Expanding Fuzzing of Critical Program Configurations via Coverage-Based Differential Testing

David Clark
University of Utah

UUCS-24-001

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

15 April 2024

Abstract

Modern software is configurable, meaning it has features that can be enabled or disabled to affect the program's output and behavior. However, configurable code can be difficult to test as each configuration (i.e., each combination of features) can introduce new execution paths, and therefore new bugs. Ideally, each configuration would be tested, but in practice, this is infeasible, as many software products have prohibitively many configurations to test in-depth. This problem can be mitigated by selecting and testing common configurations, but this is insufficient to safeguard the many variant configurations that will be used in practice.

In my work, I develop a new, semi-automated tool to reduce the difficulty of testing configurable software. The tool generates compile-time configurations and tests them against a fixed set of inputs, performing a kind of configuration-fuzzing as opposed to traditional input-fuzzing. The tool identifies critical compile-time configurations that exercise code not reached by the default configuration. These "interesting" configurations can later be tested in-depth by traditional fuzzers to reveal bugs not found in the default configuration. This coverage-driven, differential exploration of the compile-time configuration space is the main innovation of my approach. I evaluate this tool by testing open-source libraries with publicly available fuzzing harnesses (primarily libraries hosted by Google's OSS-Fuzz). With this work, I hope to provide a way of efficiently testing programs with large compile-time configuration spaces.

EXPANDING FUZZING OF CRITICAL PROGRAM CONFIGURATIONS VIA COVERAGE-BASED DIFFERENTIAL TESTING

by

David Clark

A Senior Honors Thesis Submitted to the Faculty of
The University of Utah
In Partial Fulfillment of the Requirements for the

Honors Degree in Bachelor of Science

In

Computer Science

Approved:

[illegible]

Stefan Nagy
Thesis Faculty Supervisor

Mary Hall
Director, School of Computing

Thomas C. Henderson
Honors Faculty Advisor

Monisha Pasupathi, PhD
Dean, Honors College

April 2024

ABSTRACT

Modern software is configurable, meaning it has features that can be enabled or disabled to affect the program's output and behavior. However, configurable code can be difficult to test as each configuration (i.e., each combination of features) can introduce new execution paths, and therefore new bugs. Ideally, each configuration would be tested, but in practice, this is infeasible, as many software products have prohibitively many configurations to test in-depth. This problem can be mitigated by selecting and testing common configurations, but this is insufficient to safeguard the many variant configurations that will be used in practice.

In my work, I develop a new, semi-automated tool to reduce the difficulty of testing configurable software. The tool generates compile-time configurations and tests them against a fixed set of inputs, performing a kind of configuration-fuzzing as opposed to traditional input-fuzzing. The tool identifies critical compile-time configurations that exercise code not reached by the default configuration. These "interesting" configurations can later be tested in-depth by traditional fuzzers to reveal bugs not found in the default configuration. This coverage-driven, differential exploration of the compile-time configuration space is the main innovation of my approach.

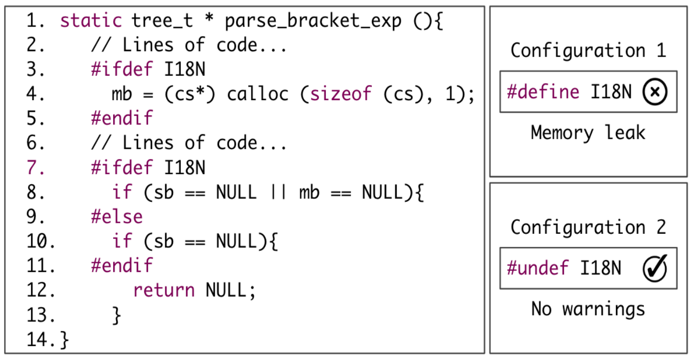
I evaluate this tool by testing open-source libraries with publicly available fuzzing harnesses (primarily libraries hosted by Google's OSS-Fuzz). With this work, I hope to provide a way of efficiently testing programs with large compile-time configuration spaces.

TABLE OF CONTENTS

INTRODUCTION

Configuration is an essential part of the way software is developed and used [1]. It allows developers to write a single program or library that can be used in many different ways. If a configuration is taken to be a combination of compile-time options, the number of different configurations available to users becomes exponential with the number of options. Unfortunately, this means there are exponentially many configurations to test, and testing them all is typically infeasible [2]. Worse still, configurable code is especially important to test, as configuration introduces unique interactions and complexities that create bugs [3]. Some configurations may even be necessary to reach certain code paths [4], so testing a code base without testing against multiple configurations may limit the amount of code coverage that can be achieved.

The figure below [5] shows an example of a configuration-related bug. In this case, if the code is compiled with the macro "I18N" enabled, then a memory buffer is allocated, but never freed, resulting in a memory leak. Bugs like this cannot be detected without statically or dynamically exploring the configuration space.



```
static tree_t * parse_bracket_exp (){
   // Lines of code...
   #ifdef I18N
     mb = (cs*) calloc (sizeof (cs), 1);
   #endif
   // Lines of code...
   #ifdef I18N
     if (sb == NULL || mb == NULL){
   #else
     if (sb == NULL){
   #endif
       return NULL;
     }
}
```

```
Configuration 1
#define I18N
Memory leak

Configuration 2
#undef I18N
No warnings
```

Figure A: A code snippet of Gawk with a memory leak when we enable macro I18N.

It’s important to note that “configuration option” typically refers to an argument to a build system, not directly to a compiler-level macro like “I18N” as shown in the example above. Configuration options may enable or disable combinations of several such macros.

This is not a new problem, so researchers and developers have created solutions for mitigating these issues. Configuration sampling is a common approach that attempts to select a subset of configurations that represents the configuration space well or that has some other desirable property (e.g., speed, code coverage)[6]. There are many distinct sampling algorithms to balance completeness and efficiency, such as t-wise sampling or LSA [5], [7], but this approach is typically unable to identify and exploit configurations with the most value, e.g., configurations that can be fuzzed to reach the highest code coverage. In an attempt to give more direction to configuration testing, researchers have also used fuzz-testing to generate and evaluate new run-time configurations [6], [8], [9]. These solutions can identify promising run-time configurations using feedback from the testing process, although they can’t directly be applied to discover compile-time configurations. More about related work is found in the discussion section below.

