POWER: Program Option-Aware Fuzzer for High Bug Detection Ability

Ahcheong Lee
School of Computing
KAIST
ahcheong.lee@kaist.ac.kr

Irfan Ariq
School of Computing
KAIST
irfanariqzaki@gmail.com

Yunho Kim
Department of Computer Science
Hanyang University
yunhokim@hanyang.ac.kr

Moonzoo Kim
School of Computing, KAIST
VPlusLab Inc.
moonzoo.kim@gmail.com

***Abstract*—Most programs with command-line interface (CLI) have dozens of command-line options (e.g., `-l`, `-F`, `-R` for `ls`) to alternate the operation of the programs. Thus, depending on the option configurations (i.e., a list of options like `-l -F` and `-F -R`) applied during fuzzing, the test coverage and crash detection results can vary significantly.**

In this paper, we propose a novel fuzzing technique POWER that detects more crashes than the cutting-edge fuzzers by actively constructing and carefully selecting various program option configurations. The salient idea of POWER is to enforce diverse executions of a target program by selecting a set of the option configurations each of which is far "different/distant" from the others in the set. Another core idea of POWER is to apply different fuzzing strategies to different input domains (i.e., option configurations and input files) to increase testing effectiveness within limited time budget. The experiment results on the 30 real-world programs show that POWER detects significantly more crash bugs than the state-of-the-art fuzzing techniques.

***Index Terms*—Automated test generation, fuzzing, program option configurations, dynamic function relevance, crash bug detection, dynamic analysis**

I. Introduction

Initial configurations of software applications can affect the behaviors of the applications in a large degree. For example, most programs with command-line interface (CLI) have dozens of command-line options to alternate the operations of programs (e.g., `ls` has more than 50 options including `-a`, `-F`, `-l`, `-n`, and `-R` [1]). In other words, program options play a crucial role in determining the target program's execution paths. Thus, when we apply fuzzing to a program with CLI, the crash detection results can vary significantly depending on which 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 an *option configuration* (i.e., a list of options given to a target program such as `-a -l -R` for `ls`) can be a huge determining factor for the effectiveness of fuzzing, most fuzzing papers have utilized only a single option configuration in their fuzzing experiments. According to the survey of the recently published 98 fuzzing papers (see Section V-A for the details), 76.5% (=(11+64)/98) of the fuzzing papers did not provide information on the option configurations in the papers. Thus, there exists large room to improve fuzzing effectiveness by systematically utilizing various option configurations.

In this paper, we propose a novel fuzzing technique POWER (Program Option-aWarE fuzzeR) that detects more crash bugs than the cutting-edge fuzzers by actively constructing and carefully selecting diverse option configurations together with conventional input file fuzzing. The salient core ideas of POWER are as follows:

1) *Different Search Strategies for Different Input Domains*: In contrast to the most fuzzers that focus and mutate only input files to a target program, POWER considers that a target program has *two different input domains* to explore (i.e., option configurations and input files). Thus, it applies *two distinct search strategies* to them for high bug detection ability within limited time budget.
   For example, POWER constructs various option configurations (e.g., `-debug -rev -num 10`) by systematically combining option keywords in the option dictionary (e.g., { `-debug`, `-num <m>`, `-rev`, `-str`, ...}) *only* for the first hour while it generates diverse input files with the various option configurations (which were constructed and selected in the previous one hour) by mutating input files in byte-level for 23 hours like conventional fuzzing.
2) *Careful Selection of Diverse Option Configurations*: To enforce diverse executions of a target program within limited time budget, after constructing various option configurations, POWER selects a set of the option configurations each of which is far "different/distant" from the others in the set (see Section II-C). This is because the set of far different option configurations can enforce a target program to execute diverse execution paths within limited time budget since an option configuration guides the target program executions in a large degree.
   For example, suppose that the executions of a target program P with an option configuration o_1 cover a set of functions $\{main, f_1\}$. Also suppose that the executions of P with another option configuration o_2 cover $\{main, f_2\}$ and the executions of P with o_3 cover $\{main, f_3\}$. Roughly speaking, o_1 is more different/distant from o_2 than o_3 if f_1 is less relevant to f_2 than f_3.

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

Based on the above two core ideas, POWER operates in the following three stages in order:

1) *Exploratory stage*: POWER actively constructs option configurations as well as input files for one hour; it semi-automatically extracts a set of options from the documents of a target program and constructs various option configurations by using a dictionary-based construction method [1], [2].
2) *Option configuration selection stage*: From the various option configurations generated in the exploratory stage, it selects a set of far "different/distant" option configurations based on the option configuration relevance metric (see Section II-C), with which POWER will generate diverse input files in the next main fuzzing stage.
3) *Main fuzzing stage*: For the remaining 23 hours, POWER fuzzes only input files with the set of the option configurations selected during the option configuration selection stage (note that this stage does not mutate option configurations at all).

To demonstrate the advantages of POWER, we have applied POWER to the 30 real-world programs. The experiment results show that POWER detects twice more unique crashes on the subject programs than the state-of-the-art fuzzers such as AFL++ [3] with ten option configurations and Eclipser [4].

The main contributions of this paper are as follows:

1) POWER is the first fuzzing technique that can detect many crash bugs by actively constructing and carefully selecting far different option configurations based on the new option configuration relevance metric (Section II-C).
2) We have performed a series of the experiments where we have empirically evaluated POWER and other cutting-edge fuzzers (i.e., AFL++ and Eclipser) and demonstrated that POWER detects significantly more unique crashes than the cutting-edge fuzzers (Section IV).
3) After detecting unique crashes in the subject programs, we have reported 51 new crash bugs detected by POWER to the original developers of the target subject programs to improve the quality of the open source subject programs. [2]

The remaining sections are organized as follows. Section II explains the three stages of POWER in detail. Section III describes the experiment design and setup. Section IV discusses the experiment results. Section V describes related work. Finally, Section VI concludes this paper with future work.

II. Program Option-Aware Fuzzer(POWER)

A. Overall Process

Figure 1 shows the overall process of POWER (Program Option-aWare fuzzER). Initially, POWER receives the following items (see the left side of Figure 1):

- a target program P
- a set of initial test inputs T_{init} for P each of which consists of
  - an initial option configuration, and
  - an initial input file
- a set of documents Doc_P for P such as a man page and help messages

POWER consists of the following three stages:

1) *Exploratory stage* (Section II-B): For the first one hour of the entire fuzzing process, POWER actively constructs various option configurations using a dictionary-based mutation method while fuzzing input files using conventional byte-level mutation.
2) *Option configuration selection stage* (Section II-C): Among all option configurations generated in the exploratory stage, POWER selects a set of the option configurations each of which is far "different/distant" from the others based on the option configuration relevance metric (Section II-C3). In other words, POWER selects a set of the option configurations with which POWER can enforce diverse executions of a target program.
3) *Main fuzzing stage* (Section II-D): Using the option configurations selected in the option configuration selection stage, POWER mutates and generates diverse input files (not option configurations) for P.

