CLIFuzzer: Mining Grammars for Command-Line Invocations

Abhilash Gupta
abhilash.gupta@cispa.de
CISPA Helmholtz Center for Information Security
Saarbrücken, Saarland, Germany

Rahul Gopinath
rahul.gopinath@cispa.de
CISPA Helmholtz Center for Information Security
Saarbrücken, Saarland, Germany

Andreas Zeller
zeller@cispa.de
CISPA Helmholtz Center for Information Security
Saarbrücken, Saarland, Germany

ABSTRACT

The behavior of command-line utilities can be very much influenced by passing command-line options and arguments—configuration settings that enable, disable, or otherwise influence parts of the code to be executed. Hence, systematic testing of command-line utilities requires testing them with diverse configurations of supported command-line options.

We introduce *CLIFuzzer*, a tool that takes an executable program and, using dynamic analysis to track input processing, automatically extract a full set of its options, arguments, and argument types. This set forms a *grammar* that represents the valid sequences of valid options and arguments. Producing invocations from this grammar, we can fuzz the program with an endless list of random configurations, covering the related code. This leads to increased coverage and new bugs over purely mutation based fuzzers.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

fuzzing, CLI Options, command-line, utilities

ACM Reference Format:
Abhilash Gupta, Rahul Gopinath, and Andreas Zeller. 2022. CLIFuzzer: Mining Grammars for Command-Line Invocations. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3540250.3558918

1 INTRODUCTION

Command line utilities are programs that use the command-line interface (CLI) as their user interface. Such programs are the mainstay of the UNIX environment as well as numerous other operating systems. The command-line interface that these utilities rely on follow a simple formula:

```
$ ⟨utility⟩ ⟨parameter⟩*
```


ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9413-0/22/11…$15.00
https://doi.org/10.1145/3540250.3558918

⟨*start*⟩ ::= ⟨*utility*⟩ ⟨*optexpr*⟩* ⟨*argument*⟩*
⟨*utility*⟩ ::= `ls`
⟨*optexpr*⟩ ::= `-a` | `-l` | `-w` ⟨*int*⟩ | ...
⟨*int*⟩ ::= {*i* | *i* ∈ positive_integers()}
⟨*argument*⟩ ::= {*f* | *f* ∈ files()}

Figure 1: The invocation grammar of `ls` (excerpt)

Here, the ⟨`parameter`⟩s are either (1) command-line switches (such as `-a`, `-v`, etc.), which control some behavior of the program being invoked; or (2) data (such as a file name) that is being passed into the program to be processed.

Given their importance, numerous previous studies have focused on testing command-line utilities. However, one limitation of all these studies is that they only fuzz the `stdin` of programs under fuzzing—either ignoring the options that the program accepts, or using a specific sequence of options [11]. Unfortunately, we cannot simply treat a command line as if it were just another input source, as CLI utilities expect specific options with a specific syntax.

A recent approach to invocation fuzzing thus has turned to *parse program documentation* [6], including the output produced by the `--help` option, to obtain a set of valid options. However, as with all documentation, this information may be non-existent, incomplete, or no longer up-to-date.

In this paper, we introduce CLIFuzzer—a tool that *automatically* determines option syntax from code. Our technique is based on the observation that most utilities use a standard *option parser* such as `getopt()` for parsing their options [5]—an observation we first made in 2019, in a chapter in the "Fuzzing Book" textbook [15, "Testing Configurations"]. Here, we prototyped an instrumention of the Python `argparse` argument parsing module in order to create a grammar of command-line parameters.

CLIFuzzer now turns this prototype into a full-fledged command line fuzzer for C programs using `getopt()`. However, the `getopt()` specification string is not sufficient to obtain a grammar, as it does not reveal the *types* of the individual arguments. Hence, we also extract the *type* of argument that the utility expects, by tracking which calls to *runtime library functions* get an argument from the command line.

As a result, CLIFuzzer obtains a *grammar* that accurately describes valid input parameters for the utility under consideration. Figure 1 shows a fragment of the final grammar recovered from `ls`. CLIFuzzer uses this grammar to produce an endless sequence of valid invocations, effectively fuzzing the command line.

2 THE GETOPT() FUNCTION

The standard `C` library functions [3] used to parse command-line invocations are (1) `getopt()`, (2) `getopt_long()` and

(3) `getopt_long_only()`. These functions have two arguments that define the possible options.

