Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems

Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S. Foster, and Adam Porter
Computer Science Department
University of Maryland
College Park, MD
{elnatan,csfalcon,kkma,jfoster,aporter}@cs.umd.edu

ABSTRACT

Many modern software systems are designed to be highly configurable, which increases flexibility but can make programs hard to test, analyze, and understand. We present an initial empirical study of how configuration options affect program behavior. We conjecture that, at certain levels of abstraction, configuration spaces are far smaller than the worst case, in which every configuration is distinct. We evaluated our conjecture by studying three configurable software systems: vsftpd, ngIRCd, and grep. We used symbolic evaluation to discover how the settings of run-time configuration options affect line, basic block, edge, and condition coverage for our subjects under a given test suite. Our results strongly suggest that for these subject programs, test suites, and configuration options, when abstracted in terms of the four coverage criteria above, configuration spaces are in fact much smaller than combinatorics would suggest and are effectively the composition of many small, self-contained groupings of options.

1. INTRODUCTION

Many modern software systems include numerous user-configurable options. For example, network servers typically let users configure the active port, the maximum number of connections, what commands are available, and so on. While this flexibility helps make software systems extensible, portable, and achieve good quality of service, it can also generate a huge number of configurations—in the worst case every combination of option settings is a distinct configuration. This *software configuration space explosion* presents real challenges to software developers. It makes testing even more costly, as it significantly magnifies testing obligations; it makes static analysis much more difficult, as different configurations can be conflated together; and it generally complicates program understanding tasks.

In this paper, we present an initial empirical study, the first of a planned family of studies, exploring how various program behaviors change in relation to system configuration. We conjecture that at certain levels of abstraction, the software configuration space is much smaller than combinatorics might suggest. For example, consider a web server that can be configured to support sequential or concurrent connections and to enable or disable logging. In this case, the block coverage achieved by all four possible configurations might be exactly the same as that achieved by two configurations: sequential connections with logging enabled, and concurrent connections with logging enabled. (Disabling logging would be unlikely to cover any new blocks.) Thus when considering block coverage, the effective configuration space for our example is half the size we would expect. If our conjecture proves true, then in future work, new techniques and heuristics might be created to partition configuration spaces in ways that greatly simplify testing, analysis, and program understanding.

To evaluate our conjecture, we studied three configurable subject systems: vsftpd, ngIRCd, and grep. For each system we identified a sizable number of run-time configuration options to analyze, determined their possible settings, and created a test suite. We then ran the test suites using Otter, a symbolic evaluator [10, 8, 2] we developed.

In our study, we marked the selected configuration options as *symbolic*, meaning they represent unknowns that can take on any value. As Otter evaluates a program, if it encounters a branch that depends on a symbolic value, it conceptually forks execution and explores both possible branches. In this way, we used Otter to compute *all* possible program paths for *all* possible settings of the selected configuration options. This required tens or hundreds of thousands of runs, varying with the application, but that is a small fraction of the tens of millions or more runs that would have been needed had we naively enumerated all configurations.

We next projected the runs onto four types of structural coverage—line, basic block, edge, and condition coverage—and used the resulting data to discover *interactions* among configuration options. Here we define an interaction to be a partial setting of configuration options such that specific line, block, edge, or condition coverage is *guaranteed* to occur under that setting, but is not guaranteed by any of its subsets. For example, if `a` and `b` are options, then `a=0, b=1` is an interaction if it guarantees some coverage that just setting `a=0` or `b=1` by themselves do not guarantee. The *strength* of an interaction is the number of options it assigns to. Interactions are interesting because they define small subsets of the configuration space that provide meaningful additional coverage.

We computed interactions incrementally starting with combinations of zero options (i.e., coverage that occurs in all runs), then one option, then two, and so on. We continued until the accumulated guaranteed coverage equaled the maximum possible coverage across all runs. We found that for our subject systems and test suites, the largest (i.e., strongest) interactions included between five and seven options. This is much smaller than the total number of options in any of the systems. Similarly, the total number of interactions is much smaller than what we would expect if we simply multiplied out the possible option combinations

naively. These trends, and the others reported below, were essentially the same under all four coverage metrics.

Next, we looked at which specific interactions are needed to achieve high coverage. We found that most coverage is supplied by relatively low-strength interactions, though all three programs had a few (one to three) *enabling options* that needed to be set a certain way to get the maximum coverage. We also used a (non-optimal) greedy algorithm to pack together interactions into full configurations, with the aim of finding the smallest set of configurations that would achieve full coverage. For example, interactions a=0, b=1 and c=0 can be joined into a single configuration, whereas a=0 and a=1 must go into different configurations. We found that we needed only at most 9 configurations for vsftpd, 8 for ngIRCd, and 10 for grep to achieve full coverage. All of these sets are again quite small. This suggests that for the programs and test suites used, the behavior of all configurations, when abstracted onto our coverage criteria, can be derived/understood from the composition of a small number of interactions.

Finally, we created graphs showing all option interactions to better understand what allows us to achieve coverage with so few configurations. From this data and the previous analyses, we observe that for our programs, coverage metrics, test suites, and configuration spaces, many options do not interact with each other; that when they do interact, they often do so at low-strength; and the interactions that exist often cluster into distinct groupings that can be combined into larger configurations. In other words, at certain levels of abstraction, the configuration spaces behave less like a monolithic cross product of all option settings, and more like the union of smaller configuration spaces.

In summary, our results strongly support our main conjecture: that in practical systems, when abstracting to specific program execution behaviors, the configuration space is much smaller than combinatorics would suggest. We believe that this work provides a basic but important starting point for understanding software configurability and for creating techniques and heuristics for scaling many software development tasks across large configuration spaces.

2. CONFIGURABLE SOFTWARE SYSTEMS

For this work, a configurable system is a generic code base and a set of mechanisms for implementing pre-planned variations in the code base's structure and behavior. In practice, these variations are wide-ranging, covering hardware and operating system platforms (e.g., Windows vs. Linux), software versions (e.g., MySQL 5.0 vs. MySQL 5.1), run-time features (e.g., enable/disable debugging output) and others. In this paper, we are focusing on run-time configuration options, which are usually given values via configuration files or command-line parameters. A *configuration* is a mapping of configuration options to their settings.

Figure 1 illustrates several ways that run-time configuration options can be used, and explains why understanding their usage requires fairly sophisticated technology. All of these examples come from our experimental subject programs, which are written in C. In this figure, variables containing configuration options are shown in boldface.

The example in Figure 1(a) shows a section of vsftpd's command loop, which receives a command and then uses a long sequence of conditionals to interpret the command and carry out the appropriate action. The example shows two

```
1  ... else if (tunable_pasv_enable &&
2              str_equal_text(&p_sess->ftp_cmd_str, "EPSV"))
3  {
4    handle_pasv(p_sess, 1);
5  }
6  ... else if (tunable_write_enable &&
7              (tunable_anon_mkdir_write_enable ||
8               !p_sess->is_anonymous) &&
9              (str_equal_text(&p_sess->ftp_cmd_str, "MKD") ||
10              str_equal_text(&p_sess->ftp_cmd_str, "XMKD")))
11 {
12   handle_mkd(p_sess);
13 }
```

(a) Boolean configuration options (vsftpd)

```
14 if ((Conf_MaxJoins > 0) &&
15     (Channel_CountForUser(Client) >= Conf_MaxJoins))
16   return IRC_WriteStrClient(Client,
17                             ERR_TOOMANYCHANNELS_MSG,
18                             Client_ID(Client), channame);
```

(b) Integer-valued configuration options (ngIRCd)

```
19 else if(Conf_OperCanMode) {
20   /* IRC-Operators can use MODE as well */
21   if (Client_OperByMe(Origin)) {
22     modeok = true;
23     if (Conf_OperServerMode)
24       use_servermode = true; /* Change Origin to Server */
25   }
26 }
27 ...
28   if (use_servermode)
29     Origin = Client_ThisServer();
```

(c) Nested conditionals (ngIRCd)

```
30 not_text =
31   (((binary_files == BINARY_BINARY_FILES && !out_quiet)
32       || binary_files == WITHOUT_MATCH_BINARY_FILES)
33      && memchr (bufbeg, eol ? '\0' : '\200', buflim - bufbeg));
34 if (not_text &&
35     binary_files == WITHOUT_MATCH_BINARY_FILES)
36   return 0;
37 done_on_match += not_text;
38 out_quiet += not_text;
```

(d) Options being passed through the program (grep)

Figure 1: Example uses of configuration options (bolded) in subjects.

such conditionals that also depend on configuration options (all of which begin with tunable_ in vsftpd). In this case, the configuration options enable certain commands, and the enabling condition can either be simply the current setting of the option (as on line 1) or may involve an interaction between multiple options (as on lines 6–7).

Not all options need to be booleans, of course. Figure 1(b) shows an example from ngIRCd, in which Conf_MaxJoins is an integer option that, if positive (line 14), specifies the maximum number of channels a user can join (line 15). In this example, error processing occurs if the user tries to join too many channels.

Figure 1(c) shows a different example in which two configuration options are tested in nested conditionals. This illustrates that it is insufficient to look at tests of configuration options in isolation; we also need to understand how they may interact based on the program's structure. Moreover, in this example, if both options are enabled then use_servermode is set on line 24, and its value is then tested on line 28. This shows that the values of configuration options can be indirectly carried through the state of the program.

Figure 1(d) shows another example of using configuration options indirectly. Here not_text is assigned the result of a complex test involving configuration options, and is then used in the conditional (lines 34–35) to change the current setting of two other configuration options (lines 37–38).

3. SYMBOLIC EVALUATION

To understand how configurations resemble and differ from each other, we have to capture their effect on a system's runtime behavior. As we saw above, configuration options can be used in quite complex ways, and so simple approaches such as searching through code for option names will be insufficient. Instead, we use symbolic evaluation [10] to capture all execution paths a program can take under any configuration.

Our symbolic evaluator, Otter,[1] is essentially a C source code interpreter, with one key difference. We allow the programmer to designate some values as *symbolic*, meaning they represent unknowns that may take on any value. Otter tracks these values as they flow through the program, and conceptually forks execution if a conditional depends on a symbolic value. Thus, if it runs to completion, Otter will simulate all paths through the program that are reachable for any values that the symbolic data can take.