Fuzz testing as an approach is particularly important, given its success and widespread use for finding input-related vulnerabilities. A core insight of this thesis is to apply the automated testing and mutation mechanisms that make fuzzing effective to find configuration-related vulnerabilities.

In this thesis, I develop and test an algorithm to effectively fuzz compile-time configurations in libraries. The algorithm discovers configurations that maximize code coverage for a given harness, and these configurations can later be fuzz tested in depth to discover new bugs. By evaluating coverage for a single harness, I limit the resulting configuration space to only those configurations that affect the execution path of the harness. The algorithm generates random configurations using a provided set of options; these configurations are built and run against a small set of test cases, and the ones that provide more coverage than a default configuration are deemed “interesting” and are used to generate new potentially interesting configurations. When the algorithm terminates, the interesting configurations are ranked, providing the user with a prioritized list of configurations to fuzz test or investigate further.

I go on to use this algorithm to test several open-source libraries with available fuzzing harnesses. I analyze configurations found this way, and discuss the findings with regard to four main research questions, shown below.

RQ1: Is configuration fuzzing useful for many already-fuzzed libraries?

RQ2: How do our generated configurations improve code coverage?

RQ3: What kinds of configurations cause increased coverage?

RQ4: Can we identify configurations that are known to have bugs?

My main contribution is the application of fuzzing techniques to the problem of testing large configuration spaces.

METHODS

Algorithm

The proposed algorithm addresses the configuration space problem by “fuzzing” configurations. The algorithm’s key assumption is that the configurations that achieve the highest code coverage when fuzzed traditionally will be the same as the configurations that achieve the highest code coverage when run on a small, fixed (but diverse) set of inputs. Thus, I can build configurations and test them against a few inputs to identify interesting configurations that yield new code coverage, and these will have the potential to reveal new bugs when tested in depth. These coverage-increasing configurations will also be referred to as “critical configurations.”

For the algorithm to work properly, it must first receive these inputs: an open-source library written in C/C++; a set of boolean-valued build-system-defined compile-time options; a driver program (with a fixed set of run-time options, if desired); and a list of input-files to be supplied to the driver program. For best results, the driver program should be a fuzzing harness. For libraries with a complex build system, a build script that compiles the library and driver program is useful.

In my implementation of this algorithm, the program receives a single command line argument: a file path to a setup.json file that defines these key inputs, along with other parameters such as how many configurations to test and what build system to use.

Now, in more detail, I will describe how configurations are generated, built, executed, and evaluated. Finally, I will describe the algorithm’s output.

For specificity's sake, a "configuration" is taken to be a set of options, where each "option" is a string. In particular, each option is a command-line flag for the build system whose inclusion is sufficient to change the build product. Since these are boolean-valued options, they can also be negated via some simple string operations. Options are defined by each library's build system, and options for various build systems may be formatted differently to account for the different ways in which build systems accept options as command line arguments.

When generating configurations, the algorithm first aims for simplicity and breadth: given N boolean-valued compile-time options, the first configuration tested will be the "default" or "null" configuration, and the next N configurations tested will activate just one option each.

After this initial step, the random mutation and feedback-driven selection step begins. The "queue" of configurations to mutate includes any configurations that were deemed "interesting" when they were run. From this queue, a random configuration is selected and "mutated," meaning that each option in the configuration is given a small chance to "flip," either from active to inactive or from inactive to active.

Active options are enabled when the library is built. Inactive options can either be explicitly disabled or else omitted (so that the library's default setting will be used), depending on a program-wide setting. Most libraries were tested by explicitly disabling options, except where doing so led to the vast majority of configurations failing.

To test these new configurations, the program must first build them, or rather, use them to configure the library and then build the library. In the implementation of this algorithm, options are stored as strings (or flags) that can be inserted as-is into a build-system’s compilation command, and configurations are stored as sets of options. These representations make building the library straightforward, as it suffices to concatenate all of the options in the configuration and append them to the terminal command that builds the library. Some configuration options may be incompatible with each other, resulting in the configuration failing to build; such configurations are logged, but otherwise ignored. To collect code coverage information, I compile all code with clang/clang++ and instrument the code with the “--coverage” option. I use llvm-cov’s gcov tool to generate coverage files.

To generate the coverage data needed to gauge a configuration’s importance, I need to execute the library’s code by running a selected driver program. Any program worth testing operates on inputs, so I need to have a small supply of input files likely to exercise a significant amount of the program’s functionality. These input files will be run against every successful build of the program. Because some inputs may stall the program, the configuration fuzzer should time out if the program does not terminate quickly.

After building the library and running the program, I can evaluate how “interesting” the configuration was by generating .gcov files and counting the number of lines unique to this configuration. To do this, I need to find the set of lines covered by the current configuration and calculate the set difference, with the set of all lines covered by the program so far. If my “new coverage” set is non-empty after removing all lines covered by previous configurations, then I really have found an interesting new configuration. If I perform traditional input fuzzing on this configuration, I can reach code that would not be covered by the same fuzzer on the default configuration. The elements of this new coverage set are then added to the set of all lines covered, and the configuration is added to the pool of interesting configurations, which will be mutated to generate new configurations.

The configuration fuzzer stops after generating and building a user-defined number of configurations. At this point, the algorithm outputs 1) the “default” configuration, with the number of lines it covered; 2) the set of interesting configurations, each with the number of lines not covered by the default configuration, expressed as a percentage of the default configuration’s number of lines covered; and 3) debug data, including a list of all lines covered for each configuration tested. The interesting configurations are ranked in descending order. Configurations are expressed in the output by concatenating their options, prepending each option with a fixed identifier (i.e., the ‘#’ character), and upper-casing the final string. In most cases, the “Default” configuration is taken to be the null configuration, i.e., the configuration without any options specified; this configuration is expressed in the output as “#DEFAULT”. For example, if a program was tested with configurations “#DEFAULT” (100 lines covered), “optA” (90 lines covered, with 10 lines not shared by the default), and “optB” (20 lines covered, with 15 lines not shared by the default), the resulting output would look like this:

```
'Default' config: #DEFAULT
#DEFAULT 100
#OPTB 15%
#OPTA  10%
```

This output can be used by a tester or developer to prioritize these critical configurations: this algorithm effectively reduces the compile-time configuration space to only these critical configurations.

