2017 IEEE/ACM 39th International Conference on Software Engineering

Balancing Soundness and Efficiency for Practical Testing of Configurable Systems

Sabrina Souto
State University of Paraíba
Paraíba, Brazil
sabrinadfs@gmail.com

Marcelo d'Amorim
Federal University of Pernambuco
Pernambuco, Brazil
damorim@cin.ufpe.br

Rohit Gheyi
Federal University of Campina Grande
Paraíba, Brazil
rohit@dsc.ufcg.edu.br

***Abstract*—Testing configurable systems is important and challenging due to the enormous space of configurations where errors can hide. Existing approaches to test these systems are often costly or unreliable. This paper proposes `S-SPLat`, a technique that combines heuristic sampling with symbolic search to obtain both breadth and depth in the exploration of the configuration space. `S-SPLat` builds on `SPLat`, our previously developed technique, that explores all reachable configurations from tests. In contrast to its predecessor, `S-SPLat` sacrifices soundness in favor of efficiency. We evaluated our technique on eight software product lines of various sizes and on a large configurable system – GCC. Considering the results for GCC, `S-SPLat` was able to reproduce all five bugs that we previously found in a previous study with `SPLat` but much faster and it was able to find two new bugs in a recent release of GCC. Results suggest that it is preferable to use a combination of simple heuristics to drive the symbolic search as opposed to a single heuristic. `S-SPLat` and our experimental infrastructure are publicly available.**

***Keywords*-sampling; testing; configuration.**

I. Introduction

Configurable systems are those that can be adapted from a set of input options, reflected in code in the form of variations. These systems are *prevalent*. The Firefox web browser [4], the Linux kernel [9], the GCC compiler infrastructure [5], and the deals-recommendation web service Groupon [8] are some well-known examples of configurable systems. Unfortunately, configuration-related errors are *not rare* [11], [19], [25], [39]. Some of these errors have been widely publicized in the media given the volume of users or data they affected [1], [3], [23].

A configuration space consists of all combinations of input options that can be used to configure a system. Configuration errors are often manifested in a small fraction of such space. Finding an adequate set of configurations for testing is therefore challenging. In one limit, testing exhaustively, against all configurations, is unacceptably expensive. In another limit, testing against one (default) configuration, albeit popular, leads to high chances of escaped defects [22], [35]. These approaches to testing make strong commitments to either cost or reliability.

Combinatorial Interaction Testing (CIT) [51] has been popularized to balance probability of finding configuration errors (i.e., efficacy) and efficiency [17], [30], [31], [38], [42]. Several sampling heuristics have been proposed in the past to support CIT. For example, pairwise testing adequacy [52] is obtained when the sample set of selected configurations covers all possible pairs of input options. The intuition is that bugs are uniformly distributed in the configuration space; a more uniform search will then have higher chances of finding bugs. Sampling heuristics are typically black-box; they do not take code (test or app) into account. This is an important source of imprecision that can lead to error misses and higher cost.

More recently, sound testing techniques have been proposed [28], [29], [33], [41]; they assure that all configuration errors that can be captured with a given test will be captured. These techniques dynamically explore *all reachable* configurations from a given test. The hypothesis is that, when each test is analyzed separately, configuration complexity is much lower compared to the (theoretical) combinatorial complexity. This is the case, for example, when test execution dynamically accesses a relatively small number of configuration variables [29]. Unfortunately, reaching soundness without compromising cost is challenging. Recent empirical studies indicate that scalability depends on many factors including the subjects and tests used [40], [47], [49].

In summary, the inability to reason about test cases constitutes an important limitation of sampling techniques and the potential high execution cost is an important drawback of dynamic sound techniques. This contrasting set of limitations motivated us to explore the synergistic integration of these approaches with the goal of finding a better balance between reliability (as to miss fewer errors) and cost (as to find errors faster). In this study, we used `SPLat` [29], [47], [49], a sound technique, previously developed by the authors, that monitors variable accesses in one execution and, based on that, decides which configurations should be executed next. Our goal is to understand the effects of combining sampling heuristics with a sound technique for testing configurable systems. We propose `S-SPLat`, a variant of `SPLat` that selects configurations according to a given sampling heuristic. Note that tension between soundness and tractability has a long history in program analysis. Recently, the static analysis community acknowledged the importance of making (and documenting) conscious unsound design choices in favor of practical soundy solutions [34]. A similar tension exists in the context of dynamic analysis techniques, such as `SPLat`.