To illustrate how Otter works, consider the example C source code in Figure 2(a). This program includes input variables a, b, c, d, and input. The first four are intended to represent run-time configuration options, and so we initialize them on lines 1–2 with *symbolic values* α, β, γ, and δ, respectively. (In the implementation, the content of a variable v is made symbolic with a special call __SYMBOLIC(&v).) The last variable, input, is intended to represent program inputs other than configuration options. Thus we leave it as concrete, and it must be supplied by the user (e.g., as part of argv (not shown)).

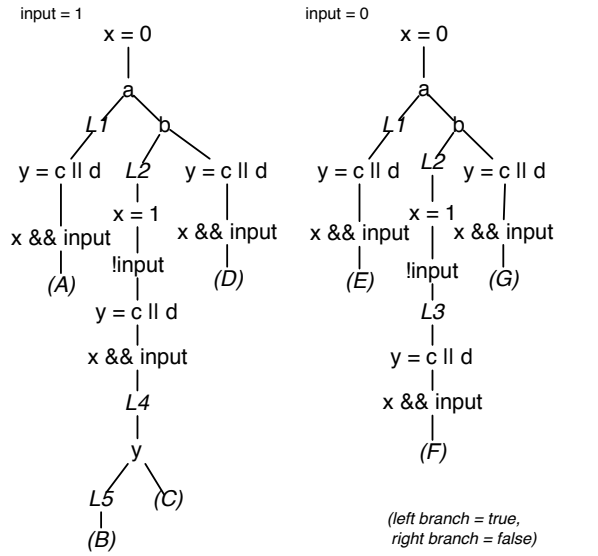
We have indicated five statements, numbered 1–5, whose coverage we are interested in. (We focus on line coverage here for illustration purposes, but the idea is the same for other forms of coverage.) Figure 2(b) shows the sets of paths explored by Otter as *execution trees* for two concrete "test cases" for this program: the tree for input=1 is on the left, and the tree for input=0 is on the right. Here nodes correspond to program statements, and branches represent places where Otter has a choice and hence "forks," exploring both possible paths.

For example, consider the tree with input=1. All executions begin by setting x to 0 and then testing the value of a, which at this point contains α. Since there are no constraints on α, both branches are possible. For the sake of simplicity

[1]DART [8] and EXE [3] are two well known symbolic evaluators. By coincidence, Dart and Exe are the names of two rivers in Devon, England. The others are the Otter, the Tamar, the Taw, the Teign, and the Torridge.

```
1  int a=α, b=β,
2      c=γ, d=δ; // symbolic
3  int input=...; // concrete
4  int x = 0;
5  if (a)
6    /* 1 */
7  else if (b)
8  {
9    /* 2 */
10   x = 1;
11   if (!input) {
12     /* 3 */
13   }
14 }
15 int y = c || d;
16 if (x && input) {
17   /* 4 */
18   if (y)
19     /* 5 */
20 }
```

(a) Example program



(b) Full execution trees

Figure 2: Example symbolic evaluation.

we will assume below that α and the other symbolic values may only represent 0 and 1, but Otter fully models symbolic integers as arbitrary 32-bit quantities.

Otter then forks its execution at the test of a: First it assumes that α is true and reaches statement 1 (left branch). It then falls through to line 15 (the assignment to y) and performs the test on line 16 (x && input). This test is false, since x was set to 0 earlier, hence there is no branch. We label this path through the execution tree as (A).

Notice that as we explored path (A), we made some decisions about the settings of symbolic values, specifically that α is true. We call this and any other constraints placed on the symbolic values a *path condition*. In this case, path (A) covers statement 1, and so any configuration that sets a=1 on line 1 (corresponding to α being true), with arbitrary choices for the values of β, γ, and δ, will cover statement 1. This is what makes symbolic evaluation so powerful: With a single predicate we characterized the behavior of many possible concrete choices of symbolic inputs (in this case, there would be 2^3 possibilities for all combinations of b, c, and d).

Otter continues by returning to the last place it forked and trying to explore the other path. In this case, it returns

to the conditional on line 5, assumes α is false by adding $\neg\alpha$ to the path condition, and continues exploring the execution tree. Each time Otter encounters a conditional, it actually calls an SMT solver to determine which branches (possibly both) of the conditional are possible based on the current path condition.

There are a few other interesting things to notice about these execution trees. First, consider the execution paths labeled (B) and (C). Because we have chosen β to be true on this path, we set x=1, and hence x && input is true, allowing us to reach statements 4 and 5. This is analogous to the example in Figure 1(c), in which a configuration option choice resulted in a change to the program state (setting x=1) that allowed us to cover some additional code. Also, notice that if input=1, there is no way to reach statement 3, no matter how we set the symbolic values. Hence coverage depends on choices of both symbolic values and concrete inputs.

In total, there are four paths that can be explored when input=1, and three paths when input=0. However, there are 2^4 possible assignments to the symbolic values α, β, γ, and δ. Hence using symbolic evaluation for these test cases enables us to gather full coverage information with only 7 paths, rather than the 32 runs required if we had tried all possible combinations of symbolic and concrete inputs. This is what makes the results in this paper even possible—we can effectively get the same result as if we had tried all possible combinations of configuration options with far fewer paths than that would entail if done concretely.

3.1 Guaranteed Coverage

Otter forms the basis for our empirical study: For each subject program, we select a number of configuration options, mark them as symbolic, and then use Otter to execute a set of test cases. The resulting execution trees contain all possible paths executed under all configuration option settings for those test cases.

Without further analysis, these paths tell us only a little about our subject programs. By definition, each path explored for a particular test case is distinct from all the other paths for the same test case. Thus with no abstraction, every configuration option combination given by a path is unique. For example, in Figure 2(b), there are four distinct paths if input=1, representing four distinct settings of configuration options. Thus far, then, we only know that that is fewer than the 16 paths we might naively expect.

However, if we are interested in more abstract properties of the program, then paths are no longer unique, and the configuration space collapses further. For example, suppose we are only interested in covering statement 2 in Figure 2. Then we can see that paths (A) and (D) are irrelevant, and either path (B) or (C) is sufficient.

For this study, we project the symbolic evaluation results onto four commonly used abstractions of program behavior: line, block, edge, and condition coverage. The principal tool we use to relate configuration options to coverage is *guaranteed coverage*.

Definition 1. *Given a particular coverage criterion, we say that a predicate p over the configuration options* guarantees coverage *(line, block, edge, condition, etc.) of X if there exists some test case such that any configuration satisfying p is guaranteed to cover X.*

For example, from Figure 2(b) we can see that any configuration satisfying $\alpha = 0 \wedge \beta = 1$ (i.e., a=0, b=1) is guaranteed to cover statement 2, no matter the choice of γ and δ.

We can use Otter's output to compute the guaranteed coverage for a predicate p, which we will write $Cov(p)$. We first find $Cov^T(p)$, the coverage guaranteed under p by test case T, for each test case; then, $Cov(p) = \bigcup_T Cov^T(p)$. To compute $Cov^T(p)$, let p_i^T be the path conditions from T's symbolic evaluation, and let $C^T(p_i^T)$ be the covered lines (or blocks, edges, conditions, etc.) that occur in that path. Then $Cov^T(p)$ is

$$\begin{aligned} Consistent^T(p) &= \{p_i^T \mid SAT(p_i^T \wedge p)\} \\ Cov^T(p) &= \bigcap_{q \in Consistent^T(p)} C^T(q) \end{aligned}$$

In words, first we compute the set of predicates p_i^T such that p and p_i^T are consistent. If this holds for p_i^T, the items in $C^T(p_i^T)$ *may* be covered if p is true. Since our symbolic evaluator explores all possible program paths, the intersection of these sets for all such p_i^T is the set guaranteed to be covered if p is true.

Going back to Figure 2, here are some predicates and the coverage they guarantee:

p	$Consistent(p)$ (input = 1)	$Consistent(p)$ (input = 0)	$Cov(p)$
α	(A)	(E)	$\{1\}$
β	$(A), (B), (C)$	$(E), (F)$	$\emptyset$
$\neg\alpha$	$(B), (C), (D)$	$(F), (G)$	$\emptyset$
$\neg\alpha \wedge \beta$	$(B), (C)$	(F)	$\{2, 3, 4\}$
$\neg\alpha \wedge \beta \wedge \gamma$	(B)	(F)	$\{2, 3, 4, 5\}$

Note that we cannot guarantee covering statement 5 without setting three symbolic values (although we could have picked δ instead of γ).

As we show in Section 5, we can use guaranteed coverage to discover *interactions* among options.

Definition 2. *An* interaction *is a set p of option settings that guarantees coverage that is not guaranteed by any subset of p.*

For example, since $Cov(\neg\alpha \wedge \beta)$ is a strict superset of $Cov(\neg\alpha) \cup Cov(\beta)$, $\neg\alpha \wedge \beta$ is an interaction. Informally, interactions indicate combinations of options that are "interesting" because they guarantee some new amount of coverage.

Definition 3. *The* strength *of an interaction is the number of option settings it contains.*

For example, $\neg\alpha \wedge \beta$ has strength 2. Lower-strength interactions place fewer requirements on configurations, whereas higher-strength interactions require more options to be set in particular ways to achieve their coverage.

3.2 Implementation

Otter is written in OCaml, and it uses CIL [13] as a front end to parse C programs and transform them into an easier-to-use intermediate representation.

The general approach used by Otter mimics KLEE [2]. A symbolic value in Otter represents a sequence of untyped bits, e.g., a 32-bit symbolic integer is treated as a vector with 32 symbolic bits in Otter. This low-level representation is important because many C programs perform bit manipulations that must be modeled accurately. When a symbolic expression has to be evaluated, Otter invokes STP [7], an SMT solver optimized for bit vectors and arrays.

Otter supports all the features of C we found necessary for our subject programs, including pointer arithmetic, function pointers, variadic functions, and type casts. Otter currently does not handle multiple processes, dereferencing symbolic pointer values, floating-point arithmetic, or inline assembly. Multiple processes are used in vsftpd's standalone mode and in ngIRCd, but we work around this. For vsftpd, in which fork() spawns a subprocess that handles client commands, we interpret fork() as driving the program to that subprocess. (The parent process would simply cycle around a loop and spawn another subprocess, so we ignore it.) For ngIRCd, where the child process parses an IP address and passes the result to the parent, we treat fork() as a branching point—we run both subprocesses, but we ignore the child process's output, instead supplying the input expected by the parent process as part of the test case. The other unsupported features either do not appear in our subject programs or do not affect the results of our study.