B. Exploratory Stage

The left part of Figure 1 illustrates the exploratory stage. Algorithm 1 describes how POWER operates in the exploratory stage. First, for a target program P, POWER semi-automatically extracts a set of available program options OPT_P from the documents of P such as its man page and help messages (line 3). Then, POWER executes P with the initial inputs by using `RunTest` (lines 4–6). `RunTest` executes P with $input$ and adds $input$ to the input priority queue $PQUEUE$ if the execution increases path coverage (lines 21–26).

Next, POWER selects an input t that has the highest priority in $PQUEUE$ (line 8). Then, it decreases the priority of t by one (line 9) so to give higher priority to the inputs newly generated from t later. Then, POWER generates two inputs t' and t'' from t as follows and executes P with t' and P with t'' by using `RunTest` (line 13 and line 16, respectively).

- t' is a new input obtained by mutating the option configuration of t (lines 11-12). To mutate option configurations, POWER applies *dictionary-based mutation* [1], [2]. When POWER mutates an option configuration, it performs the following mutation operations (`MutateOptConf` in line 11 in Algorithm 1):
  - insert a random number of random options in OPT_P into random location(s) of the option configuration, or
  - replace a random number of options in the option configuration with random options in OPT_P, or
  - remove a random number of the options in the option configuration

[2]We reported 51 out of the 88 crash bugs detected by POWER. To reduce the original developer's burden to review many crash reports, we checked if the crashes detected on the latest release version can be still replicated on the latest development version and submitted only such crash reports.

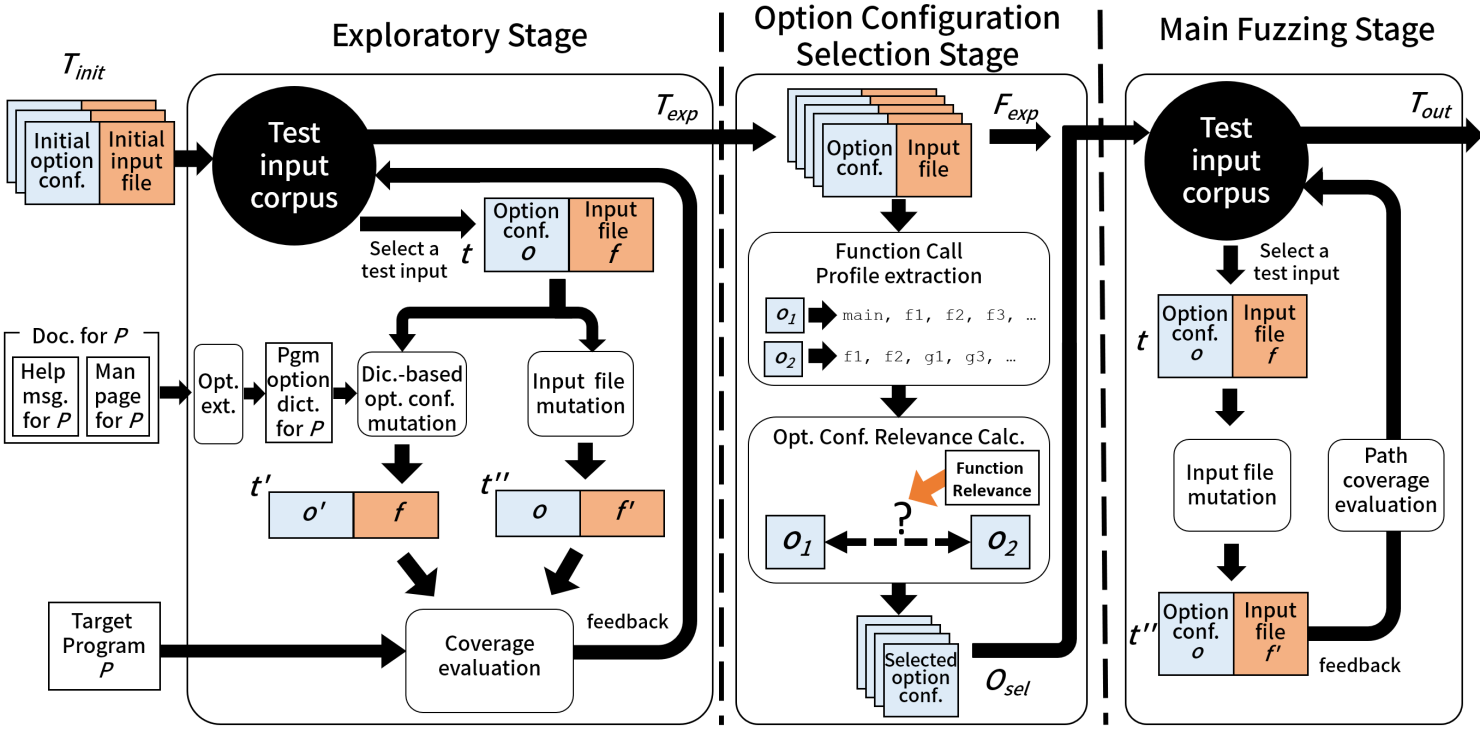


Fig. 1. Overall process of POWER in the three stages

- t'' is obtained by mutating the input file of t in a similar way to other fuzzing techniques (e.g., common byte-level mutations such as bitflip, byte-level random arithmetic addition/subtraction, and byte random replacement) (`MutateFile` in line 14)

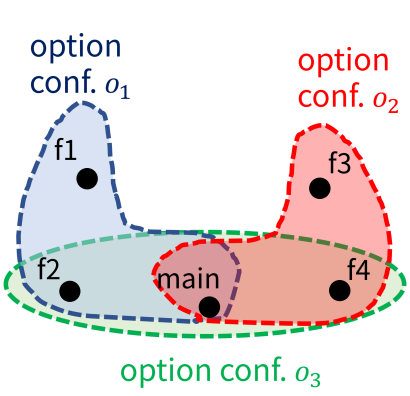
POWER repeats the above steps to generate new inputs by mutating option configurations and input files within the exploratory stage timeout (lines 7–17).

C. Option Configuration Selection Stage

The middle part of Figure 1 illustrates the option configuration selection stage. POWER selects a set of the option configurations each which is far *"different/distant"* from the others in the set based on the *option configuration relevance metric*. In other words, POWER selects a set of the option configurations with which POWER can enforce far different executions of a target program.

1) Example: How to Select Option Configurations

Figure 2 shows an example to show how POWER selects a set of the option configurations each of which is far different/distant from the others. The three dotted shapes (a left blue one, a right red one, and a bottom green one) in Figure 2(a) represent the executions of a target program P (which has the functions $main$, $f1$, $f2$, $f3$, and $f4$) with three option configurations o_1, o_2, and o_3, respectively. The left blue dotted shape contains $\{main, f1, f2\}$, which indicates that the executions of P with o_1 cover $\{main, f1, f2\}$ (simply calling that o_1 covers $\{main, f1, f2\}$). Similarly, o_2 covers $\{main, f3, f4\}$ and o_3 covers $\{main, f2, f4\}$.



(a)

Func. Relevance		
f1	f2	High
f1	f3	Low
f1	f4	Low
f1	main	High
f2	f3	Low
f2	f4	Low
f2	main	High
f3	main	Low
f3	f4	High
f4	main	High
…		

(b)

Fig. 2. (a) An example to explain relevance between option configurations (b) Function relevance table.