We analyzed `S-SPLat` with five basic heuristics that have demonstrated promising results in different studies [11], [38] and eleven combinations of these basic heuristics. We considered eight relatively-small software product lines and one large configurable system (GCC) in the evaluation. Results confirm expectations that regular `SPLat` often does not scale, i.e., it is unable to explore the space of reachable configurations completely within a long time budget. This situation occurs even on smaller subjects, where `SPLat` did not complete exploration in two of eight cases. For GCC, `SPLat` could finish exploration of only 5.7% (202 of 3,557) of the tests after one week of execution. Considering `S-SPLat`, results indicate that, compared to `SPLat`, techniques dramatically reduce the number of configurations explored yet retaining their ability to reveal failures and crashes. The heuristic variants of `S-SPLat` were able to find all five crashes that we previously-documented [49] on GCC release 4.8.2 and uncovered two new crashes on release 6.1, one of which was reported to the GCC team and was already fixed. Overall, considering our experimental setup, results suggest that it is preferable to combine some heuristics that demand a relatively low number of test requirements (e.g., `one-enabled` and `one-disabled` [11]) than using heuristics that solicit more test requirements (e.g., `pairwise` [52]).

This paper makes the following contributions: (**idea**) we propose a hybrid approach to test configurable software that integrates sampling with systematic exploration of configurations (Section IV). Our approach is similar in spirit to hybrid concolic execution [21], [36]; (**implementation**) we implemented `S-SPLat`, a variant of our previously-developed technique `SPLat` [29]. `S-SPLat` selects test-reachable configurations according to a given sampling heuristic; (**evaluation**) we evaluated `S-SPLat` on a number of subjects of various sizes and sources (Section V). Results indicate that the approach is promising, reducing cost and retaining ability to find failures and crashes; (**prototype**) we implemented a prototype of our technique. The code, datasets, containers, and scripts are all accessible from our website [48].

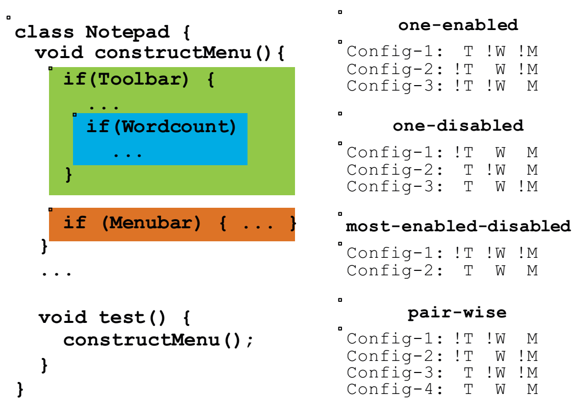
II. Background

This section introduces concepts and terminology to support the discussion along the text.

A. Configurable systems

Configurable Systems (CSs) are systems that can be adapted or configured according to a set of (configuration) *input options*, represented at the code level by *configuration variables*, also known as *feature variables*. To simplify definitions we assumed that configuration variables are boolean and that every input option maps to a single variable. Our definitions extend to any range types, booleans being a special case. If a configuration option is enabled, it means that the corresponding variable is set to `true`. Otherwise, it means that the variable is set to `false`. We make no distinction between Software Product Lines (SPLs) [12], [15] and other kinds of configurable systems.

Let ϕ be a set of boolean variables denoting the configuration variables of the system. A *configuration* $c : \phi \rightharpoonup \{false, true\}$ is a partial function from variables to boolean values; c maps *some* (not necessarily all) configuration variables to values



```
class Notepad {
  void constructMenu(){
    if(Toolbar) {
      ...
      if(Wordcount)
        ...
    }
    if (Menubar) { ... }
  }
  ...

  void test() {
    constructMenu();
  }
}
```

```
      one-enabled
Config-1:  T !W !M
Config-2: !T  W !M
Config-3: !T !W  M

      one-disabled
Config-1: !T  W  M
Config-2:  T !W  M
Config-3:  T  W !M

most-enabled-disabled
Config-1: !T !W !M
Config-2:  T  W  M

       pair-wise
Config-1: !T !W  M
Config-2: !T  W !M
Config-3:  T !W !M
Config-4:  T  W  M
```

Fig. 1. Excerpt of `Notepad` and sampling sets for testing.

false or *true*. A configuration can be encoded as a boolean formula $f_c = \bigwedge p_i$, where $p_i = (x_i | \neg x_i)$ for $x_i \in \phi$. We denote with $|f_c|$ the number of variables referenced in f_c. We say that a configuration c is *complete* iff $|f_c| = |\phi|$, i.e., all variables had their values assigned to *true* or *false*; it is *partial* otherwise, when only a subset of the variables had their values assigned.

