Fuzzing Configurations of Program Options

ZENONG ZHANG, University of Texas at Dallas, USA
GEORGE KLEES, University of Maryland, USA
ERIC WANG, Poolesville High School, USA
MICHAEL HICKS, University of Maryland and Amazon*, USA
SHIYI WEI, University of Texas at Dallas, USA

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 6 configurable, common fuzzing targets, and integrate their executions in FuzzBench. In our evaluation, ConfigFuzz outperforms two baseline fuzzers in four targets, while the results are mixed in the other targets due to program size and configuration space. We also analyze the options fuzzed by ConfigFuzz and how they affect the performance.

CCS Concepts: • **Security and privacy → Software security engineering**; • **Software and its engineering → Software testing and debugging**.

ACM Reference Format:
Zenong Zhang, George Klees, Eric Wang, Michael Hicks, and Shiyi Wei. 2023. Fuzzing Configurations of Program Options. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (February 2023), 21 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 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 at runtime. We define the combinations of command-line options of a program as its *configurations*. In addition to a configuration, a program typically takes *input data*, either from a file or `stdin`. Both the configuration and input data determine the program code that is executed at runtime. However, most fuzzers (and scientific evaluations thereof) focus on fuzzing the input data given a single, fixed program configuration (e.g., [16, 19]), and may therefore fail to properly test significant portions

*Work done prior to starting at Amazon.

Authors' addresses: Zenong Zhang, zenong@utdallas.edu, University of Texas at Dallas, Richardson, Texas, USA, 75080; George Klees, george.t.klees@gmail.com, University of Maryland, College Park, Maryland, USA, 20742; Eric Wang, ericzwang20841@gmail.com, Poolesville High School, Poolesville, Maryland, USA, 20837; Michael Hicks, mwh@cs.umd.edu, University of Maryland and Amazon, College Park, Maryland, USA, 20742; Shiyi Wei, swei@utdallas.edu, University of Texas at Dallas, Richardson, Texas, USA, 75080.

1049-331X/2023/2-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 3). 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 13% lines reached when fuzzing FFmpeg [6] 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* [17, 18], 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 [27, 29]. 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. AFL has an experimental `argv_fuzzing` feature [2] that fuzzes a program's `argv` along with program's input data. While it is possible to fuzz program configurations with `argv_fuzzing` by modeling them as unbounded strings in `argv`, we find this approach wastes most of the time trying to reach a valid configuration.

The above observations motivated us to design ConfigFuzz, which enables *efficiently fuzzing the program configurations and the input data at the same time* (Section 4). 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 data) are also mutated by the fuzzer, and given as an input of the target program's `main` function.

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. The specification is designed in a way that users can freely control which options to fuzz and can disable certain combination of options. This is because sometimes fuzzing program configurations may not be helpful. Some command-line options are not developed to be used in a security-critical context, such as options for experimental features. In addition, there may exist dependence or conflict relationship between program options, and certain combinations of these options should be avoided when fuzzing configurtions. For example, Clang's options `-march=mblaze` and `-msse4.2` have conflicts, and using them together is meaningless [14].

Using the program's options specification, ConfigFuzz outputs a C code wrapper that first 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 the target program 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. This enables the fuzzer to mutate the option and/or its settings during fuzzing. We design the expanded input to ensure that the mutation on a specific byte always updates the setting of the same option, while mutation on a byte that decides a setting has no effect when its associated option is not selected by the fuzzer. This makes the feedback mechanism built in existing fuzzers useful for fuzzing configurations. ConfigFuzz is parameterized to decide the number of program options (through a parameter that can be set in the options specification).

While the approach of ConfigFuzz is applicable to many languages, our current implementation focuses on C programs. We used ConfigFuzz to transform six common fuzzing targets and carried out the evaluation using a modified version of Google's FuzzBench [7] framework, running the AFL and AFL++ [19] fuzzers (Section 5). 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. ConfigFuzz shows better performance than the baseline setups in four targets, while on the other two targets, ConfigFuzz does not always outperform the baselines. We analyze the target programs' source code and the options fuzzed by ConfigFuzz to reason about the fuzzing performance. We also show that parameterizing ConfigFuzz to fuzz configurations with up to 2 options often leads to higher code coverage than up to 1 option, while fuzzing many more options with ConfigFuzz may decrease the performance.

This paper made 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 an analysis that provides insights on the behavior of ConfigFuzz.

2 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 was observed by the AFL authors [1], 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.

TOFU [27] 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.

Table 1. Command-line options of target programs in the empirical study.

Program	Bool	Choice	Numeric	String
nm-2.37	13	1	0	0
gif2png-2.5.8	12	0	0	0
FFmpeg-n4.4	0	1	0	0

POWER [21] is a recently developed fuzzer which fuzzes a target program with configurations in multiple steps. First, in its exploratory stage, POWER iteratively generates configurations using dictionary-based mutation while also fuzzes the input files with byte-level mutation. Next, it selects a set of configurations based on a relevance heuristic. Finally, its main fuzzing stage fuzzes the input files using the selected configurations as the seed corpus. Different from POWER, ConfigFuzz fuzzes configurations together with the input files throughout the fuzzing process, and allows reusing existing fuzzers' mutation operators by taking an expanded input that encodes the program options.

The Fuzzing Book [29] 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.

OpFuzz [28] and TypeFuzz [24] fuzz configurations of SMT solvers. Both approaches fuzz configurations by first defining a popular configuration as a default mode for each SMT solver, and then fuzzing more configurations by introducing additional options on top of the default modes. Both fuzzers found most of the bugs in the default modes, and observed that few bugs were found under the configurations that included more than 2 additional options on top of the default modes. Their results show that it is infeasible to fuzz each configuration separately, and motivate us to develop an efficient approach to allocate different resources on different configurations during fuzzing.

3 HOW PROGRAM CONFIGURATIONS AFFECT FUZZING PERFORMANCE?

We perform an 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.