POWER identifies far "different/distant" option configuration pairs based on the *option configuration relevance* metric (Section II-C3), which is calculated from the *dynamic function relevance* (Section II-C2) values between functions based on the execution profile of P. Intuitively speaking, if two functions f_i and f_j are executed together frequently in many executions, f_i and f_j are highly relevant. The table in Figure 2(b) shows function relevance values between the functions of P (e.g., $f1$ and $f2$ are highly relevant while $f1$ and $f3$ are not highly relevant).

Option configuration relevance between o_i and o_j is defined as an average of the function relevance values between all

Algorithm 1: Exploratory Stage

Input: P : a target program, T_{init} : a set of initial inputs for P, and Doc_P: documents for P (i.e., a man page or help messages)

Output: T_{exp} : a set of generated test inputs in the exploratory stage

Function `ExploratoryStage`(P, T_{init}, Doc_P)**:**
  $PQUEUE \leftarrow \emptyset$
  $OPT_P \leftarrow$ program options extracted from Doc_P
  foreach $input \in T_{init}$ **do**
    `RunTest`$(P, input)$
  end
  while a given timeout is not reached **do**
    Select an input t that has the highest priority from $PQUEUE$
    Decrease the priority of t by 1
    $(o, f) \leftarrow t$
    $o' \leftarrow$ `MutateOptConf`(o, OPT_P)
    $t' \leftarrow (o', f)$
    `RunTest`(P, t')
    $f' \leftarrow$ `MutateFile`(f)
    $t'' \leftarrow (o, f')$
    `RunTest`(P, t'')
  end
  $T_{exp} \leftarrow PQUEUE$
  return T_{exp}
End Function

Function `RunTest`$(P, input)$**:**
  Execute P with $input$
  if a new path is covered **then**
    $PQUEUE.append(input)$
  end
End Function

Algorithm 2: Option Configuration Selection Stage

Input: P : a target program, T_{exp}: a set of option configuration and input file pairs that increased path coverage in the exploratory stage

Output: $O_{distinct}$: a set of selected option configurations

Function `OptConfSelStage`(P, T_{exp})**:**
  $O_{exp} \leftarrow \emptyset$ //a set of all option configurations in T_{exp}
  $F_{exp} \leftarrow \emptyset$ //a set of all input files in T_{exp}
  foreach $(o, f) \in T_{exp}$ **do**
    $O_{exp}.add(o)$
    $F_{exp}.add(f)$
  end
  $Calls \leftarrow$ `GetFuncCalls`$(P, T_{exp}, O_{exp}, F_{exp})$
  $O_{sel} \leftarrow$ `SelectOptConfSet`$(O_{exp}, Calls)$
  return $O_{distinct}$
End Function
Function `GetFuncCalls`$(P, T_{exp}, O_{exp}, F_{exp})$**:**
  $Calls \leftarrow$ *empty map*
  foreach $o \in O_{exp}$ **do**
    $Calls[o] \leftarrow \emptyset$
    foreach $f \in F_{exp}$ *s.t.* $(o, f) \in T_{exp}$ **do**
      $CalledFuncs \leftarrow$ `ExecuteAndGetCalls`(P, o, f)
      foreach $func \in CalledFuncs$ **do**
        $Calls[o].add(func)$
      end
    end
  end
  return $Calls$
End Function
Function `SelectOptConfSet`$(O_{exp}, Calls, \tau)$**:**
  $O_{sel} \leftarrow \emptyset$ // a set of the selected option configurations
  $(o_1, o_2) \leftarrow$ a pair of option configurations $\in O_{exp}$ that has the minimum option relevance among the all pairs of option configurations
  $O_{sel}.add(o_1, o_2)$
  foreach $o \in O_{exp}$ **do**
    if $\forall o_d \in O_{sel}.OptRel_{Calls}(o, o_d) < \tau$ **then**
      $O_{sel}.add(o)$
    end
  end
  return O_{sel}
End Function

functions covered by o_i and o_j (Section II-C3). For example, the relevance between o_1 and o_2 is the average of the function relevance values between the functions covered by o_1 and o_2 (i.e., the average of the function relevance values of ($f1$,$f3$), ($f1$,$f4$), ($f1$, $main$), ($f2$,$f3$), ($f2$, $f4$), ($f2$,$main$), ($main$,$f3$), and ($main$, $f4$)).

Note that o_1 and o_2 have low relevance because more pairs of the functions covered by o_1 and o_2 have low function relevance values as shown in Figure 2(b) (i.e., five out of the above eight function pairs have low function relevance values (i.e., ($f1$,$f3$), ($f1$,$f4$), ($f2$,$f3$), ($f2$, $f4$), ($main$,$f3$),($main$, $f4$)). In contrast, o_1 and o_3 are highly relevant because more pairs of the functions covered by o_1 and o_3 (i.e., four out of the six pairs) have high relevance values.

Thus, POWER selects the lowly related (i.e., far different) option configurations o_1 and o_2 to guide diverse executions of P.

2) Dynamic Function Relevance

Among the dozens of function relevance/coupling metrics (e.g., [5]–[10]), POWER uses *dynamic function relevance metric* for its intuitive characteristics and its very low runtime cost to calculate (the concept of the dynamic function relevance was originally proposed to reduce false alarms of

unit testing [11] [12]). POWER defines and applies dynamic function relevance as follows:

Definition 1. Let TI be a set of generated test inputs with unique path coverage. A dynamic function relevance $FR_{TI}(f,g) \in [0,1]$ between two functions f and g is defined as:

$$FR_{TI}(f,g) = \frac{|\{ti \in TI|\ ti \text{ that executes both } f \text{ and } g\}|^2}{\left(\begin{array}{c}|\{ti \in TI|\ ti \text{ that executes } f\}| \\ *\,|\{ti \in TI|\ ti \text{ that executes } g\}|\end{array}\right)}$$

We say g is *highly relevant* to f if $FR_{TI}(f,g)$ is high. Intuitively speaking, a high value of $FR_{TI}(f,g)$ means that f and g are frequently executed together with TI and it means that f may have high relevance with g. Note that the runtime overhead to calculate $FR_{TI}(f,g)$ is negligible, because $FR_{TI}(f,g)$ is calculated based on function call traces and counting the number of function calls in the traces is very cheap.

3) Option Configuration Relevance

Using the function relevance, we define relevance between two different option configurations as follows:

Definition 2. For two option configurations o_1 and o_2, let F_1 and F_2 be sets of functions covered by o_1 and o_2 (i.e., that are invoked in the set of the executions with o_1 and o_2), respectively. The option configuration relevance $OptRel(o_1,o_2) \in [0,1]$ is defined as :

$$OptRel(o_1,o_2) = \frac{\sum_{f_1 \in F_1}\sum_{f_2 \in F_2} FR_{TI}(f_1,f_2)}{|F_1| * |F_2|}$$

Intuitively speaking, the option configuration relevance of two option configurations o_1 and o_2 is the average of the function relevance values between the all functions invoked in the executions with o_1 and the all functions invoked in the executions with o_2. POWER selects option configuration pairs that have low relevance (i.e., option configurations that are "far different/distant" from each other), because such two option configurations that have low relevance enforce very different function call executions from each other, which can explore diverse executions of a target program.