All of our subject programs interact with the operating system in some way. Thus, we developed "mock" libraries that simulate a file system, network, and other needed OS components. Our libraries also allow test cases to control the contents of files, data sent over the network, and so on. Our mock library functions are mostly written in C and are executed by Otter just like any other program. For example, we simulate a file with a character array, and a file descriptor points to some file and keeps the current position at which the file is to be read or written.

As Otter executes, it records the program paths explored so that we can later compute line, block, edge, and condition coverage. The precise definitions of these metrics demand some elaboration, because Otter runs on CIL's representation of the input program, which is simplified to use only a subset of the full C language.

To compute line coverage, we record which CIL statements Otter executes and project that back to the original source lines using a mapping maintained by CIL.

For block and edge coverage, we group CIL statements into basic blocks, which are sequences of statements that start at a function entry or a join point; do not contain any join point after the first statement; end in a function call, return, goto, or conditional; or fall through to a join point. Normally, CIL expands short-circuiting logical operators && and || into sequences of branches. However, for block and edge coverage, we disable that expansion as long as the right operand has no side effect, so that both operands are computed in the same basic block. Then to compute block coverage, we record which basic blocks are executed, and to compute edge coverage, we compute which control-flow edges between basic blocks are traversed.

Lastly, for condition coverage, we enable expansion of && and ||, so that each part of a compound condition is always in its own basic block. We then compute how many conditions—that is, how many branches—are taken in the expanded program.

4. SUBJECT PROGRAMS

The subject programs for our study are vsftpd, a widely-used secure FTP daemon; ngIRCd, the "next generation IRC daemon"; and GNU grep, a popular text search utility. All of our subject programs are written in C. Each has multiple configuration options that can be set either in system configuration files or through command-line parameters.

	vsftpd	ngIRCd	grep
Version	2.0.7	0.12.0	2.4.2
# Lines (sloccount)	10,482	13,601	9,124
# Lines (executable)	4,112	4,387	3,302
# Basic blocks	4,584	6,742	5,033
# Edges	5,033	7,374	6,332
# Conditions	2,528	3,432	4,094
# Test cases	64	142	113
# Analyzed conf. opts.	30	13	18
Boolean	20	5	14
Integer	10	8	4
# Excluded conf. opts.	65	16	4

Figure 3: Subject program statistics.

Figure 3 gives descriptive statistics for each subject program. The top two rows list the program version numbers and lines of code as computed by sloccount. The next group of rows lists the number of executable lines, basic blocks, edges, and conditions; these four metrics are what we measure code coverage against, and they are based on the CIL representation of the program, as discussed in Section 3.2. To get more accurate measurements, we removed some unreachable code before passing the sources to CIL. Specifically, we commented out 4 unreachable functions in grep. We also forced vsftpd to run in single-process mode, as Otter does not support multiprocess symbolic evaluation, and correspondingly eliminated 3 files in vsftpd that are reachable only in two-process mode.

One thing to note is that there are more basic blocks than executable lines of code in all 3 programs. This occurs because, in many cases, single lines form multiple blocks. For example, a line that contains a for loop will have at least two blocks (for the initializer and the guard), and lines with multiple function calls are broken into separate blocks according to our definition.

The next row in Figure 3 lists the number of test cases. In creating these test cases, we attempted to both cover the major functionality of the system and to maximize overall line coverage. We stopped creating new tests when the remaining uncovered code was overwhelmingly devoted to handling low-level system errors such as malloc() or device read() failures.

vsftpd does not come with its own test suite, so we developed tests to exercise its major functionality such as logging in; listing, downloading, and renaming files; asking for system information; and handling invalid commands.

ngIRCd also does not come with its own test suite, so we created tests based on the IRC functionality defined in RFCs 1459, 2812 and 2813. Our tests cover most of the client-server commands (e.g., client registration, channel join/-part, messaging and queries) and a few of the server-server commands (e.g., connection establishment, state exchange), with both valid and invalid inputs.

Grep comes with a test suite consisting of hundreds of tests. To build our test suite for this study, we ran all the test cases in Otter to determine their line coverage. Then, without sacrificing total line coverage, we selected 70 test cases from the original suite for our study. Next, we created 43 new test cases to improve overall line coverage. The final analysis was done using these 113 test cases.

Finally, the last group of rows in Figure 3 counts the configuration options. We give the total number of analyzed

	vsftpd	ngIRCd	grep
Coverage			
Line	62%	73%	75%
Block	63%	66%	63%
Edge	56%	61%	58%
Condition	49%	57%	52%
# Examined opts/tot			
Line, Block, Edge	22/30	13/13	17/18
Condition	24/30	13/13	17/18
# Paths			
Line, Block, Edge	30,304	53,205	625,181
Condition	136,320	95,637	764,201
Average # Paths			
Line, Block, Edge	474	375	5,533
Condition	2,130	674	6,763

Figure 4: Summary of symbolic evaluation.

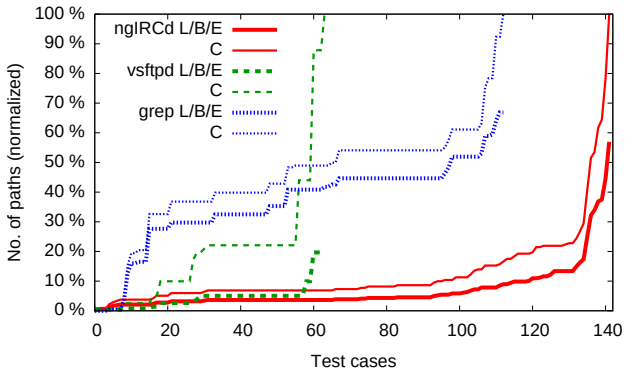


Figure 5: Number of paths per test case (L/B/E=line/block/edge, C=condition).

configuration options, i.e., those that we treated as symbolic, and also break them down by type (boolean or integer). We also list the number of configuration options we left as concrete. Our decision to leave some options concrete was primarily driven by two criteria: whether the option was likely to expose meaningful behaviors, and our desire to limit total analysis effort. This approach allowed us to run Otter numerous times on each program, to explore different scenarios, and to experiment with different kinds of analysis techniques. We used default values for the concrete configuration options, except the one used to force single-process mode in vsftpd. Grep includes a three-valued string option to control which functions execute the search; for simplicity, we introduced a three-valued integer configuration option and set the string based on this value.

5. DATA AND ANALYSIS

We ran our test suites in Otter, with symbolic configuration options as discussed above. We then performed substantial analysis on the results to explore the configuration space of each subject program. To do this we used the Skoll system, developed and housed at UMD [14]. Skoll allows users to define configurable QA tasks and run them across large virtual computing grids. For this work we used around 40 client machines. The final results reported in this section required about two weeks of elapsed time.

Figure 4 summarizes Otter's runs. The first group of rows shows the total coverage achieved by the test suites, i.e., the maximum coverage achievable for these test suites considering all possible configurations, except those options and values we left concrete. We manually inspected the uncovered lines and found that approximately another 10% of vsftpd and ngIRCd and 2% of grep comprises code for handling low-level errors. Also, another 11% of vsftpd (in addition to the three files we removed) is unreachable in one-process mode. If we adjust for the error handling and unreachable code, our test suites' line coverage exceeds 80% for all subject programs. Covering the remaining code would in many cases have required adding new mocked libraries, adding further symbolic configuration options, etc. Overall, however, based on our analysis of these systems, we believe that the test cases are reasonably comprehensive and are sufficient to expose much of the configurable behavior of the subject programs.

The next group of rows shows the number of configuration options that appear in at least one path condition (i.e., were constrained in at least one path and thus distinguish different execution paths) versus the total number of options set symbolic. In grep, the one unused option was only "used" when being printed, which did not affect any execution path. In vsftpd, there were 6 unused options total. One case was similar to grep—a configuration option specified a port number, which is ignored by our mock system. Three other options could have been covered with additional tests; the remaining two options cannot be touched without changing the settings of some of the configurations options we left concrete.

Notice that Otter constrains two more options with condition coverage than under the other metrics. This occurs because, as discussed in Section 3.2, we expand logical operators into sequences of conditionals under condition coverage. For example, under line, block, and edge coverage, the condition **if** (x||1) would be treated as a single branch that Otter would treat as always true. But under condition coverage, the conditional would be expanded, and Otter would see **if** (x) first, causing it to branch on x.

The last group of rows in Figure 4 shows the number of execution paths explored by Otter and that number averaged across all test cases for each program. While Otter found many thousands of paths, recall that these are actually *all* possible paths for these test suites under any settings of the symbolic configuration options. Had we instead naively run each test case under all possible configuration option combinations, it would have required 1.4×10^{11} executions for vsftpd, 3.7×10^{7} for ngIRCd, and 1.5×10^{12} for grep.

Notice also that expanding logical operators under condition coverage can result in many more paths. This effect is most pronounced for vsftpd, which more than quadruples the number of paths because it contains many logical expressions that test multiple configuration options at once. For example, **if** (x||y||z) would yield at most two paths before expansion, but four paths after.

Figure 5 plots the number of paths executed by each test case for each program, both with unexpanded logical operators (L/B/E) and expanded (C). The x-axis is sorted from the fewest to the most paths, and the y-axis is the percentage of paths relative to the highest number of paths seen in any test case for the expanded (C) version of the program.

One interesting feature of Figure 5 is that, for vsftpd and grep, the numbers of paths of different test cases ap-

	t=1	t=2	t=3	t=4	t=5	t=6	t=7
vsftpd							
Line	7	4	3	16	5	6	2
Block	7	4	3	16	6	6	2
Edge	9	4	4	27	7	7	2
Condition	9	4	4	32	14	9	2
ngIRCd							
Line	11	17	31	113	144	111	-
Block	15	22	31	118	147	111	-
Edge	17	26	35	118	159	111	-
Condition	17	30	35	124	174	111	-
grep							
Line	13	27	36	7	5	-	-
Block	14	34	37	7	5	-	-
Edge	23	37	45	11	7	-	-
Condition	23	45	49	16	9	2	-

Figure 6: Number of interactions at each interaction strength.