3.1 Study Setup

We chose nm-2.37 [11], gif2png-2.5.8 [9], and FFmpeg-n4.4 [6] as the target programs for this study. These programs are popular fuzzing targets [25][16][20] with command-line options. We inspected the configuration documentation of each target program to understand its allowed options. We found that each command-line option falls into one of four possible types:

- **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).

Table 2. Total and unique line coverage for FFmpeg configurations.

	Default	-f mpeg	-f mp4	-f flv	-f h264	-f webm	**all configs**
# of all lines	56183	41902	36124	33582	21082	2640	63884
# of unique lines	9578	1749	1975	2048	272	237	-
% of unique lines	17%	4%	5%	6%	1%	9%	-

Table 3. Total and unique line coverage for nm configurations.

	-l	–synthetic	-g	–w-sym-v[2]	-s	–size-sort
# of all lines	13525	12908	12678	12612	12511	12495
# of unique lines	1426	275	35	5	3	159
% of unique lines	11%	2%	0%	0%	0%	1%
	-u	**-r**	**-A**	**-n**	**–special-syms**	**–defined-only**
# of all lines	12484	12480	12451	12356	12255	12106
# of unique lines	13	11	9	44	1	5
% of unique lines	0%	0%	0%	0%	0%	0%
	-D	**-f bsd**	**-f posix**	**-f sysv**		**all configs**
# of all lines	11056	1409	1409	1409		15243
# of unique lines	29	4	4	4		-
% of unique lines	0%	0%	0%	0%		-

- **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. We used a subset of the command-line options of each program, shown in Table 1. For nm, we generated 13 configurations, each enabling one of its 13 bool options. We also generated three configurations from the choice option `-f`, which has three settings `bsd`, `posix` and `sysv`. Specifically, `bsd` is the default setting. As a result, we used 16 nm configurations for this preliminary study. For gif2png, we generated 12 configurations, each enabling one of its 12 bool options. In addition, we used a default configuration that does not turn on any of its option, totaling 13 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 5 trials for each configuration, and every trial ran for 24 hours. All programs used two seeds: a seed with corresponding file format distributed by AFL (e.g., we used a small gif file as the seed for gif2png), and the default invalid seed used by FuzzBench (a text file of string `hi`). Line coverage was extracted by llvm-cov [10] after the completion of each AFL trial.

3.2 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.

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

[2]Abbreviation for the configuration `--with-symbol-versions`.

Table 4. Total and unique line coverage for gif2png configurations.

	-h	-g	-v	-r	-m	-p	-f	
# of all lines	2898	2884	2861	2860	2847	2844	2839	
# of unique lines	63	49	18	55	12	9	4	
% of unique lines	2%	2%	1%	2%	0%	0%	0%	
	-O	Default	-i	-s	n	-w		**all configs**
# of all lines	2834	2829	2829	2818	2714	570		3108
# of unique lines	13	0	3	3	3	13		-
% of unique lines	0%	0%	0%	0%	0%	0%		-

Table 2 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 5 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 2 (“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 88% of all code covered 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., [16, 19]), 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 mpeg`, `-f mp4`, and `-f flv` configurations all covered more than 30000 lines of code, but fuzzing the configuration `-f webm` only covered about 2600 lines. Nevertheless, there are unique lines only covered by each FFmpeg configuration. About 17% of the lines covered by the configuration `Default` were unique; even the configuration that covered the least code (`-f webm`) had 237 unique lines. This result is consistent with what we observed from the code investigation on these configurations. The settings of `-f` option, which specify the format of FFmpeg’s audio and video conversion, reach very different parts of FFmpeg’s implementation.

Table 3 shows the total and unique numbers of lines covered by each configuration in nm. All configurations covered some unique lines. However, unlike FFmpeg where each configuration covered about 300-9600 unique lines (accounting for 1% to 17% of all lines covered by each configuration), 75% (12 out of 16) nm configurations covered less than 50 unique lines. Through manual investigation, we found that it is important to test the option `-l`, because it controls about 100 lines of code in the `nm.c` file to find a filename and line number for each symbol with debugging information. This explains why fuzzing the `-l` configuration covered the most lines and many unique lines in Table 3. The configurations that covered the least numbers of lines are those that set the `-f` option. This is surprising because the settings of `-f` decide the output format. After investigating the fuzzing log, we found that AFL ignored the valid seed and kept fuzzing the invalid seed, and therefore was unable to generate any new seed in 24 hours.

Table 4 illustrates the aggregated and unique lines covered for each gif2png configuration. Table 4 shows that all but one configurations covered between 2700 and 2900 lines, a smaller variance in terms of line coverage. This is because all command-line options except for `-w` differ in only a few branches in gif2png. The configuration `-w` exits gif2png earlier compared to other configurations, therefore only covered 570 lines of code.

In summary, program options often decide unique branches to execute in a target program, which make fuzzing different configurations contribute disproportionally to code coverage. The

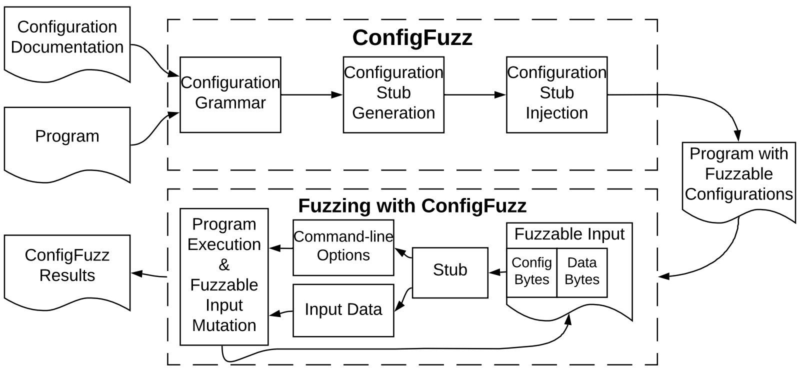

Configuration Documentation
Program
ConfigFuzz
Configuration Grammar
Configuration Stub Generation
Configuration Stub Injection
Program with Fuzzable Configurations
Fuzzing with ConfigFuzz
ConfigFuzz Results
Program Execution & Fuzzable Input Mutation
Command-line Options
Input Data
Stub
Fuzzable Input
Config Bytes
Data Bytes

Fig. 1. Overview of ConfigFuzz.

unique branches sometimes resulted in a few more statements being executed compared to the default configuration, but sometimes direct the program execution to a very different route.

