Registered Report: Fuzzing Configurations of Program Options

Zenong Zhang* George Klees† Eric Wang‡ Michael Hicks† Shiyi Wei*
*University of Texas at Dallas †University of Maryland ‡Poolesville High School

***Abstract*—While many real-world programs are shipped with configurations to enable/disable functionalities, fuzzers have mostly been applied to test single configurations of these programs. In this work, we first conduct an empirical study to understand how program configurations affect fuzzing performance. We find that limiting a campaign to a single configuration can result in failing to cover a significant amount of code. We also observe that different program configurations contribute differing amounts of code coverage, challenging the idea that each one can be efficiently fuzzed individually. Motivated by these two observations we propose ConfigFuzz, which can fuzz configurations along with normal inputs. ConfigFuzz transforms the target program to encode its program options within part of the fuzzable input, so existing fuzzers' mutation operators can be reused to fuzz program configurations. We instantiate ConfigFuzz on 3 configurable, common fuzzing targets, and integrate their executions in FuzzBench. In our preliminary evaluation, ConfigFuzz nearly always outperforms the baseline fuzzing of a single configuration, and in one target also outperforms the fuzzing of a sequence of sampled configurations. However, we find that sometimes fuzzing a sequence of sampled configurations, with shared seeds, improves on ConfigFuzz. We propose hypotheses and plan to use data visualization to further understand the behavior of ConfigFuzz, and refine it, in the full evaluation.**

I. Introduction

Fuzz testing has been successful at detecting security vulnerabilities in standalone programs. Such programs often have command-line options to enable/disable different functionalities; i.e., these options determine the *configuration* of the program code that is reachable at runtime. However, most fuzzers (and scientific evaluations thereof) only test a single program configuration (e.g., [7], [13]), and may therefore fail to properly test significant portions of a program's functionality. As a result, potential rare bugs may escape detection, and scientific evaluations of fuzzing performance may fail to account for the complete picture.

This paper considers the potential benefits of *configuration-aware* fuzzing. To start, we assess *how program configurations affect fuzzing performance* by performing an empirical study that ran AFL [2] on three common, configurable fuzzing targets (Section II). We found that fuzzing configurations with different option settings resulted in significant difference in code coverage, and some code could only be reached by a unique configuration. For example, about 35% lines reached when fuzzing FFmpeg [12] with sampled configurations were not reached by its default configuration. This result suggests a missed opportunity to achieve higher code coverage and/or to find more bugs than when a fuzzer is used to test a single configuration.

A simple remediation to this problem is to fuzz all valid program configurations. However, the configuration space of real-world programs is often large, making it infeasible to exhaustively fuzz all configurations. The software testing literature has conducted extensive research on this issue. A widely adopted solution is *combinatorial testing* [9], [10], which proposes to test a sample of configurations that covers certain properties of the configuration space (e.g., all pairs of options appear in some configurations in the sample). In addition, dictionary- or grammar-based approaches have been developed to fuzz program configurations [26], [24]. Program configurations generated by these techniques can then be used as inputs to fuzz the program's input file. However, lacking further in-advance knowledge, prior techniques would spend equal time on each configuration even though different configurations enable different amounts of reachable code; such equal treatment wastes resources.

This observation motivated us to design ConfigFuzz, which enables *efficiently fuzzing program options and the normal program input at the same time* (Section III). ConfigFuzz separates a program's input space into two parts: the *configuration bytes* and *data bytes*. We encode the program options into the *configuration bytes* in a transformed program, and allow a fuzzer's mutation operators to decide when and how to mutate the program's configurations during the fuzzing campaign. As the configuration space is highly structured, ConfigFuzz's encoding ensures that the mutations on program options always generate valid configurations. At the same time, the *data bytes* (i.e., the normal input) are also mutated by the fuzzer, and given as an input of the target program's `main` function. While it is possible to fuzz program options along with *data bytes* by modeling them as unbounded strings (similar to AFL's `argv_fuzzing` feature [2]), we find this approach wastes most of the time trying to reach a valid configuration.

Specifically, ConfigFuzz takes a program's options specification—essentially a *grammar* for the options—as input. This specification distinguishes different option types (i.e., bool, choice, numeric, and string), and specifies valid values of each option. For example, the valid values of a numeric option are integer or real numbers that can be specified with a range. Users of ConfigFuzz can use this specification to control which options to fuzz, because some options may not be useful to fuzz (e.g., they are not used in a security-critical context). ConfigFuzz then outputs a C code wrapper that first

International Fuzzing Workshop (FUZZING) 2022
24 April 2022, San Diego, CA, USA
ISBN 1-891562-77-0
https://dx.doi.org/10.14722/fuzzing.2022.23008
www.ndss-symposium.org

parses in an encoding of the options (*i.e., configuration bytes*) from the start of the program input, and then invokes the `main` function of target with its options set to the decoded values; the remaining input (i.e., *data bytes*) is used by the target program as usual. The fuzzer, e.g., AFL, fuzzes the transformed (wrapped) program. We design the expanded input to ensure that the mutation on a specific byte always updates the setting of the same option. This makes the feedback mechanism built in existing fuzzers useful for fuzzing configurations. ConfigFuzz is parameterized to decide the number of program options that can be explicitly set (through a parameter in the options specification file).

