ZigZagFuzz: Interleaved Fuzzing of Program Options and Files

AHCHEONG LEE, KAIST, South Korea
YOUNGSEOK CHOI, KAIST, South Korea
SHIN HONG, Chungbuk National University, South Korea
YUNHO KIM, Hanyang University, South Korea
KYUTAE CHO, LIG Nex1 AI R&D, South Korea
MOONZOO KIM*, KAIST, South Korea

Command-line options (e.g., `-l`, `-F`, `-R` for `ls`) given to a command-line program can significantly alternate the behaviors of the program. Thus, fuzzing not only file input but also program options can improve test coverage and bug detection. In this paper, we propose ZigZagFuzz which achieves higher test coverage and detects more bugs than the state-of-the-art fuzzers by separately mutating program options and file inputs in an iterative/interleaving manner. ZigZagFuzz applies the following three core ideas. First, to utilize different characteristics of the program option domain and the file input domain, ZigZagFuzz separates phases of mutating program options from ones of mutating file inputs and performs two distinct mutation strategies on the two different domains. Second, to reach deep segments of a target program that are accessed through an interleaving sequence of program option checks and file inputs checks, ZigZagFuzz continuously interleaves phases of mutating program options with phases of mutating file inputs. Finally, to improve fuzzing performance further, ZigZagFuzz periodically shrinks input corpus by removing similar test inputs based on their function coverage.

The experiment results on the 20 real-world programs show that ZigZagFuzz improves test coverage and detects 1.9 to 10.6 times more bugs than the state-of-the-art fuzzers that mutate program options such as AFL++-argv, AFL++-all, Eclipser, CarpetFuzz, ConfigFuzz, and POWER. We have reported the new bugs detected by ZigZagFuzz, and the original developers confirmed our bug reports.

Additional Key Words and Phrases: Automated test generation, fuzzing, command-line program options, bug detection, dynamic analysis

ACM Reference Format:
Ahcheong Lee, Youngseok Choi, Shin Hong, Yunho Kim, Kyutae Cho, and Moonzoo Kim. 2024. ZigZagFuzz: Interleaved Fuzzing of Program Options and Files . 1, 1 (September 2024), 31 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Most programs can be configured for various purposes and the configuration of a program can largely affect the behaviors of it. For example, programs with command-line interface have dozens of command-line options to alternate the operations of programs (e.g., `ls` has more than 50 program options including `-a`, `-F`, `-l`, `-n`, and

*Corresponding author

Authors' addresses: Ahcheong Lee, ahcheong.lee@kaist.ac.kr, KAIST, Daejeon, South Korea; Youngseok Choi, youngseok.choi@kaist.ac.kr, KAIST, Daejeon, South Korea; Shin Hong, hongshin@gmail.com, Chungbuk National University, Cheongju, South Korea; Yunho Kim, yunhokim@hanyang.ac.kr, Hanyang University, Seoul, South Korea; Kyutae Cho, kyutae.cho2@lignex1.com, LIG Nex1 AI R&D, Seoul, South Korea; Moonzoo Kim, moonzoo.kim@gmail.com, KAIST, Daejeon, South Korea.

XXXX-XXXX/2024/9-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

`-R` [1]). In other words, program options play a crucial role in determining a target program's execution paths. Thus, when we fuzz a program with command-line interface, our bug detection results can vary significantly depending on which program options are applied during fuzzing. For example, 36 functions of `xmllint` (an xml file parsing tool) in `libxml2` cannot be reached at all unless one of `–xinclude`, `–noxincludenode`, and `–nofixup-base-uris` options is given.

Although a *program option configuration* (i.e., a list of program options given to a target program such as "`-a -l -R`" for `ls`) is important for fuzzing, most fuzzing papers utilized only a single program option configuration in their fuzzing experiments. According to the survey of the recently published 102 fuzzing papers (see Section 2.1 for the details), 73.5% of the fuzzing papers did not provide information on the program option configurations in the papers. Thus, there exists large room to improve fuzzing effectiveness by systematically utilizing various program option configurations.

In this paper, we propose a novel fuzzing technique ZigZagFuzz that detects more bugs than the state-of-the-art fuzzers by separately fuzzing file input (FI) and program option input (POI) in an iterative/interleaving manner. Three core ideas of ZigZagFuzz are as follows (the motivations for the ideas are illustrated through a concrete example in Section 2):

- *Different mutation strategies for different input domains (i.e., POIs and FIs)*:
  To utilize different characteristics of the POI domain and the FI domain, ZigZagFuzz separates phases of mutating POIs from ones of mutating FIs. In other words, in contrast to the fuzzers that mutate only file inputs of a target program, ZigZagFuzz considers that a target program has *two different input domains* to explore (i.e., POI domain and FI domain), and it applies *two distinct mutation strategies* to them for high bug detection ability (see Section 3.4 and Section 3.5).

- *Iterative/interleaving phases of mutating POI with ones of mutating FI:*
  To penetrate the deeper segments of a target program, which are accessed via a methodical sequence of alternating checks between Program Options Inputs (POI) and File Inputs (FI), ZigZagFuzz employs a strategy of seamlessly interleaving two distinct mutation phases. This iterative and interleaved approach ensures thorough exploration and testing of the program's functionalities, optimizing the fuzzing process for efficacy and depth. We discuss the importance of this nature with a motivating example in Section 2.

- *Domain-wise corpus shrinking by reducing redundant POIs and FIs*
  To enhance fuzzing performance further, ZigZagFuzz periodically reduces redundant POI corpus and FI corpus separately. Unlike conventional corpus shrinking methods, the proposed approach independently reduces POIs and FIs, and then retains only a small set of seed inputs containing unique POIs and FIs according to their function coverage achievements (see Section 3.6).

The experiment results on the 20 real-world programs show that ZigZagFuzz improves test coverage and detects significantly more (1.9 to 10.6 times more) bugs than the state-of-the-art fuzzers that fuzz program options such as AFL++-argv, ConfigFuzz, Eclipser, CarpetFuzz, and POWER (see Section 5.1). Furthermore, we have reported the new bugs detected by ZigZagFuzz and the original developers confirmed most of our bug reports.

The main contributions of this paper are as follows:

- ZigZagFuzz is the first fuzzer that can detect significantly more bugs than the state-of-the-art fuzzers by separately mutating program options and file inputs in an iterative/interleaving manner (see Section 3).

- We have performed a series of experiments where we have empirically evaluated ZigZagFuzz and the other cutting-edge fuzzers that mutate program options (i.e., AFL++-argv, Eclipser, CarpetFuzz, ConfigFuzz, and

[1] See http://linuxcommand.org/lc3_man_pages/ls1.html

POWER) and demonstrated that ZigZagFuzz detects significantly more (1.9 to 10.6 times more) unique bugs than the cutting-edge fuzzers (Section 5).

- We have reported 61 new bugs detected by ZigZagFuzz to the original developers of the target subject programs to improve the quality of the open source subject programs. [2]

The remaining sections of this paper are as follows. Section 2 shows a motivating example for the design of ZigZagFuzz. Section 3 explains the details of ZigZagFuzz. Section 4 describes the experiment setup and Section 5 shows the experiment results and answers our research questions. Section 6 discusses our survey of POI use in fuzzing research and the benefits of ZigZagFuzz through concrete case studies. Section 7 compares ZigZagFuzz with related work. Section 8 concludes this paper and proposes future work.

2 MOTIVATION

2.1 Survey of Program Option Input (POI) Use in Fuzzing Research

Although POI can largely affect program behaviors, fuzzing researchers do not pay enough attention to fuzz POI. To find out how fuzzing papers *explicitly* utilize POI in their experiments, we have surveyed 102 fuzzing papers that (1) were published from 2015 to 2023 at top conferences and journals in software engineering and security, and (2) targeted command-line interface programs.

From the survey, we have observed that

(1) Only six papers [3, 7, 15, 38, 42, 48] directly mutate option configurations in their experiments.

(2) Only 21 papers specify the program option configurations used in their experiments (e.g., [4, 10, 22, 25, 33, 37, 45, 46]).

(3) 11 papers [1, 6, 9, 16, 23, 26, 29, 30, 41, 43, 50] do not specify the program option configurations (but implicitly expose their program option configurations through publicly available experiment data)

(4) 64 papers do not specify the program option configurations used (e.g., [8, 12, 13, 20, 24, 40, 49]).

In summary, 73.5% (=(11+64)/102) of the recently published fuzzing papers do not provide information on the program option configurations used. Moreover, most of the above papers use only one program option configuration for their experiments. We share the complete list of the surveyed papers on our paper web page (https://sites.google.com/view/zigzagfuzz).

2.2 Motivating Crash Example

Figure 1-(a) shows a buggy code example of `dwarfdump` ver.0.5.0. To trigger a use-after-free crash at Line 12 in the buggy code, a program execution must satisfy the seven branch conditions in a row (i.e., the branch conditions in Lines 2, 3, 4, 5, 7, 9, and 10). These branch conditions can be classified into Program Option Input (POI)-dependent ones (marked as Ⓟ at the end of the line) or File Input (FI)-dependent ones (marked as Ⓕ) depending on whether variables involved in the branch conditions have data-dependency on POI or FI.

In this example, `argv` and `glflags` at Line 2, `glflags.gf_1` and `glflags.gf_2` at Line 5, and `glflags.gf_eh_frame_flag` at Line 9 (colored in blue) are POI-dependent because these are data-dependent to `argv`. On the other hand, `ftype` at Lines 3 and 4 and `dobj` at Lines 7 and 10 (colored in green) are FI-dependent because their values are defined by file read operations (e.g., `open_detect_dwarf_obj` at Line 3).

[2]We reported 61 out of the 85 bugs detected by ZigZagFuzz. To reduce the original developer's burden to review many bug reports, we reported the bugs that can be replicated on the latest development version.

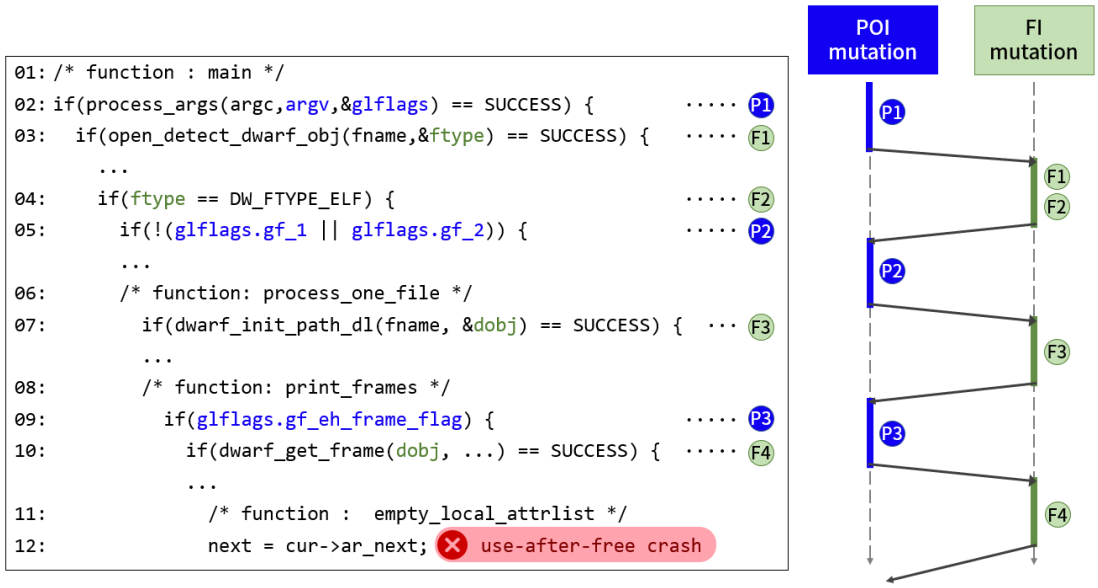

Fig. 1. (a) Simplified crashing buggy code in dwarfdump ver.0.5.0 that contains combination of POI dependent branches (marked as Ⓟ) and FI dependent branches (marked as Ⓕ). (b) An example diagram that shows interleaving behavior of ZigZagFuzz.

To trigger a crash at Line 12, a test input must satisfy the following POI-dependent and FI-dependent branch conditions in Lines 1-10 in the following order:

(1) POI-dependent branch Ⓟ1 (Line 2): The given POI should be valid (i.e., `process_args` should be able to parse the given POI). For example, each word in the POI should start with '-'; the program terminates otherwise. `process_args` sets global flag values (`glflags`) based on the given POI.

(2) FI-dependent branches Ⓕ1 and Ⓕ2 (Lines 3 and 4): The FI should contain the magic bytes to satisfy complicated checks in `open_detect_dwarf_obj` at Line 3. For example, the function has eight bytes long magic byte checks with multiple nested if/switch statements. Also, `open_detect_dwarf_obj` assigns a value to `ftype` based on the FI content. To satisfy the branch condition on `ftype` at Line 4, the FI should have proper data.

(3) POI-dependent branch Ⓟ2 (Line 5): The value of `glflags` (which is dependent on the POI) should be set properly, so that it can satisfy the branch condition at Line 5.