4 CONFIGFUZZ

Our study results suggest that different configurations can have differing levels of impact on fuzzing code coverage. To best allocate different 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 configurations 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 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 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 and their settings 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.

4.1 Configuration Grammar

The *configuration grammar* describes the allowed command-line options and settings 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.

```
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.

- **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 in 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 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.[3]

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 [22]. 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`.

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

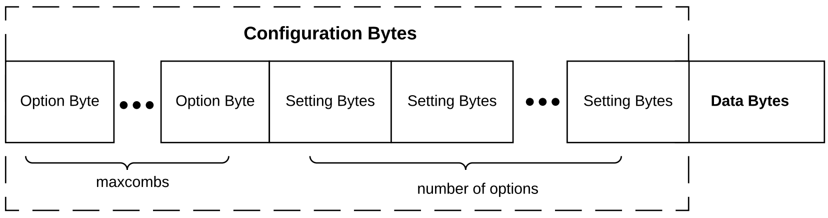


Fig. 3. Structure of ConfigFuzz expanded input.

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.

Note that the configuration grammar does not need to be accurate for ConfigFuzz to work. Invalid command-line options often cause immediate failures when a program executes, and fuzzers will not waste much resources on them. The same situation also applies on option interactions, where fuzzers will not prioritize invalid option combinations. Therefore, we expect ConfigFuzz to still work with a slightly incorrect grammar. In addition to correctly describing the configuration space of a program, the option interactions in the grammar also enables users to customize the fuzzing space. For example, users can turn off certain combinations of options that come with meaningless errors, as discussed in [14].

4.2 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 4.3).

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, 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

```
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 %
7     option = options[opt_id]
8     setting_bytes = setting_bytes_arr[opt_id]
9     if opt_id == 0:
10    setting_id = setting_bytes %
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.

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,

Table 5. Command-line options of target programs in the evaluation.

Program	Bool	Choice	Numeric	String	Total
cxxfilt-2.37	7	1	0	0	8
FFmpeg-n4.4	7	3	8	12	30
gif2png-2.5.8	14	0	1	0	15
nm-2.37	20	4	0	1	25
objdump-2.37	30	7	5	6	48
xmllint-2.9.12	49	1	1	7	58

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.

4.3 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.

5 EVALUATION

We conducted 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 experiments.

5.1 Experimental Setup

5.1.1 Target Programs and Fuzzers. ConfigFuzz-transformed programs are compatible with most existing fuzzers. In the evaluation, we ran experiments using two fuzzers: AFL-2.57b [2] and AFL++-3.14a [19]. Using more than one fuzzer, we can check if performance of ConfigFuzz is consistent in both fuzzers, and/or if the behavior is specific to a fuzzer.

The experiments were run on six popular fuzzing targets: cxxfilt-2.37 [5], FFmpeg-n4.4 [6], gif2png-2.5.8[9], nm-2.37 [11], objdump-2.37 [12], and xmllint-2.9.12 [13]. We ran both fuzzers on five programs, but ran only AFL on gif2png-2.5.8 because we could not build gif2png-2.5.8 with AFL++. Table 5 shows the configuration space of these target programs; the numbers of command-line options ranging from 8 (cxxfilt) to 58 (xmllint).

5.1.2 ConfigFuzz Settings and Baselines. We experimented with five settings of ConfigFuzz. Specifically, we set `maxopts` to 1, 2; i.e., fuzzing configurations with at most 1 and 2 options explicitly set. We also set `maxopts` to the total number of options in each program, We call these three variations as ConfigFuzz-1, ConfigFuzz-2 and ConfigFuzz-max. We excluded `string` options from ConfigFuzz's configuration grammar to be consistent with the baselines because a sample-based baseline could not always generate valid string settings, as explained below.

We compared ConfigFuzz-1, ConfigFuzz-2 and ConfigFuzz-max with two baselines. The first baseline (called Baseline-def) fuzzes only the default configuration of the target program. The second baseline (called Baseline-2-way) fuzzes a sample of configurations generated by two-way covering arrays [23]. Such sample contains two-way combinations of all option settings [18], 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 [26]. We used ACTS 3.2, a combinatorial testing tool from NIST [3], to generate the configuration samples. We included all settings of bool options and choice options with less than 50 settings. On objdump and FFmpeg, choice options with more than 50 settings cause the number of covering arrays to explode and fail the program execution. Therefore, a random sample of 10 settings are taken on these 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, 294, 22, 2567, 362 and 46 sample configurations were generated for cxxfilt, FFmpeg, gif2png, nm, 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.

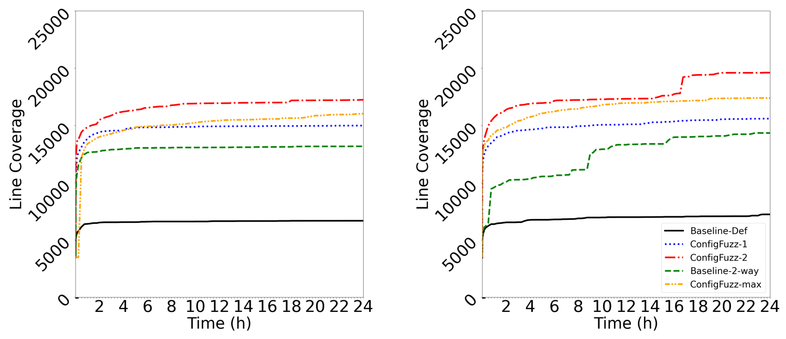
The remaining two settings enable string options. Similar to ConfigFuzz-1 and ConfigFuzz-2, we explicitly set at most 1 and 2 options during fuzzing with the entire configuration grammar, and call them ConfigFuzz-str-1 and ConfigFuzz-str-2. Results of ConfigFuzz-str-1 and ConfigFuzz-str-2 are compared with ConfigFuzz-1 and ConfigFuzz-2 to evaluate the impact of fuzzing configurations that include string options.