While the approach of ConfigFuzz is applicable to many languages, our current implementation focuses on C programs. We used ConfigFuzz to transform three common fuzzing targets—cxxfilt, objdump, and xmllint—and carried out a preliminary evaluation using a modified version of Google's FuzzBench [14] framework, running the AFL and AFL++ [13] fuzzers (Section IV). We compare ConfigFuzz's fuzzing performance against that of two baselines: (1) when always fuzzing the single default configuration; and (2) when fuzzing, in sequence and with equal time, each of a sample of configurations drawn from 2-way covering arrays. For xmllint, ConfigFuzz shows better performance than the baseline setups, and parameterizing ConfigFuzz to fuzz configurations with up to 2 options leads to higher code coverage than up to 1 option. On the other 2 programs, ConfigFuzz does not always outperform the baselines. We raise hypotheses on the internal mechanism of ConfigFuzz to understand the results, and propose future plans to test the hypotheses and expand the evaluation, using data visualization. In this registered report, we used different fuzzing targets for the empirical study and ConfigFuzz evaluation because the experimental environments were set up differently. We will update both the study and evaluation to include the same programs and run under FuzzBench.

This paper makes the following contributions:

- An empirical study that motivates the importance of fuzzing configurations of program options.
- ConfigFuzz, a tool that encodes program configurations, specified by a grammar, into the input space to allow reusing existing fuzzing algorithms to fuzz program options.
- The implementation of ConfigFuzz that automatically generates configuration stubs and the integration of ConfigFuzz into FuzzBench.
- An evaluation that shows ConfigFuzz's performance comparing to the baselines, and the proposal of an in-depth analysis of the internal mechanisms of ConfigFuzz through visualization.

Related work: The idea of encoding the options into the program input space as a prefix to the actual input was first proposed by AFL [2]; its experimental feature `argv_fuzzing` reads input from `stdin` and puts it into `argv`. A function call to `AFL_INIT_ARGV()` needs to be inserted at the beginning of the `main` function to enable `argv_fuzzing`. This approach does not require a configuration grammar and encodes configurations automatically, relying on the program itself to reject invalid ones. However, the option encoding chosen by AFL often leads to invalid configurations, causing many early terminations which waste resources; this point was observed by the AFL authors [3], and we confirmed it with our own experiments. We thus adopted a more efficient encoding that ConfigFuzz automatically generates based on a configuration grammar. Given that the grammar correctly models a program's configuration space, ConfigFuzz cannot find bugs in the program logic that handles invalid configurations, while AFL's `argv_fuzzing` feature can.

TABLE I: Command-line options of target programs in the preliminary study.

Program	Bool	Choice	Numeric	String
nm-2.28	14	3	0	0
gif2png-2.5.8	13	0	0	1
FFmpeg-n4.4	18	12	32	20

TOFU [24] is a *directed fuzzer*, meaning that it aims to drive the fuzzer to particular targets. Since such targets might require certain options to be enabled, TOFU begins by fuzzing the option space, using a grammar-based mutator to try different options and see what coverage they enable. It then selects configurations that cover code close to the targets before starting fuzzing the program's input file. ConfigFuzz is more general: it enables exploring configurations along with fuzzing program inputs, not just before, and it allows the reuse of existing general-purpose fuzzers' mutators.

The Fuzzing Book [26] introduces a tool that automatically extracts command-line options and infers a configuration grammar. The tool then uses the inferred grammar to generate configurations to fuzz, assigning equal amount of resources on each configuration. As already mentioned, assigning each configuration equal weight very likely wastes resources since different configurations offer uneven amounts of reachable code; ConfigFuzz addresses this problem by fuzzing the options and the rest of the input together. The Fuzzing Book tool also complements ConfigFuzz by automatically extracting the configuration space of the target program.

II. How program configurations affect fuzzing performance?

We perform a preliminary empirical study to understand how configurations make a difference in fuzzing outcomes. While it is expected that different configurations could result in different part of code being executed, there is no prior study that focuses on understanding how tuning a program's configurations would affect a fuzzer's results in terms of code coverage. The answer to this question can be used to motivate the design of a fuzzer that fuzzes configurations.

A. Study Setup

We chose nm-2.28 [20], gif2png-2.5.8 [15], and FFmpeg-n4.4 [12] as the target programs for this preliminary study. These programs are popular fuzzing targets [22][7][16] with command-line options. We inspected the configuration documentation of each target program to understand its allowed options. Table I shows the number of options for each program. We found that each command-line option falls into one of four possible types:

TABLE II: Total and unique line coverage for FFmpeg configurations.

	Default	-f flv	-f mpeg	-f h264	-f mp4	-f webm	**all configs**
# of all lines	17496	13511	12916	9344	1836	1782	26919
# of unique lines	4954	2073	1707	3130	149	109	-
% of unique lines	28%	15%	13%	33%	8%	6%	-

TABLE III: Total and unique line coverage for nm configurations.

	-l	–synthetic	-s	–defined-only	–with-symbol-versions	-f posix	-a	-f bsd	-r
# of all lines	4060	3913	3844	3827	3809	3788	3780	3762	3732
# of unique lines	428	41	5	3	14	19	8	3	5
% of unique lines	11%	1%	0%	0%	0%	0%	0%	0%	0%

	-g	-A	-n	–special-syms	-u	-f sysv	–size-sort	-D	**all configs**
# of all lines	3719	3676	3655	3591	2797	2728	2715	2486	4676
# of unique lines	6	6	19	0	4	38	96	7	-
% of unique lines	0%	0%	0%	0%	0%	1%	4%	0%	-

TABLE IV: Total line coverage for gif2png configurations.