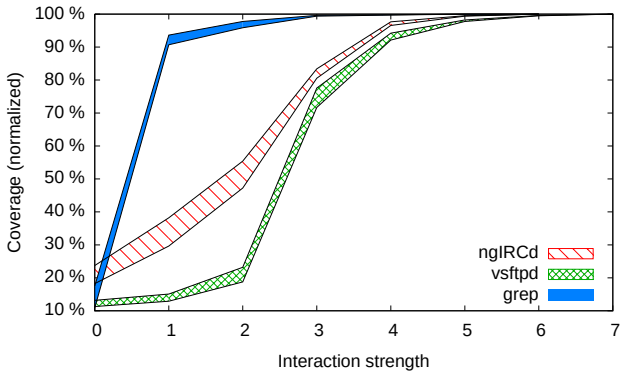


Figure 7: Guaranteed coverage versus interaction strength.

pear to cluster into a handful of groups (indicated by the plateaus in the graph). This suggests that within a group, the test cases branch on the configuration options in essentially the same manner (most likely because the programs employ common segments of code to test the configuration options). In ngIRCd, this clustering also appears but is less pronounced.

Finally, recall from Figure 3 that grep, despite still having many fewer paths than configurations, stands out as having a much larger number of paths than the other programs. We believe this is due to grep's design. In runs of grep with valid inputs, most of grep's code is executed. Therefore many of its configuration options will typically be used, resulting in significant branching in Otter. In contrast, many of vsftpd and ngIRCd's options are not necessarily used in every run. This can be seen clearly in Figure 5: only a handful of vsftpd and ngIRCd's tests exercise more than 25% of the paths, while only a handful of grep's tests exercise *fewer* than 25%.

5.1 Interaction Strength

Next, we used our guaranteed coverage analysis to explore which configuration option interactions (Section 3.1) are actually required to achieve the line, block, edge, and condition coverage reported in Figure 4. First, we computed $Cov(true)$, which we call *guaranteed 0-way coverage*. These are coverage elements that are guaranteed to be covered for any choice of options. Here when we refer to *t-way coverage*, t is the interaction strength. Then for every possible option setting $\mathsf{x} = \mathsf{v}$, we computed $Cov(\mathsf{x} = \mathsf{v})$. The union of these sets is the *guaranteed 1-way coverage*, and it captures what coverage elements will definitely be covered by 1-way interactions. Next, we computed $Cov(\mathsf{x1} = \mathsf{v1} \wedge \mathsf{x2} = \mathsf{v2})$ for all possible pairs of option settings, which is *guaranteed 2-way coverage*. Similarly, we continue to increase the number of options in the interactions until $Cov(\mathsf{x1} = \mathsf{v1} \wedge \mathsf{x2} = \mathsf{v2} \wedge ...)$ reaches the maximum possible coverage.

For boolean options, the possible settings are clearly 0 and 1. For integer-valued options, we solved the path conditions discovered by Otter to find possible concrete settings. For example, if the path condition was x>=0, then the solver might choose $\mathsf{x} = 0$ as a possible concrete setting. Because there are multiple path conditions, we sometimes found that different concrete settings were generated by the solver for the same options. In these cases we used our judgement and code examination to select appropriate values.

Figure 6 shows the number of interactions at each interaction strength. The first thing to notice is that the maximum interaction strength is always seven or less. This is significantly lower than the number of options in each program. We also see that the number of interactions is quite small relative to total number of interactions that are theoretically possible. For example, grep has 14 boolean options, which by themselves lead to (14 choose 2) $\times$ 4 = 728 possible 2-way interactions just with those options alone, yet we see at most 45 2-way interactions for grep.

Also notice that there is not much variation across different coverage criteria—they have remarkably similar numbers of interactions. We investigated further, and we found that the majority of interactions are actually identical across all four criteria. This is an encouraging finding, because it indicates that many interactions are insensitive to the particular coverage criterion.

For ngIRCd, there are significantly more interactions at higher strength than for the other subject programs. This is because almost all of ngIRCd's integer options can take on many different values across our test suite, magnifying the number of interactions.

Finally, we can see that the number of interactions peak around $t = 4$ for vsftpd, $t = 4$ or 5 for ngIRCd, and $t = 2$ or 3 for grep. We believe this corresponds to the number of *enabling options* in these programs, discussed more in the next subsection.

5.2 Guaranteed Coverage

Figure 7 presents the interaction data in terms of coverage. The x-axis is the t-way interaction strength and the y-axis is the percentage of the maximum possible coverage. Note that higher-level guaranteed coverage always includes the lower level, e.g., if a line is covered no matter what the settings are (0-way), then it is certainly covered under particular settings (1-way or higher). As it turns out, the trend lines for all four coverage criteria are essentially the same for a given program, and so the plot shows a region enclosing each set of data points. In ngIRCd, the only program with some slightly noticeable variation, line coverage corresponds to the upper boundary of the region, and edge, block, and condition coverage to the lower boundary. This commonality across coverage criteria echoes the same trend we saw in Figure 6.

Config #	1	2	3	4	5	6	7	8	9	10
vsftpd										
Line	2,521	18	8	1	1	-	-	-	-	-
Block	2,853	25	9	1	1	-	-	-	-	-
Edge	2,731	50	17	6	1	1	1	-	-	-
Condition	1,132	71	14	9	2	1	1	1	1	-
ngIRCd										
Line	3,148	30	6	6	1	1	1	-	-	-
Block	4,401	50	8	7	4	1	1	-	-	-
Edge	4,390	62	14	8	6	2	2	2	-	-
Condition	1,881	27	23	6	4	1	1	1	-	-
grep										
Line	2,218	171	34	20	5	5	3	2	2	-
Block	2,838	231	46	28	5	5	3	1	-	-
Edge	3,140	366	51	44	18	9	6	6	4	-
Condition	1,810	231	45	25	11	8	7	6	5	1

Figure 8: Additional coverage achieved by each configuration in the minimal covering sets.

One thing to notice in this figure is that the right-most portion of each region adds little to the overall coverage. For these programs and test suites then, high-strength interactions are not needed to cover most of the code. We can also see from this plot that vsftpd gains coverage slowly but then spikes with 3-way interactions, and grep has a similar spike with 1-way interactions. This suggests the presence of *enabling options*, which must be set a certain way for the program to exhibit large parts of its behavior. For example, for vsftpd (in single-process mode), the enabling options must ensure local logins and SSL support are turned off, and anonymous logins are turned on. For grep, either grep or egrep mode must be enabled to reach most of grep's code; fgrep mode touches little code. ngIRCd also has enabling options that account for the increasing coverage up to interaction strength three, but the effects of these options are less pronounced.

These enabling options also show up in Figure 6. For example, in that figure we can see that most of vsftpd's interactions are strength $t = 4$ or greater, i.e., they generally involve the three enabling options plus additional options.

5.3 Minimal Covering Configuration Sets

Our results so far show that low-strength interactions can cover most of the code. Next, we investigated how interactions can be packed together to form complete configurations, which assign values to all the configuration options. For example, the 1-way interactions `a=0` and `b=0` are consistent and can be packed into the same configuration, but `a=0` and `a=1` are contradictory and must go in different configurations.

We developed a greedy algorithm that packs options together, aiming to find a minimal set of configurations that achieves the same coverage as the full set of runs. We begin with the empty list of configurations. At each step of the algorithm, we pick the interaction that (if we also include the coverage of all subsets of that interaction) guarantees the most as-yet-uncovered lines. Then, we scan through the list to find a configuration that is consistent with our pick. We merge the interaction with the first such configuration we find in the list, or append the interaction to the list as a new configuration if it is inconsistent with all existing configurations. This algorithm will always eventually terminate with all lines covered, though it is not guaranteed to find the actual minimum set.

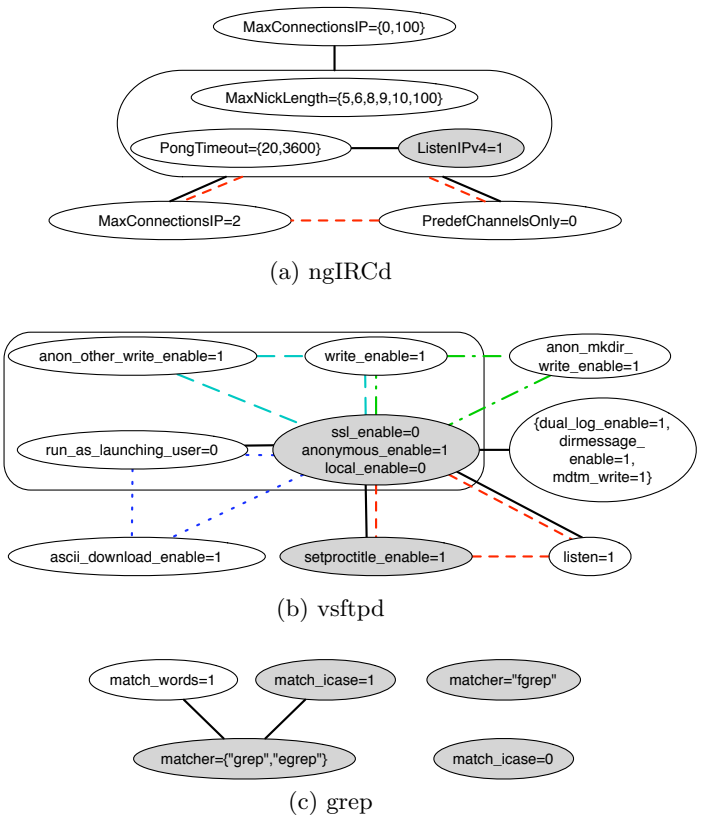


Figure 9: Interactions needed for 95% line coverage. ngIRCd and vsftpd include some approximations.

Figure 8 summarizes the results of our algorithm. The column labeled 1 shows how many lines, blocks, edges, or conditions are covered by the first configuration in the list. Then column n (for $n > 1$) shows the additional coverage achieved by the nth configuration over configurations $1..(n-1)$. Notice that minimal covering sets range in size from 5 to 10, which is much smaller than the number of possible configurations. This suggests that when we abstract in terms of coverage, in fact the configuration space looks more like a union of disjoint interactions (that can be efficiently packed together) rather than a monolithic cross-product.

