue by its greedy algorithm, thus improving the chances of being selected. If we add flags at run-time, we will dynamically change the direction of fuzzing and influence the evolution of fuzzing continuously.

Directing the Fuzzer Step by Step. The dynamic direction subsystem allows the user to dynamically add flags. For example, if we want to reproduce a vulnerability, we can add flags for the relevant paths in the top function of stack trace. However, we may find that the fuzzer does not explore the target paths and never enters the directed mode, or we may get too few seeds with the flag and may fall into a local optimum. In this context, we can add new flags dynamically. For example, we can add flags for the previous function in the stack trace, thus increasing the chance of finding flags. In this way, we can direct the fuzzing step by step.

Figure 3: Dynamic Introspector

Figure 4: Line Coverage

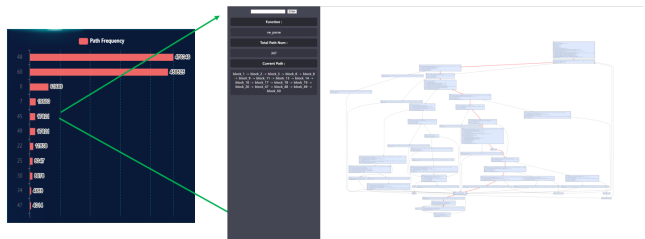


Figure 5: Path Frequency and Decoding in CFG

4.4 Implementation

We implemented dynamic directed greybox fuzzing based on AFL++. Initially, we implemented Ball-Larus path profiling algorithm ourselves, about 1500 lines of C++ code. Now we have modified and ported the Path Profiling in legacy clang-3.1 to afl-clang-lto-14. We created a web GUI based on D3.js[2] and Echarts to support various information display such as path frequency sorting, path decoding, coloring, etc. We combined it with fuzz-introspector to implement the Dynamic Introspector. We used LLVM PGO's[6] continuous mode to convert the post-processing into a real-time version. Moreover, we implemented decoding and path display with python, so that users can visually inspect and flag related paths. Note that our

CFG can be associated with source code line numbers, thus supporting path inspection on the source code. Based on memory-mapped files in Boost library[9], we implemented the real-time updating and flagging for path-profiling files. Users can add the flag for paths at any stage of fuzzing.

Figure 3 displays the dashboard of our dynamic introspection system. During the fuzzing process, we can visually observe the testing status of each function in real time based on the depth of colors. Moreover, as shown in the Figure 4, each function can be directly linked to its corresponding source code. The path execution frequency within the function is shown in real time by the dynamic sorting diagram in the bottom right corner. As shown in Figure 5, we can decode a certain path and display it in CFG.

5 Evaluation

In this section we evaluate the effectiveness of DDGF. As we mentioned in the introduction, to address two key RQs in "fuzzing challenges and reflections", DDGF is designed to facilitate collaboration between users and fuzzers to improve fuzzing efficiency. For the two subsystems of DDGF, we propose the following three more specific research questions.

RQ1. Does the dynamic introspection subsystem actually provide more insights about fuzzing ?

RQ2. Based on these insights, could we use the dynamic direction subsystem to reproduce bugs faster ?

RQ3. What is the performance overhead of DDGF?

We evaluated DDGF on the Magma[29], a benchmark now widely used for fuzzing evaluation. Magma contains real bugs, which are sourced from bug reports and forward-ported to the most recent version of the target codebase. The *magma_log* function instrumented at the source code checks whether a vulnerability is triggered.