5.1.3 Research questions. Our experiments answer three research questions:

- **RQ1:** Does ConfigFuzz outperform baselines?
- **RQ2:** How do ConfigFuzz-1, ConfigFuzz-2 and ConfigFuzz-max compare?
- **RQ3:** How does ConfigFuzz perform on string options?

For **RQ1**, we check if ConfigFuzz can result in more code coverage than fuzzing Baseline-def and Baseline-2-way. 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. Moreover, we compare the performance of ConfigFuzz-max, which does not limit the number of configuration interactions, with ConfigFuzz-1 and ConfigFuzz-2 to study how highly interacted configurations affect fuzzing performance. For **RQ3**, we compare the results of ConfigFuzz-str-1 and ConfigFuzz-str-2, with ConfigFuzz-1 and ConfigFuzz-2 to evaluate how fuzzing configurations with string options contributes to code coverage.

5.1.4 Experimental design. We integrated ConfigFuzz into FuzzBench [7] 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 [4] and



(a) AFL-xmllint (b) AFL++-xmllint

Fig. 5. Line coverage growth plots on xmllint

Table 6. Top 5 most frequently fuzzed options xmllint with AFL.

ConfigFuzz-1	ConfigFuzz-2	ConfigFuzz-max	ConfigFuzz-str-1	ConfigFuzz-str-2
−html (32%)	−html (36%)	−sax1 (87%)	−html (29%)	−html (42%)
−recover (16%)	−repeat (22%)	−recover (68%)	−xpath (16%)	−xpath (23%)
−repeat (8%)	−recover (19%)	−debug (68%)	−recover (11%)	−repeat (15%)
−stream (6%)	−maxmem (13%)	−debugent (60%)	−repeat (7%)	−recover (14%)
−maxmem (5%)	−sax1 (13%)	−copy (45%)	−stream (5%)	−sax1 (12%)

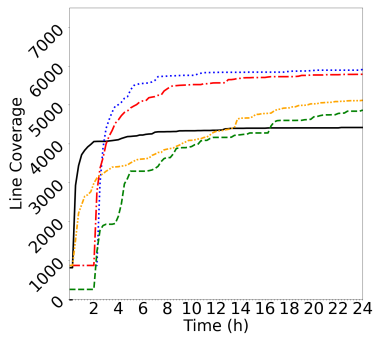
the `fsanitize-coverage=trace-pc-guard` flag), while ConfigFuzz is designed to fuzz whole programs with command-line options. We modified the FuzzBench scripts to allow running fuzzers on the whole programs following AFL's tutorial [8]. Specifically, the modified scripts build fuzzing targets using each 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 the target programs, as discussed in Section 4.3.

We ran each of ConfigFuzz-1, ConfigFuzz-2, ConfigFuzz-max, ConfigFuzz-str-1, ConfigFuzz-str-2, Baseline-def, and Baseline-2-way 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 [10]. For each of the five programs, we used an invalid seed which is Fuzzbench's default seed (a text file containing string `hi`) and one valid seed taken from AFL's repository [2] as seeds. 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 invalid seed for cxxfilt is the same text file as the other 5 programs. As ConfigFuzz takes part of the input data as configuration bytes, the remaining data bytes will not form a valid seed. Therefore, for each ConfigFuzz setup we prepended bytes on the original seeds to make sure the data bytes are valid.

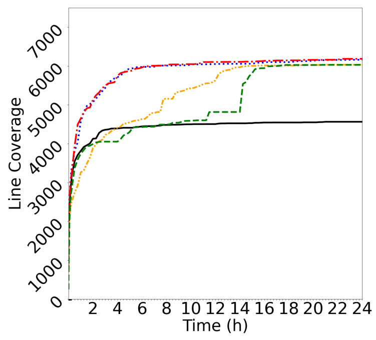
All experiments were conducted on a server with 2 Intel Xeon Silver 4116 CPUs (each with 24 cores) and 192GB of RAM running Ubuntu 16.04.

5.2 Results

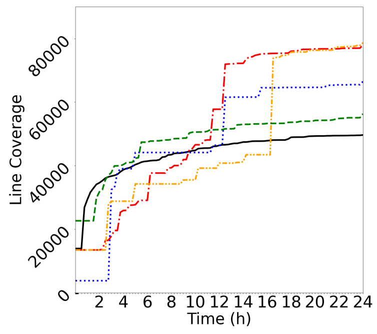
5.2.1 RQ1: Does ConfigFuzz outperform baselines? Figures 5 to 7 show line coverage growth of each fuzzer over time on all the six target programs. Results of default (Baseline-def), ConfigFuzz-1, ConfigFuzz-2,ConfigFuzz-max and covering arrays (Baseline-2-way) are represented by lines in black, blue, red, orange and green, respectively.



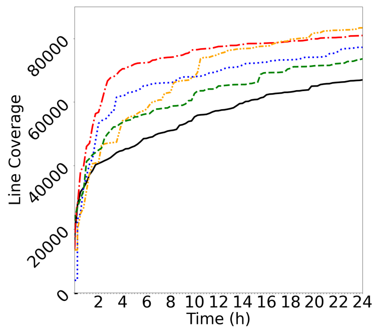
(a) AFL-cxxfilt



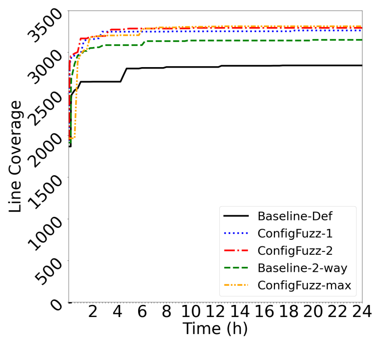
(b) AFL++-cxxfilt



(c) AFL-FFmpeg



(d) AFL++-FFmpeg



(e) AFL-gif2png

Fig. 6. Line coverage growth plots on cxxfilt, FFmpeg and gif2png.