We can also see that each subject program follows the same general trend, with most coverage achieved by just the first configuration in the set. The last several configurations in the set very often add only a single additional coverage element. This last finding hints that not every interaction offers the same level of coverage; we explore this issue in detail next.

5.4 Configuration Space Analysis

To help visualize interactions and to better understand why the minimal covering sets are so small, we mapped the interactions of each subject program, which are shown in Figure 9. These graphs show interactions based on line coverage. Because the full set of interactions is too large to display easily, we show only those interactions needed to

guarantee 95% of the maximum possible coverage.[2] In these graphs, a node represents one or more option settings; we merged nodes with common neighbors, listing all settings the node represents. 1-way interactions are shaded nodes, 2-way interactions are solid edges, and 3-way interactions are cliques of similarly patterned edges. In Figure 9(a), the box denotes a "super node" containing several options, each of which interacts with all three options outside the box. In Figure 9(b), the box instead represents a 4-way interaction. The ngIRCd options are all prefixed with Conf_, and similarly the vsftpd options are prefixed with tunable_. We omitted these prefixes from the graph, however, to save space.

To unclutter the presentation and to highlight interesting interaction patterns, we made some additional simplifications. For ngIRCd, we merged two values for PongTimeout that had similar but not identical neighbor sets, and similarly for MaxNickLength. For vsftpd, we merged the options in the center node of Figure 9(b) even though they have slightly different neighbors.

The main feature we see in ngIRCd's graph is the super node in the middle, which contains ngIRCd's enabling options. We can even see their progression: setting ListenIPv4=1 is the first crucial step that lets ngIRCd accept clients, and it forms a 1-way interaction. Next, setting PongTimeout high enough avoids early termination of client connections, and therefore this option forms a 2-way interaction with ListenIPv4=1. The last enabling option, MaxNickLength, forms a 3-way interaction with the previous two. In the full ngIRCd graph, the full set of these enabling options are similarly connected to most of the nodes in the graph.

Next, considering vsftpd's graph, we clearly see that all of the interactions involve the enabling options, which appear in the center, shaded node. There are many interactions involving just one additional option setting, such as the three options in the node at the right middle position. These options control the availability of some features, e.g., dirmessage_enable enables the display of certain messages. Moreover, notice that we can combine all the settings in the nodes of Figure 9(b) into one configuration. This helps illustrate why the minimal covering set of configurations for vsftpd is so small, and why the initial configuration is able to cover so much: one configuration can enable a range of features (writing files, logging, etc.) all at once.

For vsftpd, the full graph of interactions is very much like the image shown here, with a few additional, higher-strength interactions that include the three enabling options, plus a few low-strength interactions, including the other settings for the enabling options, which each guarantee a few additional lines.

Finally, in grep's graph, notice how few configuration options contributed to 95% of the coverage. These high-coverage interactions of grep have very low interaction strength; there are no interactions with strength higher than two, and four out of the five nodes have 1-way interactions. Also, all values of the matcher option appear in this graph, making this the most important option for grep in terms of coverage. The full configuration space graph of grep contains many more interactions and, interestingly, the important matcher option only takes part in a few interactions in the full graph.

While each program exhibits somewhat different configuration space behavior, we can see that when abstracted in terms of line coverage, many options either do not interact or interact at low strength, and thus we can combine them together into larger configurations. This supports our claim that configuration spaces are considerably smaller than combinatorics might suggest.

[2]The diagrams of the full set of interactions are presented in the appendix.

5.5 Threats to Validity

Like any empirical study, our observations and conclusions are limited by potential threats to validity. For example, in this work we used 3 subject programs. Each is widely used, but small in comparison to some industrial applications. In order to keep our analyses tractable, we focused on sets of configuration options that we determined to be important. The size of these sets was substantial, but did not include every possible configuration option. The program behaviors we studied included four structural coverage criteria for this study. Other program behaviors such as data flows or fault detection might lead to different results. Our test suites taken together have reasonable, but not complete, coverage. Individually the test cases tend to be focused on specific functionality, rather than combining multiple activities in a single test case. In that sense they are more like a typical regression suite than a customer acceptance suite. We intend to address each of these issues in future work.

6. RELATED WORK

Symbolic Evaluation. In the mid 1970's, King was one of the first to propose symbolic evaluation as an aid to program testing [10]. Theorem provers at that time, however, were fairly simple, limiting the approach's practical potential. Recent years have seen remarkable advances in Satisfiability Modulo Theory and SAT solvers, which has enabled symbolic evaluation to scale to more practical problems. Some recent symbolic evaluators include DART [8, 9], CUTE [15], SPLAT [16], EXE [3], and KLEE [2]. There are important technical differences between these systems, e.g., DART uses *concolic execution*, which mixes concrete and symbolic evaluation, and KLEE uses pure symbolic evaluation. However, at a high level, the basic idea is the same: the programmer marks values as symbolic, and the evaluator explores all possible program paths reachable under arbitrary assignments to those symbolic values. As we mentioned earlier, Otter is closest in implementation terms to KLEE.

Software Engineering for Configurable Systems. Researchers and practitioners have developed several strategies to cope with the problem of testing configurable systems. One popular approach is combinatorial testing [4, 1, 12, 5], which, given an *interaction strength* t, computes a *covering array*, a small set of configurations such that all possible t-way combinations of option settings appear in at least one configuration. The subject program is then tested under each configuration in the covering array, which will have very few configurations compared to the full configuration space of the program.

Several studies to date suggest that even low interaction strength (2- or 3-way) covering array testing can yield good line coverage while higher strengths may be needed for edge or path coverage or fault detection [1, 6, 11]. However, as far as we are aware, all of these studies have taken a black

box approach to understanding covering array performance. Thus it is unclear exactly how well and why covering arrays work. On the one hand, a t-way covering array contains all possible t-way interactions, but not all combinations of options may be needed for a given program or test suite. On the other hand, a t-way covering array must contain many combinations of more than t options, making it difficult to tell whether t-way interactions, or larger ones, are responsible for a given covering array's coverage. Our work attempts to better understand what specific configuration space characteristics control system behavior.

7. CONCLUSIONS AND FUTURE WORK

We have presented an initial experiment using symbolic evaluation to study the interactions among configuration options for three software systems. Keeping existing threats to validity in mind, we drew several conclusions. All of these conclusions are specific to our programs, test suites, and configuration spaces; further work is clearly needed to establish more general trends.

First, we found that we could achieve maximum coverage without executing anything near all the possible configurations. Most coverage was accounted for by lower-strength interactions, across all of line, basic block, edge, and condition coverage. Second, if we packed interactions into configurations greedily, it took only five to ten configurations to achieve this maximal coverage. Third, we also found that in fact it only took one configuration to get the vast majority of the maximum coverage. Finally, by mapping the interactions we gained some insight into why the minimal covering sets are so small. We observed that many options either did not interact or interacted at low strength, and it is often possible to combine different interactions together into a single configuration. Taken together, our results strongly suggest our main hypothesis—that in practical systems, configuration spaces are significantly smaller than combinatorics suggest, and they can be understood from the composition of a small number of interactions.

Based on this work, we plan to pursue several research directions. First, we will extend our studies to better understand how configurability affects software development. Some initial issues we will tackle include increasing the number and types of options and repeating our study on more and larger subject systems. Second, we plan to enhance our symbolic evaluator to improve performance, which should enable larger scale studies. One potential approach is to use path pruning heuristics to reduce the search space, although we would no longer have complete information. Finally, we will explore potential applications of our approach and results. For example, we may be able to use symbolic evaluation to discretize integer-valued configuration options and to identify enabling options. As our results show that different test cases depend on different configuration options, we will investigate how this information can be used to support a variety of software tasks, such as test prioritization, configuration-aware regression testing, and impact analysis.

8. REFERENCES

[1] R. Brownlie, J. Prowse, and M. S. Phadke. Robust testing of AT&T PMX/StarMAIL using OATS. AT&T Technical Journal, 71(3):41–7, 1992.

[2] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[3] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *CCS*, pages 322–335, 2006.

[4] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: an approach to testing based on combinatorial design. *TSE*, 23(7):437–44, 1997.

[5] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn. Constructing test suites for interaction testing. In *ICSE*, pages 38–48, 2003.

[6] I. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. M. ws, and A. Iannino. Applying design of experiments to software testing. In *ICSE*, pages 205–215, 1997.

[7] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *CAV*, July 2007.

[8] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[9] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *NDSS*. Internet Society, 2008.

[10] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[11] D. Kuhn and M. Reilly. An investigation of the applicability of design of experiments to software testing. In *NASA Goddard/IEEE Software Engineering Workshop*, pages 91–95, 2002.

[12] R. Mandl. Orthogonal Latin squares: an application of experiment design to compiler testing. *Commun. ACM*, 28(10):1054–1058, 1985.

[13] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *International Conference on Compiler Construction*, pages 213–228, 2002.

[14] A. Porter, C. Yilmaz, A. M. Memon, D. C. Schmidt, and B. Natarajan. Skoll: A process and infrastructure for distributed continuous quality assurance. *TSE*, 33(8):510–525, August, 2007.

[15] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *FSE-13*, pages 263–272, 2005.

[16] R.-G. Xu, P. Godefroid, and R. Majumdar. Testing for buffer overflows with length abstraction. In *ISSTA*, pages 27–38, 2008.

APPENDIX

The figures below depict the entire set of interactions due to line coverage for each of our subject programs: ngIRCd, grep, and vsftpd. In these graphs, a node is shaded if it guarantees coverage on its own, black edges represent interactions involving just two nodes, and interactions involving more than two nodes are cliques of similarly patterned and similarly colored edges. Nodes represent one or more option settings; we merged nodes with common neighbors, listing all settings the node represents. The ngIRCd options are all prefixed with Conf_, and similarly the vsftpd options are prefixed with tunable_; we omit these prefixes to save space.