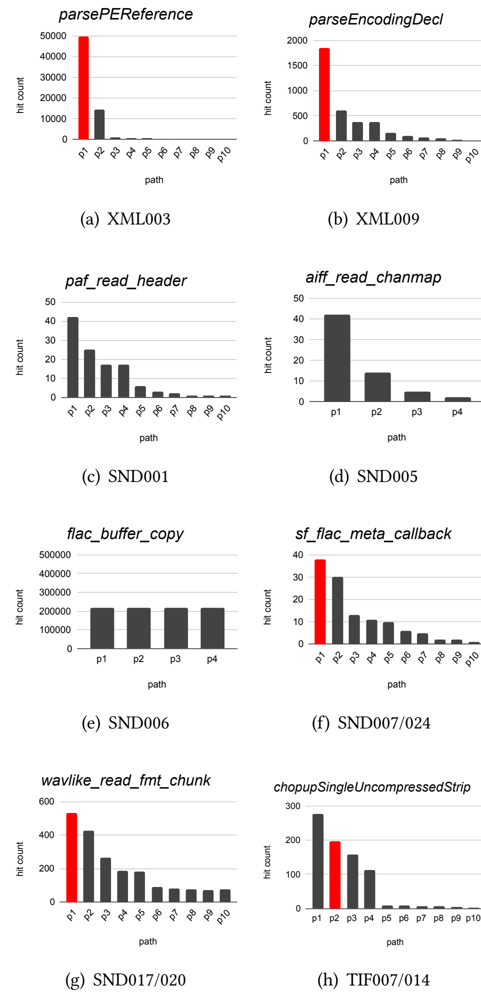
We analyzed 25 CVEs on Magma and most of them could be discovered by AFL++ within 24 hours. We conducted every experiment 10 times. For programs that the vulnerabilities were discovered within 2 hours, the time budget was 2 hours. And for vulnerabilities that take longer than 2 hours, the time budget was set to 24 hours. All experiments are conducted on a computer running Ubuntu 20.04 LTS with an i7-10700 CPU and 64GB memory. The version of Clang/LLVM is 14.0.1.

5.1 Dynamic Introspection Subsystem

In this section, we evaluate the introspection subsystem. We want to know whether introspection with path profiling can actually provide more insights for users.

As shown in Figure 6, we list the path frequency distribution of some vulnerable functions where the function *magma_log* is instrumented. The distribution has a typical long-tailed shape. For most functions, there is a huge imbalance in path frequency. Less than 20% of the paths accounted for more than 80% of the hits. During real-time monitoring, this imbalance grows as the number of seeds increases. After a while, the seeds in the queue stop increasing and the frequency distribution stabilize.

We decode and analyze the top 2 most high-frequency path IDs and find that the execution of many high-frequency paths prevents vulnerability triggering. We highlight these paths in red. These paths usually go through guard expressions that cause the function to exit directly, such as checking the front characters, or going to other cases in the switch statement and then breaking directly.



(a) XML003 (b) XML009 (c) SND001 (d) SND005 (e) SND006 (f) SND007/024 (g) SND017/020 (h) TIF007/014

Figure 6: Path frequency of the vulnerability function. Blocking paths are highlighted in red.

Listing 2: Code Snippet for Illustrating Path Frequency

```
const xmlChar *
xmlParseEncodingDecl(xmlParserCtxtPtr ctxt) {
    xmlChar *encoding = NULL;
    if (CMP8(CUR_PTR,
        'e','n','c','o','d','i','n','g')) {
            // ....
            // MAGMA(XML009)
    }
    return ;
}
```

We take XML009 as an example, and the related code is shown in code Listing 2. It is a function that parses an encoding declaration in XML. The distribution of path frequency is shown in Figure 6(b).

By decoding the top 2 most high-frequency path IDs, we find that the first path is a blocker. Nearly 80% of the hits are rejected at the first *if* statement, which is used to check the "encoding" string, and the function returns directly. The *encoding* string check is not a branch blocker in line-level fuzz-introspector because there are still some test cases that satisfy it. We can use dynamic introspection to captures well that fuzzing consumes a lot of computation resources here, which doesn't help with the testing of the *xmlParseEncodingDecl* function.

Naturally, we want to evaluate the effectiveness of dynamic direction subsystem and to see if it really can help fuzzing to focus on paths that pass the encoding branch to trigger XML009 faster.

Summary: The introspection subsystem does provide us with more insight on fuzzing. With path decoding and frequency analysis, we identified paths that hinder the detection of vulnerabilities. Fuzzing wasted too much time on these paths.

5.2 Dynamic Direction Subsystem

In this section, we evaluate the dynamic direction subsystem. We will introduce three different flagging strategies and demonstrate how DDGF helps users identify fuzzing bottlenecks through three case studies. Based on different vulnerability context and introspection results, we add flags for some paths to accelerate vulnerability reproduction. We compare with baseline AFL++(v4.07) to evaluate whether the direction subsystem in DDGF actually works, whether it actually directs fuzzer to focus on flagged paths and trigger vulnerabilities faster.