(4) FI-dependent branch Ⓕ3 (Line 7): The program reads the FI content and converts it to an internal data object (`dobj`). The FI content should be valid to satisfy complicated conditions in `dwarf_init_path_dl`. The internal data object represents complex debug information for an ELF object file. It is 9,664 bytes long and contains 83 different fields. During the conversion process, the function performs several sanity checks on the fields of the converted data object. For example, it checks if the number of the ELF sections written in the FI matches the actual number of the ELF sections converted into the object.

(5) POI-dependent branch Ⓟ3 (Line 9): The branch condition is dependent on `gflags.gf_eh_fram_flag` which is dependent on the POI.

(6) FI-dependent branch Ⓕ4 (Line 10): The program reads and checks the object `dobj` which is dependent on the FI.

As we have seen in the code example, the POI-dependent branches (i.e., Lines 2, 5, and 9) are interleaved with FI-dependent branches (i.e., Lines 3, 4, 7, and 10) to the crashing line (Line 12). To satisfy this interleaving sequence of the POI-dependent branches with the FI-dependent branch conditions, a fuzzer should solve several challenges described in Section 2.3.

2.3 Challenges and Solutions

The buggy code example of `dwarfdump` shown in Figure 1 illustrates why ZigZagFuzz alternates the POI and FI mutation phases to satisfy complex path conditions. The buggy code example shows three challenges of a mutation-based evolutionary fuzzer to trigger the crash at Line 12 (Section 2.2 explains the example in detail):

(1) Both POI-dependent and FI-dependent branches can be included in a path to a crash location. Thus, a fuzzer should mutate not only FI, but also POI effectively and efficiently.

(2) Since a pre-condition of a POI-dependent branch may rely on FI-dependent variables (and vice versa) like the interleaved sequence of the POI-dependent branches with the FI-dependent branches in Figure 1-(a), mutating both POI and FI at the same time may easily break the pre-condition and become ineffective.

(3) A branch condition often involves a complicated condition check which can be satisfied only after a fuzzer spends a long time generating a test input that satisfies multiple sub-conditions (e.g., to satisfy a branch condition at Line 3 in Figure 1-(a), a fuzzer should generate a test input that satisfies the complicated condition checks in `open_detect_dwarf_obj`).

Thus, a fuzzer should provide the following solutions to overcome the above challenges as shown in Figure 1-(b):

(1) A fuzzer should perform not only FI mutations, but also POI mutations to satisfy POI-dependent and FI-dependent branch conditions (see Section 3.1).

(2) A fuzzer should separate POI mutations from FI mutations. In other words, to satisfy a POI-dependent branch condition, it should first satisfy a FI-dependent pre-condition of the POI-dependent branch by mutating FI. Then, it should mutate POI without mutating FI to avoid violating the pre-condition of the POI-dependent branch (and vice versa for FI-dependent branches) (see Section 3.3).

(3) A fuzzer should assign enough time and resource budget to each POI mutation and FI mutation to generate a test input that satisfies complicated branch conditions (e.g., ones in `open_and_detect_dwarf_obj` and `dwarf_init_path_dl`) (see Section 3.3).

(4) A fuzzer should manage the seed corpus to maintain a high diversity of POIs and FIs separately, in order to prevent specific POIs or FIs from dominating the seed corpus and limiting exploration of different program execution scenarios (see Section 3.6).

To address the aforementioned challenges, ZigZagFuzz alternatively repeats POI mutations and FI mutations in an iterative/interleaving manner and periodically performs corpus shrinking, as suggested in the above solutions.

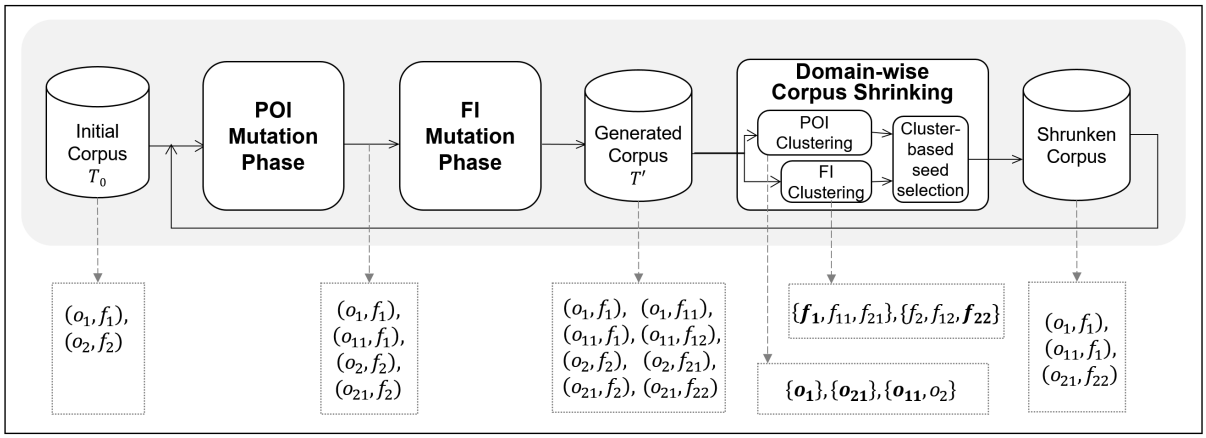

Fig. 2. Overall process of ZigZagFuzz

3 ZIGZAGFUZZ

Figure 2 shows the process of ZigZagFuzz. ZigZagFuzz considers a test input (o, f) as a pair of a *program option input* o and a *file input* f. ZigZagFuzz starts with a set of initial test inputs (e.g., (o_1, f_1), (o_2, f_2) in the Figure 2). ZigZagFuzz repeats the following three tasks as shown in Figure 2:

(1) *Phase of mutating program option input (POI)* (Section 3.4):
ZigZagFuzz mutates POIs (e.g., o_1, o_2) by applying both structural mutation [31] and random byte-level [3] mutation. Structural mutation aims to generate valid/meaningful POIs while byte-level mutation targets to generate diverse exceptional POIs.
(2) *Phase of mutating file input (FI)* (Section 3.5):
ZigZagFuzz mutates FIs through random byte-level mutation (e.g., bitflip, byteflip, arithmetic, havoc, splice, etc.) as other fuzzers like AFL++ do.
(3) *Domain-wise corpus shrinking* (Section 3.6):
To further improve fuzzing performance, ZigZagFuzz periodically decreases the redundancy in both the POI corpus and the FI corpus separately. Unlike traditional methods of corpus reduction, this approach individually shrinks POIs and FIs, subsequently preserving a compact set of test inputs that contains distinct POIs and FIs based on their functional coverage.

3.1 Overall Process of ZigZagFuzz

Algorithm 1 explains the overall process of ZigZagFuzz. First, ZigZagFuzz receives the following items to start fuzzing (see the inputs of Algorithm 1):

- A target program P.
- A set of initial test inputs T_0, each of which consists of an initial POI and an initial FI.
- A set of program option keywords (i.e., keyword dictionary) OPT_P that are semi-automatically extracted from manual pages and usage messages of a target program P (see Section 3.2).

Next, ZigZagFuzz initializes the following data (Lines 1-4 of Algorithm 1):

- T' is a set of all generated test inputs (i.e., output of fuzzing).

[3]More precisely, bit-byte-word-dword level

Algorithm 1: Overall process of ZigZagFuzz

Input: P : a test subject program
T_0 : a set of initial test inputs for P
OPT_P: a set of program option keywords of P
Output: T' : a set of all generated test inputs

```
1  T' ← ∅
2  Seeds ← ∅
3  Coverage ← ∅
4  Energy ← empty map
5  foreach (o, f) ∈ T_0 do
6  |  RunTest(P, o, f, Coverage)
7  end
8  while GlobalTimeout() do
9  |  while POI_Phase(τ_t) do
10 |  |  (o, f) ← SelectSeed(Seeds, Energy)
11 |  |  o' ← MutatePOI(o, OPT_P)
12 |  |  RunTest(P, o', f, Coverage)
13 |  end
14 |  while FI_Phase(τ_t) do
15 |  |  (o, f) ← SelectSeed(Seeds, Energy)
16 |  |  f' ← MutateFI(f)
17 |  |  RunTest(P, o, f', Coverage)
18 |  end
19 |  Seeds ← ShrinkCorpus(Seeds, Energy)
20 end
21 return T'
22 Function RunTest(P, o, f, Coverage):
23 |  Covered ← Execute(P, o, f)
24 |  if Covered ⊈ Coverage then
25 |  |  Coverage ← Coverage ∪ Covered
26 |  |  Seeds ← Seeds ∪ {(o, f)}
27 |  |  T' ← T' ∪ {(o, f)}
28 |  |  Energy(o, f) ← CalEnergy(o, f)
29 |  end
30 End Function
```

- *Seeds* is the test input corpus that will be mutated to generate diverse test inputs.
- *Coverage* contains coverage achieved by generated test inputs.
- *Energy* is a map that records assigned energy for each test input (including both FI and POI), which is the same to that of AFL++ [11]. It is used to prioritize test inputs through the power scheduling algorithm and to select test inputs at the corpus shrinking stage.

```
01: $ ./ffmpeg -help
02: ffmpeg version N-109669-g9a180f60a9
03: Hyper fast Audio and Video encoder
04: usage: ffmpeg [options] [[infile options] -i infile]... [outfile options] outfile...
05: ...
06: mpeg1video encoder AVOptions:
07:   -drop_frame_timecode  <boolean> Timecode is in drop frame format. (default false)
08:   -b_strategy           <int>     I/P/B-frames (from 0 to 2) (default 0)
09:   -b_sensitivity        <int>     Adjust sensitivity (from 1 to INT_MAX) (default 40)
```

Fig. 3. A simplified usage message example of `ffmpeg`

Then, ZigZagFuzz executes and evaluates all given initial test inputs in T_0 (Lines 5-7). When it executes each test input (o, f), it records the coverage of the execution in *Covered*. If an execution of a test input (o, f) covers a new element, ZigZagFuzz updates *Coverage* (Line 25) and adds (o, f) to *Seeds* so that it can be mutated later (Line 26). It also puts (o, f) into the output corpus T' (Line 27) and calculates the energy score of (o, f) (Line 28).

After executing and evaluating T_0, ZigZagFuzz repeats the three tasks (POI mutation phase, FI mutation phase, and domain-wise corpus shrinking) in an iterative/interleaving manner until the global timeout is reached (Lines 8-20) (see Section 3.3).

3.2 Construction of Option Keyword Dictionary

A program with a command-line interface (CLI) usually has an option (e.g., `–help`) to print usage messages and guide how to execute the program. We create an option keyword dictionary (i.e., OPT_P in Algorithm 1) by extracting option keywords from these messages. As an example, Figure 3 shows a simplified usage message of `ffmpeg`, from which we can extract option keywords by parsing each line starting with '-' (Lines 7-9). In this example, we extract `-drop_frame_timecode`, `-b_strategy`, and `-b_sensitivity` as option keywords by using a python script that creates an option keyword dictionary as follows:

(1) From the lines starting with '-' in the help (or usage) messages (e.g., "`-drop_frame_timecode`" in Figure 3), the script extracts and adds the first word of each line of the help message to the option keyword dictionary.
(2) If the first word (i.e., an option keyword) is followed by a parameter type (e.g., `<boolean>`, `<int>`), predefined values of the type are attached with the option keyword (e.g., `false` or `true` for `<boolean>` and `0`, 1, or `100` for `<int>`) in the dictionary such as "`-drop_frame_timecode false`", "`-drop_frame_timecode true`", "`-b_strategy 0`", "`-b_strategy 1`", and "`-b_strategy 100`". [4]

3.3 Iterative/Interleaving Phases of POI Mutations with FI Mutations

ZigZagFuzz repeats the POI mutation phase, the FI mutation phase, and domain-wise corpus shrinking in an iterative/interleaving manner to resolve the challenges described in Section 2.3. As described in Algorithm 1 in Section 3.1, POI mutation and FI mutation are strictly separated in the timeline (Lines 9-18).

First, ZigZagFuzz starts with the POI mutation phase (Lines 9-13) (see Section 3.4). It repeats the following tasks until a given time τ_t is reached:

- selecting a test input (based on the power scheduling algorithm (Line 10)

[4] We slightly modified the script for a program whose help message structure is different from the above example (e.g., a program which uses `INT` instead of `<int>` in its help message).

- mutating the POI of the test input (Line 11)
- evaluating the new test input with the new POI by measuring coverage of the input (Line 12).

Next, ZigZagFuzz changes the phase to the FI mutation phase (see Section 3.5). ZigZagFuzz mutates FIs similar to the POI mutation phase (Lines 14-18). After the FI mutation phase is completed, the domain-wise corpus shrinking (see Section 3.6) selects test inputs to pass to the POI mutation phase (Line 19).

Thus, POI mutation focuses on generating test inputs that satisfy a POI-dependent branch condition without breaking any pre-condition of the POI-dependent branch which may be dependent on FI (and vice versa for FI mutation).