While the algorithm stops at identifying critical configurations, there is still work to do before any bugs can be found. A common use of this algorithm might be to identify critical configurations and to automatically begin fuzzing campaigns on the most promising of them. Automating this process would be relatively straightforward, but is beyond the scope of this thesis or of the implementation described herein. Another use might be to identify a list of critical configurations for manual evaluation and development purposes.

Experiment

To test the utility of this algorithm and to improve the quality of open-source software in future work, I created a prototype (referred to as “the configuration fuzzer” or “the tool”) and ran it against 14 open-source libraries to see if there was an opportunity to increase code coverage for specified harness programs by incorporating compile-time configuration fuzzing. If libraries had untested code and features locked behind certain configurations, my tool could ideally find them.

Thirteen of these libraries were taken from OSS-Fuzz, Google’s automated open-source fuzzing platform [10]. Libraries in the ecosystem have public source code and fuzzing harnesses, as well as publicly accessible coverage data. Some of these were chosen for their scale and popularity. Others were chosen for their low overall coverage, with the idea being that a library’s low coverage may be explainable by its inaccessible, configuration-locked code.

One of these libraries, ngiflib, was chosen specifically because it was a small, simple library with a known configuration-related bug (see issue 18, which is accessible from the following URL: https://github.com/miniupnp/ngiflib/issues/18).

In general, each library was tested in whatever version was most recent as of late 2023. Specifics can be found in the data publicized with this thesis.

The experiment consists of building a fixed number of configurations for each library and evaluating the interesting and failing configurations identified this way. I ran my experiment on a 24-core, 64 GB RAM machine running the Ubuntu Linux distribution. Each processor is a 12th Gen Intel(R) Core(TM) i9-12900K unit. Some libraries were tested in a virtual Docker container used to manage dependencies and speed up development.

While open-source projects provided public harnesses, they didn't explicitly provide a list of configuration options. Configuration options are arguments to the build system, and so must be extracted from different build-systems in different ways.

Because Autotools and CMake are among the most common build systems (both in general and among libraries in the OSS-Fuzz system) [11], this project operated only on libraries that could be built using these systems. Other build systems, such as Meson, were rarely needed and are not currently supported, but are within the scope of our work.

Briefly, Autotools specific options can be gleaned by running the "configure" program with the -help option. Only options under the "optional features" and "optional packages" headers were considered in this experiment. Options are enabled by passing them as "--FEATURE" flags to either the "automake" or "configure" scripts.

CMake options are defined explicitly in a “CMakeLists.txt” file, where options that are meant to be configured by users are defined with an “option()” macro. The script that gathers options scans for anywhere the option() macro exists and grabs its argument. The pool of options is all of the strings that can be found this way. Options are enabled by passing them as “-DFEATURE” arguments when running cmake.

Technically, the tool also supports working directly with Makefiles. While Makefiles do not constitute a build system, some libraries (notably ngiflib) don’t provide anything else. In these simple cases, Makefiles can be treated as a primitive build system. In this case, options are gleaned by scanning all source files for #ifdef macros. Options are enabled by passing arguments directly to the compiler to define them as boolean-valued macros.

Not all options are likely to increase code coverage or extend the code with new features. Some options found this way are generic, being common to all configurable programs (e.g., options that define the installation prefix). Some options control platform or architecture specific attributes, some options link the library with optional dependencies (specified with a “PATH” argument), some options include debugging information, and some introduce new features. Of these, only the last category is particularly interesting. While any feature-defining option could be interesting, only boolean-valued options are explored in this experiment, as these have the smallest number of possible settings and can be most simply represented and set. The options to be fuzzed are collected in an “options” file to be passed as input to the configuration fuzzer.

Some options are necessary to test the program (e.g., any “--build-fuzzers” option). These options will not be fuzzed, but are specified with a fixed setting in a “necessary_options” file, and are included in every tested configuration. When such necessary options are used, a program’s “Default” configuration will not be the null (i.e. zero-option) configuration but rather will be the set containing only the necessary options.

Our goal with running the program is to accurately gauge how much code coverage can be gleaned from this configuration. I want to maximize code coverage while not spending too much time on any single configuration. To this end, I need to have a short list of inputs that are likely to exercise as much of the program as is possible. Practically, this means having a mix of short inputs, long inputs, and malformed inputs. If there is an existing fuzzing corpus (perhaps stored in the OSS-Fuzz repository, or in a separate “...-corpus” git repository), valid inputs can be selected from that, otherwise, it may be necessary to run a preliminary input-fuzzing campaign to generate a corpus. Malformed inputs can be collected by, again, running an input-fuzzer and selecting malformed inputs from its queue.

In general, the configuration fuzzer is meant to operate on fuzzing harnesses, which can be tested shallowly when executed with a few fixed inputs or in-depth when fuzzed with a barrage of strategically mutated inputs. It is important that the shallow and in-depth testing are done on the same program, as the configurations that are important for one code path are likely to be different from the configurations that are important to another code path. It is possible, however, to run the configuration fuzzer against non-fuzzing harness driver programs, although this eventually makes fuzz-testing the resulting critical configurations slightly more difficult.

Most of the libraries tested in this experiment were run with fuzzing harnesses used in the OSS-Fuzz project. These harnesses were either stored in the OSS-Fuzz git repository under “projects” or in the project’s own repository. In some cases projects will have multiple harnesses. In which case this experiment selected the harness with the best available corpus, and which achieved the highest code coverage in a preliminary test.

In general, building the library is accomplished by executing a simple sequence of commands determined by the build system (e.g., “./configure” then “make” for Autotools), with configuration-specific options inserted as flags into the configuration command. This works in simple cases when a library and its fuzzing executable can be built without any library-specific modifications to this process. However, most libraries tested had additional commands or scripts that needed to be run and could not be automated a priori. To address this, the configuration fuzzer accepts an optional “build script” value, which is a path to an executable that will be run in place of the stock build sequence. The build script must make use of a “CONFIG_TESTER_OPTIONS” environment variable, which is a string made by concatenating options, defined by the configuration tester.