Flagging Strategy: In the above example of XML009, according to our assumptions, if we prioritize the seeds that pass the encoding check, i.e., the input file contains the *"encoding"* string, then the probability that mutated test cases passing it will increase significantly. This would increase the probability of triggering the vulnerability, as more test cases would be directed to near the vulnerability. According to the introspection experiments and analysis above, we only need to check whether the top 1-2 highest frequency paths in the vulnerability function contribute to the triggering or not by introspection, because less than 20% of the paths accounted for more than 80% of the hits. If they prevent vulnerability from being triggered, then we exclude them and add flags for all the remaining paths. If they are not blocking, we add flags for all the paths. We set them as the first and default flagging strategy. However, we found that this flagging strategy is ineffective for some vulnerabilities that take a long time to trigger. For example, the vulnerable function in TIF008 was still not explored after 15 hours. If we only flag the vulnerable function, fuzzing will not enter into the directed mode for a long time. In this case, we can trace back to the relevant call sites through dynamic introspection and flagging the upstream functions. We summarized three main flagging strategies that we use in experiments. These different strategies will be discussed through three case studies in the next section.

Results: The experiment results are shown in the Table 1. DDGF shows amazing ability to reproduce these bugs faster. For *libsndfile*, 5 of 7 vulnerabilities are reproduced in less than 20 seconds, which is up to more than 400 times faster compared to AFL++. For vulnerabilities that take the longer to trigger, we also get good results. For TIF008/SSL020/PDF003/010, the speedup is about 10x. Especially

Table 1: Average reproduction time of 10 rounds for each target in the Magma benchmark compared with AFL++. #Flags indicates the number of paths flagged in the experiment, and Strategy indicates the specific type of flagging strategy used.

Bug ID	AFL++	DDGF	Speedup	#Flags	Strategy
SND001	7.41m	4s	>100x	13	1
SND005	15.22m	2s	>400x	4	
SND006	39.30m	16s	>140x	4	
SND007	20.01m	13s	>90x	16	
SND017	13.48m	2.21m	6x	62	
SND020	30.25m	3.19m	10x	62	
SND024	14.39m	6s	>140x	10	
TIF012	4.39m	0.98m	4.5x	1	2
TIF014	4.29m	2.11m	2x	11	1
TIF014-cp	34.24m	6.89m	5x	11	
XML003	12.71m	7.15m	1.7x	22	
XML009	11.8m	2.28m	5x	22	
PNG007	1.62h	0.49h	3.3x	2	2
TIF002	13.39h	4.35h	3.0x	1	
TIF008	19.88h	1.67h	11.9x	1	
SQL002	12.4m	10.6m	1.1x	3	
SQL014	2.07h	0.73h	2.8x	21	1
SQL018	1.31h	0.76h	1.7x	8	2
SQL020	10.0h	2.0h	5x	3	3
SSL020	17.6h	1.86h	9.5x	10	2
PDF003	1.86h	11.5m	9.7x	6	
PDF010	2.13h	2.29m	55x	36	1
PDF019	T.O	16.99h	>1.4x	110	
PDF018	19.67h	42s	>1600x	1	2
PDF021	T.O	T.O	-	6	

for PDF018, our flagging strategy achieved great success, with an average reproduction time of under 1 minute and a speedup of over 1600 times. It is worth noting that we just add one flag. **To our knowledge, it is much faster than all known distance-based directed fuzzers for most bugs**.

Why is DDGF so fast ? The key to our directing approach is the seed selection. Through path profiling and flagging, the user establishes an interaction with the seed selection algorithm in fuzzing. We can push fuzzing to prioritize seeds with the flag, which has a huge impact on the fuzzing direction.

Fairness Analysis: We only need to check introspection once to help us find the paths to be flagged, because the distribution of path frequency is basically stable under the same corpus. For example in XML009, the frequency of paths that fail the "encoding" check is always the highest. Therefore, for each experiment, our flags are determined and reusable. These flags are like the target line numbers in DGFs such as AFLGo, here we target these flagged paths. Moreover, our flags are set before fuzzing starts, just as AFLGo instrument the distance into basic blocks. So our experiments and comparison are fair.

5.3 Flagging Strategies

In this section, we will illustrate the role that humans play in DDGF with three case studies which represent three different flagging strategies. Through these cases, we explain how users can use DDGF to analyze fuzzing bottlenecks and accelerate vulnerability discovery.

Strategy 1: Exclude the Blocking Paths. The first kind of flagging strategy is to exclude high-frequency blocking paths. If there are large number of seeds enter paths which are rejected by the function, we can directly exclude these high-frequency paths and flag the remaining paths within the vulnerable function. This is our default flagging strategy and works well in most cases.