For example, as shown in the example of Figure 1, ZigZagFuzz repeats POI mutation phases and FI mutation phases to penetrate to complex interleaved pre-conditions of deep segments of programs. Suppose that we have a test input that can reach Line 10, but the test input can not satisfy the condition Ⓕ④ in the Line 10. To trigger a crash in Line 12, a fuzzer should mutate only the FI of the test input but not the POI of the test input. This is because mutating the POI of the test input will break the pre-condition of the branch condition and a new test input obtained by mutating the POI might not even reach Line 12. (i.e. mutating the POI may break the POI-dependant pre-conditions Ⓟ①, Ⓟ②, and Ⓟ③). This is why ZigZagFuzz uses iterative and interleaved fuzzing of POI mutations and FI mutations for high bug detection effectiveness.

3.4 Program Option Input Mutation Phase

Algorithm 2 explains how it mutates a given POI o. ZigZagFuzz mutates POI in two ways: structural mutation and byte-level mutation. Structural mutation considers a POI as a list of words and mutates the list by randomly inserting, removing, or replacing a word in the list. (e.g., the structural mutation mutates “-a 100 -d” to “-a 100 -d -f” by adding one word “-f”). Byte-level mutation considers a POI (including option arguments) as a string and mutates the string randomly (e.g., the byte-level mutation mutates “-a 100 -d” to “-ab 21a -de”). ZigZagFuzz randomly applies mutations of both types to POI (See Section 5.4) with probability 50% each (Line 2). We select this probability value (i.e., 50%) based on our exploratory study.

The left part of Figure 2 shows how POI mutation phase operates. In this example, it receives an initial corpus T_0 (=$\{(o_1, f_1), (o_2, f_2)\}$) and generates two test inputs ((o_{11}, f_1) and (o_{21}, f_2)) from T_0; (o_{11}, f_1) is generated by mutating o_1 of (o_1, f_1), and(o_{21}, f_2) is generated by mutating o_2 of (o_2, f_2).

3.4.1 Structural Mutation on POI. Structural mutation considers a POI as a list of words and focuses on generating valid/meaningful POIs. The else branch of the Algorithm 2 (Lines 6-26) shows how ZigZagFuzz applies structural mutation to POI. First, ZigZagFuzz splits a given POI o into a list of option words *opt_list* (Line 7). Then, ZigZagFuzz applies a random number (e.g. between one and 32) of structural mutations on *opt_list*. There are three structural mutation operators: insertion, deletion, and replacement. The insertion operator inserts a random option keyword in a dictionary OPT_P at a random location of the list (Lines 10-14). The deletion operator removes a random word in the list (Lines 15-18). The replacement operator replaces a random word in the list with a random option keyword in OPT_P (Lines 19-23). Lastly, ZigZagFuzz concatenates the option words in the list to generate a new POI o' (Line 26).

For an example of POI "-a 100 -d -e", the three structural mutation operators work as follows:

(1) The insertion operator inserts a random word from the dictionary in a random place. One example output is “-a 100 -d **-f** -e”, which has added an option ‘-f’.
(2) The removal operator deletes a random word from the POI. One example output is “-a 100 -d”, which has removed an option ‘-e’.
(3) The replacement operator replaces a random word in the POI with another random word in the dictionary. One example output is “-a 100 **-g** -e”, which has replaced the option ‘-d’ with ‘-g’. Another example output is “-a **-e** -d -e”, which has replaced the option argument ‘100’ with ‘-e’.

Algorithm 2: Mutation of Program Option Input

Input: o : a given POI to mutate
OPT_P : command-line keywords of the subject program P
Output: o' : a new generated POI.

```
1  Function MutatePOI(o, OPT_P):
2      if Rand({0, 1})== 0 then
3          // Byte-level mutation on POI
4          o' ← ByteLevelMut(o)
5      else
6          // Structural mutation on POI
7          opt_list ← Split(o)
8          for i ← 0 to Rand({1, 2, 3, ··· , 32}) do
9              switch GetRandMutOp() do
10                 case Insertion do
11                     opt_new ← Rand(OPT_P)
12                     pos ← Rand({0, 1, 2, ..., |opt_list| + 1})
13                     InsertAt(opt_list, pos, opt_new)
14                 end
15                 case Deletion do
16                     pos ← Rand({0, 1, 2, ..., |opt_list|})
17                     RemoveAt(opt_list, pos)
18                 end
19                 case Replacement do
20                     opt_new ← Rand(OPT_P)
21                     pos ← Rand({0, 1, 2, ..., |opt_list|})
22                     Replace(opt_list, pos, opt_new)
23                 end
24             end
25         end
26         o' ← Concat(opt_list)
27     end
28     return o'
29 End Function
```

3.4.2 Byte-level Mutation on POI. Byte-level mutation considers a POI as a string and mutates the string randomly (Line 3-4 in Algorithm 2). Thus, it is likely to generate broken POIs that violate the constraints of POI. These invalid POIs can detect crashes that occur only in exceptional executions caused by invalid POIs.

Also, this byte-level mutation can contribute to generating diverse option arguments with infinite domains. Some options in POIs take an integer or a string value as an argument. For example, in Figure 3, `-b_sensitivity` option in Line 9 takes a string that represents a positive integer value as an argument. While structural mutation can insert only pre-defined strings in the dictionary, byte-level mutation has more chances to generate strings

that represent diverse integers including exceptional ones such as negative numbers or a huge number larger than INT_MAX.

3.5 File Input Mutation Phase

After generating diverse POIs in the POI mutation phase, ZigZagFuzz generates diverse FIs through random mutation like other fuzzers (e.g., AFL++).

The center part of Figure 2 shows a process of the FI mutation phase. It receives the test inputs generated by the preceding POI mutation phase (e.g., (o_1, f_1), (o_{11}, f_1), (o_2, f_2), (o_{21}, f_2)). Then, it generates more test inputs (e.g., (o_1, f_{11}), (o_{11}, f_{12}), (o_2, f_{21}) and (o_{21}, f_{22})) by mutating the FIs of the given test inputs.

3.6 Domain-wise Corpus Shrinking

After each FI mutation phase, ZigZagFuzz removes test inputs with redundant POI and FI combinations from the seed corpus. ZigZagFuzz tries to retain a test input only if both of its components are *unique*. To this end, the domain-wise corpus shrinking algorithm performs POI reduction and FI reduction independently, and selects a test input whose both components are unique. ZigZagFuzz determines two POIs o_1 and o_2 are similar to each other if two test inputs with the two POIs (e.g., (o_1, f_1) and (o_2, f_2)) cover almost same set of functions. For each POI, ZigZagFuzz measures the function coverage of the POI (i.e., the set of functions covered by the test inputs containing the POI). Likewise, ZigZagFuzz determines which FIs are similar.

Algorithm 3 shows how the domain-wise corpus shrinking works. The inputs of the algorithm are a test input corpus T, *Energy*, and *Cov*. T contains test inputs generated in the mutation phases. *Energy* is a map that records assigned energy score for each test input. *Cov* is a map that records function coverage of each test input $(o, f) \in T$. The domain-wise corpus shrinking operates as follows:

(1) Initialization (Lines 2-5)
O is a set of POIs in T. F is a set of FIs in T. Cov_{POI} is a map that records the set of covered functions of each POI in O. Similarly, Cov_{FI} is a map that saves the set of covered functions of each FI in F.

(2) Coverage processing (Lines 6-10)
ZigZagFuzz calculates function coverage of each POI and each FI. A POI's function coverage is defined as the set of functions covered by the test inputs consisting of the POI and all associated FIs, as specified in Definition 1 (similarly for the function coverage of FIs as specified in Definition 2). While iterating each test input (o, f) in T, ZigZagFuzz takes a list of covered functions *Coverage* that are covered by (o, f) (Line 7). *Coverage* is aggregated to the function coverage of each POI (Line 8) and each FI (Line 9).

Definition 1. For a POI o in a test input $(o, f) \in T$, the function coverage Cov_{POI} of o is defined as:

$$Cov_{POI}[o] = \{func \mid func \in Cov[(o, f)] \text{ for all } f \text{ such that } (o, f) \in T\}$$

Definition 2. For a FI f in a test input $(o, f) \in T$, the function coverage Cov_{FI} of f is defined as:

$$Cov_{FI}[f] = \{func \mid func \in Cov[(o, f)] \text{ for all } o \text{ such that } (o, f) \in T\}$$

(3) Clustering (Lines 11-12)
ZigZagFuzz clusters POIs (and similarly FIs) based on Jaccard distance metric $Dist_J$ [36] which is defined in Definition 3.

Definition 3. For two lists of functions F_1 and F_2, the Jaccard distance between F_1 and F_2 is defined as:

$$Dist_J(F_1, F_2) = 1 - \frac{|F_1 \cap F_2|}{|F_1 \cup F_2|}$$

ZigZagFuzz performs K-Means clustering on POIs (and similarly on FIs) where the distance between elements o_1 and o_2 is measured by the Jaccard distance metric on their function coverage ($Cov_{POI}(o_1)$ and $Cov_{POI}(o_2)$). Two POIs with similar function coverage will have a short distance between them and will be clustered together. The number of clusters is set from predefined configuration values k_{opt} (and similarly k_{file}).

(4) Selection of POIs (Lines 13-16)
ZigZagFuzz selects a small set of POIs O' to represent clusters of POIs. From each cluster, SelTopN selects top-N POIs that have high score. The score of POI is calculated by averaging AFL++ scores of the test inputs that contain the POI. SelTopN utilizes AFL++ score of each test input recorded in *Energy* [5]. ZigZagFuzz selects a predefined number of elements from each cluster as specified by N_{opt}.

(5) Selection of FIs (Lines 17-20)
ZigZagFuzz does the similar process on FIs; it selects a small set of FIs F' from each cluster by calculating the averaged AFL++ score of test inputs that contain each FI. ZigZagFuzz picks a predefined number of elements from each cluster, as indicated by N_{file}.

(6) Selection of test inputs (Lines 21-26)
ZigZagFuzz selects only test inputs in T, each of which contains both selected POI in O' and selected FI in F'.

The right part of the Figure 2 shows a process of the domain-wise corpus shrinking. In this example, it receives eight test inputs (o_1, f_1), ..., (o_{21}, f_{22}) passed from the preceding mutation phases and shrinks the test input corpus to contain only three test inputs (o_1, f_1), (o_{11}, f_1), (o_{21}, f_{22}) as follows. First, ZigZagFuzz generates three POI clusters $\{\mathbf{o_1}\}$, $\{\mathbf{o_{21}}\}$, $\{\mathbf{o_{11}}, o_2\}$ (and two FI clusters $\{\mathbf{f_1}, f_{11}, f_{21}\}$, $\{f_2, f_{12}, \mathbf{f_{22}}\}$). Then, ZigZagFuzz selects o_1, o_{21}, and o_{11} (shown in a **bold** font) from each POI cluster (similarly f_1 and f_{22} from each FI cluster). Lastly, ZigZagFuzz selects three test inputs (o_1, f_1), (o_{11}, f_1), and (o_{21}, f_{22}) each of which consists of selected POI and selected FI. This shrunken corpus will be delivered to the next POI mutation phase.

By clustering POIs and FIs separately, ZigZagFuzz preserves a high diversity of input components (both POIs and FIs) in a shrunken corpus, preventing the potential bloating of specific input components. This dominance of specific POIs or FIs reduces the chances of other POIs being mutated, limiting exploration of various program option scenarios (similarly, this problem can occur to FIs, too).

For example, suppose that we perform corpus shrinking on a corpus $T = \{(o_1, f_1), (o_1, f_2), ..., (o_1, f_{100}), (o_2, f_{101}), (o_3, f_{102}), (o_4, f_{103}), (o_5, f_{104})\}$. In this corpus, a POI o_1 is associated with many FIs while the other POIs o_2, o_3, o_4, and o_5 are not. [6] Also, suppose that, by separately clustering POIs and FIs, ZigZagFuzz generates POI clusters $\{o_1, o_5\}$, $\{o_2\}$, $\{o_3\}$, and $\{o_4\}$ and selects test inputs containing four POIs o_1, o_2, o_3, and o_4, respectively. In contrast, if we cluster test inputs in T without separating POIs and FIs, we might end up with four clusters like $\{(o_1, f_1), ...(o_1, f_{25}), (o_2, f_{101}), \}$, $\{(o_1, f_{26}), ..., (o_1, f_{50}), (o_3, f_{102})\}$, $\{(o_1, f_{51}), ..., (o_1, f_{75}), (o_4, f_{103})\}$, and $\{(o_1, f_{76}), ..., (o_1, f_{100}), (o_5, f_{104})\}$ where each cluster contains a high number of test inputs with the dominant POI o_1. This could result in selecting only test inputs with o_1 from each cluster, thereby excluding o_2, o_3, and o_4 which were selected by ZigZagFuzz. ZigZagFuzz's domain-wise clustering strategy can prevent this problem and ensure a more balanced clustering of POIs and FIs.