	-g	-r	-b	-v	-h	-O	-i	-p
# of all lines	2398	2392	2373	2369	2352	2352	2350	2334

	-f	-n	-m	Default	-vv	-s	-w	**all configs**
# of all lines	2310	2248	2237	2232	2167	2130	404	2667

- **Bool**: the setting of a bool option is a binary value to decide the presence.
- **Choice**: the setting of a choice option is an element in a finite set of possible choices.
- **Numeric**: the setting of a numeric option is either an integer (i.e., the *intnum* subtype) or a real number (i.e., the *realnum* subtype).
- **String**: the setting of a string option is an arbitrary string.[1]

To answer how program configurations affect fuzzing performance, we generate multiple configurations of each program, and perform fuzzing runs on each generated configuration to compare their code coverage. For this preliminary study, we used a subset of the command-line options of each program in Table I (see Section V for the planned expanded study). For nm, we generated 14 configurations, each enabling one of its 14 bool options. We also generated three configurations from the choice option `-f`, which has three settings `posix`, `sysv`, and `bsd`. As a result, we used 17 nm configurations for this preliminary study. For gif2png, we generated 13 configurations, each enabling one of its 13 bool options. In addition, we used a default configuration that does not turn on any of its option, totaling 14 gif2png configurations. FFmpeg has a much larger configuration space; we selected 5 settings (`flv`, `mpeg`, `h264`, `mp4`, and `webm`) for an important choice option `-f`, which forces the format of FFmpeg's audio and video conversion. We also used a *default* configuration with only `-i` option turned on to accept input file, totaling 6 FFmpeg configurations.

We used AFL-2.52b as the fuzzer for this preliminary study. We ran 3 trials for each configuration, and every trial ran for 24 hours. All programs used an empty file as their seed, except for gif2png which used the seed distributed with AFL (because AFL had issues making progress with gif2png when using the empty seed). Line coverage was extracted by afl-cov [5] after the completion of each AFL trial.

[1]The constraints of a string option are usually not shown in the configuration documentation, even if the programs are checking the valid settings of a string option at runtime (e.g., through regular expressions).

B. Study Results

Overall, we observed that *different configurations contributed disproportionally to code coverage, while almost every individual configuration enabled some unique code to be reached.* This result strongly motivates the design of an effective fuzzer for program configurations.

Table II shows the total and unique numbers of lines covered by each configuration in FFmpeg. We report the number of lines covered by each configuration by aggregating the distinct lines in all 3 trials. A unique line (third row) means that this code was only reached in the fuzzing runs of a specific configuration, and we also show the percentage of these unique lines of all the lines covered by each configuration (fourth row). The last column of Table II ("all configs") shows the number of distinct lines covered in the fuzzing runs of the FFmpeg configurations.

We observe that while the default configuration covered the most code, only 65% of all code covered (17496 out of 26919) by fuzzing these six configurations was due to the default configuration. This indicates limiting runs to a single program configuration, as most past fuzzing experiments have done (e.g., [7], [13]), is a missed opportunity to reach more code. We also see that different configurations make different contributions to the overall code coverage. The default, `-f flv`, and `-f mpeg` configurations all covered more than 10000 lines of code, but fuzzing the configuration `-f webm` only covered 1782 lines. Nevertheless, there are unique lines only covered by each FFmpeg configuration. About one third of the lines covered by the configuration `-f h264` were unique; even

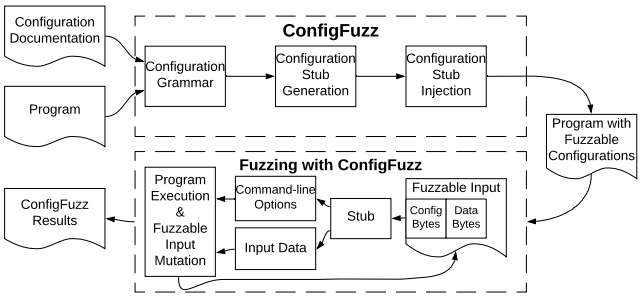


Fig. 1: Overview of ConfigFuzz.

the configuration that covered the least code (`-f webm`) had 109 unique lines.

Table III shows the total and unique numbers of lines covered by each configuration in nm. All but one configuration covered some unique lines. However, unlike FFmpeg where each configuration covered 109-4954 unique lines (accounting for 6% to 33% of all lines covered by each configuration), 76% (13 out of 17) nm configurations covered less than 20 unique lines. The option `-l` covered the most (4060) lines and many (428) unique lines, an important option to test. On the other hand, while the option `-size-sort` only covered 2715 lines, it produced the second most unique lines (96).

Table IV illustrates the aggregated lines covered for each gif2png configuration. As of right now, we have not computed the unique lines covered. Table IV shows that all but one configurations covered between 2100 and 2400 lines, a smaller variance in terms of line coverage. The configuration `-w` covered 404 lines of code. We will add the unique lines covered by each configuration to further analyze the impact of gif2png's configurations.

III. ConfigFuzz

Our study results suggest that different configurations can have differing levels of impact on fuzzing code coverage. To best allocate resources to a fuzzing task, we should prioritize fuzzing the configurations that are likely to lead to more coverage. However, it is difficult to know in advance which are the high-coverage configurations.

We propose ConfigFuzz to address this challenge by transforming the target program to integrate command-line options into the program input that is subject to fuzzing. This allows the fuzzer to change the configuration on the fly if doing so will improve coverage. ConfigFuzz requires a grammar of the configuration options accepted on the target program's command line, and uses these to drive the transformation. The transformed program effectively allows the fuzzer to mutate *expanded* inputs, which include both a configuration part and a normal input data part.