Example: Let $\phi = \{A, B, C, D, E\}$. Configuration $f_{c_1} = A \wedge B \wedge \neg C \wedge D \wedge E$ is complete. Configurations $f_{c_2} = A \wedge \neg B$ and $f_{c_3} = A \wedge B \wedge \neg C \wedge E$ are incomplete. c_2 can be also written as `10???`, where the i_{th} position corresponds to the i_{th} configuration variable according to a given total order on variables. We use numbers `1` and `0` to indicate, respectively, the *true* or *false* values and the symbol `?` to indicate "undefined", it means that no value was assigned to a given variable during test execution.

A *feature model (FM)* [24] distinguishes which combinations of variables are legal from those that are not. A complete configuration is *valid* if it satisfies the constraints of the feature model. A partial assignment is *satisfiable* if it can be extended to a valid complete assignment. In practice, the FM may not be always documented or available.

B. Sampling heuristics

Notepad is a visual text editor, implemented as a configurable system, that has been previously used in related studies [26], [27], [29]. Figure 1 shows an excerpt of `Notepad` to illustrate sampling heuristics. Function `constructMenu` creates the menu on screen whereas the corresponding test checks if the menu has been properly created. Note that the call to `constructMenu` accesses the configuration variables `Toolbar (T)`, `Wordcount (W)`, and `Menubar (M)`, which are relevant for the presentation of both the toolbar and menubar on screen. The names in parentheses denote corresponding input configuration options.

Among the various sampling heuristics proposed in the literature related to testing configurable systems, we considered those that have been recently evaluated and are applicable in our context [11], [38]. The heuristic `one-disabled` samples configurations that have only one option disabled and all other options enabled whereas `one-enabled` samples configurations that have only one option enabled and all the rest disabled. The heuristic `most-enabled-disabled` combines two sample sets independently: one set with most options enabled and one set with most options disabled. In case feature constraints are not considered, this heuristic will only check two configurations: the one with all options enabled and the one with all options disabled. Finally, *t-wise* [52] samples all combinations of t configuration options. In particular, `pairwise` (*t-wise* with t=2), checks all pairs of configuration options, and it selects four configurations of the example of Figure 1. Considering options `T` and `W`, we can see that there is a configuration where both options are disabled (Config-1), two other configurations with only one of them enabled (Config-2 and Config-3), and another configuration where both configuration options are enabled (Config-4). The same situation occurs for configuration options `T` and `M` and options `W` and `M`.

Figure 1 shows, at the right-hand side, sampling sets for each of these heuristics. Notepad has a total of 17 variables, however, to simplify illustration we focused only on variables `M`, `T`, and `W`. For example, a sample set for `one-enabled` can be obtained with only three configurations; each configuration corresponding to the selection of one variable.

It is important to note that these heuristics are "black-box"; they do *not* take into consideration test and code. All tests will be executed against the same set of pre-computed configurations. This means that this approach can miss relevant configurations (leading to error misses) and can also include irrelevant configurations (leading to additional cost). Again, note that we used three variables on Figure 1 to simplify illustration. Had all 17 variables of Notepad been considered the size of the sample set would be significantly larger.

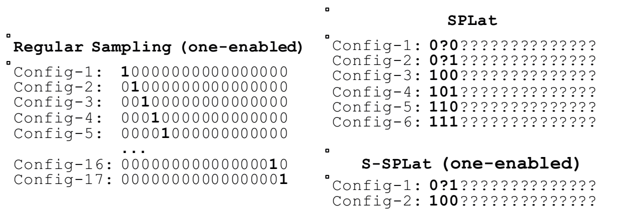
C. SPLat

In a nutshell, `SPLat` [29] works as follows. It executes the test on one configuration, observes the values of configuration variables that have been accessed during the execution, and uses these values to determine what other configurations should be considered in subsequent test executions. For example, if a test execution accessed only one configuration variable, f, with value *false*, then `SPLat` re-executes the test with f set to *true*. If that second execution accesses no other configuration variables, the search stops. Otherwise, it continues to explore the combinations of values of other accessed variables. `SPLat` repeats this process until it explores all dynamically reachable configurations or until it reaches a specified bound on number of configurations. As output `SPLat` returns, for each test, the configurations explored and the respective results (pass or fail).