This domain-wise corpus shrinking algorithm incurs run-time overhead to measure function coverage and calculating distances between seeds. However, this run-time overhead is negligible because measuring function coverage incurs much less run-time overhead than path coverage and the distance calculation is simple (the time

[5] AFL++ calculates each test input's score based on its execution statistics such as execution speed and the number of covered branches.
[6] This scenario can occasionally occur due to the domain-wise mutation strategy of ZigZagFuzz. Suppose that a FI mutation phase starts with a corpus $T_0 = \{(o_1, f_1), (o_2, f_2), (o_3, f_3), (o_4, f_4), (o_5, f_5)\}$ and mutates only (o_1, f_1) due to the limited time budget of the phase. Then, the resulting corpus T will have many test inputs containing o_1.

Algorithm 3: Domain-wise corpus shrinking

Input: T : a set of the test inputs given to the domain-wise corpus shrinking stage
Energy : a map that records assigned energy score for each test input
Cov : a map that records function coverage of each test input $(o, f) \in T$
Output: T' : a reduced set of test inputs.

Function ShrinkCorpus(T, *Energy*)**:**
  $O \leftarrow \{o : (o, f) \in T\}$
  $F \leftarrow \{f : (o, f) \in T\}$
  $Cov_{POI} \leftarrow map\ \{(o, \emptyset) : o \in O\}$
  $Cov_{FI} \leftarrow map\ \{(f, \emptyset) : f \in F\}$
  foreach $(o, f) \in T$ **do**
    $Coverage \leftarrow Cov[(o, f)]$
    $Cov_{POI}[o] \leftarrow Cov_{POI}[o] \cup Coverage$
    $Cov_{FI}[f] \leftarrow Cov_{FI}[f] \cup Coverage$
  end
  $Cl_O \leftarrow \texttt{KMeans}(O, Cov_{POI}, k_{opt}, Dist_J)$
  $Cl_F \leftarrow \texttt{KMeans}(F, Cov_{FI}, k_{file}, Dist_J)$
  $O' \leftarrow \emptyset$
  foreach $cl \in Cl_O$ **do**
    $O' \leftarrow O' \cup$ SelTopN(cl, *Energy*, N_{opt})
  end
  $F' \leftarrow \emptyset$
  foreach $cl \in Cl_F$ **do**
    $F' \leftarrow F' \cup$ SelTopN(cl, *Energy*, N_{file})
  end
  $T' \leftarrow \emptyset$
  foreach $(o, f) \in T$ **do**
    if $o \in O'$ *and* $f \in F'$ **then**
      $T' \leftarrow T' \cup \{(o, f)\}$
    end
  end
  return T'
End Function

consumed for the domain-wise corpus shrinking is 633.8 seconds on average over 12 hours run in the experiments in Section 4).

3.7 Implementation

We have implemented ZigZagFuzz based on AFL++-4.05a [11]. The core components of ZigZagFuzz (e.g., automated program option keyword extraction, program instrumentation for POI mutation, POI mutation strategies, interleaving scheme, and corpus shrinking) are implemented in an additional 7,000 lines of C and C++ code. The implementation is publicly available on our paper web page (https://sites.google.com/view/zigzagfuzz).

4 EXPERIMENT SETUP

To evaluate ZigZagFuzz, we set four research questions with six state-of-the-art fuzzing techniques and four variants of ZigZagFuzz. The following sections explain the research questions with the experiment setup details such as target subject programs and measurement. We share all detailed experiment setups on our paper web page.

4.1 Research Questions

RQ1. How much does ZigZagFuzz outperform the other state-of-the-art program option fuzzers?

To what extent does ZigZagFuzz achieve bug detection and branch coverage, compared to the state-of-the-art program option fuzzers? Since the previous studies already showed that mutating POI can significantly improve test coverage and bug detection ability (see Section 7.1), we focus on comparing ZigZagFuzz with the following six state-of-the-art fuzzers that mutate POI (i.e., not comparing to the fuzzers that do not mutate POI).

- **AFL++-argv**: AFL++ [11] has a feature called *argv-fuzzing* which inserts a test driver at the entry point of the subject program to randomly mutate POI bytes without mutating FI.
- **AFL++-all**: We implemented a new variant of AFL++-argv (calling it AFL++-all) that mutates both POI and FI at the same time. After AFL++-all mutates an input byte sequence to generate new byte sequences, it interprets the first 256 bytes of a generated byte sequence as POI and the remaining bytes as FI.
- **Eclipser** [7]: We selected Eclipser because it supports mutating both POI and FI (we used `'v1.x'` branch of Eclipser that can utilize both initial POI and FI).
- **CarpetFuzz** [38]: CarpetFuzz employs natural language processing (NLP) and pairwise testing techniques to identify effective POIs. It then mutates only FI with the selected POIs. CarpetFuzz introduces an NLP-based tool that extracts relationships between program options. It begins by filtering out invalid program option combinations using the extracted relationships. Subsequently, it prunes out further by applying N-wise testing technique. In our study, we utilized the POIs previously identified by the authors of CarpetFuzz in their research paper.
- **ConfigFuzz** [48]: ConfigFuzz mutates both POI and FI. Based on a manually written program option grammar, it automatically generates a test driver that can generate a program option configuration from a generated byte sequence. Instead of manually writing program option grammars for all 20 target subjects, which would cost significant human effort, we converted the dictionaries of program option keywords for the 20 target subjects (originally made for ZigZagFuzz) to json files that ConfigFuzz can accept as program option grammars.
- **POWER** [19]: It is a predecessor of ZigZagFuzz. Unlike ZigZagFuzz, POWER does not interleave POI mutation phases with FI mutation phases. Also, unlike ZigZagFuzz which applies both structural mutation and byte-level mutation to POI, POWER applies only structural mutation to POI. Another difference is that POWER selects useful POIs based on the expensive function relevance [18] while ZigZagFuzz selects both POIs and FIs based on their function coverage. These differences between POWER and ZigZagFuzz are described in Section 7.1.1.

RQ2. How much does the interleaving of POI mutation phases with FI mutation phases affect the performance of ZigZagFuzz?

To what extent does the interleaving feature of ZigZagFuzz contribute to achieving high bug detection ability and branch coverage? For RQ3, we developed ZZF$^{no\text{-}int}$ that runs a POI mutation phase for one hour, and then a FI mutation phase for another hour, and terminates without domain-wise corpus shrinking (which is meaningless without repeated mutation phases). To focus on the effect of the interleaving in ZigZagFuzz, we compared the

Table 1. Target subjects

Programs	Package name and version	Size (LoC)	#Prog. opt. keywords	Programs	Package name and version	Size (LoC)	#Prog. opt. keywords
avconv	libav-12.3	600,955	761	nasm	nasm-2.16.01	105,260	218
bison	bison-3.8	82,075	51	objdump	binutils-2.40	1,255,876	84
cjpeg	libjpeg-turbo-2.1.4	18,594	33	pdftohtml	poppler-22.12.0	127,859	26
dwarfdump	libdwarf-0.5.0	104,578	103	pdftopng	xpdf-4.04	104,472	18
exiv2	exiv2-0.27.6	112,168	76	pspp	pspp-1.6.2	209,790	20
ffmpeg	ffmpeg-N-109669-g9a180f60a9	1,087,592	1817	readelf	binutils-2.40	154,470	98
gm	GraphicsMagick-1.3.40	287,198	757	tiff2pdf	libtiff-4.5.0	51,294	30
gs	ghostscript-10.0.0	1,517,673	350	tiff2ps	libtiff-4.5.0	42,839	34
jasper	jasper-4.0.0	46,331	20	xmllint	libxml2-2.10.3	186,900	66
mpg123	mpg123-1.31.2	44,675	122	xmlwf	expat-2.5.0	16,721	15
Average						307,866	235.0

performance of ZZF$^{no\text{-}int}$ with ZZF$^{no\text{-}shr}$ for two hours (i.e., ZZF$^{no\text{-}shr}$ performs POI mutation (30 mins) → FI mutation (30 mins) → POI mutation (30 mins) → FI mutation (30 mins)).

- **ZZF**$^{no\text{-}int}$: ZZF$^{no\text{-}int}$ does not repeat mutation phases; it performs a POI mutation phase for the first one hour and an FI mutation phase for the next one hour without the domain-wise corpus shrinking.
- **ZZF**$^{no\text{-}shr}$: It is the same as ZigZagFuzz except that it skips the corpus shrinking (i.e., ZZF$^{no\text{-}shr}$ uses the entire test input corpus generated from the preceding mutation phases).

RQ3. How much does domain-wise corpus shrinking technique affect the performance of ZigZagFuzz?
To what extent does ZigZagFuzz achieve bug detection ability and branch coverage, compared to the variant of ZigZagFuzz that does not perform the input corpus shrinking? For RQ3, we compare the performance results of ZigZagFuzz with ZZF$^{no\text{-}shr}$.

RQ4. How much do different mutation schemes to POI affect the performance of ZigZagFuzz?
To what extent does ZigZagFuzz achieve bug detection and branch coverage, compared to the variants of ZigZagFuzz that perform only one mutation scheme on POI? We would like to evaluate the effectiveness of applying both structural mutation and byte-level mutation to POI in ZigZagFuzz. For that purpose, we compare ZigZagFuzz with the following two variants of ZigZagFuzz:

- **ZZF**^{struct}: It applies only structural mutation [31] to POI, not byte-level random mutation. Based on a dictionary of program option keywords for a target program, it makes random combination of the keywords by randomly inserting one keyword, removing one keyword, or replacing one keyword with another random keyword.
- **ZZF**^{byte}: It applies only byte-level mutation to POI (similar to random mutation of FI).

4.2 Fuzzing Subjects

As ConfigFuzz [48] stated, we could not utilize common fuzzing benchmarks such as Google's FuzzBench [28] because they are not developed to mutate POI along with FI.

Instead, we employ the latest versions of the 20 C/C++ open-source real programs which were frequently tested by other fuzzing papers. The subject details are listed in Table 1. The size of the subjects ranges from 16,721 LoC (i.e., `xmlwf`) to 1,517,673 LoC (i.e., `gs`) and the average size is 307,866 LoC. The number of the program option keywords extracted from the documents including usage messages ranges from 15 (i.e., `xmlwf`) to 1,817 (i.e., `ffmpeg`).

4.3 Fuzzing Setup

We execute each studied fuzzing technique for 12 hours (except ZZF$^{no\text{-}int}$ and ZZF$^{no\text{-}shr}$ in RQ2), since our preliminary study has shown that most fuzzing campaigns in the experiments show consistent results within 12 hours. Also, we repeated each experiment run five times to mitigate the random variance of fuzzing experiments. We used a 30-minute timeout (τ_t) for each POI mutation phase and FI mutation phase. All experiments were performed on a cluster where each machine is equipped with AMD Ryzen 9 5950X (3.4GHz) and 32GB RAM, running Ubuntu 18.04.6.

4.3.1 Initial Seed Setup. ZigZagFuzz requires both initial POIs and initial FIs. We collected and utilized commonly used POIs and FIs from the recent fuzzing papers surveyed in Section 2.1. The initial seed setup is publicly available at our paper web page.

4.3.2 Configuration of Input Corpus Clustering. For the domain-wise input corpus shrinking (Section 3.6), we set the number of input clusters and the numbers of selected POIs and FIs as follows (obtained from our preliminary study). We set ZigZagFuzz to make 20 POI clusters and 800 FI clusters and selected top-2 POIs and top-2 FIs from each cluster (i.e., $k_{opt} = 20, k_{file} = 800$, and $N_{opt} = N_{file} = 2$ in Algorithm 3). We performed several exploratory studies with different settings, and we selected the best values we observed.

4.4 Measurement

4.4.1 Branch Coverage. To measure the coverage achievement of each technique, we replayed all generated test inputs and counted the number of covered branches using `gcov`. We report the average branch coverage over the five experiment runs.

4.4.2 Unique Bug Detection. To measure bug detection ability of fuzzing techniques, we count the number of unique bugs found by each technique. First, we applied LLVM AddressSanitizer [35] to collect crashes raised by generated test inputs for each technique. After that, we identified unique crash bugs based on the collected alarm messages. Following the most widely used practice [27], we first removed the crashes that show the identical stack trace to each other. Then, we manually identified the crashes whose stack traces are different but are suspected to share the same root cause, to the best of our ability. We reported the number of unique bugs detected in any of the five fuzzing runs.