- The `optstring` argument is a string which contains information about the `short options` of the utility. The structure of `optstring` is described by the grammar below.
  ⟨*optstring*⟩ ::= ⟨*prefix*⟩ ⟨*optionletter*⟩+
  ⟨*prefix*⟩ ::= `-` | `:` | `+` | ϵ
  ⟨*optionletter*⟩ ::= ⟨*letter*⟩
  | ⟨*letter*⟩ `:`
  | ⟨*letter*⟩ `::`
  | `W;`
  ⟨*letter*⟩ ::= {c | c ∈ ascii() ∩ is_graph(c) } - {;,:,-}
  Each letter in the ⟨*optstring*⟩ represents an option that may be present in command-line parameters. A letter may be followed by a `':'` which means that that option, if present in the parameters, requires an option-argument. As an example of processing, consider the options to `ls`. The `ls` utility accepts (among others) two short options `-a` and `-l` that are essentially boolean switches while `-w` takes an argument that specifies the column width as an `unsigned int`. This is encoded into an optstring `"alw:"`.
- The `longopts` argument is a pointer to an array of the `struct option` which describes the `long options` accepted by a utility. This `struct` is described in Listing 1.

Listing 1: struct option

```
struct option {
  const char *name;// Name of the option
  int has_arg;     // Does the option expect an
     argument?
  int *flag;       // Flag for returning results
  int val;         // The value to return
};
```

Long options such as `--long` are stored as string in the `name` field of an `option`. `val` is the value to return when encountered; a value of `'l'` makes `--long` an alias for `-l`, which is intended.

3 MINING PARAMETER SPECIFICATIONS

CLIFuzzer uses *context-free grammars* enriched by a few generative predicates as the parameter specification. We construct this grammar in three steps. (1) Converting option string to a context free grammar; (2) Converting option arguments to predicates; and (3) arguments to predicates.

3.1 Constructing Grammars from Option Specs

In this step, we convert the short option (`optstring`) and long option (`longopts`) specifications to a context-free grammar.

In order to mine these specifications, we use shadow versions of `getopt()` variants that log the parameters they were called with. We use a shared library that contains the shadow variants. This shared library is force-loaded into the utility under evaluation by overriding `LD_PRELOAD`. Hence, when the utility is invoked, the option specifications are logged.

Once CLIFuzzer has extracted the `optstring`, the algorithm in Listing 2 constructs the list of short options expected by a utility in the grammar. At this stage, the grammar encodes the argument requirement of all options but does not reflect their type and defaults them to strings. Similarly, the set of long options is also extracted and inserted into the grammar.

Listing 2: Constructing grammar options from optstring

```
def gen_options(optstring):
  grammar = {}
  options = []
  if optstring[0] == '-':
     options.append('-<letter>')
     optstring = optstring[1:]

  elif optstring[0] in {':', '+'}:
     optstring = optstring[1:]

  while optstring:
     optchar, *optstring = optstring
     if optstring[:2] == '::':
        option = '-%s %s' % (optchar, '<str>')
        options.append(option)
        option = '-%s' % optchar'
        options.append(option)
        optstring = optstring[2:]
     elif optstring[:1] == ':':
        option = '-%s %s' % (optchar, '<str>')
        options.append(option)
        optstring = optstring[1:]
     else:
        option = '-%s' % optchar
        options.append(option)

  grammar['<option>'] = options
  return grammar
```

The algorithm in Listing 2 first checks whether the given `optstring` starts with a hyphen. If it does, it indicates that the utility accepts any unspecified option letters without an immediate error, postponing validation for later (Section 2). Hence, we append `-<letter>` to the context-free grammar to account for this fact. If the `optstring` starts with `:` or `+` it affects how missing arguments are indicated to the program. However, it has no direct impact on option specification. Hence, we skip over these letters.

Within the `while` loop, we extract each option letter, and check if any are followed by `:` or `::`. If an option letter is followed by a single colon (`:`), it indicates a *mandatory* option-argument, while a double colon (`::`) indicates an *optional* option-argument. These are used to update the context-free grammar.

Long options from `longopts` is a data structure that does not require parsing, and is directly translated to the context-free grammar.

3.2 Mining Option Argument Types

The next step in CLIFuzzer is ascertaining the type of option-arguments expected by options. For this, CLIFuzzer starts by scanning `libc` looking for functions that take a string argument. CLIFuzzer then injects *shadow variants* for each of these functions such that invoking any of these functions would result in the arguments being logged.