The configuration fuzzer itself is a Python 3 program that takes as input a user-written .json file that specifies options, inputs, and which driver program to use, among other things. Importantly, a user must specify a number of configurations to test. The output of this program is a list of configurations that increase code coverage, ranked by how many lines they covered as a percentage of the number of lines covered by the default configuration. Additional debug information, such as how many configurations failed to build and what specific lines were covered by each configuration, is written to a designated folder.

This experiment does not fuzz-test the configurations identified by the configuration fuzzer. In a practical bug-hunting use case, this would be the logical next step. Input fuzzing can be accomplished with any of the state-of-the-art techniques used in academia or industry.

To increase development speed, several libraries were tested inside Docker containers. The OSS-Fuzz repository defines Dockerfiles and scripts for automatically building libraries in a default configuration, and these can be easily modified to build diverse configurations. The OSS-Fuzz Dockerfile installs all necessary dependencies which avoids a tedious development step. However, the extra virtualization layer incurs an unknown performance cost and should be avoided in optimizations of this code or in future work.

As a brief aside, a substantial number of configurations generated by the tool were impossible to build. There are two reasons why these builds failed. In some cases, certain combinations of options were incompatible with each other, and this incompatibility caused the build process to fail. In other cases, certain criteria that were required by a certain option (e.g., a certain dependency being installed) were not met, and all configurations containing this option failed. Knowingly wasting time on an option that will not build is not ideal, so currently, the choices we have are either to remove the option from the set of tested options (after identifying the option as useless, and while the configuration fuzzer isn’t running) or otherwise to accept the penalty of partially building configurations that are not useful. In the future, the algorithm can be modified to notice options or combinations of options that result in failed builds and to avoid them automatically.

RESULTS

The experiment aimed to answer four main research questions, listed below. The experiment evaluated 14 open-source libraries.

RQ1: Is configuration fuzzing useful for many already-fuzzed libraries?

RQ2: How do our generated configurations improve code coverage?

RQ3: What kinds of configurations cause increased coverage?

RQ4: Can we identify configurations that are known to have bugs?

RQ1: Is configuration fuzzing useful for many already-fuzzed libraries?

Of the 14 libraries tested, the configuration fuzzer was able to identify coverage-increasing configurations in 11. Most of the libraries evaluated were tested with 100 configurations, though some were tested with fewer. Some libraries had significantly more coverage increasing configurations (notably libxml2, and to a lesser extent libssh and Simd) while most had 1 or 2. Figure A summarizes this information for the libraries in the experiment.

API	API Description	Lines of Code	Number of Configurations Tested	Number of Covg-Increasing Configurations
ngiflib	GIF library	1.6K	32	3
libxml2	xml library	416.9K	100	24
libpng	png library	104.9K	100	2
libssh	ssh library	100.3K	100	7
matio	MATLAB MAT file I/O library	36.0K	100	2
Simd	optimized image processing library	399.8K	100	5
cups	network printing library	348.1K	100	3
mosh	shell	16.4K	100	1
libgit2	Git implementation	264.1K	100	1
unbound	DNS resolver	183.9K	100	2
neomutt	email program	341.5K	100	2
tmux	terminal multiplexer	72.8K	30	0
lldpd	LLDP protocol implementation	40.8K	30	0
sqlite	database	398.2K	100	0

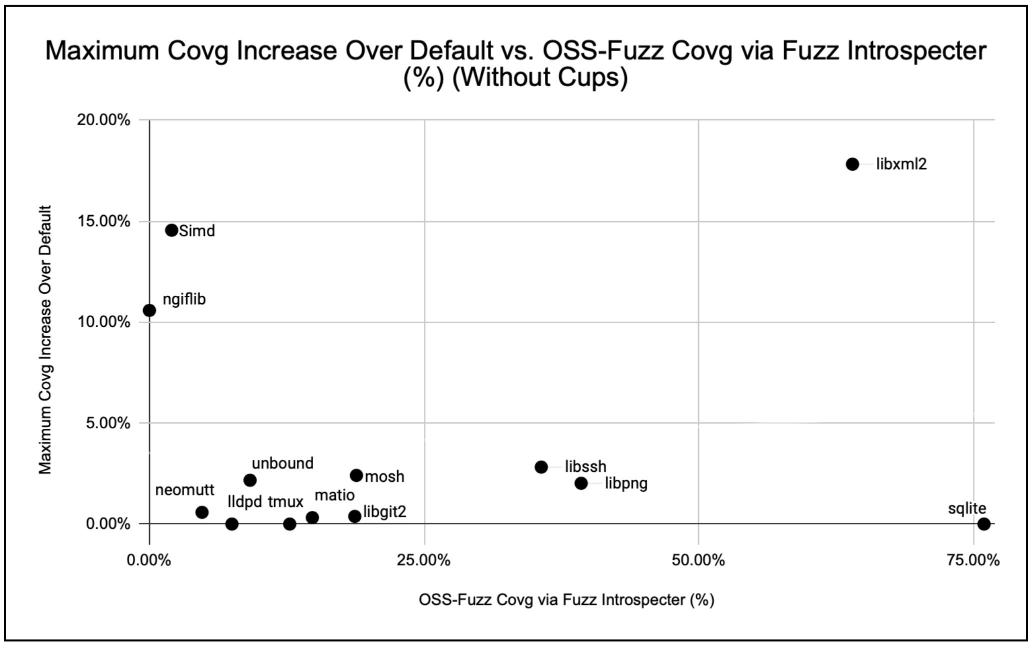
Table A: Libraries Tested with # of Critical Configurations

The figure shows a diverse set of libraries of differing sizes (as measured in lines of code, using the “cloc” utility).

Logically, it seems likely that testing a library in more configurations increases the probability of finding more coverage-increasing configurations. Future iterations of this work should focus on increasing the speed at which configurations can be tested, so that higher numbers of configurations can be tested efficiently.

It’s worth noting that some libraries contain more variability than others due to having more or less code enabled by optional features. It remains an open question how much code we are failing to test and what libraries are most vulnerable. With that noted, one of this project's aims is to identify what might make a library a good candidate for configuration fuzzing.