4.5 Threats to Validity

External. To our best knowledge, there exist no benchmark programs for the fuzzers that fuzz POI. By selecting 20 diverse popular open source programs that were used to evaluate other recent fuzzers, we believe that this threat is limited (i.e., our experiment result can be applicable to various programs with program options). Also, we compared the results obtained from each experiment for 12 hours and this time budget might not be enough to compare the overall performance of each technique. However, we could observe that 12-hour timeout was long enough to find the consistent results of the studied fuzzing techniques.
Internal. The implementation of ZigZagFuzz may contain bugs that can affect the experiment results. To control this threat, we have tested our implementation extensively. Another threat may be that we gave ConfigFuzz a program option grammar that was semi-automatically generated from manual pages/usage messages of a target

Table 2. The total numbers of the unique bugs detected and the average numbers of branches covered by the state-of-the-art program option fuzzers

Programs	AFL++-argv		AFL++-all		Eclipser		CarpetFuzz		ConfigFuzz		POWER		ZigZagFuzz	
	B	Cov	B	Cov	B	Cov	B	Cov	B	Cov	B	Cov	B	Cov
avconv	0	4224.6	1	16454.0	0	6148.4	4	15543.2	1	9978.4	3	10201.4	**13**	**18881.6**
bison	0	1342.4	**4**	4218.4	1	3340.6	0	5213.4	2	5433.8	0	5212.2	**4**	**6207.6**
cjpeg	0	210.6	0	3239.8	0	2654.2	0	2841.2	0	4305.0	0	4376.6	0	**4416.6**
dwarfdump	1	853.6	2	2408.6	2	6128.4	2	8074.6	3	8097.8	5	9072.2	**6**	**9320.4**
exiv2	0	2042.0	0	4722.0	0	2944.4	**2**	3581.0	0	5846.0	0	6222.2	0	**7967.0**
ffmpeg	1	26321.6	1	33832.2	0	17641.6	1	37310.0	1	19967.8	3	39008.4	**11**	**42761.0**
gm	10	14711.8	0	5599.8	1	3875.8	0	6539.6	3	13162.6	8	14116.0	**17**	**19298.0**
gs	6	13384.0	0	19009.4	0	15880.2	0	19531.4	1	20789.0	1	29345.6	**8**	**35604.6**
jasper	0	641.0	0	1486.2	0	3389.0	1	3713.2	**2**	3740.4	1	3598.6	1	**4073.0**
mpg123	0	134.0	1	4292.0	0	2797.2	1	3115.8	**2**	3587.4	0	3802.4	1	**4587.2**
nasm	3	5200.8	9	6750.0	2	3141.8	1	5325.4	11	8536.6	**13**	**8857.2**	11	8448.2
objdump	0	6285.2	0	17036.2	0	5728.8	**2**	**31429.4**	0	23615.8	**2**	29538.0	**2**	27759.6
pdftohtml	**1**	662.4	0	4163.0	0	1967.8	0	4633.4	0	4973.6	0	**5703.4**	**1**	5029.4
pdftopng	0	1007.8	1	5496.8	0	5014.8	1	6700.2	1	6538.4	1	7199.6	**2**	**8225.2**
pspp	0	2577.4	2	4105.4	2	3003.4	**8**	6465.2	5	5727.6	6	**7174.8**	5	6690.2
readelf	0	894.8	0	5712.0	0	3796.4	**2**	**10065.8**	**2**	9843.8	**2**	9987.4	**2**	9600.0
tiff2pdf	0	147.0	0	2907.4	0	2435.2	0	4472.8	**1**	4404.2	0	**4576.8**	0	4384.8
tiff2ps	0	168.4	0	2015.0	0	1808.0	0	3864.2	0	3939.0	0	**4212.6**	0	4091.6
xmllint	1	13752.4	1	16045.2	0	5418.0	**4**	12886.0	0	15531.0	0	15663.0	1	**17118.2**
xmlwf	0	568.0	0	2086.2	0	2403.4	0	**4019.2**	0	3685.0	0	3087.0	0	3628.0
Total # of bugs	23		22		8		29		35		45		**85**	
Avg. # of branches covered		4756.5		8079.0		4975.9		9766.3		9085.2		11047.8		**12404.6**

* The B means the number of unique bugs detected, and the Cov means the average number of branches covered.

program, which might reduce performance of ConfigFuzz. We think that this threat is unavoidable for the fair comparison of ConfigFuzz with the other fuzzers.

5 EXPERIMENT RESULTS

Tables 2, 3, 4, 5 and Figure 6 report the number of unique bugs detected and the number of branches covered by each fuzzing technique on the 20 fuzzing subjects.

5.1 RQ1. Fuzzing Effectiveness of ZigZagFuzz Compared to the State-of-the-art Program Option Fuzzers

5.1.1 Bug Detection Achieved. The experiment results show that ZigZagFuzz detects significantly more (1.9 to 10.6 times more) unique bugs than the other state-of-the-art POI fuzzing techniques. Table 2 shows the number of unique bugs detected and the number of branches covered by AFL++-argv, AFL++-all, Eclipser, CarpetFuzz, ConfigFuzz, POWER, and ZigZagFuzz (the best numbers are marked in a **bold** font).

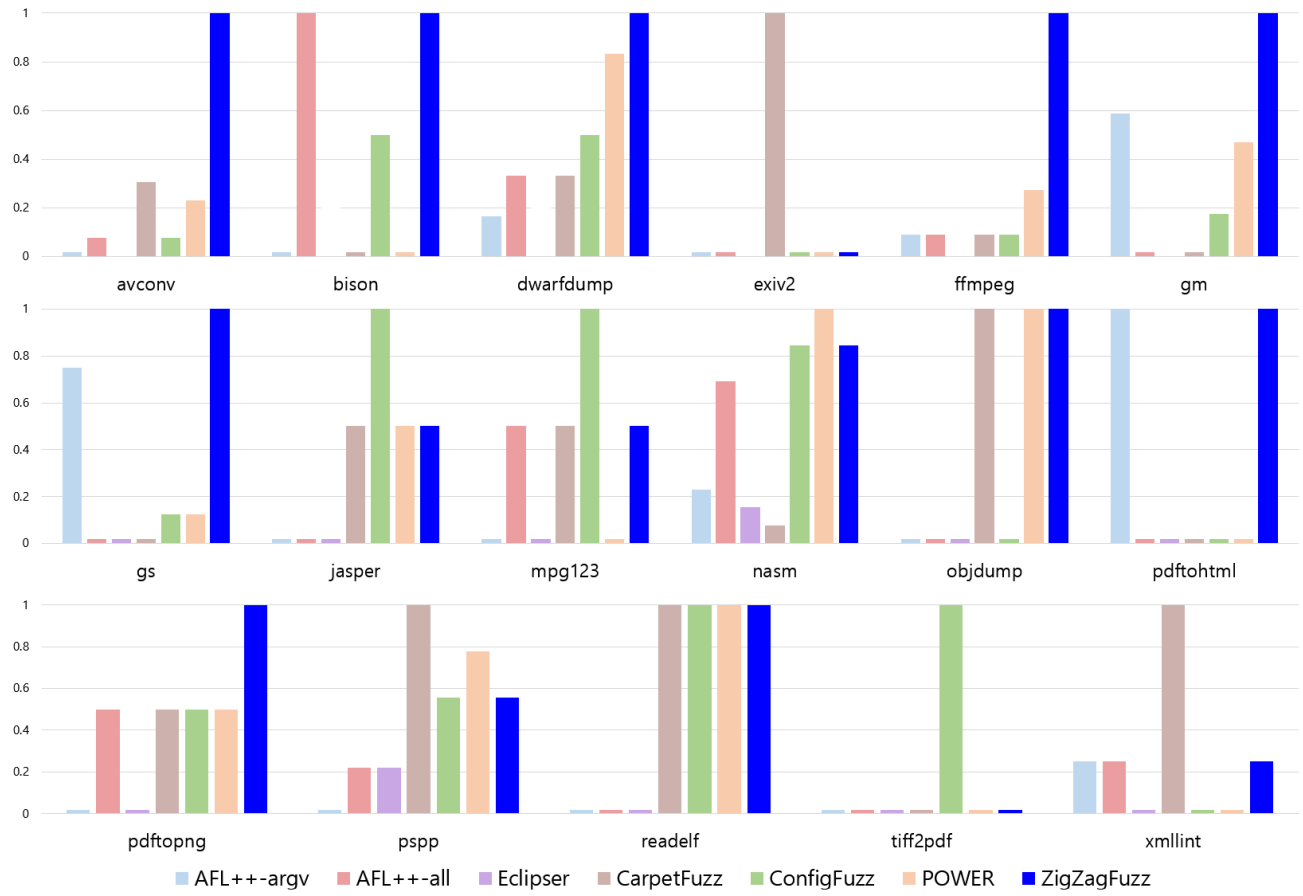

Fig. 4. The relative ratio of the unique bugs detected for each target subject program (the most effective fuzzer's effectiveness is normalized to one)

ZigZagFuzz detected 85 unique bugs on 15 subjects. ZigZagFuzz detected 3.7 (= 85/23), 3.9 (= 85/22), 10.6 (= 85/8), 2.9 (= 85/29), 2.4 (= 85/35), and 1.9 (= 85/45) times more unique bugs than AFL++-argv, AFL++-all, Eclipser, CarpetFuzz, ConfigFuzz, and POWER, respectively. Also, note that, among the 17 target subjects from which at least one of the fuzzers detected a bug, ZigZagFuzz is most effective for the 10 subjects (i.e., `avconv`, `bison`, `dwarfdump`, `ffmpeg`, `gm`, `gs`, `objdump`, `pdftohtml`, `pdftopng`, and `readelf`) and the second most effective for the four subjects (`jasper`, `mpg123`, `nasm`, and `xmllint`).

Figure 4 visually illustrates the relative fuzzing effectiveness in terms of the uniquely detected bugs of the fuzzers for each target subject. For each target subject, the most effective fuzzer's effectiveness is normalized to one, and the relative effectiveness of the other fuzzers is computed in relation to the normalized value. As shown in the figure, ZigZagFuzz is obviously the most effective fuzzer among the compared techniques (i.e., ZigZagFuzz's bars are higher than the other techniques for most subjects).

In addition, the Venn diagram in Figure 5 shows how many unique bugs were found by each of the top four fuzzing techniques (i.e. CarpetFuzz, ConfigFuzz, POWER, and ZigZagFuzz). It shows that ZigZagFuzz detected the largest number of unique crashes that were not found by the other techniques (46). ZigZagFuzz also detected

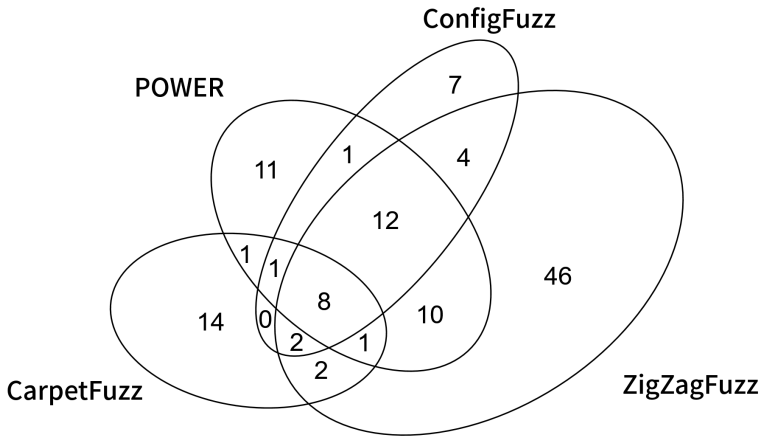

Fig. 5. The relation of the unique bugs found by the top four fuzzers

most of the unique bugs found by the other three fuzzers (i.e., 44.8% (=13/29), 68.9% (= 31/45), and 74.3% (=26/35) of the bugs detected by CarpetFuzz, POWER, and ConfigFuzz, respectively).

We can make a few additional observations as follows:

- *ZigZagFuzz has superior bug detection ability for large target programs* than the other fuzzers:
  For the top three largest subjects (`ffmpeg`, `gs`, and `objdump` whose sizes are larger than 1 MLoC), ZigZagFuzz detected the far larger number of bugs among the compared fuzzing techniques. For example, for `ffmpeg`, ZigZagFuzz detected 11 bugs while the second most effective fuzzer (POWER) did only three.
- *ZigZagFuzz has superior bug detection ability for hard-to-find bugs* than the other fuzzers:
  For the eight subjects with hard-to-find crashes (i.e., the subjects with eight or fewer bugs detected by all seven fuzzers such as `jasper`, `mpg123`, `objdump`, `pdftohtml`, `pdftopng`, `readelf`, `tiff2pdf`, and `xmllint`), ZigZagFuzz detected the largest number of bugs for the four subjects (`objdump`, `pdftohtml`, `pdftopng`, `readelf`) and the second largest number of bugs for the three subjects (`jasper`, `mpg123`, and `xmllint`).

5.1.2 Branch Coverage Achieved. ZigZagFuzz covered significantly more branches than the other techniques. For example, on average, ZigZagFuzz covered 2.6 (=12404.6/4756.5) times more branches than AFL++-argv and 1.4 (=12404.6/9085.2) times more branches than ConfigFuzz. Figure 6 shows the branch coverage increase over time. X-axis and y-axis represent execution time in hours and the number of covered branches, respectively.

ZigZagFuzz achieved the highest branch coverage for the 12 subjects (`avconv`, `bison`, `cjpeg`, `dwarfdump`, `exiv2`, `ffmpeg`, `gm`, `gs`, `jasper`, `mpg123`, `pdftopng`, and `xmllint`) and the second highest branch coverage for the three subjects (`pdftohtml`, `pspp`, and `tiff2ps`). Even for the remaining five subjects (`nasm`, `objdump`, `readelf`, `tiff2pdf`, and `xmlwf`), ZigZagFuzz's coverage is almost same to the most effective fuzzer per subject (i.e., ZigZagFuzz covered 88.3% (=27759.6/31429.4 on `objdump`) to 95.8% (=4384.8/4576.8 on `tiff2pdf`) of the branches covered by the most effective fuzzer per subject).