4) Option Configuration Selection

Algorithm 2 describes how to select option configurations far different from each other. From the set of inputs T_{exp} (a set of pairs of option configurations and input files that increased path coverage) gathered from the exploratory stage, POWER selects option configurations as follows:

1) It gets a set of option configurations O_{exp} and a set of input files F_{exp} from T_{exp} (line 2-7).
2) `GetFuncCalls` (lines 8, 12-24) gets sets of functions that are called with each option configuration in O_{exp} by executing and extracting function call profiles.
3) `SelectOptConfSet` (lines 9, 25–35) computes relevance values between the all option configurations by using the function call profile information obtained by `GetFuncCalls`. Then, it selects a set of the option configurations each of which has low relevance to the others in the set (i.e. a set of the diverse option configurations with which a target program runs diverse executions paths) with respect to a user given threshold τ (see Section III-C).

D. Main Fuzzing Stage

The right part of Figure 1 illustrates the main fuzzing stage, which fuzzes only input files with the option configurations selected in the previous stage. In this stage, POWER operates like other fuzzing techniques except that it exercises various executions with *carefully selected far different option configurations* (Section II-C). As a result, POWER can explore much more diverse execution paths than other fuzzing techniques even with the multiple different option configurations (e.g., AFL++ with ten option configuration) and/or with continuously mutating option configurations (e.g., Ecliper) (Section IV-B).

E. Implementation

We have implemented POWER on top of AFL++ [3]. The core components of POWER including automated program option extraction, dictionary-based mutation of option configurations, option configuration selection strategy, option configuration execution interface for fuzz engine are implemented in additional 6,000 lines of C and C++ code.

III. Experiment Setup

A. Research Questions

RQ1. Fuzzing effectiveness of POWER compared to the state-of-the-art fuzzing techniques: To what extent does POWER achieve crash detection ability and branch coverage in 24 hours, compared to the state-of-the-art fuzzing techniques? For RQ1, we compare POWER with AFL++ [3] with ten initial option configurations. We modified AFL++ to accept multiple initial option configurations and make AFL++ continue fuzzing with the given multiple initial options (similar to the main fuzzing stage of POWER).

- *AFL++ [3]*: it is a fork of AFL [1], which integrates diverse features from fuzzing research such as AFLFast's power scheduling [13] and MOPT's mutation scheduling scheme [14]. We selected AFL++ because AFL++ shows the best performance on the fuzzbench service [15] provided by Google.

To make a fair comparison with POWER, we provide ten initial option configurations to AFL++ in the following way:

1) From the 97 fuzzing papers in the survey (Section V-A), if there exist option configurations that are used by other papers, we use the option configurations in the papers.
2) If we get only $n(< 10)$ option configurations from the papers, we randomly generate $10 - n$ option configurations with the same option dictionary used for POWER. We restrict the maximum number of options in each option configuration as ten because it is unlikely that

testers use an option configuration with more than ten options. In addition, we do not use option configurations that are not accepted by the target programs (i.e., that cause the target programs to terminate early with printing command-line usage messages). The full list of the option configurations we used is available at https://sites.google.com/view/power-icst2022.

Also, we compare POWER with Eclipser [4] that supports mutating both option configurations and input files.

- *Eclipser [4]*: We select Eclipser because, in our best knowledge, it is the only open-source state-of-the-art fuzzer that officially supports mutating both option configurations and input files. [3]

RQ2. Fuzzing effectiveness of the option configuration relevance based option configuration selection strategy of POWER: To what extent does the option configuration relevance values of the selection strategy of POWER affect crash detection ability and branch coverage achievement? For RQ2, we have developed a variant of POWER, POWER^{Rnd} which uses random option configuration relevance values in the option configuration selection stage.

RQ3. Fuzzing effectiveness of the explicit option configuration selection of POWER: To what extent does the option configuration selection strategy of POWER affect crash detection ability and coverage, compared to a variant of POWER, POWER^{KMO} that keeps mutating option configurations without selecting option configurations? In other words, POWER^{KMO} runs in the exploratory stage for the entire fuzzing time.

B. Target subjects

We have collected the latest release versions (as of September 1st, 2021) of the popular real-world C/C++ programs that have been used by other fuzzing papers. As like other fuzzing papers, we target the latest release version (not a development version) to avoid unnecessary confusion caused by frequent changes of target program code in a development version. If the latest release version is distributed earlier than two years ago, we used the latest development versions. Table I shows the information (the size and the number of available program options). The sizes of these subjects range from 2,920 LoC to 1,174,673 LoC (the average is 137,570 LoC). The numbers of the program options range from 10 to 760. We selected these real-world subjects with the following criteria:

- The subject should have at least ten program options.
- The subject should be actively maintained (i.e., the last commit of the subjects was made within around two year ago).

[3]We used Eclipser version '1.x' instead of the most recent version (v2.0) because Eclipser 2.0 does not mutate the option configurations anymore.

C. Fuzzing Setup

1) Timeout Setup

We ran AFL++, Eclipser, POWER, and the variants of POWER for 24 hours, which follows the guideline on evaluating fuzzers proposed by Klees et al. [16].

2) Control of Random Variance

To reduce the random variance in the experiment results, we repeated the same experiment ten times.

3) Testbed Setup

All the experiments were performed on our own cluster in which each node is equipped with AMD Ryzen 7 3800XT (4.3 Ghz) and 16GB RAM, running Ubuntu 18.04 64 bit version.

D. Measurement

1) Crash Bug Detection

To measure the crash bug detection ability of the fuzzing techniques, we report the number of the crashes detected by the fuzzing techniques. Among the various crash counting methods [17], we first used stack backtrace hashing which counts crashes with the same stack trace as one crash (the most widely used method). Then, we manually deduplicate those crashes with our best effort, since one unique crash bug can generate several different crash stack traces. We report the number of the crashes detected in any of the ten experiment runs.

2) Coverage Achievement

To measure the coverage achievement of each technique, we count the number of the covered branches obtained by `gcov` and report the average numbers of the covered branches over the ten experiment runs.

E. Initial Seed Setup

An initial seed consists of an initial option configuration and an initial input file. All detailed list of initial seed setup is uploaded at https://sites.google.com/view/power-icst2022.

1) Initial option configuration

We provide an initial option configuration for each subject as follows:

a) If the papers in the survey (Section V-A) provide an option configuration for the target program, we used it.
b) If we cannot find such one, we used the simplest option configuration that can be handled by the subject (e.g., '@@', '-i @@ -o /dev/null', ...)

2) Initial input file

We provide initial input files for each subject as follows:

a) If the papers in the survey (Section V-A) provide input files for the target program, we used them.
b) If we cannot find such one, we used example input file(s) in a subject repository or repositories of similar subjects (e.g., we can use an example input file in `pdftops` for `pdftohtml` and `pdftopng`).