Figure 1 shows an overview of ConfigFuzz and how to fuzz with ConfigFuzz. Given a target program and its configuration documentation as inputs, the test engineer first constructs a *formatted configuration grammar file*. This grammar describes each fuzzable program option with its type and constraints (e.g., valid range of a numeric option). The *configuration stub generator* then uses this grammar to create a C-code stub

```
1  {
2  "input options": ["-i"],
3  "options":
4  [{"id": 0,"opt": "-f", "type": "choice",
5  "choices":
6  ["mp4", "mpeg", "webm", "h264", "flv"]},
7  {"id": 1,"opt": "-vframes",
8   "type": "numeric", "range": [0,432000]},
9  {"id": 2,"opt": "-vn", "type": "bool"},
10 {"id": 3,"opt": "-filter",
11 "type": "string"}],
12 "dependence": [],
13 "conflict":
14 [["-vn", "-vframes"]],
15 "strmax": 19,
16 "maxopts": 2
17 }
```

Fig. 2: An excerpt of configuration grammar of FFmpeg.

that decodes binary input into a set of options. In particular, the stub is fed the *expanded fuzzable input*, whose prefix consists of *configuration bytes* and whose remainder consists of *data bytes*. It decodes the *configuration bytes* into a set of command-line options which it writes into `argc` and `argv`. It directs the remaining *data bytes* into the program's input stream (e.g., `stdin` or an input file path). The generated stub is *injected* into the start of the target program's `main` function. Doing so allows any fuzzer's original algorithm (e.g., the mutator) to fuzz both the program's input configuration and its normal input at once.

A. Configuration Grammar

The *configuration grammar* describes the allowed command-line options of a program. Each option is specified using an *identifier* (id), *name*, and *type*. There are five possible types:

- **bool**: a command-line option that is either present or not present. Its setting is a boolean value to decide the presence.
- **choice**: a command-line option whose setting is a element in a finite set of possible choices.
- **intnum**: a command-line option whose setting is a number with no fractional part.
- **realnum**: a command-line option whose setting is a number with a fractional part.
- **string**: a command-line option whose setting is an arbitrary string.

Type *bool* and *choice* have finite number of settings, while *intnum*, *realnum* and *string* have arbitrarily large setting space.

The configuration grammar is expressed using a simple JSON format; an example is shown in Figure 2. A program may take input files in different ways. For example FFmpeg takes an input file with its `-i` option, specified on line 2. For programs taking input data from `stdin`, an input option `<` will be specified in the grammar, so that data in input file will be

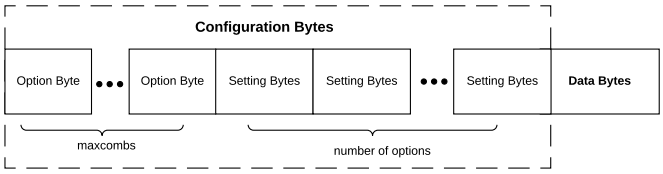


Fig. 3: Structure of ConfigFuzz expanded input.

redirected to `stdin`. For a choice option, the possible settings are listed in the *choices* field. For example, lines 4-6 specify that the choice option `-f` has the five settings: `flv`, `mpeg`, `h264`, `mp4` and `webm`. For a numeric option, the `range` field is used to specify the range of its valid values. For example, lines 7-8 specify that the valid range of the intnum option `-vframes` is 0 to 43200. When the range of a numeric option is not given, this option is potentially unbounded; instead, we use the range of `int` and `double` types in C as the default range for intnum and realnum options, respectively. Line 9 specifies a bool option `-vn`. For all string options, like `-filter` on lines 10-11, the `strmax` field is used to specify their maximum number of characters. Line 15 specifies that at most 19 characters are allowed for all string option settings in FFmpeg.[2]

When manually inspecting the configuration documentation, we also identified that there exist two types of interactions among command-line options: *dependence* and *conflict*. Similar interactions between the program options were identified by Mordahl and Wei in other configurable software [18]. We say option A depends on option B when option A can only be set if the bool option B is set. We say option A conflicts with option B when at most one of the two options can be set in the program's configuration. For example, lines 13-14 specifies that `-vn` conflicts with `-vframes`.

Lastly, we use the `maxopts` field to enforce a maximum degree of option combinations that may be generated during fuzzing. Line 16 specifies that configurations with up to 2 options explicitly set can be generated.

B. Configuration Stub Generation

ConfigFuzz transforms the target program to take an expanded input that contains both its configuration and its data input. This expanded input, with *configuration bytes* followed by *data bytes* is processed by an automatically generated C-code stub. The stub decodes the options from the configuration bytes, and redirects the remaining *data bytes* to program's main input channel. This stub is injected at the beginning of a program's `main` function (see Section III-C).

The configuration data is encoded in the expanded input as shown in Figure 3. The *configuration bytes* precede the *data bytes*, and these *configuration bytes* are divided into two parts: the bytes responsible for encoding which options to turn on (i.e., *option bytes*), and the bytes responsible for encoding which setting to use for an option (i.e., the group of *setting bytes* that follows the *option bytes*).