In each of our programs, there were some settings that were involved in many interactions. In ngIRCd, this is ListenIPv4=1; in vsftpd, it is a 3-way interaction among ssl_enable=0, local_enable=0, and anonymous_enable=1; and in grep, it is a 2-way interaction between match_words=0 and match_lines=0. For vsftpd and grep, we grouped this key interaction into a single node. Then, to help keep the graphs legible, we omitted the edges incident on these key nodes for interactions involving more than one other node. Instead, in Figure 10, interactions involving the key node are marked by thin edges while others are marked by thick edges; Figures 11 and 12 have the roles of thick and thin edges reversed.

ngIRCd is depicted differently than grep and vsftpd; many of ngIRCd's option settings have *nearly* identical neighbors as some other settings, so most options are depicted as a single node which contains all of the possible values for that option. When multiple values of an option interact with the same other settings, a single edge is used to represent *all* such interactions, with the set of values for these interactions enclosed together in a subnode of the option's node. For example, the thin black edge connecting the MaxNickLength node to the values 20 and 3600 of PongTimeout represents 10 different 3-way interactions: the interaction among ListenIPv4=1 (indicated by the line being thin), each of the 5 values of MaxNickLength, and each of PongTimeout=20 and PongTimeout=3600. (The colors of the subnodes of MaxNickLength are only to help distinguish the subnodes one from another.)

Two options, UID and ListenIPv4, are not depicted with a single node containing all the values because both options' settings have very few edges in the graph, so this would not have helped keep the graph sparse.

While the graphs are intended to give a rough sense of what options interact and how, they are difficult to decipher, even with our attempts to keep them tidy. Therefore, we also list the interactions themselves in Figures 13 through 16.

Finally, in Figure 17, we list the entire set of options we set symbolic during our tests. For the non-boolean options, some had constraints on what values they could take, either implicitly in the program, or imposed by us (in an attempt to maximize coverage while keeping symbolic evaluation practical); the figure lists their possible values. The remaining options were integer-valued options on which we put no constraints during symbolic evaluation. For these unconstrained options, we manually selected the values to use in the guaranteed coverage calculations and in Figures 10 through 12, as described in section 5.1.

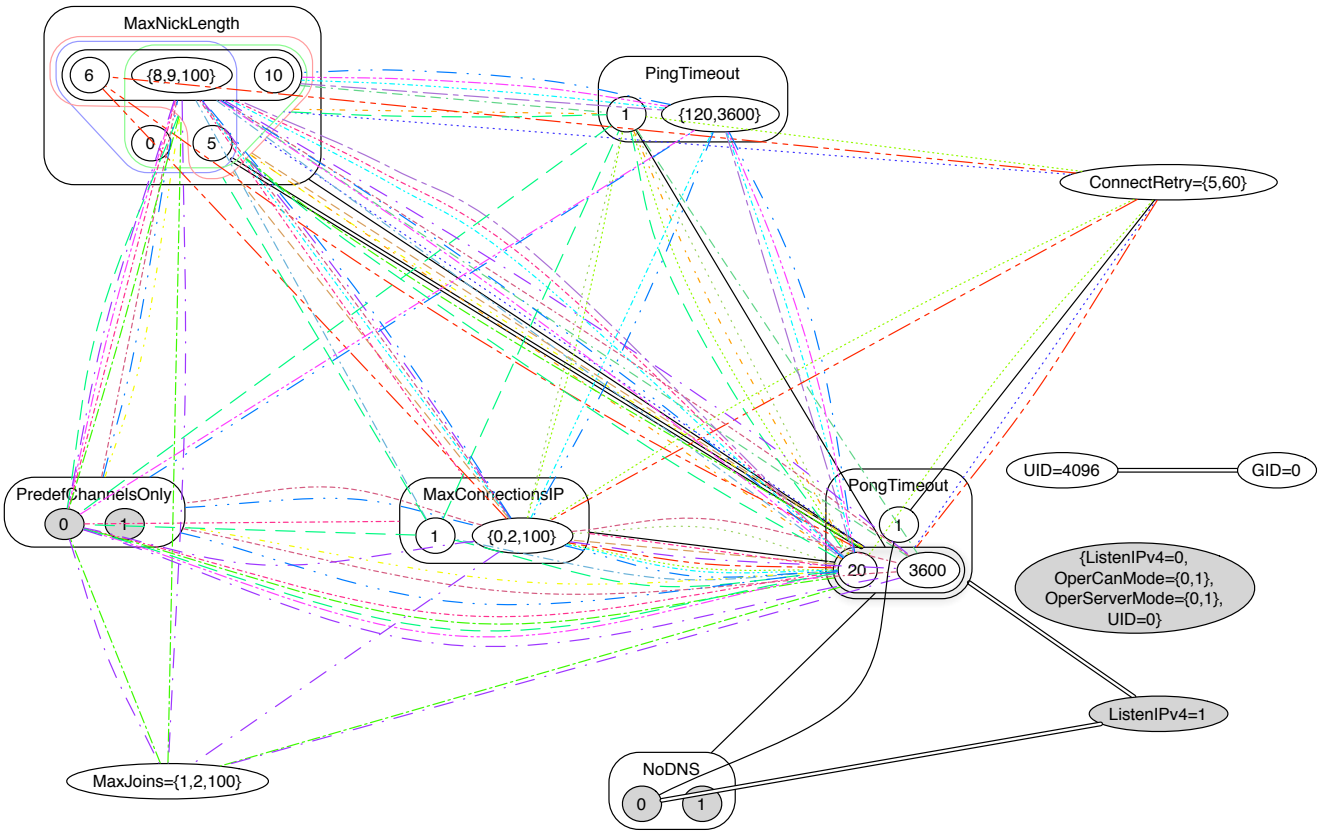


Figure 10: All line-coverage interactions for ngIRCd. Thin-edge cliques implicitly include `ListenIPv4=1`.

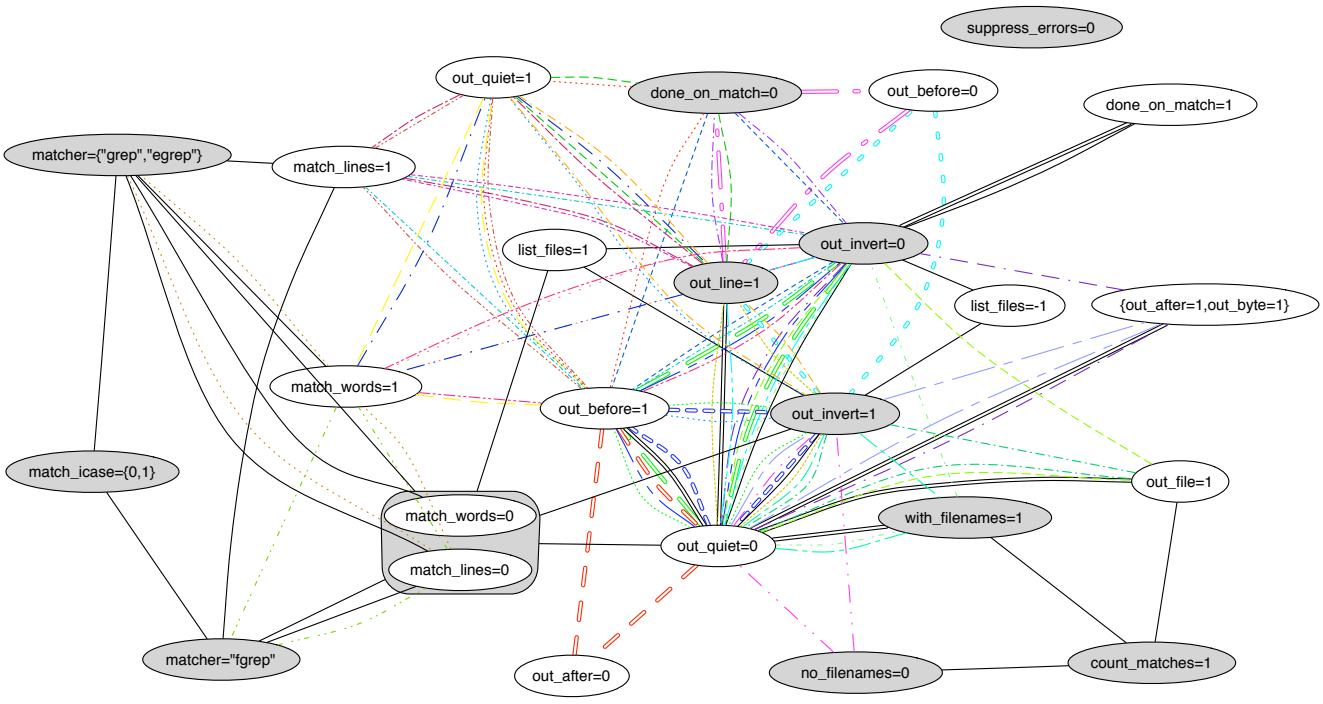


Figure 11: All line-coverage interactions for grep.
Thick-edge cliques implicitly include `match_words=0,match_lines=0`.