Listing 3: Code Snippet for Flagging Stragety 1

```
1  sf_flac_meta_callback () {
2      switch (metadata->type){
3          case FLAC_METADATA_TYPE_STREAM_INFO:
4              ...
5              // MAGMA_LOG(SND007)
6              // MAGMA_LOG(SND024)
7              break;
8          case FLAC_METADATA_TYPE_UNDEFINED:
9              break;
10         }
11 }
```

We use SND007 and SND024 as examples, the related code is shown in the Listing 3. *sf_flac_meta_callback* is a callback function that is called when decoding metadata types. The switch statement in this function checks the metadata block type *metadata->type* of the FLAC audio, and only seeds with the type of *STREAM_INFO* is likely to trigger both vulnerabilities. We can decode and see in Figure 6(f) that the highest frequency path is the type of *UNDEFINED*, highlighted in red. This means that a large number of seeds in the queue are *UNDEFINED* type, which is rejected by this function. If we can avoid mutating these seeds with *UNDEFINED* type as much as possible, the probability of triggering the vulnerability will be much higher. So we excluded this blocking path, and add flags for all the remaining paths within the function. We reproduced both vulnerabilities in less than 20 seconds. It proves that our flagging strategy and seed selection method work well.

Strategy 2: Identify the Key Control-Flow. The second kind of flagging strategy is to identify the key control-flow. In DDGF, the path frequency distribution of the vulnerability function is an important indicator. It helps us quickly identify the blocking paths. We can simply exclude them to accelerate vulnerability detection, filtering out better seeds that are more conducive to triggering the vulnerability. However, for some cases, the vulnerability function has not been reached for a long time, which makes the first flagging strategy ineffective. We take TIF008 as an example to illustrate it. The code Listing 4 is the functions related to this vulnerability.

AFL++ took an average of 20 hours to discover this vulnerability, with 6 instances exceeding 24 hours. So, what exactly hinders the triggering? Through introspection, We found that in the first 15 hours of testing, the vulnerability function *NeXTDecode* was not tested at all. The path hit count is 0 , meaning that none of the more

Listing 4: Code Snippet for Flagging Stragety 2

```
1  static int NeXTDecode(TIFF* tif, ..) {
2      switch (n) {
3          case 1: .. break
4          case 2: .. break
5              ...
6          default: // MAGMA_LOG(TIF008)
7      }
8  }
9
10 int TIFFInitNeXT(TIFF* tif , int scheme) {
11     (void) scheme;
12     tif ->tif_predecode = NeXTPreDecode;
13     tif ->tif_decoderow = NeXTDecode;
14     tif -> tif_decodestrip = NeXTDecode;
15     tif -> tif_decodetile = NeXTDecode;
16     return (1) ;
17 }
```

than 6,000 seeds could reach this function. It also means that the strategy of only flagging the vulnerable function is not feasible in such cases. Furthermore, the fuzzing was already in the coverage plateau, and the discovery of new seeds is becoming increasingly difficult. If it is so difficult to reach the vulnerable function, how can the fuzzing satisfy the stricter data-flow conditions to trigger it? Naturally, We want to know why function *NeXTDecode* was not called.

We found the potential call site of the function *NeXTDecode*. In the function *TIFFInitNeXT*, the *NeXTDecode* function pointer is assigned to *tif_decode* for initialization. What surprised us is that this function was only hit 6 times. This means that , after 15 hours, out of the 6,000 seeds, at most only 6 seeds did the initialization of the vulnerable function pointer. If the initialization function was not even reached, the function *NeXTDecode* could not be called at all. The initialization function is a necessary condition for triggering the vulnerability. We gained the key insight and discovered the reason for the low efficiency of fuzzing. After reaching the coverage plateau, the probability of these seeds being randomly selected is only about 6/6000, which greatly hinders the triggering of the vulnerability in *NeXTDecode*. Therefore, we believe that once fuzzing explores this initialization function, the seeds should be given the highest priority.

We added a flag for the key initialization path in function *TIFFInitNeXT* and then restarted fuzzing. To ensure fairness, we have to still use the original corpus, rather than the 6,000 seeds. In 10 rounds of fuzzing, we discovered this vulnerability in an average of only 1.67 hours. With DDGF, we provided key insights to fuzzing in the form of path flagging. Just a key flag reduced the triggering time by more than 10 times. The combination of human and fuzzing has greatly enhanced the capabilities of full-automated fuzzing.