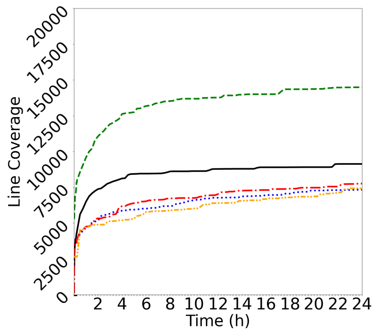
Overall, the performance of ConfigFuzz varied based on the target programs. ConfigFuzz clearly outperformed the baselines on xmllint, gif2png, cxxfilt and FFmpeg, as a result of ConfigFuzz prioritizing configurations that may lead to larger code coverage. However, Baseline-2-way and/or Baseline-def achieved higher coverage on nm and objdump. In general, ConfigFuzz performed better than the baseline settings on programs that can be well-explored in the 24-hour timeout (i.e., xmllint, gif2png, and cxxfilt) by effectively allocating resources in the whole configuration space. It is also possible for ConfigFuzz to perform well on more complicated programs (i.e., FFmpeg) due to the same reason. While on other complicated programs (i.e., nm and objdump), ConfigFuzz may spend a lot resources fuzzing the configurations that continuously generate new coverage, but not noticing that exploration of the other configurations may lead to even more coverage. We now analyze the performance on each program in detail.

For xmllint (Figures 5a and 5b), every setting of ConfigFuzz outperformed the two baselines by a large margin for both AFL and AFL++. ConfigFuzz-1, ConfigFuzz-2 and ConfigFuzz-max not only achieved higher coverage than Baseline-def and Baseline-2-way at a very early stage of the fuzzing campaign, but also grew faster. This is a strong indication that ConfigFuzz outperformed the baselines by exploring more command-line options. For both AFL and AFL++, Baseline-def only covered less than 7000 lines while every setting of ConfigFuzz covered more than 15000 lines. This result indicates that a large portion of xmllint code may not be reachable through its default configuration. Using more preset configurations (Baseline-2-way) did increase the coverage of the fuzzers, reaching 13000 lines, but still less effective than ConfigFuzz.

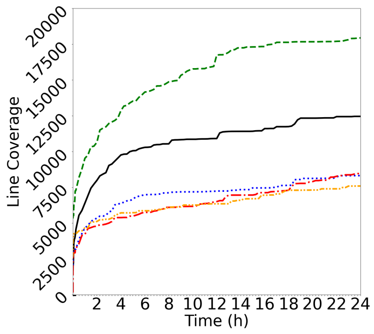
To provide more insights on how ConfigFuzz performed during the fuzzing campaign, we collected all the generated seeds and extracted their configurations. We analyzed the distribution of the options that appeared in these configurations. Columns 1-3 in Table 6 show the 5 most frequent options in ConfigFuzz-1, ConfigFuzz-2, and ConfigFuzz-max results. Most options rarely appeared (less than 5%) in the generated configurations. The three most frequent options in ConfigFuzz-1 results were `-html`, `-recover`, and `-repeat`; they appeared 15523, 7622, and 4004 times, respectively. These options were also frequently fuzzed by ConfigFuzz-2 and ConfigFuzz-max. This shows that ConfigFuzz was able to frequently fuzz these non-default options that led to higher coverage. By investigating the source code, we observed that enabling `-html` and/or `-recover` allows ConfigFuzz to reach many unique lines. On the other hand, enabling the `-repeat` option would iteratively execute xmllint's main functionality (which parses and prints the input xml file) 100 times. This may confuse the fuzzers on its potential to generate new coverage.

Results on cxxfilt (Figures 6a and 6b), FFmpeg (Figures 6c and 6d) and gif2png (Figure 6e) show similar trend where all settings of ConfigFuzz outperformed the two baselines, while Baseline-def always performed the worst. ConfigFuzz also consistently fuzzed a small set of options more frequently across the three ConfigFuzz settings. Different from xmllint, these frequently fuzzed options in gif2png did not contribute much to ConfigFuzz's performance. Actually, the number of unique lines exposed by enabling each individual option in gif2png is usually small. Nevertheless, fuzzing all options led to ConfigFuzz outperforming Baseline-def. The option `-w` caused the poor performance of Baseline-2-way. By enabling `-w`, gif2png lists images without animation or transparency, and exits earlier compared to other options. Baseline-2-way has about half of its 2-way covering arrays enabling `-w`. On the other hand, ConfigFuzz allocated much less resources on `-w`. Unlike xmllint, the baselines produced better coverage at some stages of the fuzzing campaigns. For example, for AFL-cxxfilt, Baseline-def achieved higher coverage than ConfigFuzz in the first two hours but was eventually surpassed in a few hours. This result suggests that ConfigFuzz may not always find the most effective configurations to fuzz at the beginning but was capable of finding such configurations given time. The most frequently fuzzed options by ConfigFuzz in cxxfilt, `--format` and `-t`, are both important. The main functionality of cxxfilt is to demangle a string, and the settings of `--format` decide the demangling style. Similar to the `-w` option in gif2png, disabling `-t` terminates cxxfilt's execution earlier compared to other options.