The number of bytes needed for *option bytes* is decided by `maxopts` specified in the configuration grammar, that is,

[2]We use strmax=19 as the default value for ConfigFuzz.

```
1  opt_bytes = read_next_k_bytes(2)
2  setting_bytes_arr[0] = read_next_k_bytes(1)
3  ...
4  data_bytes = read_remaining_bytes()
5  for opt_byte in opt_bytes:
6    opt_id = opt_byte % 4
7    option = options[opt_id]
8    setting_bytes = setting_bytes_arr[opt_id]
9    if opt_id == 0:
10     setting_id = setting_bytes % 5
11     if setting_id == 0:
12       setting = "mp4"
13     else if setting_id == 1:
14       setting = "mpeg"
15     else if setting_id == 2:
16     ...
17     argv = argv + option + setting
18     argc += 2
19   ...
20 fn = dump_bytes_to_file(data_bytes)
21 argv = argv + "-i" + fn
22 argc += 2
```

Fig. 4: Configuration stub pseudocode generated for FFmpeg.

at most `maxopts` options are turned on in the generated configurations. One byte is needed to encode each option assuming a program does not have more than 256 options. The rest of the *configuration bytes* are used for encoding the setting of each option in the configuration grammar. For the *setting bytes* of each option, the number of bytes needed is decided by the option type and its constraint. Specifically, (1) a bool option needs 1 byte, (2) a choice option needs ceiling of $(\log_{256}[number\ of\ choices])$ bytes, (3) a numeric option needs ceiling of $(\log_{256}[size\ of\ range])$ bytes, where the size of range is computed by subtracting the lower bound from the upper bound, both specified in the configuration grammar, and (4) a string option needs (strmax+1) bytes, where `strmax` is specified in the configuration grammar.

Figure 4 shows the pseudocode for part of a configuration stub generated with the grammar described in Figure 2 for FFmpeg. Line 1 reads two *option bytes* from the input (the first two bytes), as `maxopts` is set as 2 in our example. Line 2 reads the next byte from the input as the *setting bytes* of `option_id=0`; 1 byte is read because in our example, this option (`-f`) is a choice option with 5 choices, and 1 byte is large enough to encode them. The omitted code at line 3 reads the setting types for each remaining option. Line 4 reads the remaining data into *data bytes*, which is later used as program input. For each option byte, lines 6 and 7 transform it into an option. The option id is determined by taking the reminder of the option byte divided by the total number of options (4 in our example, as shown on line 6). The option is then looked up in the `options` array (which is generated with the rest of the stub when processing the configuration grammar), per line 7. Line 8 retrieves the *setting bytes* per the option id. Lines 9–16 decode the setting of option `-f`, which is a *choice*-type option. First, a `setting_id` is calculated from the *setting bytes*, and then this byte is used to select the actual choice. Other option types are encoded as follows:

- For a bool option, its setting is empty (i.e., turning on the bool option only requires adding the option to the argument without a setting).
- For a numeric option, we first obtain the `range` of the option and then the encoding is based on the *setting bytes* and the range. Specifically, we follow two equations for the calculation: `rsize=range.max-range.min+1` and `setting=range.min+setting_bytes%rsize`.
- For a string option, its setting is directly transformed from the *setting bytes* with string type cast.

Returning to Figure 4, lines 17 and 18 append the option and setting strings to `argv` and increment `argc` of the program based on the encoding. These simulate the command-line arguments given to the `main` function. We update `argc` and `argv` the same way on all options except bool options. Because bool options do not have any setting, we only append the option string to `argv` and increment `argc` by 1. Finally, in lines 20 – 22, we dump the *data bytes* into an in-memory file, append `argv` with the input options (i.e., `-i` for FFmpeg) and file name, and increment `argc` accordingly.

This structure for *configuration bytes*' encoding ensures that each byte can always be legally interpreted as specifying an option or its setting. As a result, a mutation performed on the same byte always properly updates the encoded option or setting, making the coverage feedback of a fuzzer more efficient. We call this encoding the *hash encoding*. One limitation of hash encoding is that it may not encode options and settings with equal probability. For example, we use one option byte to encode a program with 255 command-line options; one option (`option_id=0`) will have a higher probability of being selected. We may allocate more bytes for selecting an option or a setting to remediate this problem, but it makes the input larger which may reduce fuzzing effectiveness.

C. Configuration Stub Injection

The stub injection step of ConfigFuzz takes the source code of the generated stub, and injects it into the beginning of the target program's `main` function, implemented with a Python script. The stub modifies `main`'s parameters `argc` and `argv` to hold the decoded options. ConfigFuzz assumes that a fuzzer will run the transformed program with the expanded input and no other command-line options because the command-line options are written by the injected stub. Therefore, the inputs of the stub will usually be `argc` of 2 and `argv[1]` being the path to a file storing the expanded input.

The modified parameters are then given to the rest of the `main` function, mimicking the situation where command-line options are stored in `argv`, and a fuzzer fuzzes the program along with configurations.

IV. Evaluation

We conducted preliminary experiments to evaluate ConfigFuzz, comparing its performance on different settings against two baseline setups: one fuzzes a default configuration and the other samples configurations drawn from covering arrays. In this section, we present the setup and results of the preliminary experiments. We discuss additional planned evaluation in Section V.

Program	Bool	Choice	Numeric	String
cxxfilt-2.37	7	1	0	0
objdump-2.37	26	7	3	1
xmllint-2.9.12	36	0	1	4

TABLE V: Command-line options of target programs in the preliminary evaluation.

A. Setup of Preliminary Experiments

1) Target Programs and Fuzzers: ConfigFuzz-transformed programs are compatible with most existing fuzzers. In the preliminary experiments, we ran experiments using two fuzzers: AFL-2.57b [2] and AFL++-3.14a [13]. Using more than one fuzzer, we can check if benefits of ConfigFuzz are observed in both fuzzers, and/or if the behavior is specific to a fuzzer.