Strategy 3: Identify the Key Data-Flow. The third kind of flagging strategy is to identify the key data-flow. For some vulnerabilities, their triggering conditions may be very strict, such as the initial allocation of some variables, the assignment of certain function pointers, etc. In such cases, the above two flagging strategies will both fail because the vulnerability function (or its parent function) may be reached a large number of times during the

fuzzing process, but due to these constraints, the vulnerability is difficult to trigger. We take SQL020 as an example. The code Listing 5 is a list of functions related to this vulnerability.

Listing 5: Code Snippet for Flagging Stragety 3

```
1  static ExprList *exprListAppendList (...) {
2      // MAGMA_LOG(SQL020);
3  }
4
5  int sqlite3WindowRewrite(Select *p, ...) {
6      // pWin use, data-flow constraint
7      if (p -> pWin && ...) {
8          pSort = exprListAppendList (...) ;
9      }
10 }
11
12 Window* sqlite3WindowAlloc(Parse *pParse, ...) {
13     // pWin def
14     pWin = (Window*)sqlite3DbMallocZero(pParse->db, sizeof(
           Window));
15 }
16
17 YYACTIONTYPE yy_reduce( ..) {
18     switch(yyruleno) {
19         case 315:
20         case 316:
21         case 317: sqlite3WindowAlloc(pParse, ...)
22          ...
23     }
24 }
```

AFL++ took an average of 10 hours to discover this vulnerability. However, when adopting the flagging strategy which excludes high-frequency paths in the vulnerable function or its caller function, the fuzzer also took almost the same amount of time as AFL++. It indicates that in this case, the root cause preventing the fuzzer from triggering the vulnerability is not the high-frequency blocking paths.

The vulnerability point is in the *exprListAppendList* function, and *sqlite3WindowRewrite* is the only parent function that calls it. By analyzing the path frequency distribution diagram provided by our introspection subsystem during the testing, we found that many test cases that have not triggered the vulnerability for a long time have certain characteristics in common: 1. no paths were executed in *exprListAppendList* and 2. its parent function *sqlite3WindowRewrite* was executed more than 50,000 times, but no paths entered the if statement. This implies that the vast majority of seeds cannot meet the conditions of the if statement. By analyzing the path profiling statistics results, we discovered that they were rejected because they did not meet the fisrt condition *p -> pWin*, meaning *pWin* was not properly initialized. We found the place where *pWin* is initialized and allocated in the *sqlite3WindowAlloc* function, and identified the location where this function is called in a switch statement in *yy_reduce function*. Therefore, we believe that these are the critical paths to trigger the vulnerability.

Then we added flags to three paths in *yy_reduce* function (corresponding to three case statements shown above) and restarted fuzzing test with the original corpus provided by Magma. In 10 rounds of fuzzing, the average time to reproduce the vulnerability was reduced to 2 hours, which is 5 times faster compared to AFL++ and the default flagging strategy. Obviously, when the default flagging strategy doesn't work, the human-specific ability to understand and analyze program semantics is crucial for fuzzer.

> Summary: Based on the insights from dynamic introspection, we are able to direct fuzzing to reproduce vulnerabilities faster with different flagging strategies. We achieved amazing improvements on some projects. This demonstrates the effectiveness of DDGF.

5.4 Unknown Vulnerability Detection

We tested the real programs to evaluate the effectiveness of DDGF in discovering unknown vulnerabilities. As shown in the Table 2, we found 4 new vulnerabilities, 2 of which were assigned CVE IDs. These programs are heavily tested in OSS-Fuzz, but the low-frequency regions remains to be low-frequency during daily fuzzing.

Table 2: Unknown Vulnerability Detection Results

Program	Version	Type	CVE Status
pdftoppm	v21.10	heap overflow	req. & assigned
		assertion failure	req.
ImageMagick	f71ca65	assertion failure	req.
		heap overflow	req. & assigned

With dynamic introspection, we observed which parts of the program were not fully tested and should be directed. Our primary strategy is to flag the low-frequency part, excluding the blocking paths, and use a combination of strategy 2 and 3 as we mentioned above to identify the key control-flow and data-flow. It makes the flagged parts to be tested more intensively. We expect to conduct larger-scale experiments in the future to discover more vulnerabilities.

5.5 Performance Overhead

Table 3: Comparison of Runtime Overhead between DDGF and AFL++.