TABLE I
TARGET SUBJECTS

Subjects	Package name	Size (LoC)	# prog. option	Subjects	Package name	Size (LoC)	# prog. option
avconv	libav-git-c464278	454,936	80	pdftohtml	poppler-21.07.0	38,111	32
bison	bison-3.7.6	54,423	54	pdftopng	xpdf-4.03	97,890	33
cflow	cflow-1.6	18,197	45	pdftops	xpdf-4.03	103,077	46
cjpeg	libjpeg-turbo-2.1.0	6,308	37	pngfix	libpng-1.6.37	7,020	15
djpeg	libjpeg-turbo-2.1.0	5,792	37	pspp	pspp-1.4.1	4,901	25
dwarfdump	libdwarf-20210528	83,545	48	readelf	binutils-2.36.1	74,789	169
exiv2	exiv2-0.27.4	33,417	79	size	binutils-2.36.1	436,055	19
ffmpeg	ffmpeg-N-103440-g2f0113be3f	774,186	230	tiff2pdf	libtiff-4.3.0	8,234	35
gm	GraphicsMagick-1.3.36	197,891	760	tiff2ps	libtiff-4.3.0	5,646	41
gs	ghostpdl-9.54.0	1,174,673	53	tiffinfo	libtiff-4.3.0	3,752	10
jasper	jasper-2.0.32	2,920	16	vim	vim-8.2.3113	296,916	54
mpg123	mpg123-1.28.2	11,298	123	xmlcatalog	libxml-2.9.12	2,609	27
mutool	mupdf-git-d00de0e	364,318	224	xmllint	libxml-2.9.12	11,285	94
nasm	nasm-2.15.05	70,903	33	xmlwf	libexpat-2.4.1	4,147	19
objdump	binutils-2.36.1	877,165	145	yara	yara-4.1.1	5,862	37

F. POWER configuration

We give one hour to the exploratory stage because it shows best performance during our experimental study. For the user-given threshold τ of the option configuration selection (Section II-C), we make POWER to adaptively use the average value of the maximum and minimum values of the option relevance as τ.

G. Threats to Validity

A threat to external validity is the representativeness of our target subjects. We expect that this threat is limited since we choose the target programs widely used by many fuzzing researchers. A threat to internal validity is possible bugs in the implementation of POWER. To control this threat, we have tested our implementation extensively.

IV. EXPERIMENT RESULTS

A. Summary of the Experiment Data

Table II shows the average length (i.e., the number of options) and the total number of option configurations selected and generated by POWER. For example, for avconv (on the third row), POWER generated 133.9 option configurations on average and each of the option configurations had 30.7 options on average. Among the 133.9 option configurations, POWER selects only 10.0 option configurations on average (each of these 10.0 selected option configurations has 17.2 options on average). On average, POWER selected 34.7% of the option configurations generated in the exploratory stage.

Table III and Table IV report the number of unique crashes detected and the number of branches covered by the fuzzing techniques on the 30 target subjects. All experiment data are publicly available at https://sites.google.com/view/power-icst2022.

TABLE II
THE AVERAGE LENGTH AND TOTAL NUMBER OF THE OPTION CONFIGURATIONS GENERATED AND SELECTED BY POWER

Targets	All option conf. generated by POWER		Option conf. selected by POWER		Selection Ratio
	# of opt. (length)	# of opt. conf.	# of opt. (length)	# of opt. conf.	
avconv	30.7	133.9	17.2	10.0	7.5%
bison	24.8	145.6	10.1	37.8	26.0%
cflow	24.1	202.5	28.9	20.4	10.1%
cjpeg	16.2	124.7	6.4	30.3	24.3%
djpeg	14.8	138.3	5.1	25.5	18.4%
dwarfdump	22.3	111.2	10.4	21.6	19.4%
exiv2	39.9	519.1	40.8	192.8	37.1%
ffmpeg	41.7	226.0	43.3	113.6	50.3%
gm	303.1	1277.3	281.7	57.1	4.5%
gs	25.0	54.7	15.5	4.0	7.3%
jasper	15.4	107.2	13.8	20.9	19.5%
mpg123	41.2	246.3	43.6	138.5	56.2%
mutool	47.6	147.7	53.7	44.9	30.4%
nasm	15.7	161.2	13.2	25.5	15.8%
objdump	33.3	243.0	51.7	81.4	33.5%
pdftohtml	11.2	83.8	6.2	14.0	16.7%
pdftopng	11.8	49.2	15.3	27.6	56.1%
pdftops	10.2	47.6	14.6	27.6	58.0%
pngfix	8.0	43.1	6.5	13.0	30.2%
pspp	12.2	132.3	12.1	99.5	75.2%
readelf	54.9	307.8	58.1	230.7	75.0%
size	8.4	85.8	11.3	20.5	23.9%
tiff2pdf	16.9	153.9	14.0	30.2	19.6%
tiff2ps	14.8	220.1	14.8	114.9	52.2%
tiffinfo	10.9	100.1	10.4	3.8	3.8%
vim	23.4	457.6	22.9	179.3	39.2%
xmlcatalog	17.7	253.1	18.1	107.2	42.4%
xmllint	51.9	1576.2	52.3	1523.8	96.7%
xmlwf	12.5	231.9	12.3	64.8	27.9%
yara	18.4	95.4	20.7	60.7	63.6%
Avg.	32.6	255.9	30.8	111.4	34.7%

B. RQ1. Fuzzing effectiveness of POWER compared to the state-of-the-art fuzzing techniques

The experiment results clearly show that POWER detects far more unique crashes than the other fuzzing techniques.

TABLE III
THE TOTAL NUMBER OF CRASHES DETECTED AND THE AVERAGE NUMBERS OF BRANCHES COVERED BY THE FUZZERS

Targets	Eclipser		AFL++ w/ 10 opt. conf.		POWER	
	#uniq. crash	#branch covered	#uniq. crash	#branch covered	#uniq. crash	#branch covered
avconv	0	8778.1	0	20226.7	5	15006.2
bison	0	1636.1	0	6526.2	5	6138.0
cflow	0	1792.8	0	1386.8	2	1675.3
cjpeg	0	2961.9	2	3119.7	0	4086.7
djpeg	0	438.5	0	2254.5	0	2513.7
dwarfdump	0	472.6	2	6259.9	2	7240.6
exiv2	0	4243.6	0	8082.2	1	9567.0
ffmpeg	0	18520.7	0	43967.0	2	45392.8
gm	0	2497.4	0	7636.2	1	9710.1
gs	0	13234.9	0	20495.9	0	24161.6
jasper	0	1855.6	0	2051.2	0	4101.0
mpg123	0	3202.1	0	2944.9	1	3809.3
mutool	0	13315.1	0	4076.0	0	13647.7
nasm	2	2150.0	0	6737.6	4	6506.6
objdump	0	5116.1	13	31327.6	13	33070.5
pdftohtml	0	1159.4	0	5997.7	4	7600.7
pdftopng	0	766.0	0	8404.7	9	8687.5
pdftops	0	763.1	0	9738.6	9	9354.9
pngfix	0	535.5	0	1166.8	0	1143.1
pspp	0	2935.0	4	6564.5	8	5650.0
readelf	0	520.3	2	10550.1	8	10321.6
size	0	3812.1	5	9078.9	3	9054.8
tiff2pdf	0	494.4	0	4133.0	0	4177.1
tiff2ps	0	898.1	0	3514.1	0	3379.0
tiffinfo	1	552.2	0	3509.4	4	3228.1
vim	0	27141.3	7	50842.5	5	45654.3
xmlcatalog	0	374.7	0	7607.0	0	7598.9
xmllint	0	4233.9	0	11132.5	2	14420.5
xmlwf	0	1984.3	0	3821.2	0	3733.8
yara	0	745.6	0	3002.1	0	3118.9
Total	3		35		88	