CLIFuzzer invokes the program under test with a random argument for each option expecting arguments to determine the kind of argument required. For example, if the option takes in a filename as argument, the utility calls `open` or `stat` variants to operate on

the passed argument. Similarly, a utility would call one of `atoi()` or `strtol()`'s variant functions and one of `atof()` or `strtod()`'s variant functions to parse integer and floating-point number arguments respectively.

3.3 Arguments to Predicates

The final step in CLIFuzzer grammar construction involves determining the argument requirements of the utility. Arguments to utilities are similar to option-arguments except that utilities can expect multiple arguments. Hence, the utility is invoked with multiple number of arguments to determine how many arguments the utility expects, and the type of argument is determined similar to option-arguments from Section 3.2.

3.4 Using CLIFuzzer

CLIFuzzer is implemented as `clifuzzer`. A sample invocation that extracts the grammar from the `ls` command is as follows:

```
clifuzzer --get-grammar -o ls.json ./coreutils/ls
```

This extracts the invocation grammar of `ls` to the file `ls.json`. Given this grammar, one can fuzz the `ls` command as follows:

```
clifuzzer -f 1000 -g ls.json -o ls.out ./coreutils/ls
```

The `-g` option specifies the invocation grammar, and `-f` option specifies how many invocations to produce.

4 EVALUATION

For our evaluation, we fuzzed the latest versions of 44 command-line utilities written in `C` and `C++` in Linux. These utilities use one of the `getopt()` variants to parse their invocation. We specifically chose those utilities that used one of the `getopt()` variants for parsing as our grammar construction is dependent on mining specifications from `getopt()`. We also limited our study to utilities that take at least one file argument so that we have at least one entry point for delivering completely random input for fuzzing. Our subjects include seven of the nine utilities (`as`, `bison`, `dc`, `gdb`, `ptx`, `spell`, and `troff`) that reported failures in Linux by Miller et al [11]. The number of options of the utilities ranged from 0 to 103 (mean 27.18, std deviation 24.13). The lines of code of the utilities ranged from 101 to 81215 (mean 10246.79, std deviation 20458.43).

CLIFuzzer logs the execution results in five groups: (1) Crashes exit-code > 128. (2) Unresolved invocations $2 <$ exit-code ≤ 128. (3) Passing invocations exit-code $= 0$. (4) Graceful handling of invalid options: exit-code $\in 1, 2$. (5) Exceptions such as `TimeoutExpired`. For each utility, we look for crashes and exceptions. Any exception found is manually verified to check whether it was a hang or happened by design (e.g., waiting to read from `stdin`). We then manually replicate and confirm the failures.

4.1 Comparison with AFL++

To evaluate whether CLIFuzzer is able to progress beyond the state of the art, we used AFL++ [2] as the baseline, producing file and `stdin` inputs. For fuzzing with AFL++, each utility had a set of small valid seed inputs (size < 1K) which we produced by hand such that when the utility in question read the file, it returned exit-code `0`. We ran AFL++ for three hours on each utility without any options in its invocation. We evaluate how much options can influence coverage by comparing the coverage achieved by CLIFuzzer and AFL++, respectively.

Table 1: Utilities reporting failures in their latest versions

as	bc	*bison*	cat	cmp	col	colcrt
column	colrm	comm	cut	*dc*	diff	expand
fmt	fold	*gdb*	grep	head	join	look
m4	nl	nm	od	paste	pr	*ptx*
rev	sdiff	*spell*	strings	strip	sort	**tac**
tail	**tee**	tr	*troff*	**tsort**	unexpand	uniq
wc	xargs					

Utilities where Miller [11] reported a bug, but has not yet been fixed in the latest versions are *italicised*. New failures found in latest versions by CLIFuzzer are **bolded**.

AFL++ fuzzed each utility for three hours. CLIFuzzer performs better than most (41/44) of those utilities. It performs significantly better for some utilities such as `spell` and `column` than others. This observation can be explained on the basis of the utilities' set of options. The improvement in CLIFuzzer's code coverage for a utility is directly proportional to its number of valid options. When a utility has a large set of valid options, CLIFuzzer achieves significantly better coverage than AFL++ since options do not form a part of AFL++'s fuzzing process. When a utility does not have a lot of valid options or even no options (e.g., `rev` or `tsort`), then CLIFuzzer covers comparable code (or marginally more) than AFL++. During the fuzzing campaign, AFL++ found crashes in `gdb` and `col`. The crash in `gdb` is same as what CLIFuzzer found while CLIFuzzer could not replicate the crash in `col`. This is because the input needs to contain particular characters in a particular order which CLIFuzzer's input generation technique did not achieve during its 20 runs but AFL++ did, because of its coverage driven fuzzing nature.