One hypothesis is that libraries that are already being fuzzed, with low overall coverage, might reveal more code coverage when fuzzed across multiple configurations. To test this, I collected and plotted each library's code coverage, as collected in the OSS-Fuzz dataset, and plotted it against the maximum coverage increase for a single configuration from my experiment (Figure B). Unfortunately, my data shows little connection between these two variables, indicating that low OSS-Fuzz coverage is not a good predictor of high configuration related coverage potential.



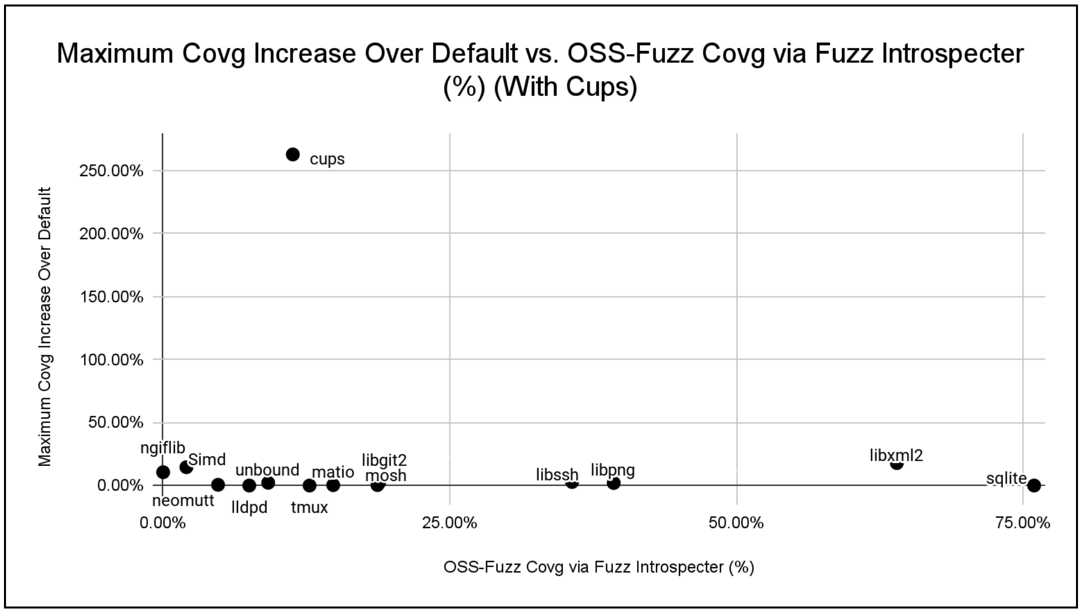


Figure B - OSSFuzz Coverage vs Highest Critical Configuration's Coverage

Future work might apply static analysis to see if low OSS-Fuzz coverage is concentrated in configuration-dependent code. Libraries meeting this condition may be more productively tested via configuration fuzzing.

RQ2: How do our generated configurations improve code coverage?

For this technique to be worth the investment, it must be shown that code coverage can be increased by a meaningful amount, not just that code coverage can be increased. To measure this, I collected the total number of lines of code run for each coverage-increasing configuration. As a way of normalizing the data, each configuration's coverage is reported relative to the default configuration's coverage. For example, if a library's default configuration covers 100 lines of code and a new configuration covers 120 lines of code, the coverage-increase over default would be reported as 20%.

This way of representing coverage avoids a key problem with the way code coverage is often calculated. Typically, code coverage is expressed not as a count of lines executed, but rather as a percentage of the total number of lines in the program, where this total only represents the number of lines in the compiled executable; this measurement excludes source code that was left out of the current configuration, and so underestimates the true amount of code left untested. Importantly for this analysis, the typical measure of code coverage makes comparisons across configurations (which will have different denominators, i.e., total lines of compiled-in code) unnecessarily challenging.

The table below shows the number of coverage-increasing configurations found for each library, along with the single highest coverage-increase over default for any configuration (Figure B).

API	Driver Program	Default Config LoC	Number of Configurations Tested	Number of Covg-Increasing Configurations	Highest Covg Increase Over Default
ngiflib	gif2tga	165	32	3	10.58%
libxml2	xmllint	3861	100	24	17.82%
libpng	pngtest	4159	100	2	2.02%
libssh	ssh_client_fuzzer	3828	100	7	2.82%
matio	matio_fuzzer	1878	100	2	0.32%
Simd	simd_load_fuzzer	1870	100	5	14.55%
cups	FuzzCUPS	596	100	3	262.75%
mosh	terminal_fuzzer	290	100	1	2.41%
libgit2	midx_fuzzer	1056	100	1	0.38%
unbound	parse_packet_ fuzzer	276	100	2	2.17%
neomutt	address-fuzz	1901	100	2	0.58%
tmux	input-fuzzer	1760	30	0	-
lldpd	fuzz_lldp	117	30	0	-
sqlite	ossfuzz	3973	100	0	-

Table B: Coverage Increasing Configurations

The figure above shows substantial variation among libraries, both in terms of the number of coverage-increasing configurations found and in terms of the amount of coverage contributed by the best.

RQ3: What kinds of configurations cause increased coverage?

Shown below is a table correlating a subset of libraries with the two most interesting configurations identified, where each configuration is identified as a set of uppercase options. While this is meant to be a somewhat ad-hoc exercise (different libraries will require different feature options to unlock code) there are some noticeable trends. Importantly, some options (e.g., “debug” or “enable-unit-test” options) increase coverage significantly, though it’s doubtful these configurations are likely to discover new bugs. The “cups” library, with its 232% increase in coverage over the baseline, is an example of this. In other cases, we seem to be identifying true feature-related code coverage with the potential for identifying new bugs.