TABLE IV
THE TOTAL NUMBER OF CRASHES DETECTED AND THE AVERAGE NUMBERS OF BRANCHES COVERED BY THE VARIANTS OF POWER

Targets	POWER^{Rnd}		POWER^{KMO}		POWER	
	#uniq. crash	#branch covered	#uniq. crash	#branch covered	#uniq. crash	#branch covered
avconv	4	11709.6	7	17197.7	5	15006.2
bison	1	5728.0	3	6637.6	5	6138.0
cflow	3	1553.2	4	1689.1	2	1675.3
cjpeg	0	3920.1	0	4192.8	0	4086.7
djpeg	0	2598.1	0	2651.7	0	2513.7
dwarfdump	1	6565.5	4	7563.7	2	7240.6
exiv2	0	8679.3	0	9636.8	1	9567.0
ffmpeg	1	36252.6	1	48122.8	2	45392.8
gm	0	6492.3	0	9454.0	1	9710.1
gs	0	22586.2	1	24905.8	0	24161.6
jasper	0	3674.4	0	3660.1	0	4101.0
mpg123	0	3744.1	1	4006.3	1	3809.3
mutool	0	12423.8	0	15746.1	0	13647.7
nasm	4	6403.2	3	6578.8	4	6506.6
objdump	13	26237.9	8	24639.1	13	33070.5
pdftohtml	0	7184.0	0	8100.5	4	7600.7
pdftopng	0	7341.9	0	8947.8	9	8687.5
pdftops	0	8177.3	0	9719.0	9	9354.9
pngfix	0	1107.8	0	1191.2	0	1143.1
pspp	9	3389.2	7	4462.3	8	5650.0
readelf	0	9402.0	1	8799.3	8	10321.6
size	1	5078.7	4	7621.5	3	9054.8
tiff2pdf	0	4126.1	0	4226.8	0	4177.1
tiff2ps	1	2950.8	1	3274.1	0	3379.0
tiffinfo	1	2732.4	1	3060.9	4	3228.1
vim	0	39844.8	2	45466.5	5	45654.3
xmlcatalog	0	6598.8	0	6413.9	0	7598.9
xmllint	2	14245.7	2	14406.5	2	14420.5
xmlwf	0	3590.3	0	3733.0	0	3733.8
yara	0	3455.6	0	3954.2	0	3118.9
Total	41		50		88	

Table III show the number of unique crashes detected and the number of the branches covered by Eclipser, AFL++, and POWER. POWER detected 88 unique crashes on the 30 target programs, which is significantly more than the number of the unique crashes detected by the other state-of-the-art fuzzing techniques. In other words, POWER detects 29.3 (= 88/3) times more unique crashes than Eclipser and 2.51 (= 88/35) times more crashes than AFL++ with ten option configurations. Also, POWER covers more branches than the other techniques (i.e. POWER achieved 5.7% more branches than AFL++ on average with very low p-value(0.0011)). For example, on `exiv2`, POWER covered 2.3 (=9567.0/4243.6) times and 1.2 (= 9567.0/8082.2) times more branches than Eclipser and AFL++ respectively (see the ninth row in Table III).

C. RQ2. Fuzzing effectiveness of the option configuration relevance based option configuration selection strategy of POWER

From the experiment results, we can conclude that the option configuration selection strategy using the option configuration relevance contributes to significantly increase the testing effectiveness of POWER. Table IV show the number of the unique crashes detected and the branches covered by POWER and POWER^{Rnd}. POWER^{Rnd} uses random option configuration relevance values to select arbitrary option configurations. In total, POWER found 2.15 (= 88/41) times more crashes than POWER^{Rnd}, and POWER covered 12.1% more branch coverage than POWER^{Rnd} on average. For example, on `tiffinfo`, POWER found four times more crashes and covered 18.1% (=(3228.1-2732.4)/2732.4) more branches than POWER^{Rnd}.

D. RQ3. Fuzzing effectiveness of the explicit option configuration selection of POWER

The experiment results show that the explicit option configuration selection of POWER contribute to detect a large number of unique crashes. Table IV shows the number of the unique crashes detected and the branches covered by POWER and POWER^{KMO} where POWER^{KMO} keeps mutating option configurations for 24 hours *without* selecting option configurations. POWER detected 76% (=(88-50)/50) more crashes than POWER^{KMO} and covered similar number of branches on average. Thus, we can conclude that the explicit option configuration selection contributes to improve crash detection.

E. Real-world Crash Bugs Detected by POWER

For complex bugs that require specific combination of multiple options to trigger, POWER can successfully detect such bugs that have not been detected even after the extensive fuzzing effort.

For example, a new crash bug of `mpg123` is detected by POWER, but not by AFL++ with ten option configurations nor Eclipser. This is because the bug requires a specific option configuration to trigger. POWER detects the bug by generating an option configuration with 12 different command-line options [4]. 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 POWER because although `mpg123` had been extensively fuzzed by using Google's OSS-fuzz [18], the reported bug was not detected before (the exact message of the developer is "Interesting approach you find stuff where oss-fuzz didn't anymore."). This demonstrates that POWER can detect many crash bugs in practice by systematically constructing and carefully selecting option configurations by using option configuration relevance.

For another example, POWER detected a new crash of `vim` (but not by AFL++ with ten option configurations nor Eclipser). The option configuration used to detect the crash consists of 19 options. [5] The developers of `vim` responded to us as follows: "thanks for fuzzing and finding those bugs. Out of curiosity, which fuzzer did you use? I hope you continue fuzzing vim to find more bugs". The bug report is available at https://github.com/vim/vim/issues/8955

V. Related Work

A. Survey of Fuzzing Papers

While command-line options can largely affect program behaviors, research communities pay little attention to fuzzing command-line option configurations. To find out how many papers *explicitly* utilize program option configurations in their experiments, we have surveyed 98 fuzzing papers that (1) were published recently (from 2015 to 2021) on top conferences and journals of software engineering and security, and (2) targeted CLI programs. From the survey, we have found that

1) three papers [4], [13], [19] directly mutate option configurations in their experiments.
2) 20 papers specify the option configurations used in their experiments (e.g., [14], [20]–[26]).
3) 11 papers [27]–[37] did not specify the option configurations (but implicitly exposed their option configurations through publicly available experiment data)
4) 64 papers failed to specify program option configurations used (e.g. [38]–[44]).

In summary, 76.5% (=(11+64)/98) of the recently-published fuzzing papers did not provide information on the program option configurations used. Moreover, most of the above papers (except [4], [13], [19]) use only one fixed option configuration for their experiments.