4.2 Comparison with Miller et al.

Another relevant baseline is the fuzzing effort by Miller et al. [11]. Our effort here is to verify that CLIFuzzer can at least replicate the bugs that were found by Miller et al. on the particular versions that Miller et al. tested.

CLIFuzzer is able to replicate all of the failures reported in the latest study on utilities in Linux (conducted by Miller et al). In fact, our approach finds an extra failure, a crash, in `spell`. We also observe that `as` and `ptx`'s failures are triggered only when particular options are used[1][2].

4.3 Exploration of Utilities

Finally, to understand whether current utilities are robust, we also run our fuzzer against the current versions of all utilities, looking for any bugs that still exist.

We tested the latest versions of all 44 utilities. Our findings were that in almost all cases, the bugs have not been fixed, despite their publication in the previous study. The only exceptions are `spell`, which fixed its crash issue in the latest version, but not its hang issue and `bison`, which fixed its hang issue in the latest version, but

[1] `as-new -a < 18`
[2] `ptx --references --traditional testopt`

now it has a new crash failure. Interestingly, this failure is caused only when a `hidden option` is part of the invocation[3]. This option `--trace` or `-T` is not documented on `bison`'s manpage or infopage but is part of the valid options set that it expects. Since CLIFuzzer extracts this set of valid options from the code, this option is part of `bison`'s grammar and we were able to discover this failure. Finally, the same input crashes `gdb`'s older and latest version but with different return codes (`SIGSEGV` in older version, `SIGABRT` in latest version).

Among the other 37 utilities, CLIFuzzer discovered failures in four. In particular, two different input files induced crashes in `column` with two unique return codes (`SIGSEGV` and `SIGABRT`). Furthermore, the hangs in `tac` and `tee` are induced only when particular options are used in the invocation[4] [5]. For `tac`, the option-argument to `--separator` option (which expects any string) needs to be of the form `.+<int>`. This (option,option-argument) combination, along with the `--regex` option and a particular inputfile leads to a hang. Interestingly, code segment where it gets stuck in during the hang is in regexec.c of `coreutils` which is used by `tac`. This file is included by four other utilities – `csplit`, `expr`, `nl` and `ptx`. Hence, it is possible that this failure could be induced in these utilities too.

`tee` is a utility that reads from `stdin` and writes to the `stdout` and files. One of its options is `--append`, which appends to a file instead of overwriting it. When the contents of a file is redirected into `tee` while also writing to the same file in the append mode, `tee` enters an infinite loop. That is, the file keeps doubling in size until all memory is exhausted.

5 RELATED WORK

5.1 Fuzzing CLIs

Fuzzing was first mentioned by Miller et al [9] where they conducted *random* black-box tests on CLI utilities. They tested 88 utilities across 7 different versions of UNIX[6] and found failures in at least 24% of utilities tested on each system. This study focussed solely on the non-option argument (mostly `stdin` and files) of the utilities as the source of random input. The input consisted of random files of size 1KB to 1MB.

Miller et al [8] repeated their previous experiment by fuzzing 135 CLI utilities on MacOS X. The study reported a 7% failure rate, which is comparable to the best results (GNU utilities) of their previous study [10].

American Fuzzy Lop (AFL) [14] is a popular fuzzing tool, and is used in fuzzing programs such as CLI utilities. However, it focusses on the `stdin` and file input to a program. Other variants derived from AFL such as AFLGo [1] and AFL++ [2] also limit their focus to `stdin` and file input.

5.2 Fuzzing Command-Line Arguments

A small number of approaches focuses on command-line *arguments* as fuzzing targets.

AFL++ has an experimental "`argv` fuzzing" mode in which it sends its random input to the command line rather than files or `stdin`. As "`argv` fuzzing" does not specifically aim for creating options or arguments, it is anything but efficient; its creator states that "it's just not horribly useful in practice" [7].

Ghosh et al [4] conducted black-box random tests on eight GNUWin32 CLI utilities on Windows NT platform. They developed a tool named RIDDLE which utilized some of the options of the utilities (via a grammar) to fuzz them. The study reported that 23.41% of test runs resulted in utilities exiting abnormally with system error conditions and 1.55% of the test runs resulted in hung applications (after 33600 test runs for each utility).

Sutton et al [12] developed iFUZZ, a tool that requires the user to submit `optstring` arguments of the `getopt()` function of applications. They report more than 50 failures in IBM AIX 5.3 using iFUZZ.