5.2 RQ2. Fuzzing Effectiveness of the Interleaving of POI Mutation Phases with FI Mutation Phases

The experiment results in two hours show that the interleaving scheme improves the performance of ZigZagFuzz. Table 3 shows that, by interleaving POI mutation phases with FI mutation phases, ZZF$^{no\text{-}shr}$ detected 25.0% (=(40-32)/32) more unique bugs and covered 21.7% (=(9114.7- 7487.0)/7487.0) more branches than ZZF$^{no\text{-}int}$. For example, on `objdump`, `pdftopng`, and `xmllint`, ZZF$^{no\text{-}shr}$ detected a bug while ZZF$^{no\text{-}int}$ did not. Moreover,

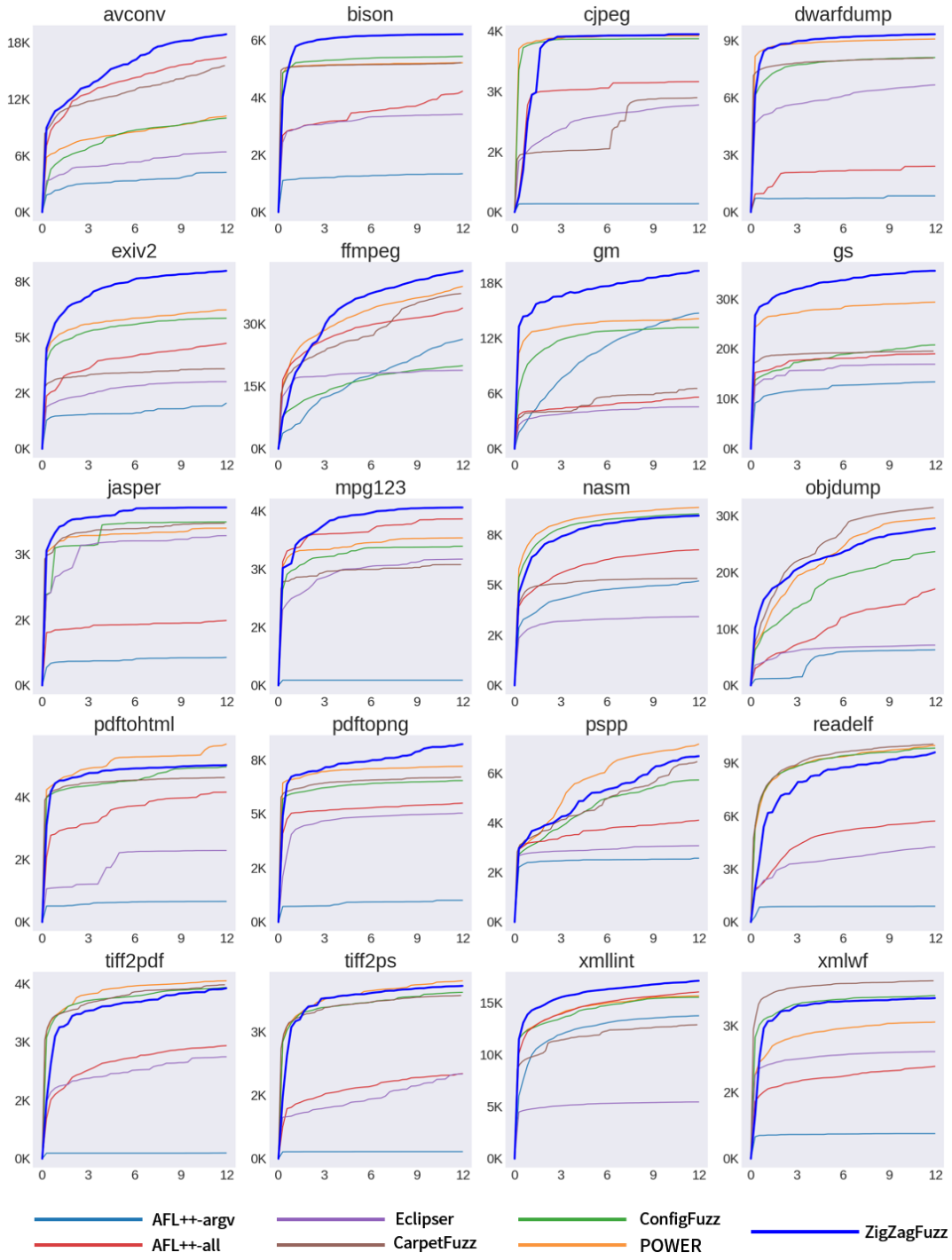

Fig. 6. The branch coverage results for the state-of-the-art fuzzing techniques over time

Table 3. The total number of the unique bugs detected and the average numbers of the branches covered in two hours by the variants of ZigZagFuzz with/without interleaving

Programs	ZZF$^{no\text{-}int}$ (two hours run)		ZZF$^{no\text{-}shr}$ (two hours run)		Programs	ZZF$^{no\text{-}int}$ (two hours run)		ZZF$^{no\text{-}shr}$ (two hours run)	
	#uniq. bugs	#branch covered	#uniq. bugs	#branch covered		#uniq. bugs	#branch covered	#uniq. bugs	#branch covered
avconv	**4**	7966.2	3	**10784.0**	nasm	5	5934.4	**8**	**7084.6**
bison	1	4970.0	**2**	**5676.8**	objdump	0	13844.8	**1**	**18544.2**
cjpeg	0	1537.2	0	**4206.6**	pdftohtml	0	3968.6	0	**4442.2**
dwarfdump	**5**	8069.0	4	**8593.2**	pdftopng	0	6211.8	**1**	**6774.0**
exiv2	0	5098.4	0	**6361.2**	pspp	1	3219.2	**2**	**3419.6**
ffmpeg	**2**	13175.0	**2**	**20895.8**	readelf	**1**	4875.2	**1**	**7893.4**
gm	9	12118.2	**10**	**14591.6**	tiff2pdf	0	2920.6	0	**3739.2**
gs	3	**31288.2**	**5**	30680.8	tiff2ps	0	2418.6	0	**3496.8**
jasper	**1**	3706.2	0	**3724.0**	xmllint	0	14228.6	**1**	**14978.0**
mpg123	0	2647.8	0	**3752.2**	xmlwf	0	1541.4	0	**2656.4**
Total # of bugs						32		**40**	
Avg. # of branches covered							7487.0		**9114.7**

ZZF$^{no\text{-}shr}$ covered more branches than ZZF$^{no\text{-}int}$ for 19 out of the 20 subject programs (except `gs`). For example, ZZF$^{no\text{-}shr}$ covered 18544.2 branches while ZZF$^{no\text{-}int}$ did only 13844.8 branches of `objdump` on average.

Also, to see the effectiveness of the iterative/interleaving POI mutation phases with FI mutation phases to reach deep code segments through complex condition checks, we counted the number of the test inputs that satisfied all branch conditions to reach the crashing line (Line 12) in the `dwarfdump` crash example in Figure 1. On average, ConfigFuzz made only 409.6 test inputs while ZigZagFuzz made 3754.4 test inputs that satisfied all branch conditions and detected bug in Figure 1. This observation shows that the interleaving of POI mutation phases with FI mutation phases of ZigZagFuzz can successfully generate effective test inputs (pairs of POIs and FIs) to reach hard-to-reach deep code segments, which leads to significantly higher bug detection and higher branch coverage than the other fuzzers.

5.3 RQ3. Fuzzing Effectiveness of Corpus Shrinking of ZigZagFuzz

The experiment results show that the domain-wise corpus shrinking improves the performance of ZigZagFuzz. Table 4 shows that ZigZagFuzz detected 1.15 (=85/74) times more unique bugs and covered 5.4% (= (12404.6-11743.3)/11743.3) more branches than ZZF$^{no\text{-}shr}$. For example, on `bison`, ZigZagFuzz detected four bugs while ZZF$^{no\text{-}shr}$ did only two bugs. For another example, on `ffmpeg`, ZigZagFuzz covered 42761.0 branches while ZZF$^{no\text{-}shr}$ did only 34122.2 branches on average.

In addition, the Venn diagram in Figure 7 shows how many unique bugs were detected by ZigZagFuzz and ZZF$^{no\text{-}shr}$. ZigZagFuzz utilizes function coverage information (instead of fine-grained branch or path coverage which incurs heavy run-time overhead) for efficient domain-wise corpus shrinking. The Venn diagram clearly shows that ZigZagFuzz detected most bugs detected by ZZF$^{no\text{-}shr}$ (62/74), and also it detected a large number (23) of the unique bugs that were not detected by ZZF$^{no\text{-}shr}$.

Table 4. The total number of unique bugs detected and the average numbers of branches covered by the variants of ZigZagFuzz with/without corpus shrinking

Programs	ZZF$^{no\text{-}shr}$		ZigZagFuzz		Programs	ZZF$^{no\text{-}shr}$		ZigZagFuzz	
	#uniq. bugs	#branch covered	#uniq. bugs	#branch covered		#uniq. bugs	#branch covered	#uniq. bugs	#branch covered
avconv	10	15653.2	**13**	**18881.6**	nasm	8	8420.6	**11**	**8448.2**
bison	2	6092.8	**4**	**6207.6**	objdump	**2**	27591.2	**2**	**27759.6**
cjpeg	0	4398.2	0	**4416.6**	pdftohtml	**1**	4858.4	**1**	**5029.4**
dwarfdump	5	9139.0	**6**	**9320.4**	pdftopng	**2**	7359.4	**2**	**8225.2**
exiv2	0	7638.8	0	**7967.0**	pspp	**5**	5656.8	**5**	**6690.2**
ffmpeg	7	34122.2	**11**	**42761.0**	readelf	**2**	**10032.8**	**2**	9600.0
gm	16	18846.6	**17**	**19298.0**	tiff2pdf	0	**4484.8**	0	4384.8
gs	**11**	**37919.6**	8	35604.6	tiff2ps	0	3969.0	0	**4091.6**
jasper	**1**	4025.4	**1**	**4073.0**	xmllint	**1**	**17268.6**	**1**	17118.2
mpg123	**1**	4391.8	**1**	**4587.2**	xmlwf	0	2998.0	0	**3628.0**
Total # of bugs						74		**85**	
Avg. # of branches covered							11743.4		**12404.6**

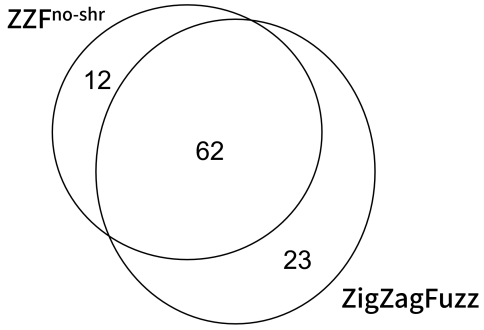

Fig. 7. The relation of the unique bugs detected by ZigZagFuzz and ZZF$^{no\text{-}shr}$

5.4 RQ4. Fuzzing Effectiveness of Different Schemes for POI Mutation of ZigZagFuzz

The experiment results show that using both structural mutation and byte-level mutation on POI significantly increases testing effectiveness. Table 5 shows the number of the unique bugs detected and the number of branches covered by ZZF^{byte}, ZZF^{struct}, and ZigZagFuzz (the largest numbers are marked in a **bold** font). On average, ZZF^{byte} and ZZF^{struct} detected similar number of unique bugs and covered similar number of branches while ZigZagFuzz detected significantly more unique bugs and covered more branches.

5.4.1 Bug Detection Achieved. ZigZagFuzz detected 1.7 (= 85/49) and 1.6 (= 85/53) times more unique bugs than ZZF^{byte} and ZZF^{struct}, respectively. For example of avconv, ZigZagFuzz detected 13 unique bugs while ZZF^{byte} and ZZF^{struct} did only four and five bugs, respectively.

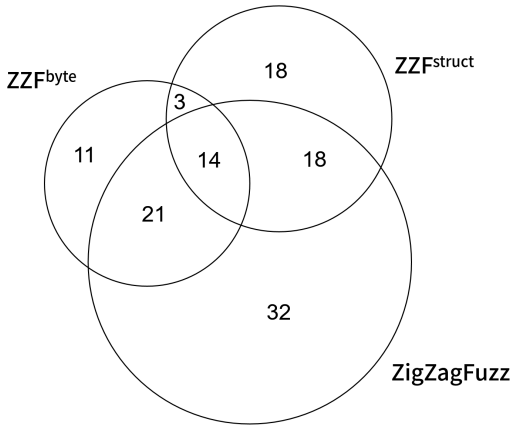

Fig. 8. The relation of the unique bugs detected by ZZF^{byte}, ZZF^{struct}, and ZigZagFuzz

The Venn diagram in Figure 8 shows the number of unique bugs detected by ZZF^{byte}, ZZF^{struct}, and ZigZagFuzz. From the diagram, we make the following observations:

- *Combining the two mutation schemes on POI improves bug detection ability*:
  ZigZagFuzz detected 32 unique bugs that were not detected by the two variants with different mutation schemes. Also, ZigZagFuzz detected the majority of the unique bugs detected by the two variants (i.e., ZigZagFuzz detected 71.4% (= 35/49) of the unique bugs detected by ZZF^{byte} and 60.3% (= 32/53) of the unique bugs of ZZF^{struct}).
- *The two variants with different mutation schemes detected much different sets of unique bugs*:
  ZZF^{byte} detected only 32.1% (=17/53) of the unique bugs detected by ZZF^{struct}. Similarly, ZZF^{struct} detected only 34.6% (= 17/49) of the unique bugs detected by ZZF^{byte}. This indicates that the mutation scheme has a high impact on detecting unique bugs.