We have run both fuzzers on three particular programs:[3] cxxfilt-2.37 [11], objdump-2.37 [21], and xmllint-2.9.12 [25]. The configuration space of these programs is shown in Table V. We excluded `string` options from ConfigFuzz's configuration grammar to be consistent with the baselines, as explained in the following subsection.

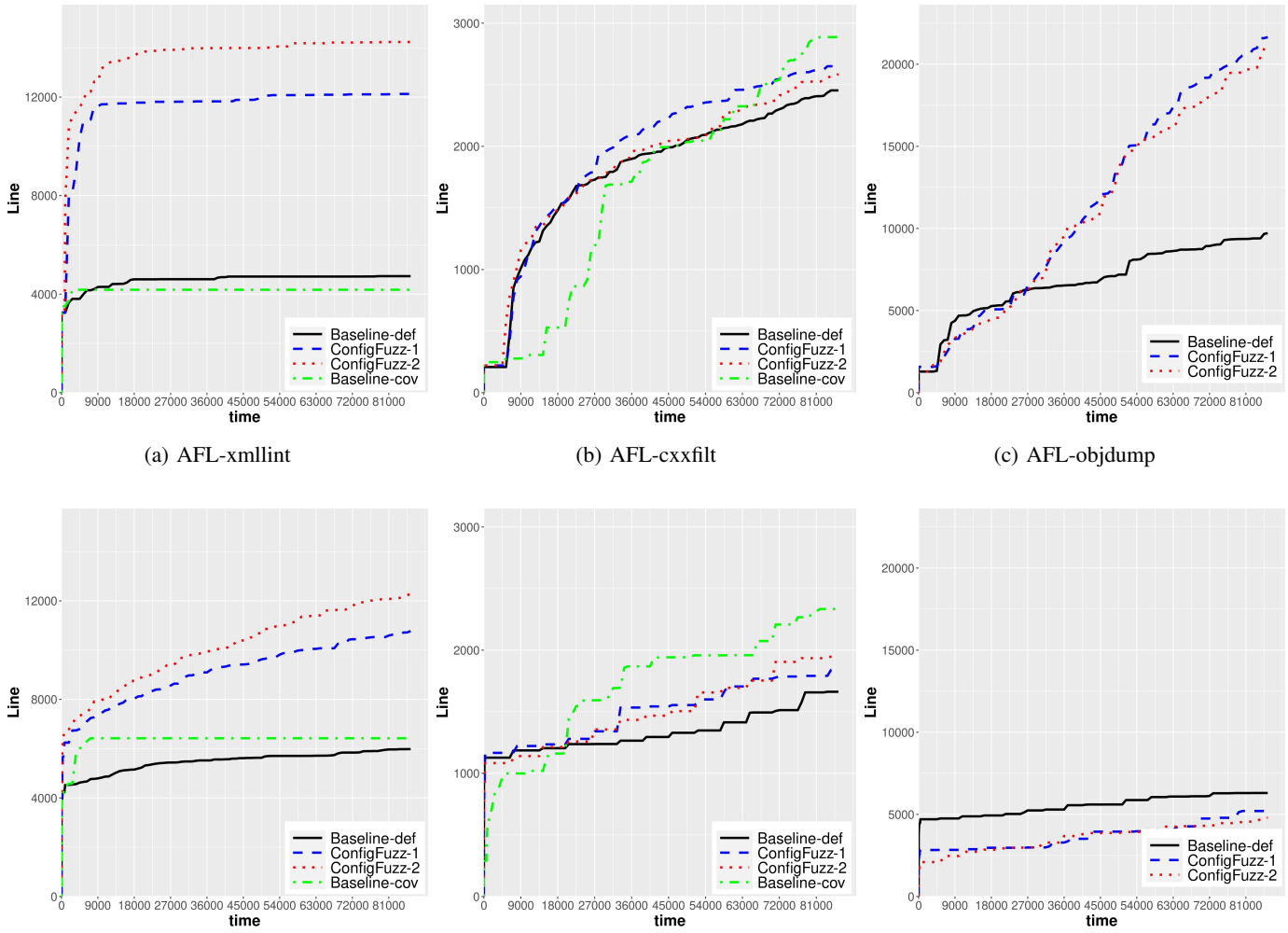
2) ConfigFuzz Settings and Baselines: We experimented with two settings of ConfigFuzz. Specifically, we set `maxopts` to 1 and 2, i.e., fuzzing configurations with at most 1 and 2 options explicitly set, respectively. We call these two variations as ConfigFuzz-1 and ConfigFuzz-2.

We compared ConfigFuzz-1 and ConfigFuzz-2 with two baselines. The first baseline (called Baseline-def) fuzzes only the default configuration of the target program. The second baseline (called Baseline-cov) fuzzes a sample of configurations generated by two-way covering arrays [19]. Such sample contains two-way combinations of all option settings [10], enhancing the likelihood of discovering interactions compared to just a random sample. We considered two-way interactions to reduce the configuration space, and because it is commonly assumed that most faults are caused by the interaction of only a few features [23]. We used ACTS 3.2, a combinatorial testing tool from NIST [1], to generate the configuration samples. We included all settings of bool and choice options. For numeric options, we took the lower and upper bound of the range, and randomly sampled 8 numbers in between to generate 10 choices. For string options, considering a randomly sampled string is mostly likely to yield an invalid setting, we removed all strings options from the configuration grammar. As a result, 17, 1865, and 24 sample configurations were generated for cxxfilt, objdump, and xmllint, respectively. To fairly compare with ConfigFuzz, we modified the fuzzing process to fuzz an equal amount of time for each sampled configuration (i.e., [total time]/[number of configurations]) in sequence, while retaining the seeds from previous fuzzing progress. We could have chosen to use seeds afresh, but we thought that retaining seeds between configurations might lead to more useful results, and indeed that is what happened, as we will see.