API	Best + Percent Covg Increase Over Default	Second Best + Percent Covg Increase Over Default
ngiflib	-DDEBUG	-DNGIFLIB_NO_FILE
	19.39%	9.09%
libxml2	--WITH-C14N --WITH-CATALOG --WITH-DEBUG --WITH-FTP --WITH-ICU --WITH-MEM-DEBUG --WITH-OUTPUT --WITH-READER --WITH-SAX1 --WITH-THREADS --WITH-TREE --WITH-VALID --WITH-WRITER --WITH-XPATH --WITH-XPTR --WITH-MINIMUM --WITH-LEGACY --WITH-TLS --WITH-COVERAGE --WITH-PYTHON-SYS-PREFIX --WITH-PYTHON_PREFIX	--WITH-GNU-LD --WITH-C14N --WITH-DEBUG --WITH-HISTORY --WITH-HTTP --WITH-MEM-DEBUG --WITH-OUTPUT --WITH-RUN-DEBUG --WITH-SCHEMATRON --WITH-TREE --WITH-XINCLUDE --WITH-XPATH --WITH-MINIMUM --WITH-LEGACY --WITH-FEXCEPTIONS --WITH-COVERAGE --WITH-PYTHON_EXEC_PREFIX
	17.82%	16.14%
matio	--ENABLE-EXTENDED-SPARSE	--ENABLE-MAT73
	0.32%	0.32%
Simd	SIMD_SHARED=OFF SIMD_AVX512 SIMD_AVX512VNNI SIMD_AMXBF16 SIMD_PERF SIMD_GET_VERSION SIMD_SYNET SIMD_HIDE SIMD_INSTALL SIMD_UNINSTALL	SIMD_SHARED=OFF SIMD_PERF
	14.55%	14.01%
cups	--ENABLE-DEBUG-PRINTFS --ENABLE-UNIT-TESTS --WITH-CUPS-BUILD --WITH-CACHEDIR --WITH-PKGCONFPATH --WITH-RCDIR --WITH-SNMP-ADDRESS	--ENABLE-UNIT-TESTS
	262.75%	232.72%
libgit2	DEBUG_STRICT_ALLOC	N/A
	0.38%	N/A

Table C: Top 2 Critical Configurations for a Subset of Libraries

RQ4: Can we identify configurations that are known to have bugs?

The results for ngiflib are particularly interesting. Among the libraries tested, ngiflib is unique. It is the smallest library tested, and it isn't hosted on OSS-Fuzz. More importantly, ngiflib has a known configuration bug in the version identified by commit SHA 0245fd4. The bug can be reproduced by compiling the library with the "-DNGIFLIB_NO_FILE" option defined as a macro, then running gif2tga against a certain pathological input. With this bug, ngiflib serves as a valuable test case for whether or not the configuration tester can identify critical configurations that lead to real bugs. As expected, the configuration tester correctly identified "-DNGIFLIB_NO_FILE" as a coverage-increasing configuration (as seen in Table C above) within 8 seconds, meaning a fuzzing campaign could be run on this configuration to reveal the bug. This supports the idea that configuration fuzzing can complement traditional fuzzing to improve testing effectiveness.

As a side note, the proportion of configurations that failed to build varied significantly across libraries. Interestingly, the libraries with zero failing builds are the same as the libraries that failed to reveal any coverage increasing configurations. This potentially shows something going wrong in the build process, for instance, the configuration fuzzer may be failing to modify the output when it intends to change the configuration.

API	Number of Covg-Increasing Configurations	Percent of Builds Failed
ngiflib	3	65.63%
libxml2	24	36.00%
libpng	2	56.00%
libssh	7	76.00%
matio	2	0.00%
Simd	5	34.00%
cups	3	40.00%
mosh	1	27.00%
libgit2	1	22.00%
unbound	2	22.00%
neomutt	2	38.00%
tmux	0	0.00%
lldpd	0	0.00%
sqlite	0	0.00%

Table D: Failed Builds

Finally, while these results overall seem somewhat promising, it’s important to consider the cost. Recompiling the library for each configuration is an expensive, time-consuming process taking as long as an hour and seventeen minutes (in the case of libssh) to test just 100 configurations. For this process to see wide-spread application, it is important that future work is done to accelerate this process.

The figure below shows each library's total testing time (in seconds) plotted against its total lines of code. The plot indicates that the total time to test each configuration varied significantly across libraries and cannot be well explained on the basis of the size of the code base. The median time to generate and test the configurations in this experiment is 860 seconds.

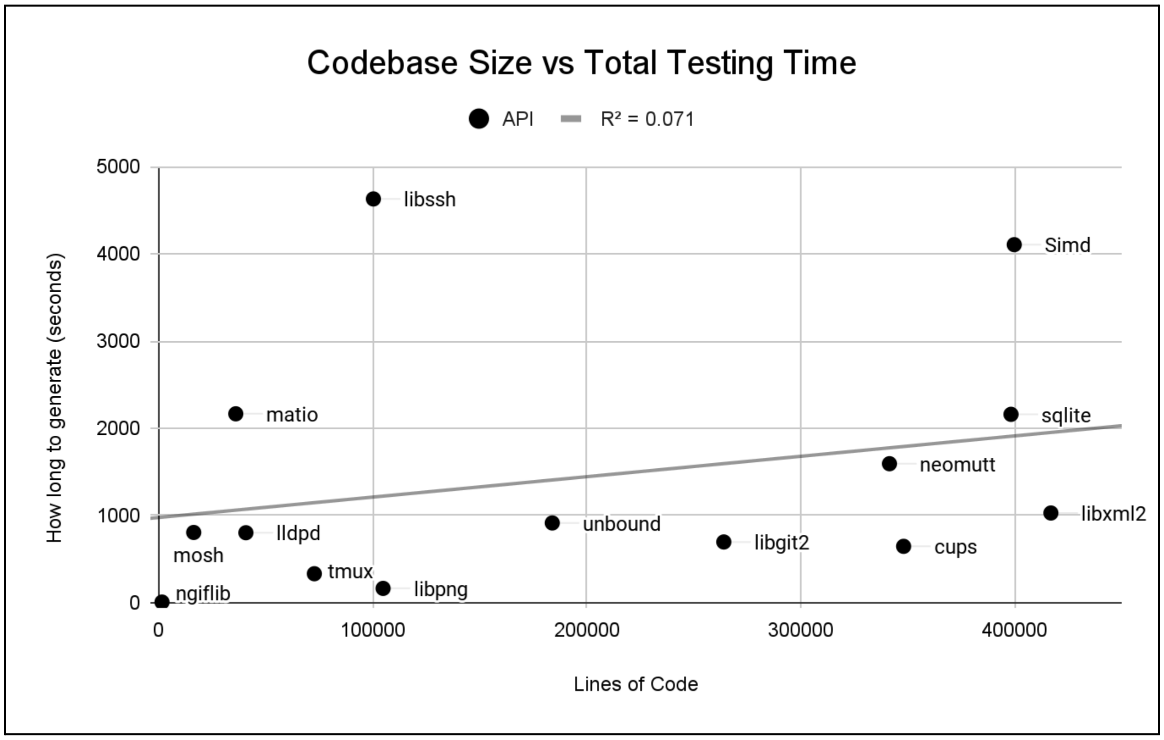


Figure C: Codebase Size vs Total Testing Time

DISCUSSION

The experiment here shows that there is some potential for increasing code coverage by identifying and testing critical configurations. Of 14 diverse libraries, 11 of them had coverage-increasing configurations. For the median library, the single most critical configuration covered a modest 2% more code than the default configuration. While these gains are slight, they indicate that there is room for software testing to be improved by fuzzing the configuration space in this way.