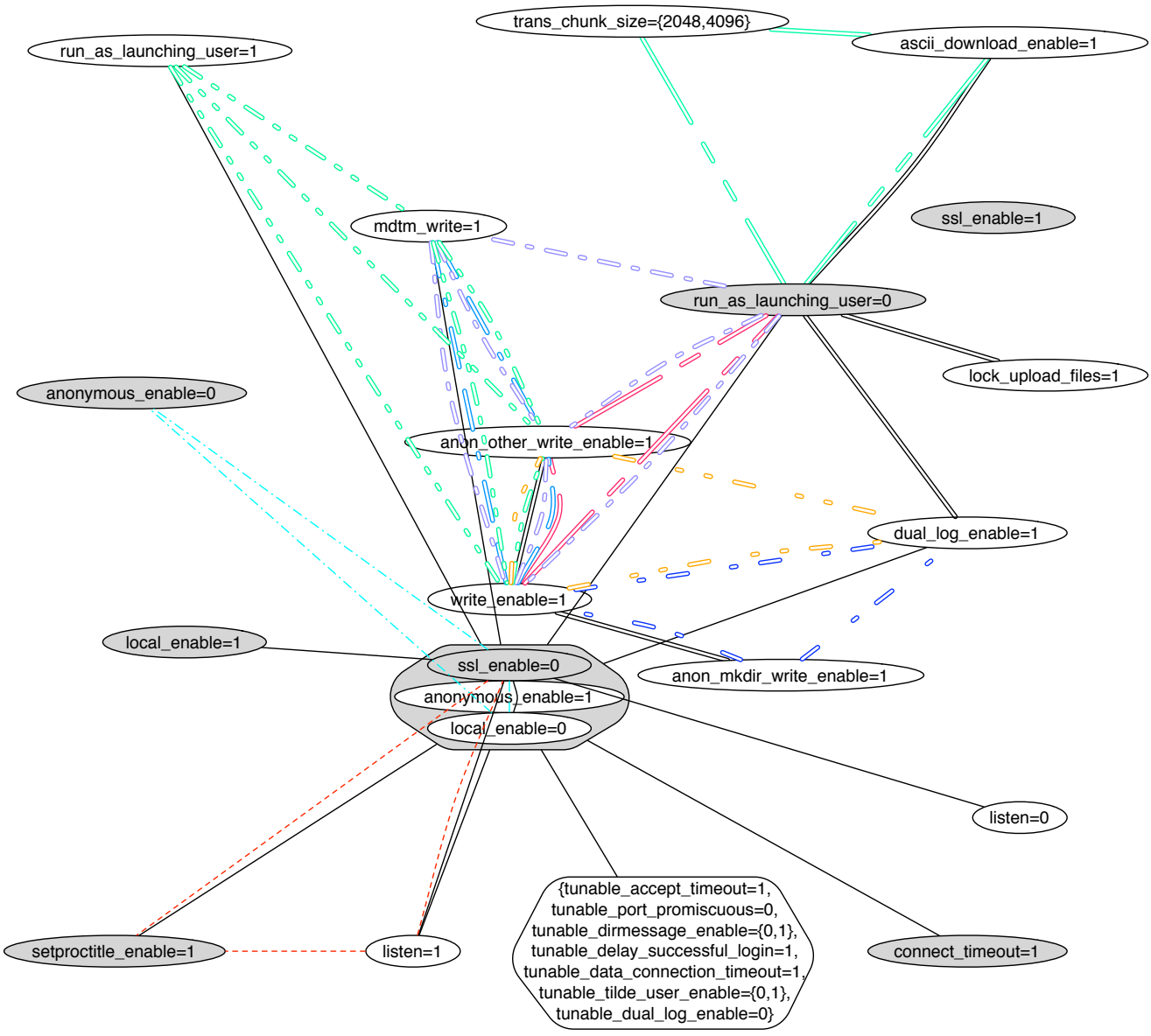


Figure 12: All line-coverage interactions for vsftpd.
Thick-edge cliques implicitly include ssl_enable=0,local_enable=0,anonymous_enable=1.

{ListenIPv4=0,OperCanMode={0,1},OperServerMode={0,1},UID=0}
ListenIPv4=1
NoDNS=0
NoDNS=1
PredefChannelsOnly=0
PredefChannelsOnly=1
GID=0:UID=4096
ListenIPv4=1:NoDNS=0
ListenIPv4=1:PongTimeout=1
ListenIPv4=1:PongTimeout=20
ListenIPv4=1:PongTimeout=3600
MaxNickLength=0:PongTimeout=20
MaxNickLength=0:PongTimeout=3600
MaxNickLength=5:PongTimeout=20
MaxNickLength=5:PongTimeout=3600
MaxNickLength=6:PongTimeout=20
MaxNickLength=6:PongTimeout=3600
MaxNickLength={8,9,100}:PongTimeout=20
MaxNickLength={8,9,100}:PongTimeout=3600
ListenIPv4=1:ConnectRetry={5,60}:PongTimeout=1
ListenIPv4=1:MaxConnectionsIP={0,2,100}:PongTimeout=20
ListenIPv4=1:MaxConnectionsIP={0,2,100}:PongTimeout=3600
ListenIPv4=1:MaxConnectionsIP=1:PongTimeout=20
ListenIPv4=1:MaxConnectionsIP=1:PongTimeout=3600
ListenIPv4=1:MaxNickLength=0:PongTimeout=20
ListenIPv4=1:MaxNickLength=0:PongTimeout=3600
ListenIPv4=1:MaxNickLength=10:PongTimeout=20
ListenIPv4=1:MaxNickLength=10:PongTimeout=3600
ListenIPv4=1:MaxNickLength=5:PongTimeout=20
ListenIPv4=1:MaxNickLength=5:PongTimeout=3600
ListenIPv4=1:MaxNickLength=6:PongTimeout=20
ListenIPv4=1:MaxNickLength=6:PongTimeout=3600
ListenIPv4=1:MaxNickLength={8,9,100}:PongTimeout=20
ListenIPv4=1:MaxNickLength={8,9,100}:PongTimeout=3600
ListenIPv4=1:NoDNS=0:PongTimeout=1
ListenIPv4=1:NoDNS=0:PongTimeout=20
ListenIPv4=1:NoDNS=0:PongTimeout=3600
ListenIPv4=1:NoDNS=1:PongTimeout=20
ListenIPv4=1:NoDNS=1:PongTimeout=3600
ListenIPv4=1:PingTimeout=1:PongTimeout=20
ListenIPv4=1:PingTimeout=1:PongTimeout=3600
ListenIPv4=1:ConnectRetry={5,60}:MaxNickLength=0:PongTimeout=20
ListenIPv4=1:ConnectRetry={5,60}:MaxNickLength=0:PongTimeout=3600
ListenIPv4=1:ConnectRetry={5,60}:MaxNickLength=10:PongTimeout=20
ListenIPv4=1:ConnectRetry={5,60}:MaxNickLength=10:PongTimeout=3600
ListenIPv4=1:ConnectRetry={5,60}:MaxNickLength=5:PongTimeout=20
ListenIPv4=1:ConnectRetry={5,60}:MaxNickLength=5:PongTimeout=3600
ListenIPv4=1:ConnectRetry={5,60}:MaxNickLength={8,9,100}:PongTimeout=20
ListenIPv4=1:ConnectRetry={5,60}:MaxNickLength={8,9,100}:PongTimeout=3600
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength=10:PongTimeout=20
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength=10:PongTimeout=3600
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength=5:PongTimeout=20
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength=5:PongTimeout=3600
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength=6:PongTimeout=20
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength=6:PongTimeout=3600
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength={8,9,100}:PongTimeout=20
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength={8,9,100}:PongTimeout=3600
ListenIPv4=1:MaxConnectionsIP={0,2,100}:PingTimeout=1:PongTimeout=20
ListenIPv4=1:MaxConnectionsIP=1:MaxNickLength=10:PongTimeout=20
ListenIPv4=1:MaxConnectionsIP=1:MaxNickLength=6:PongTimeout=20
ListenIPv4=1:MaxConnectionsIP=1:MaxNickLength={8,9,100}:PongTimeout=20
ListenIPv4=1:MaxNickLength=10:PingTimeout={120,3600}:PongTimeout=20
ListenIPv4=1:MaxNickLength=10:PingTimeout=1:PongTimeout=20
ListenIPv4=1:MaxNickLength=10:PingTimeout=1:PongTimeout=3600
ListenIPv4=1:MaxNickLength=10:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxNickLength=10:PongTimeout=20:PredefChannelsOnly=1
ListenIPv4=1:MaxNickLength=10:PongTimeout=3600:PredefChannelsOnly=0
ListenIPv4=1:MaxNickLength=10:PongTimeout=3600:PredefChannelsOnly=1
ListenIPv4=1:MaxNickLength=5:PingTimeout=1:PongTimeout=20
ListenIPv4=1:MaxNickLength=5:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxNickLength=5:PongTimeout=20:PredefChannelsOnly=1
ListenIPv4=1:MaxNickLength=5:PongTimeout=3600:PredefChannelsOnly=0
ListenIPv4=1:MaxNickLength=5:PongTimeout=3600:PredefChannelsOnly=1
ListenIPv4=1:MaxNickLength=6:PingTimeout={120,3600}:PongTimeout=20
ListenIPv4=1:MaxNickLength=6:PingTimeout=1:PongTimeout=20
ListenIPv4=1:MaxNickLength=6:PingTimeout=1:PongTimeout=3600

Figure 13: ngIRCd interactions

```
ListenIPv4=1:MaxNickLength=6:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxNickLength=6:PongTimeout=20:PredefChannelsOnly=1
ListenIPv4=1:MaxNickLength=6:PongTimeout=3600:PredefChannelsOnly=0
ListenIPv4=1:MaxNickLength=6:PongTimeout=3600:PredefChannelsOnly=1
ListenIPv4=1:MaxNickLength={8,9,100}:PingTimeout={120,3600}:PongTimeout=20
ListenIPv4=1:MaxNickLength={8,9,100}:PingTimeout=1:PongTimeout=20
ListenIPv4=1:MaxNickLength={8,9,100}:PingTimeout=1:PongTimeout=3600
ListenIPv4=1:MaxNickLength={8,9,100}:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxNickLength={8,9,100}:PongTimeout=20:PredefChannelsOnly=1
ListenIPv4=1:MaxNickLength={8,9,100}:PongTimeout=3600:PredefChannelsOnly=0
ListenIPv4=1:MaxNickLength={8,9,100}:PongTimeout=3600:PredefChannelsOnly=1
ListenIPv4=1:ConnectRetry={5,60}:MaxConnectionsIP={0,2,100}:MaxNickLength=6:PongTimeout=20
ListenIPv4=1:ConnectRetry={5,60}:MaxConnectionsIP={0,2,100}:MaxNickLength=6:PongTimeout=3600
ListenIPv4=1:ConnectRetry={5,60}:MaxConnectionsIP={0,2,100}:PingTimeout=1:PongTimeout=20
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength=10:PingTimeout={120,3600}:PongTimeout=20
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength=10:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength=10:PongTimeout=3600:PredefChannelsOnly=0
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength=10:PongTimeout=3600:PredefChannelsOnly=1
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength=6:PingTimeout={120,3600}:PongTimeout=20
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength=6:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength=6:PongTimeout=3600:PredefChannelsOnly=0
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength=6:PongTimeout=3600:PredefChannelsOnly=1
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength={8,9,100}:PingTimeout={120,3600}:PongTimeout=20
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength={8,9,100}:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength={8,9,100}:PongTimeout=3600:PredefChannelsOnly=0
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength={8,9,100}:PongTimeout=3600:PredefChannelsOnly=1
ListenIPv4=1:MaxConnectionsIP=1:MaxNickLength=10:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxConnectionsIP=1:MaxNickLength=6:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxConnectionsIP=1:MaxNickLength={8,9,100}:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxJoins={1,2,100}:MaxNickLength=10:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxJoins={1,2,100}:MaxNickLength=10:PongTimeout=3600:PredefChannelsOnly=0
ListenIPv4=1:MaxJoins={1,2,100}:MaxNickLength=5:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxJoins={1,2,100}:MaxNickLength=5:PongTimeout=3600:PredefChannelsOnly=0
ListenIPv4=1:MaxJoins={1,2,100}:MaxNickLength=6:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxJoins={1,2,100}:MaxNickLength=6:PongTimeout=3600:PredefChannelsOnly=0
ListenIPv4=1:MaxJoins={1,2,100}:MaxNickLength={8,9,100}:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxJoins={1,2,100}:MaxNickLength={8,9,100}:PongTimeout=3600:PredefChannelsOnly=0
ListenIPv4=1:MaxNickLength=10:PingTimeout={120,3600}:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxNickLength=6:PingTimeout={120,3600}:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxNickLength={8,9,100}:PingTimeout={120,3600}:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxJoins={1,2,100}:MaxNickLength=10:PongTimeout=3600:PredefChannelsOnly=0
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxJoins={1,2,100}:MaxNickLength=6:PongTimeout=3600:PredefChannelsOnly=0
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxJoins={1,2,100}:MaxNickLength={8,9,100}:PongTimeout=3600:PredefChannelsOnly=0
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength=10:PingTimeout={120,3600}:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength=10:PingTimeout={120,3600}:PongTimeout=20:PredefChannelsOnly=1
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength=6:PingTimeout={120,3600}:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength=6:PingTimeout={120,3600}:PongTimeout=20:PredefChannelsOnly=1
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength={8,9,100}:PingTimeout={120,3600}:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxConnectionsIP={0,2,100}:MaxNickLength={8,9,100}:PingTimeout={120,3600}:PongTimeout=20:PredefChannelsOnly=1
ListenIPv4=1:MaxConnectionsIP=1:MaxNickLength=10:PingTimeout=1:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxConnectionsIP=1:MaxNickLength=5:PingTimeout=1:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxConnectionsIP=1:MaxNickLength=6:PingTimeout=1:PongTimeout=20:PredefChannelsOnly=0
ListenIPv4=1:MaxConnectionsIP=1:MaxNickLength={8,9,100}:PingTimeout=1:PongTimeout=20:PredefChannelsOnly=0
```