[4] --smooth --listentry -z -w 1 --quiet --index - -4to1 -2 -q --fifo --outfile

[5] -o1 -b -S -y --noplugin -O2 -E -i --startuptime -A --ttyfail NONE -u 'input file' -S -R -o2 + -V1

B. Fuzzing Techniques to Directly Mutate Option Configurations

TOFU [19] is a fuzzer that mutates command-line option configurations for directed fuzzing. It generates many different option configurations by using dictionary-based mutation and tries to find an option configuration that gives the closest distance to a target basic block. TOFU receives a specification of command-line options (i.e., the name of options and the type of option argument if any) from a user and performs a dictionary-based mutation on the command-line option configurations by using the specification as a dictionary. Unlike TOFU, POWER *automatically* extracts the specification of command-line options from the man page and the help messages of target programs. Also, POWER actively generates diverse option configurations with accompanying input files to explore large path space while TOFU mutates option configurations only until it finds a path to a target block.

Zeller et al. [45] (an online course, not a published paper) 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. However, unlike POWER that generates both option configurations and accompanying input files together and evaluates/selects far different/distant option configurations, they did not evaluate the generated option configurations.

Eclipser [4] also 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 on not only input file bytes but also on input option configuration's bytes. After tracking the relation, it searches for correct values of the related bytes with multiple executions to cover the branches.

C. Dictionary-based Mutation in Fuzzing Techniques

Dictionary-based mutation was developed for effective fuzzing for simply structured input files (to complexly-structured input files, grammar-based fuzzing [2], [46], [47] are applied). The dictionary consists of tokens provided by users or automatically extracted from target programs' source code and/or documents. Dictionary-based mutation adds or deletes a token and mutates one token into another to effectively generate test inputs that satisfy the input constraints of the target programs. To guide fuzzing an input file, AFL [1] provides an API to use either a user-provided dictionary or an automatically extracted one. Superion [2] improves AFL's dictionary-based mutation to align with their grammar-aware fuzzing. The main difference between POWER and the above fuzzers is that POWER applies dictionary-based mutation to option configurations but AFL and Superion do to input files without recognizing the importance of diverse option configurations.

D. Prioritization Heuristics on Input Files

As POWER assigns high priority to the inputs containing far different/distant option configurations, other fuzzers apply various prioritization heuristics to increase coverage and crash detection power.

AFLfast [48] favors input files that execute rarely executed paths. FairFuzz [49] and Vuzzer [50] favor input files which execute rarely executed branches and which execute basic blocks located in deep control-structure, respectively. CollAFL [40] favors input files whose execution paths have many uncovered neighbor branches. Ankou [27] defines a distance between two different execution paths, and it scores each input file according to its execution path's "uniqueness" which is measured using the distances to other paths. TortoiseFuzz [35] favors input files which execute many functions, loops, and basic blocks that have many memory access operators. SAVIOR [51] statically labels suspicious basic blocks which contain (or which can reach) operators that can lead to undefined behaviors, and it scores each input file in terms of a number of the suspicious basic blocks visited by the test input.

Although the prioritization heuristics of these fuzzers consider only input files (not option configurations), POWER focuses on option configurations as well as input files/executions so to improve bug detection power further.

E. Testing Configurable Software Systems

A command-line option configuration can be considered as a kind of system configurations. There exist several methodologies that utilize combinatorial interation testing to test highly configurable systems [52]. Because a highly configurable system can produce a huge number of different products, the following papers estimated and priortized products to test by selecting corresponding configurations. A.B. Sánchez et al. [53] suggested methods to measure complexity of each product and assign a high priorty to highly complex products to detect faults as early as possible. Henard et al. [54] measured *similarity* between two different configurations and suggested to test distinct products before similar products.

VI. Conclusion and Future Work

This paper presents a novel fuzzing technique POWER, which improves bug detection ability of fuzzing by actively fuzzing and selecting option configurations as well as input files. The experiment results on the 30 popular real-world subjects confirm that POWER significantly outperforms the state-of-the-arts fuzzing techniques and the core ideas of POWER are effective to improve fuzzing performance.

As future work, we will consider constraints on command line option configurations (curretly POWER does not consider constraints on the options; it applies dictionary-based mutation to generate diverse option configurations). Also, we will apply mutation-based heuristics [55]–[57] with cost-effective mutation [58] and symbolic execution heuristics [59] to improve testing effectiveness of POWER.

Acknowledgments

We thank Robert Sebastian Herlim for the help on the command line survey of fuzzing papers. This work was supported by the NRF grant (NRF-2020R1C1C1013996, NRF-2021R1A2C2009384, NRF-2021R1A5A1021944) and the IITP grant (no. 2021-0-00905-001) funded by the Korea government (MSIT).

References

[1] M. Zalewski, "American fuzzy lop (afl) fuzzer," http://lcamtuf.coredump.cx/afl/, 2017.

[2] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," in *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 2019, p. 724–735. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00081

[3] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "Afl++ : Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/fioraldi

[4] J. Choi, J. Jang, C. Han, and S. K. Cha, "Grey-box concolic testing on binary code," in *Proceedings of the International Conference on Software Engineering*, 2019, pp. 736–747.

[5] S. Chidamber and C. Kemerer, "Towards a metrics suite for object oriented design," in *'OOPSLA*, 1991.

[6] W. Li and S. Henry, "Object-oriented metrics that predict maintainability," *Journal of Systems and Software*, vol. 23, no. 2, pp. 111 – 122, 1993. [Online]. Available: http://www.sciencedirect.com/science/article/pii/016412129390077B

[7] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering (TSE)*, vol. 20, no. 6, pp. 476–493, 1994.

[8] Y. Lee, B. Liang, S. Wu, and F. Wang, "Measuring coupling and cohesion of object-oriented programs based on information flow," in *International Conference on Software Quality (ICSQ)*, 1995.

[9] G. A. Hall, W. Tao, and J. C. Munson, "Measurement and validation of module coupling attributes," *Software Quality Journal*, vol. 13, no. 3, pp. 281–296, 2005.

[10] E. Arisholm, L. C. Briand, and A. Foyen, "Dynamic coupling measurement for object-oriented software," *IEEE Transactions on Software Engineering*, vol. 30, no. 8, pp. 491–506, 2004.

[11] Y. Kim, Y. Choi, and M. Kim, "Precise concolic unit testing of c programs using extended units and symbolic alarm filtering," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 315–326.

[12] Y. Kim, S. Hong, and M. Kim, "Target-driven compositional concolic testing with function summary refinement for effective bug detection," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 16–26.

[13] M. Böhme, V. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2019.

[14] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "MOPT: Optimized mutation scheduling for fuzzers," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1949–1966. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/lyu

[15] J. Metzman, A. Arya, and L. Szekeres, "FuzzBench: Fuzzer Benchmarking as a Service," March 2020. [Online]. Available: https://security.googleblog.com/2020/03/fuzzbench-fuzzer-benchmarking-as-service.html

[16] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2123–2138.

[17] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, 2019.