3) Research questions: Our preliminary experiments answer two research questions:

- **RQ1:** Does ConfigFuzz outperform baselines?

[3]We plan to include the 3 programs used in section II in future experiments



(a) AFL-xmllint (b) AFL-cxxfilt (c) AFL-objdump

(d) AFL++-xmllint (e) AFL++-cxxfilt (f) AFL++-objdump

Fig. 5: Line coverage growth plots.

- **RQ2:** How do ConfigFuzz-1 and ConfigFuzz-2 compare?

For **RQ1**, we check if ConfigFuzz can result in more code coverage than fuzzing Basedef-def and Baseline-cov. For **RQ2**, we check if ConfigFuzz-2, which can generate configurations that interact with 2 options, can result in more code coverage than ConfigFuzz-1 to assess the importance of fuzzing the interactions among program options.

4) Experimental design: We integrated the use of ConfigFuzz into FuzzBench [14] to allow reproducible and reusable experiments. We added ConfigFuzz transformed programs into FuzzBench benchmarks, and specified the fuzzing targets to be the executables containing the modified `main` functions. FuzzBench originally expected LibFuzzer harnesses as entry points (compiled with Clang [8] and the `fsanitize-coverage=trace-pc-guard` flag), while ConfigFuzz is designed to fuzz whole programs with command-line options. We modified the FuzzBench scripts to enable running fuzzers on the whole programs following AFL's tutorial [4]. Specifically, the modified script builds fuzzing targets using each fuzzer's own compiler for instrumentation, and the command to run each fuzzer was also updated accordingly to ensure the expanded input is correctly passed to target programs, as discussed in section III-C.

We ran each of ConfigFuzz-1, ConfigFuzz-2, Baseline-def, and Baseline-cov on each program with 5 trials and 24-hour timeout. We report the median number of lines covered by each fuzzer after post-processing code coverage using llvm-cov [17]. We used two seeds for objdump and xmllint: an empty file [7], [16], and one valid seed taken from AFL's repository [2]. There does not exist a valid seed for cxxfilt in AFL's repository; instead, we used the mangled version of function name `f()` as the valid seed for cxxfilt. The empty seed for cxxfilt is the same empty file as the other 2 programs. In particular, the valid seeds are only valid for Baseline-def and Baseline-cov. After taking *configuration bytes* by ConfigFuzz, the remaining *data bytes* of a valid seed become invalid for ConfigFuzz-1 and ConfigFuzz-2. We plan to revisit this setup in future experiments.

All experiments were conducted on a server with 2 Intel Xeon Silver 4116 CPUs (each with 24 cores) and 192GB of RAM running Ubuntu 16.04. The experiment were run in

parallel per target program at a time, which used 20 cores.

B. Results

Figure 5 shows line coverage growth of each fuzzer over time. Results of default (Baseline-def), ConfigFuzz-1, ConfigFuzz-2, and covering arrays (Baseline-cov) are represented by lines in black, blue, red and green, respectively.

For xmllint, ConfigFuzz outperformed the two baselines. ConfigFuzz-1 and ConfigFuzz-2 not only achieved higher coverage than Baseline-def and Baseline-cov at a very early stage of the fuzzing campaign (within 15 minutes), but also grew faster. This is a strong indication that ConfigFuzz outperforms the baselines by exploring more command-line options. In addition, ConfigFuzz-2 covered 9% more lines than ConfigFuzz-1 in 24 hours (statistically significant through Mann Whitney U test [6], [16]), which can be attributed to option interactions. The covering arrays achieved similar coverage as default, but at a much faster speed. It is possible that the default configuration reached code by "going deep", while covering arrays benefit from "going wide" in terms of covering configurations. We will perform further investigation to explain this result (see Section V).

Results of cxxfilt, however, did not show ConfigFuzz always outperformed the baseline setups, with both fuzzers having covering arrays as the best approach. The difference between ConfigFuzz-1 and ConfigFuzz-2 was not statistically significant, but both of them performed better than Baseline-def. A possible explanation is that fuzzers benefit more in "going wider" by exploring different configurations than going deeper into execution paths within a configuration, on cxxfilt.

Covering arrays experiments for objdump were not completed as the time of submission. Coverage of the default configuration was similar between AFL and AFL++, but ConfigFuzz-1 and ConfigFuzz-2 performed much better in AFL; we have yet to ascertain the reason for this. Another observation is that ConfigFuzz-1 performed slightly better than ConfigFuzz-2. However, the difference is not statistically significant. A possible reason is that option interactions may not be very significant in objdump, and potentially resulted in fewer options covered in ConfigFuzz-2.

In summary, we observed mixed answers to our research questions. During analysis of the experiment results, we raised the hypotheses that there are 3 major parts in code coverage during option fuzzing: exploring more options, exploring option combinations, and exploring more inputs with a given option. Performance of ConfigFuzz is decided by the characteristics of these three parts in a program. Further analysis is needed to test the hypotheses and find the best application of fuzzing configurations with ConfigFuzz.

C. Threats to Validity

A potential threat to validity is that we measure code coverage instead of ground-truth bugs when comparing the fuzzers, which may not accurately measure the effectiveness of fuzzers [16]. To our knowledge, there do not exist any ground-truth benchmarks that are suitable for evaluating fuzzers that consider program configurations. It is future work to develop such benchmarks to allow fair comparison between fuzzers on programs with configurations. If any approaches in our evaluations discover crashes in the target programs, we will perform deduplication and report the bugs found.