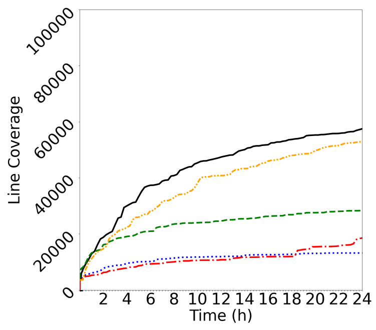
The results on nm (Figures 7a and 7b), on the other hand, show that ConfigFuzz performed worse than both baselines. Baseline-2-way was significantly better than other approaches, reaching 15000 and 17500 lines using AFL and AFL++, respectively. Even Baseline-def outperformed all ConfigFuzz settings. To understand this behavior, in Table 7, we extracted the 10 most frequently



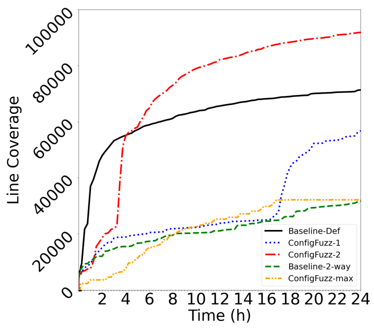
(a) AFL-nm



(b) AFL++-nm



(c) AFL-objdump



(d) AFL++-objdump

Fig. 7. Line coverage growth plots on nm and objdump.

Table 7. Top 10 most frequently fuzzed options of nm with AFL.

ConfigFuzz-1	ConfigFuzz-2	ConfigFuzz-max
−target (25%)	−target (60%)	−target (81%)
−defined-only (22%)	-a (15%)	-a (50%)
-a (11%)	-S (13%)	−special-syms (43%)
−quiet (10%)	−format (8%)	-s (39%)
−u (6%)	−no-recurse-limit (5%)	-l (28%)
-A (4%)	−size-sort (4%)	−demangle (26%)
-D (3%)	-D (4%)	−defined-only (24%)
−format (3%)	−defined-only (4%)	−quiet (24%)
-s (3%)	−no-demangle (4%)	−format (24%)
−size-sort (2%)	−radix (3%)	-A (21%)

fuzzed options in ConfigFuzz-1, ConfigFuzz-2, and ConfigFuzz-max using AFL. We observe that different options were fuzzed more frequently in these settings. Over the 24 hours fuzzing, the frequently fuzzed options such as `--target`, `-defined-only` and `-a` continuously generated new coverage, a potential reason why ConfigFuzz did not allocate more resources on other options. This indicates that ConfigFuzz might not be the most efficient when handling the large program and configuration space in nm.

AFL and AFL++ produced different results on objdump (Figures 7c and 7d). Baseline-def had the best performance using AFL with ConfigFuzz-max being the second. While using AFL++, ConfigFuzz-2 was the best and Baseline-def was the second. First, unlike most other programs, the coverage growth on objdump did not flatten after a few hours of fuzzing. This indicates that the search space of this program is large and all fuzzers; even when only the default configuration is fuzzed, the fuzzers continue discovering new coverage steadily over the 24 hours. Second, the results in Figures 7c and 7d show that the effectiveness of ConfigFuzz also depends on the fuzzer. Using AFL and AFL++, ConfigFuzz-2 performed significantly different, covering about 20000 and 90000 lines, respectively.

5.2.2 RQ2: How do ConfigFuzz-1, ConfigFuzz-2, and ConfigFuzz-max compare? Comparing the performance of ConfigFuzz-1, ConfigFuzz-2, and ConfigFuzz-max in Figures 5 to 7, ConfigFuzz-max did not always produce the highest coverage among these ConfigFuzz settings and ConfigFuzz-2 outperformed ConfigFuzz-1 in most cases. The performance of these ConfigFuzz settings was impacted by two aspects of the configuration space of the target programs. First, there exist interactions between two options, where some source code lines can only be reached by either enabling both options, or enabling one and disabling the other. Second, some options cause longer running time of the program execution, and when ConfigFuzz generates configurations including such option(s), the fuzzing process will be slower.

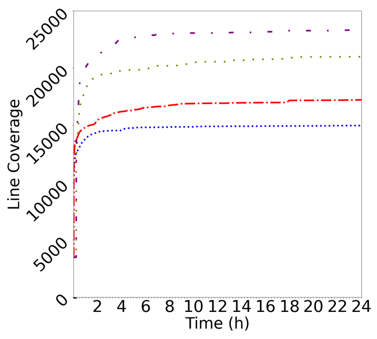
On xmllint (Figures 5a and 5b), ConfigFuzz-2 covered 18% more lines than ConfigFuzz-1 in 24 hours (statistically significant through Mann-Whitney U test [15, 20]), which can be attributed to option interactions. One of the options generated frequently by ConfigFuzz was `--html`. It interacts with `--push`, `--memory`, `--insert`, `--xlmout` and `--debugent` and was the main reason why ConfigFuzz-2 outperformed ConfigFuzz-1. Although ConfigFuzz-max outperformed ConfigFuzz-1, it was worse than ConfigFuzz-2. This result indicates that while covering option interactions may help improve the coverage, considering many option interactions may make ConfigFuzz less effective.

In Figure 6e the coverages of ConfigFuzz-1 and ConfigFuzz-2 on gif2png are close. This is because there is only one interaction in gif2png's options by enabling `-O` and disabling `-r`, and the interaction is associated with only a few unique lines.

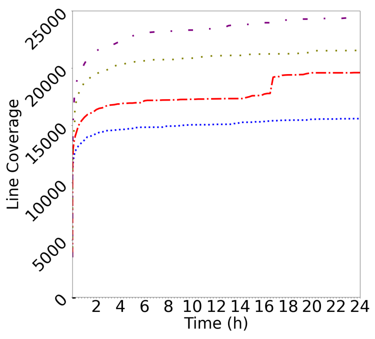
On objdump, as discussed above, due to the large size of this program, the performance of ConfigFuzz-1, ConfigFuzz-2, and ConfigFuzz-max varied significantly between AFL and AFL++. ConfigFuzz-max covered more than 3 times the number of lines than that of ConfigFuzz-1 or ConfigFuzz-2 using AFL, while ConfigFuzz-2 was the best when using AFL++. The performance between ConfigFuzz-1 and ConfigFuzz-2 is not significantly different in other programs, in most cases. Interestingly, ConfigFuzz-max in a few cases performed worse than both ConfigFuzz-1 and ConfigFuzz-2. This is because ConfigFuzz-max was slowed down by including too many options in its generated configurations. For example, on gif2png AFL ran about 11,000,000 executions with ConfigFuzz-2 and about 6,800,000 executions with ConfigFuzz-max.