[18] "Oss-fuzz," https://google.github.io/oss-fuzz/, accessed: 2021-10-03.

[19] Z. Wang, B. Liblit, and T. Reps, "TOFU: Target-oriented fuzzer," 2020.

[20] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu, K. Lu, and X. Zhou, "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, Aug. 2020, pp. 2307–2324. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/yue

[21] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 711–725.

[22] T. Yue, Y. Tang, B. Yu, P. Wang, and E. Wang, "Learnafl: Greybox fuzzing with knowledge enhancement," *IEEE Access*, vol. 7, pp. 117 029–117 043, 2019.

[23] V. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering*, 2019.

[24] A. Fioraldi, D. C. D'Elia, and D. Balzarotti, "The use of likely invariants as feedback for fuzzers," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2829–2846. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/fioraldi

[25] X. Li, L. Sun, R. Jiang, H. Qu, and Z. Yan, "Ota: An operation-oriented time allocation strategy for greybox fuzzing," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021, pp. 108–118.

[26] B. Wang, K. Lu, Q. Wu, and A. Pakki, "Unleashing fuzzing through comprehensive, efficient, and faithful exploitable-bug exposing," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, 2021.

[27] V. J. Manès, S. Kim, and S. K. Cha, "Ankou: Guiding grey-box fuzzing towards combinatorial difference," in *Proceedings of the 42nd International Conference on Software Engineering*, 2020.

[28] S. Dinesh, N. Burow, D. Xu, and M. Payer, "Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization," *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 1497–1511, 2020.

[29] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence," in *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[30] P. Zong, T. Lv, D. Wang, Z. Deng, R. Liang, and K. Chen, "Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2255–2269. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/zong

[31] S. Nagy, A. Nguyen-Tuong, J. D. Hiser, J. W. Davidson, and M. Hicks, "Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1683–1700. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/nagy

[32] Y. Li, S. Ji, Y. Chen, S. Liang, W.-H. Lee, Y. Chen, C. Lyu, C. Wu, R. Beyah, P. Cheng, K. Lu, and T. Wang, "Unifuzz: A holistic and pragmatic metrics-driven platform for evaluating fuzzers," 2020.

[33] A. Herrera, H. Gunadi, S. Magrath, M. Norrish, M. Payer, and A. L. Hosking, "Seed selection for successful fuzzing," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2021, p. 230–243. [Online]. Available: https://doi.org/10.1145/3460319.3464795

[34] C. Wen, H. Wang, Y. Li, S. Qin, Y. Liu, Z. Xu, H. Chen, X. Xie, G. Pu, and T. Liu, "Memlock: Memory usage guided fuzzing," in *2020 IEEE/ACM 42nd International Conference on Software Engineering*, Seoul, South Korea, 2020.

[35] Y. Wang, X. Jia, Y. Liu, K. Zeng, T. Bao, D. Wu, and P. Su, "Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization," in *Symposium on Network and Distributed System Security (NDSS)*, 01 2020.

[36] S. Nagy and M. Hicks, "Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 787–802.

[37] Y. Chen, D. Mu, J. Xu, Z. Sun, W. Shen, X. Xing, L. Lu, and B. Mao, "Ptrix: Efficient hardware-assisted fuzzing for cots binary," in *AsiaCCS 2019 - Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. Association for Computing Machinery, Inc, 2019, pp. 633–645.

[38] N. Coppik, O. Schwahn, and N. Suri, "Memfuzz: Using memory accesses to guide fuzzing," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019, pp. 48–58.

[39] C. Lemieux, R. Padhye, K. Sen, and D. Song, "Perffuzz: Automatically generating pathological inputs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2018, p. 254–265. [Online]. Available: https://doi.org/10.1145/3213846.3213874

[40] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *2018 IEEE Symposium on Security and Privacy (SP)*, vol. 00, 2018, pp. 660–677. [Online]. Available: doi.ieeecomputersociety.org/10.1109/SP.2018.00040

[41] Y. Li, Y. Xue, H. Chen, X. Wu, C. Zhang, X. Xie, H. Wang, and Y. Liu, "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*. New York, NY, USA: Association for Computing Machinery, 2019, p. 533–544. [Online]. Available: https://doi.org/10.1145/3338906.3338975

[42] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, and Z. Chen, "GREYONE: Data flow sensitive fuzzing," in *29th USENIX Security Symposium (USENIX Security 20)*. Boston, MA: USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/gan

[43] Z. Zhang, W. You, G. Tao, Y. Aafer, X. Liu, and X. Zhang, "Stochfuzz: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 659–676.

[44] X. Wang, C. Hu, R. Ma, D. Tian, and J. He, "Cmfuzz: context-aware adaptive mutation for fuzzers," *Empirical Software Engineering*, vol. 26, 01 2021.

[45] "Testing configurations - the fuzzing book," https://www.fuzzingbook.org/html/ConfigurationFuzzer.html, accessed: 2020-10-13.

[46] T. Pham, M. Boehme, A. Santosa, A. Caciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering*, vol. PP, pp. 1–1, 09 2019.

[47] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX Association, Aug. 2012, pp. 445–458. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler

[48] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2016, p. 1032–1043. [Online]. Available: https://doi.org/10.1145/2976749.2978428

[49] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2018, p. 475–485. [Online]. Available: https://doi.org/10.1145/3238147.3238176

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

[51] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu, "Savior: Towards bug-driven hybrid testing," in *2020 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2020, pp. 2–2. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP.2020.00002

[52] I. do Carmo Machado, J. D. McGregor, Y. C. Cavalcanti, and E. S. de Almeida, "On strategies for testing software product lines: A systematic literature review," *Information and Software Technology*, vol. 56, no. 10, pp. 1183–1199, 2014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584914000834

[53] A. B. Sánchez, S. Segura, and A. Ruiz-Cortés, "A comparison of test case prioritization criteria for software product lines," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, 2014, pp. 41–50.

[54] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. L. Traon, "Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product

lines," *IEEE Transactions on Software Engineering*, vol. 40, no. 07, pp. 650–670, jul 2014.

[55] Y. Kim, S. Mun, S. Yoo, and M. Kim, "Precise learn-to-rank fault localization using dynamic and static features of target programs," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 4, oct 2019. [Online]. Available: https://doi.org/10.1145/3345628

[56] Y. Kim, S. Hong, B. Ko, D. L. Phan, and M. Kim, "Invasive software testing: Mutating target programs to diversify test exploration for high test coverage," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation*, April 2018.

[57] S. Hong, T. Kwak, B. Lee, Y. Jeon, B. Ko, Y. Kim, and M. Kim, "MUSEUM: Debugging real-world multilingual programs using mutation analysis," *Information and Software Technology (IST)*, vol. 82, pp. 80–95, Feb 2017.

[58] Y. Kim and S. Hong, "Learning-based mutant reduction using fine-grained mutation operators," *Software Testing, Verification and Reliability*, p. e1786. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1786

[59] Y. Kim, D. Lee, J. Baek, and M. Kim, "Concolic testing for high test coverage and reduced human effort in automotive industry," in *41st ACM/IEEE IEEE International Conference on Software Engineering*, 2019.