Threats to Validity

Before discussing related work and future directions for this project, I will now discuss any threats to validity relevant to my experiment.

Unfortunately, my evaluation did not use an explicit development-test split over the libraries, so it’s possible the algorithm may be overly tuned to the specific libraries used. However, the libraries used are diverse in function and size, which might mitigate problems of overfitting, and the specific performance metrics of the tool aren’t as important to my thesis as is the underlying concept and the demonstration that this is a productive direction for future research.

Additionally, my experiment doesn’t compare my algorithm to any competing tools, only to a default-configuration baseline. In part, this is because of how new this area of research is, and how few competing products there are.

Related Work

This work is not the first to address the problem of testing large configuration spaces, so I will now acknowledge other works that have motivated or attempted to solve this problem.

Previous research has established a need for more thorough testing of programs' configuration spaces. Papers have shown that different configurations exercise different code paths, making it possible that full code coverage can only be achieved by testing diverse configurations [6]. Configurability itself may be responsible for bugs, due to the added complexity of configurable code [3]. This increases the need for better tools to test configurable code.

To support research into the testing of configurable software, researchers have collected and qualitatively analyzed an assortment of configuration-related bugs [12].

One apparent solution is to test all configurations, though this isn't feasible for many programs and teams. Research has given us some idea of how costly it is to test all configurations of a (moderately sized) program. For JHipster, a web development library, building and testing each of 26,257 configurations took 4376 hours of CPU time [2]. Given this cost, few researchers or developers will decide to test all configurations, but many researchers don't test more than a single configuration. In a survey of 98 fuzzing papers from 2015 to 2021, only 3 papers ran tests with more than one (run-time) configuration [8].

A more practical solution is to test a manually selected set of configurations that is determined to be representative of all populations or otherwise to be representative of essential use cases. The JHipster development team, for example, uses such a sample of twelve configurations [2]. This approach allows developers or researchers to apply domain knowledge about which configurations are most important, and the resulting set of configurations is usually small enough to test frequently, e.g., each time new code is committed. While this approach is likely to safeguard the most common configurations, it is also likely to miss lesser-used configurations that can be just as dangerous when built.

A more thorough approach to testing variability involves sampling the configuration space systematically. Various methods have been proposed, with varying levels of complexity and efficiency [6], [7].

Researchers have also tried testing run-time configuration spaces via fuzz-testing. Some have tried directly fuzzing a program's configuration space [6]. This kind of fuzzing is difficult to apply to compile-time configuration spaces, as changing the configuration requires rebuilding the code base as opposed to just modifying a string of options. The researchers behind the fuzzing tool POWER [8] tried fuzzing in distinct stages, using a preliminary stage that identifies interesting configurations and a main stage that tests the input spaces for those specific configurations. My work takes these configuration-fuzzing ideas and applies them in a build-time configuration context.

Earlier this year, researchers from the University of Florida and Iowa State University published a configuration fuzzing tool called CONFIZZ [13], which operates similarly to the tool I've presented in this thesis. It generates and tests combinations of "configuration variables," (e.g., the argument to #ifdef directives) which I have referred to as compiler-level macros. My approach generates and tests combinations of build-system defined options, with the idea that real, deployed applications are more likely to be configured this way, rather than by defining specific compiler-level macros. From a software user's perspective, it may be sufficient to only test configurations that are possible to build via the build system. From an optimization perspective, it may be desirable to test combinations of build-system defined options rather than combinations of compiler-level macros, as there may be fewer of the former; this limits the combinatorial explosion inherent to configuration testing.

Future Work

While this thesis develops the concept of configuration fuzzing and introduces a prototype implementation, there remains much work to do before this technique can be usefully applied to improve real software.

Importantly, additional work needs to be done to develop and test the full configuration fuzzing pipeline, starting from gathering data to generating and evaluating configurations (done in this thesis) to running input fuzzing campaigns and ultimately finding bugs.

While this thesis has shown that automatically generated configurations can improve code coverage when testing shallowly (i.e., running a program with 10 to 20 fixed inputs), it has not shown what effect these configurations have when tested deeply (i.e., fuzzing a program with potentially thousands of inputs each second). It may be that these configurations provide dramatically more coverage, or else that they don't contribute much new coverage beyond the shallow testing phase. It may be that fuzzing these configurations reveals many new bugs, or else that the expense of running these extra tests is not justified by the number of new bugs discovered. We will not know if this work is not continued.

A good intermediate goal would be to expand on the results provided by ngiflib, i.e., to apply the configuration fuzzer to more libraries with known configuration-specific bugs to see if the fuzzer can identify the critical configurations that produce the bugs. If it can, this strengthens the case that configuration fuzzing can be used to find real bugs in real software, and thus can be used to improve the quality of software testing.

Additionally, a more thorough study might shed light on why some libraries didn't have any automatically identifiable critical configurations. My hypothesis is that the configuration-related code may not have been executed because certain properties were not present in the inputs fed to the driver programs, but I cannot give an evidence-supported description of why some libraries behaved this way, so this can only be answered by future research.

Future work should also focus on optimizing and accelerating the tool. Much of the success of modern fuzz testing is attributable to fuzzing's speed. With that in mind, fuzzing compile-time options is a comparatively daunting task, because rather than simply changing an input file or command line argument, each new test case must be compiled from source code. Optimizing the configuration fuzzing algorithm is therefore an important step in future work. I will now briefly discuss some potential targets for optimization.