Program	AFL++	DDGF	Overhead
sndfile_fuzzer	6.52M	6.11M	6%
tiff_read_rgb_fuzzer	101M	89.9M	11%
sqlite3_fuzzer	41.2M	37.5M	9%
xml_read_memory_fuzzer	22.3M	17.6M	21%
pdf_fuzzer	209k	173k	19%
libpng_read_fuzzer	312M	281M	10%

We evaluate the performance overhead of DDGF. If the performance overhead is too high, it will seriously affect the speed of fuzzing. We test each program in DDGF for 2 hours and compare the number of executions of test cases with AFL++. The flags added by

the user will affect the direction of fuzzing and lead to different mutation strategies. So we compare with AFL++ in DDGF un-directed mode. The results are shown in the Table 3.

We can see that the average performance overhead of DDGF is only 13%, because we only do path profiling for the seeds. It proves that our two-files fuzzing strategy works well.

5.6 Discussion

Compatible with Distance-Based DGF. The instrumentation of path profiling are target-independent. It encodes the different paths with integers. The flexibility of this encoding makes it also compatible with distance-based DGFs such as AFLGo. Traditional DGFs instrument constant distances into each basic block, and we cannot change the target after compilation. However, we can transform hard-code distance into variables by using path ID as an index. We can compute the distance for different targets offline, and feed them to the fuzzer as path-level distance with ID. This makes our DDGF compatible with distance-based directing methods.

Directing methods can be designed in any stage of the fuzzing loop. The core of our directing method is based on seed selection, while the popular distance-based method is based on energy scheduling, which still wastes a lot of time on non-optimal seeds, and relies on evolutionary algorithms to get close to the target slowly and smoothly. Our approach prioritizes the seeds that go through the flagged paths. It is able to approach the vulnerability target rapidly and test intensively against it. In addition, DDGF provides users with the chance to change the runtime strategy of fuzzing by adding flags in real time. In the future, we will continue to explore more types of interaction based on DDGF.

6 Related Works

6.1 Coverage Metrics

Coverage feedback[27, 34, 39, 46, 46] is the most critical part of greybox fuzzing. Test cases that explore new coverage are considered good seeds and added to the queue for further mutation. Popular fuzzers such as AFL++[25] , Angora[22] use edge coverage by default, while other tools such as LibFuzzer and Honggfuzz also provide function and basic block level coverage for users. Ankou[37] points out that current edge coverage feedback can easily get stuck in local optima because it cannot distinguish edge combination order. They propose a distance-based fitness function to differentiate execution coverage. AFLNet[40] and SGFuzz [13] propose that code coverage can not consider state transitions for stateful programs like network protocols. They provide state coverage feedback using server return values and edges in the state transition graph. Heng Yin et al. [43] formally define the sensitivity of coverage feedback metrics and analyze the impact of different fitness functions systematically based on vulnerability detection results. However, as they said, the choice of coverage feedback strategy is a trade-off, and there is no absolute advantage or disadvantage for different metrics.

The improvements in coverage metric selection and related strategies are directly reflected in the seed queue. However, there are still more intermediate states not exposed to users. We can only judge whether new coverage is discovered or fuzzing gets stuck by the console dashboard in AFL, which hinders us from evaluating the fuzzing progress. At the same time, the fuzzer also cannot sense well which structure in the program hinders testing. This inspires us to do the research to facility the communication between users and fuzzers. Our work exposes the path-level status and shows the great potential for human-in-the-loop for fuzzing.

6.2 Human-in-the-Loop for Fuzzing

In recent years, a large number of research have tried to improve different stages of fuzzing. However, code coverage has always been a key issue for testing. A critical problem is that fuzzers can not understand the high-level semantics for different programs and make dynamic adjustments according to the testing status. After the CGC[3, 42] competition, the UCSB team[8] proposed to improve the existing fully automated vulnerability detection paradigm and built the prototype HaCRS[41]. When the machine gets stuck, humans can participate by feeding new seeds, and providing key tokens to help the fuzzer adjust in time.