5.2.3 RQ3: How does ConfigFuzz perform on string options? For the four programs with string options (FFmpeg, nm, objdump, and xmllint), we additionally fuzzed them including the string options. Figures 8 and 9 show the coverage growth plots comparing ConfigFuzz-1 and ConfigFuzz-2 with ConfigFuzz-str-1 and ConfigFuzz-str-2 for these programs. Results of ConfigFuzz-1, ConfigFuzz-2, ConfigFuzz-str-1 and ConfigFuzz-str-2 are represented by lines in blue, red, yellow and purple, respectively.

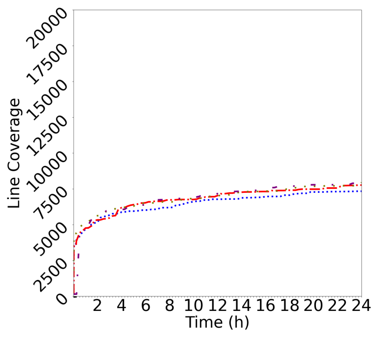
For xmllint (Figures 8a and 8b), ConfigFuzz performed better with string options included in the fuzzing space. ConfigFuzz-str-1 and ConfigFuzz-str-2 achieved higher coverage than ConfigFuzz-1 and ConfigFuzz-2 from early stage. At the end of 24 hours, fuzzing configurations with string



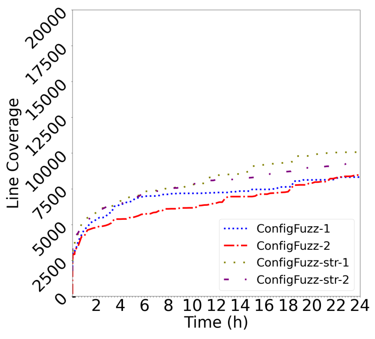
(a) AFL-xmllint



(b) AFL++-xmllint



(c) AFL-nm



(d) AFL++-nm

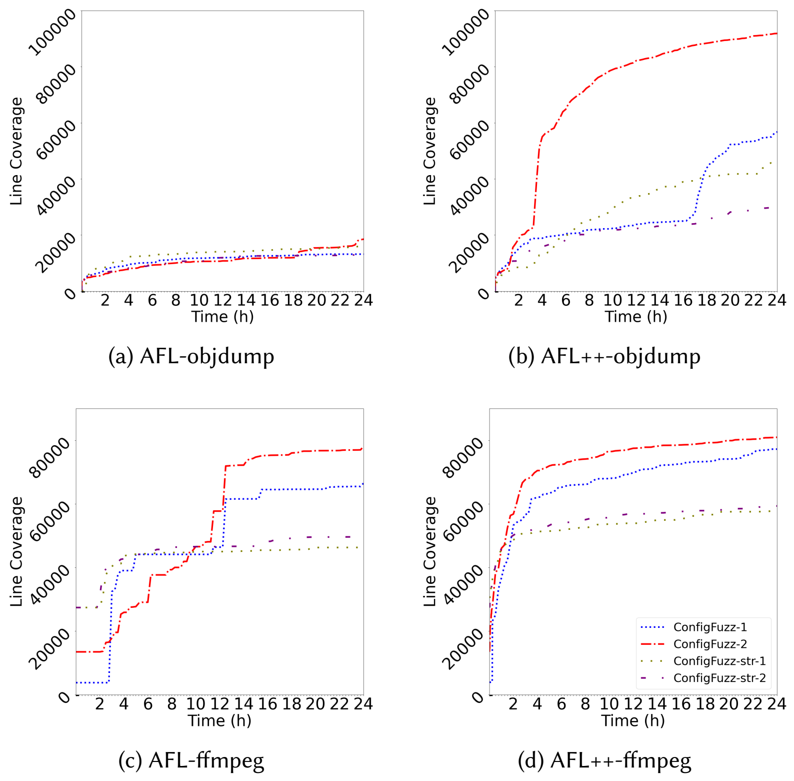
Fig. 8. Line coverage growth plots on string options with ConfigFuzz including string options on xmllint and nm.

options on xmllint resulted in more than 50% increase in line coverage. The string option `-xpath` was frequently fuzzed and enabled ConfigFuzz to reach many unique lines. Also, this string option was more frequently fuzzed by ConfigFuzz-str-2 than ConfigFuzz-str-1, as it was more likely to be generated in a configuration with 2 options, compared to a configuration with a single option.

Adding the string options did not help ConfigFuzz improve the performance on nm. As shown in Figures 8c and 8d, the coverages of ConfigFuzz-str-1 and ConfigFuzz-str-2 are similar to the coverages of ConfigFuzz-1 and ConfigFuzz-2. The only string option `--ifunc-chars` in nm was not frequently fuzzed by ConfigFuzz-str-1 and ConfigFuzz-str-2. This is likely because `--ifunc-chars` only controls 4 unique lines in nm and was quickly exploited by ConfigFuzz.

String options in objdump are mostly structured strings. For example, the `--ctf` and `--ctf-parent` options take section names which are hard to be generated by ConfigFuzz. An exception is `--source-comment`, which prefixes its setting to the source code lines. ConfigFuzz spent a lot of resources on this option, but did not achieve higher coverage. In Figures 9a and 9b, ConfigFuzz-str-1 and ConfigFuzz-str-2 did not outperform ConfigFuzz-1 and ConfigFuzz-2.

FFmpeg has 12 string options and we set the max size of string setting to be 19 in this evaluation. The implementation of ConfigFuzz enforced all options that are not bool type (23 in FFmpeg, as shown in Table 5) to have the same size of setting bytes, meaning that there needs to be 460 setting bytes to represent these options. When the input data size is smaller than the configuration bytes, ConfigFuzz will run the target program with the default configuration. ConfigFuzz-str-1 and



(a) AFL-objdump (b) AFL++-objdump

(c) AFL-ffmpeg (d) AFL++-ffmpeg

Fig. 9. Line coverage growth plots on string options with ConfigFuzz including string options on objdump and FFmpeg.