This project can already use a limited degree of parallel computing to speed up fuzzing. At present, recompiling the library under test for each new configuration is a major time sink, but this can be mitigated by using multiple cores to handle build tasks. With future work, it may be possible to run more tasks in a distributed way, thus accelerating the process.

Future work might be able to eliminate unimportant build-time tasks, such as performing library checks or irrelevant tests.

Currently, the configuration fuzzer makes heavy use of strings and string operations to store line numbers and calculate difference-sets. Future work might benefit from using more compact representations and data structures.

At present, the tool's configuration generation strategy is very limited. It effectively has one mutation operator (i.e., randomly flipping options' settings), and it doesn't infer or act-on potentially useful information, such as what options or combinations of options are likely to result in failed builds. This could be made smarter in future work.

As a final direction for improvements, future versions of the tool should be extended to new situations and improved with more features.

For this tool to ever be more than a prototype, it will of course need certain usability features, such as the ability to specify a desired runtime (as opposed to the current "number of configurations to test"), or to terminate the fuzzer early without losing data. These aren't a high priority, as they aren't important to the overall functioning of the fuzzer, but their absence occasionally hinders testing efforts.

This tool could be made more general by supporting other, less common build systems. While CMake and Autotools are de facto standards, alternatives such as Meson may be common enough to justify adding support for them.

The tool could be expanded to explore more of the configuration space by allowing it to build configurations including non-boolean valued options. Currently, the algorithm generates configurations containing only boolean-valued options, i.e., options that can be simply enabled or disabled without any other parameters. In the future, the algorithm could be improved with support and mutation strategies for integer-, string-, and enum-valued options in configurations as well. This could potentially increase the number of critical configurations identified, and may expand the total amount of new coverage gained from this technique.

Currently, the tool only operates on C or C++ code, but the technique could be extended to other languages. The compiler and instrumentation used would need to change, among other things, but expanding the idea this way would be a valuable result.

Main Conclusions

In this thesis, I developed configuration fuzzing as a concept and implemented a prototype. This configuration fuzzing scheme addresses the widespread problem of failing to adequately test configurable software by applying an automated testing approach with insights from traditional fuzzing.

I demonstrated that this technique can improve code coverage for a reasonably high proportion of libraries (11 of 14 studied), and that code coverage is typically improved by a modest amount (a median 2% increase for the highest-coverage critical configuration). I was able to characterize some of the configurations that increase coverage (importantly, which configurations increase coverage without being likely to reveal new bugs), although I was unable to identify an a priori pattern in which libraries were likely to yield substantial new coverage when tested with the configuration fuzzer.

If this approach is developed further, it may complement existing fuzz testing efforts by identifying critical configurations that are prerequisites for reaching certain regions of (potentially bug-containing) code.

ACKNOWLEDGMENTS

I would like to thank my research advisor, Dr. Stefan Nagy, for his guidance and expertise–I've certainly needed it. I am infinitely grateful to my wonderful partner London Ruff, who has encouraged and supported me through the process of researching and writing this thesis. Additional thanks go to my mother and father, Ana and Jonathan Clark, and to my sisters, Joanne and Lauren, who inspire me every day. I would not be here without them.

REFERENCES

[1] N. Siegmund, N. Ruckel, and J. Siegmund, “Dimensions of software configuration: on the configuration context in modern software development,” *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020. doi:10.1145/3368089.3409675

[2] A. Halin *et al.*, “Test them all, is it worth it? assessing configuration sampling on the JHIPSTER web development stack,” *Empirical Software Engineering*, vol. 24, no. 2, pp. 674–717, 2018. doi:10.1007/s10664-018-9635-4

[3] G. Ferreira, M. Malik, C. Kästner, J. Pfeffer, and S. Apel, “Do #ifdefs influence the occurrence of vulnerabilities? an empirical study of the linux kernel,” *Proceedings of the 20th International Systems and Software Product Line Conference*, 2016. doi:10.1145/2934466.2934467

[4] Z. Zhang, G. Klees, E. Wang, M. Hicks, and S. Wei, “Fuzzing configurations of program options,” *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 2, pp. 1–21, 2023. doi:10.1145/3580597

[5] F. Medeiros *et al.*, “An empirical study on configuration-related code weaknesses,” *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*, 2020. doi:10.1145/3422392.3422409

[6] C. Kaltenecker, A. Grebhahn, N. Siegmund, J. Guo, and S. Apel, “Distance-based sampling of software configuration spaces,” *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019. doi:10.1109/icse.2019.00112

[7] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel, “A comparison of 10 sampling algorithms for configurable systems,” *Proceedings of the 38th International Conference on Software Engineering*, 2016. doi:10.1145/2884781.2884793

[8] A. Lee, I. Ariq, Y. Kim, and M. Kim, “Power: Program option-aware Fuzzer for high bug detection ability,” *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2022. doi:10.1109/icst53961.2022.00032

[9] H. Dai, C. Murphy, and G. Kaiser, “Configuration fuzzing for software vulnerability detection,” *2010 International Conference on Availability, Reliability and Security*, 2010. doi:10.1109/ares.2010.22

[10] “Announcing OSS-Fuzz: Continuous Fuzzing for Open Source Software,” *Google Online Security Blog*.

https://security.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html (accessed Apr. 03, 2024).

[11] "Multiple Build Systems — Spack 0.17.1 documentation," *spack.readthedocs.io*. https://spack.readthedocs.io/en/v0.17.1/build_systems/multiplepackage.html (accessed Apr. 03, 2024).

[12] Abal, Iago & Melo, Jean & Stănciulescu, Ştefan & Brabrand, Claus & Ribeiro, Márcio & Wasowski, Andrzej. (2018). "Variability Bugs in Highly Configurable Systems: A Qualitative Analysis," A*CM Transactions on Software Engineering and Methodology.* 26. 1-34. 10.1145/3149119.

[13] T. Yavuz, C. Khor, K. Yihang, Bai, and R. Lutz, "Generating Maximal Configurations and Their Variants Using Code Metrics." April 2024. https://arxiv.org/pdf/2401.07898

Name of Candidate:	David Clark
Date of Submission:	April 23, 2024