IJON[12] presents the code annotation method for deep state fuzzing. Users can customize the coverage feedback and guide fuzzing to explore the state space of target regions by adding annotations to the source code. It helps to complete challenges that were previously difficult to solve, such as mazes, Mario games, and other state-sensitive vulnerabilities. JMPscare[36] explicitly points out that existing fuzzing lacks introspection. It proposes an offline analysis of the seed queue, displaying obstacles and current coverage boundaries. It is implemented as a plugin for Binary Ninja. Security auditors can directly flip the blocked conditional branches through the UI to force the fuzzer to bypass obstacles. Learn2Fix[20] applies interactive fuzzing for data-driven automated vulnerability fixing. It asks users to mark whether errors occur during testing, which is used in active learning to train test oracles. HM-Fuzz[19] presents the concept of compartment analysis, combining the dominator tree and dynamic analysis to divide the control flow graph into different compartments. It helps users to locate areas that are more likely to expand coverage.

7 Conclusion

In this paper, we propose a new directed fuzzing system called DDGF to facilitate the collaboration between humans and fuzzers. We present the concept of dynamic directed fuzzing for the first time, which allows users to dynamically direct fuzzer at runtime. DDGF consists of two subsystems, dynamic introspection system and dynamic direction system, connected by the Ball-Larus path profiling algorithm. On the one hand, path profiling reveals the huge imbalance in path frequencies, which gives the user more insights during the fuzzing process. On the other hand, we can select paths to influence the seed selection stage in the fuzzing loop, which leads fuzzer to focus on these paths and accelerate vulnerability discovery. We implemented DDGF based on AFL++ and fuzz-introspector. Experiments on Magma and real programs show that DDGF can significantly improve the efficiency and effectiveness of fuzzing.

Acknowledgments

The authors thank the anonymous reviewers for their constructive comments. Yuanyuan Zhang is the corresponding author.

References

[1] 2023. AFL. https://lcamtuf.coredump.cx/afl. Accessed: 2023-12-01.

[2] 2023. D3.js. https://d3js.org/. Accessed: 2023-12-01.

[3] 2023. DARPA CGC Project. https://www.darpa.mil/program/cyber-grand-challenge. Accessed: 2023-12-01.

[4] 2023. DARPA CHESS Project. https://www.darpa.mil/program/computers-and-humans-exploring-software-security. Accessed: 2023-12-01.

[5] 2023. Fuzz-Introspector. https://github.com/ossf/fuzz-introspector. Accessed: 2023-12-01.

[6] 2023. LLVM-PGO. https://clang.llvm.org/docs/UsersManual.html#profile-guided-optimization. Accessed: 2023-12-01.

[7] 2023. MATE by Galois Inc. https://galois.com/project/mate/. Accessed: 2023-12-01.

[8] 2023. Mechanical Phish. https://github.com/mechaphish. Accessed: 2023-12-01.

[9] 2023. Memory Mapping in Boost. https://www.boost.org/doc/libs/1_85_0/doc/html/interprocess/sharedmemorybetweenprocesses.html. Accessed: 2023-12-01.

[10] 2023. OSS-Fuzz. https://github.com/google/oss-fuzz. Accessed: 2023-12-01.

[11] 2023. OSS-Fuzz improvements in Google. https://security.googleblog.com/2023/02/taking-next-step-oss-fuzz-in-2023.html. Accessed: 2023-12-01.

[12] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. 2020. Ijon: Exploring deep state spaces via fuzzing. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1597–1612. https://doi.org/10.1109/SP40000.2020.00117

[13] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. 2022. Stateful greybox fuzzing. In *31st USENIX Security Symposium (Usenix Security 22)*. 3255–3272.

[14] Thomas Ball and James R Larus. 1996. Efficient path profiling. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*. IEEE, 46–57. https://doi.org/10.1109/MICRO.1996.566449

[15] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. 2021. Fuzzing: Challenges and Reflections. *IEEE Softw.* 38, 3 (2021), 79–86. https://doi.org/10.1109/MS.2020.3016773

[16] Marcel Böhme and Brandon Falk. 2020. Fuzzing: On the exponential cost of vulnerability discovery. In *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 713–724.