ConfigFuzz-str-2 were not able to generate large enough seeds during most of the fuzzing time, resulting in their worse performance than ConfigFuzz-1 and ConfigFuzz-2.

5.3 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 [20]. 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 (e.g., based on RevBugBench we recently developed [30] to allow fair comparison between fuzzers on programs with configurations.

6 CONCLUSIONS & FUTURE WORK

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. Our experiments on six programs show that ConfigFuzz outperformed the baselines on four programs while the results were mixed in the other two. We provided insights of these results through the analysis of options fuzzed by ConfigFuzz. We also compared the performance of ConfigFuzz using different parameters and including different options.

We identify several directions to explore in future work. First, as our empirical study shows, the coverage achieved by fuzzing configurations is associated with the unique branches covered by those configurations. While ConfigFuzz enables the fuzzers to allocate different resources on different configurations, the allocation may be further augmented with static analysis of each configuration. Second, fuzzing too many options together slows down the fuzzing process. Instead of enforcing the number of options in a configuration generated by ConfigFuzz, there could be a feedback loop adjusting the number of options in the generated configurations. Third, we observe that ConfigFuzz was unable to efficiently fuzz large configuration space and may frequently fuzz options that do not lead to the largest coverage. ConfigFuzz can be further improved by introducing an exploration stage, in which all configurations are allocated the same resources in order to identify the configurations that may lead to better performance.

ACKNOWLEDGEMENTS

This work was partly supported by NSF grants CCF-1816951, CCF-1955610, and CCF-2047682.

REFERENCES

[1] [n.d.]. AFL User Conversation. https://groups.google.com/g/afl-users/c/ZBWq0LdHBzw/m/zBlo7q9LBAAJ.
[2] [n.d.]. American Fuzzy Lop (AFL). https://lcamtuf.coredump.cx/afl/.
[3] [n.d.]. Automated Combinatorial Testing for Software (ACTS). https://www.nist.gov/programs-projects/automated-combinatorial-testing-software-acts.
[4] [n.d.]. Clang: a C language family frontend for LLVM. https://clang.llvm.org/.
[5] [n.d.]. cxxfilt. https://sourceware.org/binutils/docs/binutils/c_002b_002bfilt.html.
[6] [n.d.]. FFmpeg. https://ffmpeg.org.
[7] [n.d.]. FuzzBench: Fuzzer Benchmarking As a Service. https://github.com/google/fuzzbench/.
[8] [n.d.]. Fuzzing with afl-fuzz. https://afl-1.readthedocs.io/en/latest/fuzzing.html.
[9] [n.d.]. gif2png. http://www.catb.org/esr/gif2png/.
[10] [n.d.]. llvm-cov. https://llvm.org/docs/CommandGuide/llvm-cov.html.
[11] [n.d.]. nm. https://sourceware.org/binutils/docs/binutils/nm.html.
[12] [n.d.]. objdump. https://sourceware.org/binutils/docs/binutils/objdump.html.
[13] [n.d.]. xmllint. http://xmlsoft.org/xmllint.html.
[14] [n.d.]. Z3 Issue #4461. https://github.com/Z3Prover/z3/issues/4461#issuecomment-633988515.
[15] Andrea Arcuri and Lionel Briand. 2011. A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/1985793.1985795
[16] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-Based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1032–1043. https://doi.org/10.1145/2976749.2978428
[17] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. 1997. The AETG System: An Approach to Testing Based on Combinatorial Design. *IEEE Trans. Softw. Eng.* 23, 7 (July 1997), 437–444. https://doi.org/10.1109/32.605761
[18] Myra B. Cohen, Peter B. Gibbons, Warwick B. Mugridge, and Charles J. Colbourn. 2003. Constructing Test Suites for Interaction Testing. In *Proceedings of the 25th International Conference on Software Engineering* (Portland, Oregon) *(ICSE '03)*. IEEE Computer Society, USA, 38–48.
[19] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi

[20] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3243734.3243804

[21] Ahcheong Lee, Irfan Ariq, Yunho Kim, and Moonzoo Kim. 2022. POWER: Program Option-Aware Fuzzer for High Bug Detection Ability. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 220–231. https://doi.org/10.1109/ICST53961.2022.00032

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

[23] Changhai Nie and Hareton Leung. 2011. A Survey of Combinatorial Testing. *ACM Comput. Surv.* 43, 2, Article 11 (Feb. 2011), 29 pages. https://doi.org/10.1145/1883612.1883618

[24] Jiwon Park, Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2021. Generative Type-Aware Mutation for Testing SMT Solvers. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 152 (oct 2021), 19 pages. https://doi.org/10.1145/3485529

[25] V. Pham, M. Bohme, A. E. Santosa, A. Caciulescu, and A. Roychoudhury. 2021. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering* 47, 09 (sep 2021), 1980–1997. https://doi.org/10.1109/TSE.2019.2941681

[26] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.* 47, 1, Article 6 (June 2014), 45 pages. https://doi.org/10.1145/2580950

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

[28] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. On the Unusual Effectiveness of Type-Aware Operator Mutations for Testing SMT Solvers. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 193 (nov 2020), 25 pages. https://doi.org/10.1145/3428261

[29] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2021. Testing Configurations. In *The Fuzzing Book*. CISPA Helmholtz Center for Information Security. https://www.fuzzingbook.org/html/ConfigurationFuzzer.html Retrieved 2021-11-07 22:56:29+01:00.

[30] Zenong Zhang, Zach Patterson, Michael Hicks, and Shiyi Wei. 2022. FIXREVERTER: A Realistic Bug Injection Methodology for Benchmarking Fuzz Testing. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 3699–3715. https://www.usenix.org/conference/usenixsecurity22/presentation/zhang-zenong