V. Planned Additional Study and Evaluation

We plan to expand the preliminary study (Section II) and evaluation (Section IV).

First, the same set of programs will be used for the study and evaluation, including FFmpeg, nm, cxxfilt, objdump, and xmllint. We will also search for more programs suitable for our evaluation: common fuzzing targets built with well-documented command-line options. Specifically, we expect a well-documented program to come with detailed explanation of each option in order for us to decide its type and constraints.

Second, we will perform deeper investigation of the experimental results to understand the behavior of ConfigFuzz and of covering-array based fuzzing. We will summarize which configurations are fuzzed overtime, and how they contribute to the overall performance. Specifically, we will develop multiple visualizations that show the sequence of changes to the command-line options that take place during a fuzzing run. It would be interesting to see how often they flip back and forth and which ones contributed the most to the code coverage. We will also examine how sharing seeds between sequenced covering array-based configurations aids its performance.

Third, we will experiment with other variants of ConfigFuzz, especially ConfigFuzz with larger `maxopts`. We will set `maxopts` to the number of options of each program to allow fuzzing all possible configurations to further evaluate the interactions among program options. We will also enable string options to be fuzzed by ConfigFuzz in a separate experiment. Finally, we may examine how covering array-based configuration sampling could be integrated into ConfigFuzz.

Fourth, we will use more fuzzers (those that are built in FuzzBench and suitable for fuzzing whole programs) to evaluate ConfigFuzz.

VI. Conclusions

In this paper, we present ConfigFuzz, an approach that enables fuzzing program options and program input at the same time. A key idea of ConfigFuzz is to transform the target to take an expanded input that encodes the program options, so existing fuzzers' mutation operators can be reused to fuzz program configurations. The options are specified by a grammar, and code is generated and injected into the target to decode them from the expanded input; the option encoding is designed to ensure that mutations are effective. ConfigFuzz's design was motivated by an empirical study that showed different configurations can have differing levels of impact on fuzzing code coverage. We integrated ConfigFuzz into FuzzBench in order to evaluate it. Preliminary experiments on three programs show that ConfigFuzz nearly always outperforms the baseline of fuzzing the default configuration, while fuzzing a sequence of configurations sampled using covering array improves on ConfigFuzz in cxxfilt. We proposed to perform deeper investigation of the experimental results using data visualization.

REFERENCES

[1] "Automated Combinatorial Testing for Software (ACTS)," https://www.nist.gov/programs-projects/automated-combinatorial-testing-software-acts.

[2] "American Fuzzy Lop (AFL)," https://lcamtuf.coredump.cx/afl/.

[3] "Struggling to give inputs to afl," https://groups.google.com/g/afl-users/c/ZBWq0LdHBzw/m/zBlo7q9LBAAJ.

[4] "afl-tutorial," https://afl-1.readthedocs.io/en/latest/fuzzing.html.

[5] "afl-cov," https://github.com/mrash/afl-cov.

[6] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 1–10. [Online]. Available: https://doi.org/10.1145/1985793.1985795

[7] 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*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1032–1043. [Online]. Available: https://doi.org/10.1145/2976749.2978428

[8] "Clang: a C language family frontend for LLVM," https://clang.llvm.org/.

[9] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The aetg system: An approach to testing based on combinatorial design," *IEEE Trans. Softw. Eng.*, vol. 23, no. 7, p. 437–444, Jul. 1997. [Online]. Available: https://doi.org/10.1109/32.605761

[10] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn, "Constructing test suites for interaction testing," in *Proceedings of the 25th International Conference on Software Engineering*, ser. ICSE '03. USA: IEEE Computer Society, 2003, p. 38–48.

[11] "cxxfilt," https://sourceware.org/binutils/docs/binutils/c_002b_002bfilt.html.

[12] "FFmpeg," https://ffmpeg.org.

[13] 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

[14] "FuzzBench: Fuzzer benchmarking as a service," https://github.com/google/fuzzbench/.

[15] "gif2png," http://www.catb.org/esr/gif2png/.

[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*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 2123–2138. [Online]. Available: https://doi.org/10.1145/3243734.3243804

[17] "llvm-cov," https://llvm.org/docs/CommandGuide/llvm-cov.html.

[18] A. Mordahl and S. Wei, *The Impact of Tool Configuration Spaces on the Evaluation of Configurable Taint Analysis for Android*. New York, NY, USA: Association for Computing Machinery, 2021, p. 466–477. [Online]. Available: https://doi.org/10.1145/3460319.3464823

[19] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Comput. Surv.*, vol. 43, no. 2, Feb. 2011. [Online]. Available: https://doi.org/10.1145/1883612.1883618

[20] "nm," https://sourceware.org/binutils/docs/binutils/nm.html.

[21] "objdump," https://sourceware.org/binutils/docs/binutils/objdump.html.

[22] V. Pham, M. Bohme, A. E. Santosa, A. Caciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering*, vol. 47, no. 09, pp. 1980–1997, sep 2021.

[23] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, "A classification and survey of analysis strategies for software product lines," *ACM Comput. Surv.*, vol. 47, no. 1, Jun. 2014. [Online]. Available: https://doi.org/10.1145/2580950

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

[25] "xmllint," http://xmlsoft.org/xmllint.html.

[26] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, "Testing configurations," in *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2021, retrieved 2021-11-07 22:56:29+01:00. [Online]. Available: https://www.fuzzingbook.org/html/ConfigurationFuzzer.html