Figure 14: ngIRCd interactions, continued

count_matches=1
done_on_match=0
matcher={“grep”,“egrep”}
matcher=“fgrep”
match_icase={0,1}
no_filenames=0
out_invert=0
out_invert=1
out_line=1
suppress_errors=0
with_filenames=1
count_matches=1:no_filenames=0
count_matches=1:out_file=1
count_matches=1:with_filenames=1
done_on_match=1:out_invert=0
matcher={“grep”,“egrep”}:match_icase={0,1}
matcher={“grep”,“egrep”}:match_lines=0
matcher={“grep”,“egrep”}:match_lines=1
matcher={“grep”,“egrep”}:match_words=0
matcher={“grep”,“egrep”}:match_words=1
matcher=“fgrep”:match_icase={0,1}
matcher=“fgrep”:match_lines=0
matcher=“fgrep”:match_lines=1
list_files=-1:out_invert=0
list_files=1:out_invert=0
list_files=-1:out_invert=1
list_files=1:out_invert=1
match_lines=0:match_words=0
out_invert=0:out_quiet=0
out_invert=1:out_quiet=0
done_on_match=0:out_before=1:out_invert=0
done_on_match=0:out_before=1:out_quiet=1
done_on_match=0:out_invert=0:out_line=1
done_on_match=0:out_line=1:out_quiet=1
match_lines=0:match_words=0:matcher={“grep”,“egrep”}
matcher={“grep”,“egrep”}:match_lines=0:match_words=1
match_lines=0:match_words=0:matcher=“fgrep”
matcher=“fgrep”:match_lines=0:match_words=1
match_lines=0:match_words=0:list_files=1
match_lines=0:match_words=0:out_invert=1
match_lines=0:match_words=0:out_quiet=0
match_lines=1:out_before=1:out_invert=0
match_lines=1:out_before=1:out_quiet=1
match_lines=1:out_invert=0:out_line=1
match_lines=1:out_line=1:out_quiet=1
match_words=1:out_before=1:out_invert=0
match_words=1:out_before=1:out_quiet=1
match_words=1:out_invert=0:out_line=1
match_words=1:out_line=1:out_quiet=1
no_filenames=0:out_invert=1:out_quiet=0
{out_after=1,out_byte=1}:out_invert=0:out_quiet=0
{out_after=1,out_byte=1}:out_invert=1:out_quiet=0
out_before=1:out_invert=0:out_quiet=0
out_before=1:out_invert=1:out_quiet=0
out_before=1:out_invert=1:out_quiet=1
out_file=1:out_invert=0:out_quiet=0
out_file=1:out_invert=1:out_quiet=0
out_invert=0:out_line=1:out_quiet=0
out_invert=0:out_quiet=0:with_filenames=1
out_invert=1:out_line=1:out_quiet=0
out_invert=1:out_line=1:out_quiet=1
out_invert=1:out_quiet=0:with_filenames=1
match_lines=0:match_words=0:done_on_match=1:out_invert=0
match_lines=0:match_words=0:{out_after=1,out_byte=1}:out_quiet=0
match_lines=0:match_words=0:out_before=1:out_quiet=0
match_lines=0:match_words=0:out_file=1:out_quiet=0
match_lines=0:match_words=0:out_line=1:out_quiet=0
match_lines=0:match_words=0:out_quiet=0:with_filenames=1
match_lines=0:match_words=0:done_on_match=0:out_before=0:out_line=1
match_lines=0:match_words=0:out_after=0:out_before=1:out_quiet=0
match_lines=0:match_words=0:out_before=0:out_invert=1:out_line=1
match_lines=0:match_words=0:out_before=1:out_invert=0:out_quiet=0
match_lines=0:match_words=0:out_before=1:out_invert=1:out_quiet=0

Figure 15: grep interactions

anonymous_enable=0
connect_timeout=1
local_enable=1
run_as_launching_user=0
setproctitle_enable=1
ssl_enable=0
ssl_enable=1
listen=0:ssl_enable=0
listen=1:ssl_enable=0
local_enable=0:ssl_enable=0
local_enable=1:ssl_enable=0
anonymous_enable=0:local_enable=0:ssl_enable=0
anonymous_enable=1:local_enable=0:ssl_enable=0
listen=1:setproctitle_enable=1:ssl_enable=0
anonymous_enable=1:local_enable=0:ssl_enable=0:{accept_timeout=1,data_connection_timeout=1,delay_successful_login=1,
dirmessage_enable={0,1},dual_log_enable=0,port_promiscuous=0,tilde_user_enable={0,1}}
anonymous_enable=1:local_enable=0:ssl_enable=0:connect_timeout=1
anonymous_enable=1:local_enable=0:ssl_enable=0:dual_log_enable=1
anonymous_enable=1:local_enable=0:ssl_enable=0:listen=1
anonymous_enable=1:local_enable=0:ssl_enable=0:mdtm_write=1
anonymous_enable=1:local_enable=0:ssl_enable=0:run_as_launching_user=0
anonymous_enable=1:local_enable=0:ssl_enable=0:run_as_launching_user=1
anonymous_enable=1:local_enable=0:ssl_enable=0:setproctitle_enable=1
anonymous_enable=1:local_enable=0:ssl_enable=0:anon_mkdir_write_enable=1:write_enable=1
anonymous_enable=1:local_enable=0:ssl_enable=0:anon_other_write_enable=1:write_enable=1
anonymous_enable=1:local_enable=0:ssl_enable=0:ascii_download_enable=1:run_as_launching_user=0
anonymous_enable=1:local_enable=0:ssl_enable=0:dual_log_enable=1:run_as_launching_user=0
anonymous_enable=1:local_enable=0:ssl_enable=0:lock_upload_files=1:run_as_launching_user=0
anonymous_enable=1:local_enable=0:ssl_enable=0:anon_mkdir_write_enable=1:dual_log_enable=1:write_enable=1
anonymous_enable=1:local_enable=0:ssl_enable=0:anon_other_write_enable=1:dual_log_enable=1:write_enable=1
anonymous_enable=1:local_enable=0:ssl_enable=0:anon_other_write_enable=1:mdtm_write=1:write_enable=1
anonymous_enable=1:local_enable=0:ssl_enable=0:anon_other_write_enable=1:run_as_launching_user=0:write_enable=1
anonymous_enable=1:local_enable=0:ssl_enable=0:ascii_download_enable=1:run_as_launching_user=0:trans_chunk_size={2048,4096}
anonymous_enable=1:local_enable=0:ssl_enable=0:anon_other_write_enable=1:mdtm_write=1:run_as_launching_user=0:write_enable=1
anonymous_enable=1:local_enable=0:ssl_enable=0:anon_other_write_enable=1:mdtm_write=1:run_as_launching_user=1:write_enable=1

Figure 16: vsftpd interactions

Name	vsftpd	ngIRCd	grep
Booleans	anon_mkdir_write_enable anon_other_write_enable anon_upload_enable anonymous_enable ascii_download_enable ascii_upload_enable* delete_failed_uploads* dirmessage_enable dual_log_enable listen local_enable lock_upload_files mdtm_write pasv_addr_resolve* port_promiscuous run_as_launching_user setproctitle_enable ssl_enable tilde_user_enable write_enable	ListenIPv4 NoDNS OperCanMode OperServerMode PredefChannelsOnly	count_matches done_on_match filename_mask* match_icase match_lines match_words no_filenames out_byte out_file out_invert out_line out_quiet suppress_errors with_filenames
Other	accept_timeout chown_upload_mode* connect_timeout data_connection_timeout delay_successful_login ftp_data_port* listen_port* max_clients max_per_ip trans_chunk_size	ConnectRetry ∈ {5,60} GID MaxConnectionsIP MaxJoins MaxNickLength PingTimeout ∈ {1,20,3600} PongTimeout ∈ {1,20,3600} UID	list_files ∈ {-1,0,1} matcher ∈ {“grep”,“egrep”,“fgrep”} out_after ∈ {0,1} out_before ∈ {0,1}

Figure 17: Symbolic configuration options. Asterisks indicate options that never led to branching during symbolic evaluation.