5.4.2 Branch Coverage Achieved. ZigZagFuzz covered about 1.1 (= 12404.6/10877.1) and 1.2 (= 12404.6/10877.1) times more branches than ZZF^{byte} and ZZF^{struct}, respectively. For example of avconv, ZigZagFuzz covered 18881.6 branches while ZZF^{byte} and ZZF^{struct} did only 15312.6 and 12475.0 branches on average, respectively.

6 DISCUSSION

6.1 Real-world Bugs Detected by ZigZagFuzz

Among the 85 bugs detected by ZigZagFuzz, we reported 61 bugs to the original developers of the target programs. To reduce the developer's burden to review many bug reports, we excluded 11 reports detected in the latest release version but could not replicate in the latest development version. Also, we did not report the 13 bugs in avconv because it is no longer supported by the developers. We received the following positive responses from the developers:

- 44 reported bugs have been fixed by the original developers
- 17 bugs are waiting to be confirmed

As shown in the following case studies, ZigZagFuzz can successfully detect complex bugs that require a specific POI and a specific FI to trigger.

Table 5. The total number of the unique bugs detected and the average numbers of the branches covered by the variants of ZigZagFuzz with different mutation schemes

Programs	ZZF^{byte}		ZZF^{struct}		ZigZagFuzz	
	#uniq. bugs	#branch covered	#uniq. bugs	#branch covered	#uniq. bugs	#branch covered
avconv	4	15312.6	5	12475.0	**13**	**18881.6**
bison	3	6118.4	1	5299.2	**4**	**6207.6**
cjpeg	0	4275.8	0	4165.0	0	**4416.6**
dwarfdump	**7**	9284.2	4	8952.8	6	**9320.4**
exiv2	0	**8121.2**	**1**	7034.2	0	7967.0
ffmpeg	1	38096.2	5	28570.6	**11**	**42761.0**
gm	5	11760.4	15	**19608.4**	**17**	19298.0
gs	7	23253.6	3	31589.2	**8**	**35604.6**
jasper	0	3789.0	**2**	3664.0	1	**4073.0**
mpg123	**1**	4536.8	0	3661.4	**1**	**4587.2**
nasm	9	8158.6	**12**	8247.0	11	**8448.2**
objdump	**2**	**31134.4**	0	28593.0	**2**	27759.6
pdftohtml	**1**	4699.4	0	4986.4	**1**	**5029.4**
pdftopng	**2**	7033.8	1	7257.2	**2**	**8225.2**
pspp	4	4505.2	3	6258.0	**5**	**6690.2**
readelf	**2**	8595.6	1	**9769.0**	**2**	9600.0
tiff2pdf	0	4255.2	0	4139.6	0	**4384.8**
tiff2ps	0	4018.6	0	3835.6	0	**4091.6**
xmllint	**1**	**17215.6**	0	14422.2	**1**	17118.2
xmlwf	0	3376.8	0	2655.2	0	**3628.0**
Total # of bugs	49		53		**85**	
Avg. # of branches		10877.1		10759.2		**12404.6**

```
mpg123 -smooth --listentry -z -w l --quiet --index --4to1 -2 -q --fifo --outfile @@
```

Fig. 9. POI generated by ZigZagFuzz that triggers a crash in `mpg123`

6.1.1 Case Study 1: `mpg123`. ZigZagFuzz detected a new crash bug in `mpg123` by generating a program option configuration containing the 13 command-line options shown in Figure 9. `-2` or `-2to1` options make the program to downsample an audio file. `-index` option makes the program to scan through an audio file.

The crash bug was triggered when `mpg123` tries to scan an audio file with invalid sampling rate value. `mpg123` supports specific sampling rate values that range from 8kHz to 48.0kHz. If an audio file with low sampling rate value (e.g., 11,025Hz) is given as FI to `mpg123` with `-2` option, the program tries downsampling the audio file and makes an invalid audio file with an unsupported sampling rate value (e.g., 5,512 Hz). Thus, by using `-index` option with `-2` option, it results in a wrong memory access when the program reads an invalid track in the audio file.

```
objdump @@ --adjust-vma=4 --start-address=0x0 -Wc -S -x -Ud 8-S o@rchite -g -f -Wm -Ud
-mh --file-offsets -f -a -mnf -Wf -Wo
```

Fig. 10. POI generated by ZigZagFuzz that triggers a segmentation violation error in objdump

We reported the bug to the developer of mpg123 (the bug report is available at https://sourceforge.net/p/mpg123/bugs/322/) and the developer fixed the bug within 33 hours from the initial bug report. The developer was highly interested in ZigZagFuzz because, although mpg123 had been extensively fuzzed by using Google's OSS-fuzz [14], the reported bug was not detected before. The response of the developer is as follows: "Interesting approach you find stuff where oss-fuzz didn't anymore".

6.1.2 Case Study 2: objdump. The following two new features of ZigZagFuzz enabled the successful detection of a segmentation violation error in objdump:

- *Iterative/interleaved POI mutations with FI mutations*:
  To trigger the crash, a test input should satisfy a complex intermixed sequence of POI- and FI-dependent conditions like Figure 1. The interleaving of POI mutations with FI mutations of ZigZagFuzz can generate a proper pair of a POI and a FI that can satisfy conditions in a complex intermixed sequence of POI- and FI-dependent conditions.
  ZigZagFuzz triggered this crash bug when objdump is commanded to show the disassembled code of an object file for Netronome Flow Processor (NFP) architecture. `-m nfp` or `-mnf` option generated by ZigZagFuzz causes objdump to show the disassembled code of an object file for NFP architecture.
  To detect the crash error, a test input should contain both a proper POI (i.e., `-mnf`) that triggers objdump to handle NFP architecture and a proper FI that contains a corrupted NFP file which contains an empty section owner information; when objdump tries to access the corresponding section, it causes a crash error. By using the interleaved mutation approach for POI and FI, ZigZagFuzz successfully generated a test input that caused the crash error.
- *Combined application of both structural and byte-level mutations to POI*:
  ZigZagFuzz could generate a POI that leads to the crash by utilizing both structural mutation and byte-level mutation on POI. Figure 10 shows the POI generated by ZigZagFuzz that triggers the segmentation violation error. It shows that ZigZagFuzz created a complex POI by using both structural mutation and byte-level mutation. The tokens such as `-start-address` is generated by structural mutation and `-mnf` is generated by byte-level mutation (`-mnf` is not included in a dictionary used by ZigZagFuzz, since it is not an officially documented program option).

We reported this bug to objdump developers and they fixed the bug by applying the patch in Figure 11 (the patch checks if the given section owner information is empty or not in Lines 5-6). The patch indicates that the crash bug is induced when an NFP file has an empty section owner information.

6.2 Dependency of Bugs on POIs and FIs Generated by ZigZagFuzz

Table 6 presents the 22 bugs of gm and pspp detected by ZigZagFuzz as an example to show dependencies between the bugs and input components generated by ZigZagFuzz. The third row shows a failure type of each bug. The fourth and fifth rows indicate whether each bug is dependent solely on a POI or a FI, respectively. The sixth row indicates whether each bug is dependent on both POIs and FIs. Lastly, the seventh row indicates if the bug was detected by ZigZagFuzz only among the top four program option fuzzers in RQ1.

We determined the dependency between a bug and an input component as follows:

- A bug is dependent solely on a POI if the bug is triggered with a specific POI with any FIs (or no FI).

```
01:    @@ -2676,7 +2676,9 @@ init_nfp6000_priv

02:      memset (mecfg_orders, -1, sizeof (mecfg_orders));
03: -    if (!dinfo->section)
04: +    if (dinfo->section == NULL
05: +      || dinfo->section->owner == NULL
06: +      || elf_elfsections (dinfo->section->owner) == NULL)
         ...
07:      /* No section info, will use default values. */
08:      return true;
```

Fig. 11. The fix commit of the corrupted NFP file bug in `objdump`

Table 6. Dependency analysis results on 22 bugs detected by ZigZagFuzz

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
Prog.	gm																	pspp				
Fail. ty.	a.v.	a.v.	a.v.	a.v.	f.p.	f.p.	h.b.	s.f.	h.b.	s.f.	s.f.	s.f.	s.f.	s.f.	s.f.	h.b.	s.f.	a.v.	a.v.	a.v.	a.v.	h.b.
POI		✓						✓			✓			✓			✓					
FI																		✓	✓	✓		✓
Both	✓		✓	✓	✓	✓	✓		✓	✓		✓	✓		✓	✓					✓	
ZZF		✓	✓	✓		✓	✓		✓	✓	✓					✓	✓					

a.v.: assert violation, f.p.: floating point exception, h.b.: heap-buffer-overflow, s.f.: segmentation fault

- A bug is dependent solely on a FI if the bug is triggered with a specific FI with a default initial POI in Section 4.3.1.
- A bug is dependent on both POIs and FIs if the bug can be triggered by a combination of a specific POI and a specific FI.

The five out of the 22 bugs are dependent solely on POIs (i.e., bug indices 2, 8, 11, 14, and 17), four bugs are dependent solely on FIs (i.e., bug indices 18, 19, 20, and 22), and the remaining 13 bugs are dependent on both POIs and FIs. ZigZagFuzz detected various types of crash bugs. In detail, ZigZagFuzz detected eight assert violations, two floating point exception bugs, four heap-buffer-overflow bugs, and eight segmentation fault bugs. Note that, among the top four fuzzers, only ZigZagFuzz detect three POI-dependent bugs and seven POI-FI-dependent bugs.

7 RELATED WORK

7.1 Fuzzers that Mutate Program Option Input (POI)

TOFU [42] is a fuzzer that mutates program option configurations for directed fuzzing. It generates many different option configurations by using structural mutation and tries to find an option configuration that gives the closest distance to a target basic block. TOFU receives a specification of program options (i.e., the name of options

and the type of option argument if any) from a user and performs structural mutation on the program option configurations by using the specification as a dictionary. Unlike TOFU, ZigZagFuzz just requires a list of program option keywords that are described in the manual pages and/or the help messages of target programs. Also, ZigZagFuzz actively generates diverse POIs with accompanying FIs to explore large path space while TOFU mutates POI only until it finds a path to a target block.

Zeller et al. [47] (an online course) developed a fuzzer that automatically infers the program option grammar of Python programs that use `argparse` function. They use the inferred program option grammar to generate valid program option configurations and fuzz input files with the generated option configurations.

CLIFuzzer [15] also tried to automatically infer program option input grammar from the usage of standard C library function `getopt`. CLIFuzzer's applicability is limited, since many real-world programs use its own program option handling logic, not `getopt` (CLIFuzzer was evaluated on small C/C++ programs with a maximum size of 81,215 lines). In contrast, we target much larger real-world programs whose average size is 307,866 lines (only six of the target subjects use `getopt`).

Eclipser [7] supports option configuration mutation. Eclipser tracks relation between each input byte and branch constraints with light-weight instrumentation on binary code, and it supports tracking not only input file bytes but also POI's bytes. After tracking the relation, it searches for correct values of the related bytes with multiple executions to cover the branches.

CarpetFuzz [38] does not mutate POIs, but it selects effective POIs from possible option combinations by utilizing relationship information extracted using natural language processing techniques. It parses CLI documentation and extracts dependency or conflict among program options. Then, it selects only valid POIs that satisfy the extracted relationship conditions. After selecting POIs, CarpetFuzz mutates only FIs using the selected POIs.

ConfigFuzz [48] mutates both POI and FI by mutating input bytes and interpreting the first few bytes as POI and the remaining bytes as FI. Based on a manually written program option grammar, ConfigFuzz inserts, at the entry point of the `main` function, code that converts input bytes into a program option configuration. In contrast, ZigZagFuzz's clear separation and interleaving of POI mutation phases and FI mutation phases enables to explore deeper state of a target program and, thus, detects more bugs.

7.1.1 Comparison between ZigZagFuzz and POWER. ZigZagFuzz has the following new features compared to its predecessor POWER [19]:

(1) Unlike POWER, ZigZagFuzz separately mutates POIs and FIs in an iterative/interleaving manner because a target program may have a complex intermixed sequence of POI-dependent branches and FI-dependent branches that depend on each other (see Section 2). In contrast, POWER mutates both POIs and FIs together for the first one hour and it mutates only FI for the remaining time.
We illustrate the necessity of iterative/interleaved application of POI mutations with FI mutations in Figure 1 in Section 2. Section 4.1 (RQ 2: Fuzzing Effectiveness of the Interleaving of POI Mutation Phases with FI Mutation Phases) and Section 5.2 show that this iterative/interleaving mutations of POIs with FIs improve bug detection ability by 25.0% and branch coverage achievements by 21.7%.