[17] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2329–2344. https://doi.org/10.1145/3133956.3134020

[18] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1032–1043. https://doi.org/10.1109/TSE.2017.2785841

[19] Joshua Bundt, Andrew Fasano, Brendan Dolan-Gavitt, William Robertson, and Tim Leek. 2022. Homo in Machina: Improving Fuzz Testing Coverage via Compartment Analysis. https://doi.org/10.48550/ARXIV.2212.11162

[20] Marcel Böhme, Charaka Geethal, and Van-Thuan Pham. 2020. Human-In-The-Loop Automatic Program Repair. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 274–285. https://doi.org/10.1109/ICST46399.2020.00036

[21] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2095–2108. https://doi.org/10.1145/3243734.3243849

[22] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 711–725. https://doi.org/10.1109/SP.2018.00046

[23] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. 2020. Savior: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1580–1596. https://doi.org/10.1109/SP40000.2020.00002

[24] Zhengjie Du, Yuekang Li, Yang Liu, and Bing Mao. 2022. WindRanger: a directed greybox fuzzer driven by deviation basic blocks. In *Proceedings of the 44th International Conference on Software Engineering*. 2440–2451. https://doi.org/10.1145/3510003.3510197

[25] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies*. 10–10.

[26] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. 2022. LibAFL: A Framework to Build Modular and Reusable Fuzzers. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1051–1065. https://doi.org/10.1145/3548606.3560602

[27] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 679–696. https://doi.org/10.1109/SP.2018.00040

[28] Wentao Gao, Van-Thuan Pham, Dongge Liu, Oliver Chang, Toby Murray, and Benjamin IP Rubinstein. 2023. Beyond the Coverage Plateau: A Comprehensive Study of Fuzz Blockers (Registered Report). In *Proceedings of the 2nd International Fuzzing Workshop*. 47–55.

[29] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A ground-truth fuzzing benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 3 (2020), 1–29. https://doi.org/10.1145/3543516.3456276

[30] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2022. Beacon: Directed grey-box fuzzing with provable path pruning. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 36–50. https://doi.org/10.1109/SP46214.2022.9833751

[31] Heqing Huang, Peisen Yao, CHIU Hung-Chun, Yiyuan Guo, and Charles Zhang. 2023. Titan: Efficient Multi-target Directed Greybox Fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 59–59.

[32] Tae Eun Kim, Jaeseung Choi, Kihong Heo, and Sang Kil Cha. 2023. {DAFL}: Directed Grey-box Fuzzing guided by Data Dependency. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4931–4948.

[33] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 2123–2138. https://doi.org/10.1145/3243734.3243804

[34] Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*. 475–485. https://doi.org/10.1145/3238147.3238176

[35] Changhua Luo, Wei Meng, and Penghui Li. 2023. Selectfuzz: Efficient directed fuzzing with selective path exploration. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2693–2707.

[36] Dominik Maier and Lukas Seidel. 2021. Jmpscare: Introspection for binary-only fuzzing. In *Workshop on Binary Analysis Research (BAR)*, Vol. 2021. 21. https://doi.org/10.14722/bar.2021.23003

[37] Valentin JM Manès, Soomin Kim, and Sang Kil Cha. 2020. Ankou: Guiding grey-box fuzzing towards combinatorial difference. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1024–1036. https://doi.org/10.1145/3377811.3380421

[38] Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 47, 11 (2021), 2312–2331. https://doi.org/10.1109/TSE.2019.2946563

[39] Stefan Nagy and Matthew Hicks. 2019. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 787–802. https://doi.org/10.1109/SP.2019.00069

[40] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNet: a greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 460–465. https://doi.org/10.1109/ICST46399.2020.00062

[41] Yan Shoshitaishvili, Michael Weissbacher, Lukas Dresel, Christopher Salls, Ruoyu Wang, Christopher Kruegel, and Giovanni Vigna. 2017. Rise of the HaCRS: Augmenting autonomous cyber reasoning systems with human assistance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 347–362. https://doi.org/10.1145/3133956.3134105

[42] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution.. In *NDSS*, Vol. 16. 1–16. https://doi.org/10.14722/ndss.2016.23368

[43] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. 2019. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. 1–15.

[44] Qian Yan, Huayang Cao, Shuaibing Lu, and Minhuan Huang. 2023. InFuzz: An Interactive Tool for Enhancing Efficiency in Fuzzing through Visual Bottleneck Analysis (Registered Report). In *Proceedings of the 2nd International Fuzzing Workshop*. 56–61.

[45] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 745–761. https://doi.org/10.5555/3277203.3277260

[46] Chijin Zhou, Mingzhe Wang, Jie Liang, Zhe Liu, and Yu Jiang. 2020. Zeror: Speed up fuzzing with coverage-sensitive tracing and scheduling. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 858–870. https://doi.org/10.1145/3324884.3416572

[47] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)* 54, 11s (2022), 1–36. https://doi.org/10.1145/3512345

Received 2024-04-12; accepted 2024-07-03