Wang et al [13] included options to execute guided fuzzing on CLI utilities with the specific goal of reaching specific targets in programs and maximizing its coverage. The handpicked options are specified manually as a grammar in a `Protobuf` specification and fed to their fuzzing tool to guide its fuzzing.

Lee et al. [6] designed a study to fuzz 30 programs incorporating both options and arguments. They first extracted a set of options from the documentation of programs (e.g., man pages and help messages). Then they determined a subset of options that cover as much of the program's functions as possible. These options are used to construct ten invocation strings which are then used to fuzz the programs, during which, only the argument input files are mutated. The source of random input was still the argument even if some options were included in the invocation strings. Lee et al. reported crashes in 19 out of the 30 programs they tested.

In contrast to CLIFuzzer, all of the above approaches require some amount of human effort to infer full command-line invocations.

6 CONCLUSION

Command line utilities are one of the most commonly used programs in operating systems such as UNIX. Hence, they need to be highly reliable. Previous fuzzing research on these utilities focused only on the standard input stream, relying on a few hand-picked options. This can, however, be non-optimal, as option interactions may also contain bugs and vulnerabilities.

In this research, we show how to extract the parameter specifications from `getopt()` calls, and enhance it with generative predicates mined from program executions.

We fuzzed 44 CLI utilities in Linux, and found failures in 25% of the fuzzed utilities. Within the reported failures, 45% of the failures could only be found due to parameter interactions.

Finally, CLIFuzzer achieves more coverage in lesser time on average than AFL++, thus increasing the likelihood of finding failures faster.

CLIFuzzer and all data, including replication and demonstration packages, are available at:

https://github.com/vrthra/fse2022-clifuzzer

[3] `bison --trace s1`
[4] `tac --separator=.+5 --regex E.coli`
[5] `tee --append FILE/README < README`
[6] Not all the utilities were available on all operating systems.

REFERENCES

[1] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *ACM SIGSAC Conference on Computer and Communications Security*. 2329–2344.

[2] 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)*.

[3] Free Software Foundation, Inc. 2021. *Parsing program options using getopt.* https://www.gnu.org/software/libc/manual/html_node/Getopt.html Accessed: 2021-12-15.

[4] Anup K Ghosh, Viren Shah, and Matt Schmid. 1998. An approach for analyzing the robustness of Windows NT software. In *Proc. 21st National Information Systems Security Conference, Crystal City, VA, USA*. 383–391.

[5] GNU Project. 2021. *Standards for Command Line Interfaces.* https://www.gnu.org/prep/standards/standards.html#Command_002dLine-Interfaces Accessed: 2021-12-15.

[6] Ahcheong Lee, Irfan Ariq, Yunho Kim, and Moonzoo Kim. 2022. POWER: Program Option-Aware Fuzzer for High Bug Detection Ability. In *ICST*.

[7] Michal Zalewski. 2015. *Struggling to give inputs to AFL.* https://groups.google.com/g/afl-users/c/ZBWq0LdHBzw/m/zBlo7q9LBAAJ Accessed: 2022-03-15.

[8] Barton P Miller, Gregory Cooksey, and Fredrick Moore. 2006. An empirical study of the robustness of macOS applications using random testing. In *international workshop on Random testing*. 46–54.

[9] Barton P Miller, Louis Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33, 12 (1990), 32–44.

[10] Barton P Miller, David Koski, Cjin Pheow Lee, Vivekandanda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. 1995. *Fuzz revisited: A re-examination of the reliability of UNIX utilities and services.* Technical Report. University of Wisconsin-Madison Department of Computer Sciences.

[11] Barton P Miller, Mengxiao Zhang, and Elisa Heymann. 2020. The Relevance of Classic Fuzz Testing: Have We Solved This One? *IEEE Transactions on Software Engineering* (2020).

[12] Michael Sutton, Adam Greene, and Pedram Amini. 2007. *Fuzzing: brute force vulnerability discovery.* Pearson Education.

[13] Zi Wang, Ben Liblit, and Thomas Reps. 2020. TOFU: Target-Orienter FUzzer. *arXiv preprint arXiv:2004.14375* (2020).

[14] Michal Zalewski. 2021. *American fuzzy lop (2.52b).* https://lcamtuf.coredump.cx/afl/ Accessed: 2021-12-15.

[15] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2019. *The Fuzzing Book.* https://www.fuzzingbook.org Accessed: 2021-12-15.