(2) ZigZagFuzz considers the distinct characteristics of POIs and FIs and applies a reduction strategy by clustering POIs and FIs separately (Section 3.6). In contrast, POWER adopts a selective approach, greedily targeting promising POIs. Also, ZigZagFuzz employs function coverage as a criterion for corpus reduction while POWER prioritizes expensive function relevance. This new corpus shrinking method improves test coverage 5.6% and bug detection 14.9% (RQ3 in Section 5.3).
(3) ZigZagFuzz applies both byte-level mutations and structural mutations to POIs (which improves bug detection by 60.4% (= (85-53)/53) compared to applying only structural mutations in RQ4) while POWER does only structural mutations to POI (see Section 3.4 and Section 5.4).

7.2 Structural Mutation in Fuzzing Techniques

Structural mutation was developed for effective fuzzing for simply structured file inputs (to structured file inputs of high complexity, grammar-based fuzzing [17, 32, 39] are applied). It receives a dictionary containing tokens provided by users or automatically extracted from source code and/or documents of a target program. Structural mutation adds/deletes/replaces a token to effectively generate test inputs that satisfy the input constraints of a target program. Zest [31] performs structural mutation by utilizing manually written *parametric generators* which convert a byte sequence into a structured input such as a XML format file. Yoo et al. [44] utilizes grammar-aware mutation operators for effective continuous unit-level fuzzing. Superion [39] improves AFL's dictionary-based mutation to align with their grammar-aware fuzzing.

The main difference between ZigZagFuzz and the above fuzzers is that ZigZagFuzz applies both structural mutation and byte-level mutation to POI while Zest and Superion applies structural mutation to FI without recognizing the importance of diverse POI.

7.3 Heuristics to Select Test Inputs

AFLfast [2] favors inputs that execute rarely executed paths. FairFuzz [21] and Vuzzer [34] favor inputs which execute rarely executed branches and which execute basic blocks located in deep control-structure, respectively. CollAFL [13] favors inputs whose execution paths have many uncovered neighbor branches. Ankou [26] defines a distance between two different execution paths and scores each input according to its execution path's "uniqueness" which is measured using the distances to other paths. TortoiseFuzz [41] favors inputs which execute many functions, loops, and basic blocks that have many memory access operators. SAVIOR [5] statically labels suspicious basic blocks which contain (or which can reach) operators that can lead to undefined behaviors and it scores each input in terms of a number of the suspicious basic blocks visited by the test input.

While the aforementioned selection/prioritization heuristics of these fuzzers consider only FI (not POI), ZigZagFuzz selects test inputs based on both POI and FI and, thus, improves bug detection ability further (see Section 3.6 and Section 5.3).

8 CONCLUSION

This paper presents a novel fuzzing technique ZigZagFuzz, which improves bug detection ability by separately fuzzing file input and program option input in an interleaving manner. We have applied ZigZagFuzz to the 20 popular real-world subjects and confirmed that ZigZagFuzz significantly outperforms the state-of-the-art fuzzing techniques (i.e., ZigZagFuzz detected 1.9 to 10.6 times more unique bugs than the compared cutting-edge fuzzers). Also, we have demonstrated that the core ideas of ZigZagFuzz (i.e., different fuzzing strategies for different input domains, interleaving phases of mutating program option input with ones of mutating file input, domain-wise corpus shrinking by reducing POIs and FIs separately, and applying both structural and random byte mutations to POIs) are effective to improve fuzzing performance.

For future work, we will apply the key ideas of ZigZagFuzz to not only programs with program option input, but also to programs with configurations of different types such as build configurations. Also, we observed that different fuzzing techniques detected different sets of unique bugs as shown in Figure 5 in Section 5.1. We plan to study the differences between the unique bugs detected by each fuzzer, which can help improve fuzzing effectiveness.

ACKNOWLEDGMENTS

This research was supported by SW Research and Development Program (No. UC210018AD), the National Research Foundation grants funded by the Korea government (NRF-2021R1A5A1021944, NRF-RS-2023-00253977,

NRF-RS-2024-00357348, and NRF-2020R1C1C1013512), IITP grant (No. 2021-0-00905-001) funded by the Korea government (MSIT), research fund of Hanyang University (HY-2020), and Samsung Electronics.

REFERENCES

[1] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Symposium on Network and Distributed System Security (NDSS)*.

[2] 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* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1032–1043. https://doi.org/10.1145/2976749.2978428

[3] M. Böhme, V. Pham, and A. Roychoudhury. 2019. Coverage-Based Greybox Fuzzing as Markov Chain. *IEEE Transactions on Software Engineering* 45, 5 (2019), 489–506.

[4] P. Chen and H. Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP)*. 711–725.

[5] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu. 2020. SAVIOR: Towards Bug-Driven Hybrid Testing. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 2–2. https://doi.org/10.1109/SP.2020.00002

[6] Yaohui Chen, Dongliang Mu, Jun Xu, Zhichuang Sun, Wenbo Shen, Xinyu Xing, Long Lu, and Bing Mao. 2019. PTrix: Efficient hardware-assisted fuzzing for COTS binary. In *AsiaCCS 2019 - Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security (AsiaCCS 2019 - Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security)*. Association for Computing Machinery, Inc, 633–645. https://doi.org/10.1145/3321705.3329828

[7] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. 2019. Grey-box Concolic Testing on Binary Code. In *Proceedings of the International Conference on Software Engineering*. 736–747.

[8] N. Coppik, O. Schwahn, and N. Suri. 2019. MemFuzz: Using Memory Accesses to Guide Fuzzing. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 48–58.

[9] S. Dinesh, Nathan Burow, D. Xu, and M. Payer. 2020. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. *2020 IEEE Symposium on Security and Privacy (SP)* (2020), 1497–1511.

[10] Andrea Fioraldi, Daniele Cono D'Elia, and Davide Balzarotti. 2021. The Use of Likely Invariants as Feedback for Fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2829–2846. https://www.usenix.org/conference/usenixsecurity21/presentation/fioraldi

[11] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi

[12] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. GREYONE: Data Flow Sensitive Fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/usenixsecurity20/presentation/gan

[13] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen. 2018. CollAFL: Path Sensitive Fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, Vol. 00. 660–677. https://doi.org/10.1109/SP.2018.00040

[14] Google. 2016. OSS-Fuzz. https://google.github.io/oss-fuzz/. Accessed: 2021-10-03.

[15] Abhilash Gupta, Rahul Gopinath, and Andreas Zeller. 2022. CLIFuzzer: Mining Grammars for Command-Line Invocations. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 1667–1671. https://doi.org/10.1145/3540250.3558918

[16] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking. 2021. Seed Selection for Successful Fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 230–243. https://doi.org/10.1145/3460319.3464795

[17] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *21st USENIX Security Symposium (USENIX Security 12)*. USENIX Association, Bellevue, WA, 445–458. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler

[18] Yunho Kim, Yunja Choi, and Moonzoo Kim. 2018. Precise Concolic Unit Testing of C Programs Using Extended Units and Symbolic Alarm Filtering *(International Conference on Software Engineering (ICSE))*. 315–326.

[19] Ahcheong Lee, Irfan Ariq, Yunho Kim, and Moonzoo Kim. 2022. POWER: Program option-aware fuzzer for high bug detection ability. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 220–231.

[20] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: Automatically Generating Pathological Inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 254–265. https://doi.org/10.1145/3213846.3213874

[21] 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* (Montpellier, France) *(ASE 2018)*. Association for Computing Machinery, New York, NY, USA, 475–485. https://doi.org/10.1145/3238147.3238176

[22] Xumei Li, Lei Sun, Ruobing Jiang, Haipeng Qu, and Zhen Yan. 2021. OTA: An Operation-oriented Time Allocation Strategy for Greybox Fuzzing. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 108–118. https://doi.org/10.1109/SANER50967.2021.00019

[23] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, Kangjie Lu, and Ting Wang. 2020. UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers. arXiv:2010.01785 [cs.CR]

[24] Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. 2019. Cerebro: Context-Aware Adaptive Fuzzing for Effective Vulnerability Detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 533–544. https://doi.org/10.1145/3338906.3338975

[25] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1949–1966. https://www.usenix.org/conference/usenixsecurity19/presentation/lyu

[26] Valentin JM Manès, Soomin Kim, and Sang Kil Cha. 2020. Ankou: Guiding Grey-box Fuzzing towards Combinatorial Difference. (2020).

[27] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* (2019).

[28] Jonathan Metzman, Abhishek Arya, and Laszlo Szekeres. 2020. FuzzBench: Fuzzer Benchmarking as a Service. https://security.googleblog.com/2020/03/fuzzbench-fuzzer-benchmarking-as-service.html

[29] 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)*. 787–802. https://doi.org/10.1109/SP.2019.00069

[30] Stefan Nagy, Anh Nguyen-Tuong, Jason D. Hiser, Jack W. Davidson, and Matthew Hicks. 2021. Breaking Through Binaries: Compiler-quality Instrumentation for Better Binary-only Fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 1683–1700. https://www.usenix.org/conference/usenixsecurity21/presentation/nagy

[31] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 329–340. https://doi.org/10.1145/3293882.3330576

[32] Thuan Pham, Marcel Boehme, Andrew Santosa, Alexandru Caciulescu, and Abhik Roychoudhury. 2019. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering* PP (09 2019), 1–1. https://doi.org/10.1109/TSE.2019.2941681

[33] V. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury. 2019. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering* (2019). https://doi.org/10.1109/TSE.2019.2941681

[34] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing.. In *NDSS*, Vol. 17. 1–14.

[35] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker *(USENIX ATC)*.

[36] Mushfeq-Us-Saleheen Shameem and Raihana Ferdous. 2009. An efficient k-means algorithm integrated with Jaccard distance measure for document clustering. In *2009 First Asian Himalayas International Conference on Internet*. 1–6. https://doi.org/10.1109/AHICI.2009.5340335

[37] Bowen Wang, Kangjie Lu, Qiushi Wu, and Aditya Pakki. 2021. Unleashing Fuzzing Through Comprehensive, Efficient, and Faithful Exploitable-Bug Exposing. *IEEE Transactions on Dependable and Secure Computing* (2021), 1–1. https://doi.org/10.1109/TDSC.2021.3079857

[38] Dawei Wang, Ying Li, Zhiyu Zhang, and Kai Chen. 2023. CarpetFuzz: Automatic Program Option Constraint Extraction from Documentation for Fuzzing. In *Proceedings of the 32nd USENIX Conference on Security Symposium*. USENIX Association, Anaheim, CA, USA.

[39] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-Aware Greybox Fuzzing. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 724–735. https://doi.org/10.1109/ICSE.2019.00081

[40] Xiajing Wang, Changzhen Hu, Rui Ma, Donghai Tian, and Jinyuan He. 2021. CMFuzz: context-aware adaptive mutation for fuzzers. *Empirical Software Engineering* 26 (01 2021). https://doi.org/10.1007/s10664-020-09927-3

[41] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. 2020. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization. In *Symposium on Network and Distributed System Security (NDSS)*. https://doi.org/10.14722/ndss.2020.24422

[42] Zi Wang, Ben Liblit, and Thomas Reps. 2020. TOFU: Target-Oriented FUzzer. arXiv:2004.14375 [cs.SE]

[43] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. 2020. MemLock: Memory Usage Guided Fuzzing. In *2020 IEEE/ACM 42nd International Conference on Software Engineering*. Seoul, South Korea.

[44] Hanyoung Yoo, Jingun Hong, Lucas Bader, Dong Won Hwang, and Shin Hong. 2021. Improving Configurability of Unit-level Continuous Fuzzing: An Industrial Case Study with SAP HANA. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1101–1105. https://doi.org/10.1109/ASE51524.2021.9678685

[45] T. Yue, Y. Tang, B. Yu, P. Wang, and E. Wang. 2019. LearnAFL: Greybox Fuzzing With Knowledge Enhancement. *IEEE Access* 7 (2019), 117029–117043.

[46] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. 2020. EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2307–2324. https://www.usenix.org/conference/usenixsecurity20/presentation/yue

[47] Andreas Zeller. 2022. Testing Configurations - The Fuzzing Book. https://www.fuzzingbook.org/html/ConfigurationFuzzer.html. Accessed: 2020-10-13.

[48] Zenong Zhang, George Klees, Eric Wang, Michael Hicks, and Shiyi Wei. 2023. Fuzzing Configurations of Program Options. *ACM Trans. Softw. Eng. Methodol.* 32, 2, Article 53 (mar 2023), 21 pages. https://doi.org/10.1145/3580597

[49] Zhuo Zhang, Wei You, Guanhong Tao, Yousra Aafer, Xuwei Liu, and Xiangyu Zhang. 2021. StochFuzz: Sound and Cost-effective Fuzzing of Stripped Binaries by Incremental and Stochastic Rewriting. In *2021 IEEE Symposium on Security and Privacy (SP)*. 659–676. https://doi.org/10.1109/SP40001.2021.00109

[50] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. 2020. FuzzGuard: Filtering out Unreachable Inputs in Directed Grey-box Fuzzing through Deep Learning. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2255–2269. https://www.usenix.org/conference/usenixsecurity20/presentation/zong