`SPLat` only explores configurations which are reachable through tests. In contrast to sampling, `SPLat` does not explore configurations unrelated to the test and cannot miss configurations that would otherwise reveal failures. However, `SPLat` can access several variables through a test leading to high execution cost, a factor that can be controlled through sampling.

III. Example

This section presents an example comparing three approaches to testing: sampling with `one-enabled`, sound exploration with `SPLat`, and a combination of sampling and sound exploration. We used the Notepad test from Figure 1 in this comparison. Figure 2 shows the sample sets obtained. The values corresponding to variables `Toolbar`, `Wordcount`, and `Menubar` appear in the initial positions of the configuration vectors (see Section II-A) in these sample sets.



```
Regular Sampling (one-enabled)
Config-1:  10000000000000000
Config-2:  01000000000000000
Config-3:  00100000000000000
Config-4:  00010000000000000
Config-5:  00001000000000000
           ...
Config-16: 00000000000000010
Config-17: 00000000000000001
```

```
SPLat
Config-1: 0?0??????????????
Config-2: 0?1??????????????
Config-3: 100??????????????
Config-4: 101??????????????
Config-5: 110??????????????
Config-6: 111??????????????
```

```
S-SPLat (one-enabled)
Config-1: 0?1??????????????
Config-2: 100??????????????
```

Fig. 2. Configurations generated with `one-enabled`, `SPLat`, and `S-SPLat`, for the Notepad test from Figure 1.

`Notepad` has 17 configuration variables. Ignoring constraints, a sampling set adequate to `one-enabled` includes 17 configurations, one for each configuration variable to be enabled. Note, however, that the function under test, `constructMenu`, accesses only three of these variables (see Figure 1). There are 14 configurations in this set that produce the same output as they do not access any variable reachable through the test.

In contrast to sampling, `SPLat` runs the test only on reachable configurations, i.e., configurations that relate to accessed variables. For this case, `SPLat` runs the test against six configurations, as shown in Figure 2. Furthermore, note that variable `Wordcount` is only accessed if variable `Toolbar` is accessed; such conditional accesses enable further reduction of the search space. To provide soundness guarantees, `SPLat` needs to explore all reachable configurations; it assumes that tests reach a relatively small number of variables. Unfortunately, we observed that, for a large system, such as GCC, with hundreds of configuration options, exploring all reachable configurations is impractical for several tests [47], [49].

This paper proposes `S-SPLat` (for Sampling with `SPLat`) to mitigate the individual limitations of sampling and `SPLat`. `S-SPLat` uses the set of reachable configurations from `SPLat` (see Section II-C) to sample those configurations that satisfy a given sampling heuristic. `S-SPLat` is by definition unsound. The intuition is that the use of heuristics can provide a better balance between cost and reliability and that balance is essential for practicality in this domain [10], [34], [36].

Figure 2 shows results of `S-SPLat` with `one-enabled`. The corresponding sample set includes only two configurations. Note that it is not possible to build a `one-enabled`-configuration with variable `Wordcount` set as it is not possible to access `Wordcount` without also accessing variable `Toolbar`.

`S-SPLat` proceeds as follows to explore these two configurations. In its first iteration on this test, `S-SPLat` assigns default values to every variable it accesses (as `SPLat` does), reaching the partial configuration $\neg T \wedge \neg M$, which clearly does *not* satisfy `one-enabled`. At this point, `S-SPLat` looks for neighbor configurations to $\neg T \wedge \neg M$ that satisfy `one-enabled` and it finds $\neg T \wedge M$, which corresponds to the first configuration vector in the figure: `0?1...`. After executing the test on configuration `0?1...`, `S-SPLat` backtracks. Note that, at that point, variable

M was fully explored under $\neg T$. Then, S-SPLat initiates a new iteration from a partial configuration where T holds. At the end of execution of the corresponding path, S-SPLat explores the configuration $T \wedge \neg W \wedge \neg M$, which also satisfies one-enabled. At that point S-SPLat cannot find any other reachable configurations that satisfy one-enabled, and the execution finishes reporting two configurations.

IV. Sampling with SPLat

This section describes S-SPLat (for Sampling with SPLat), a modified version of the SPLat algorithm [29]. S-SPLat samples configurations from the set of dynamically reachable configuration from a given test that SPLat explores. In the following we present the workflow of S-SPLat, highlighting key modifications made to the original SPLat algorithm. A complete version of SPLat can be found elsewhere [29].

A. Basic workflow

Figure 3 shows the pseudo-code of S-SPLat. It takes as input a test t for a configurable system and an optional feature model fm. To drive the search, the algorithm uses a map that stores the values of feature variables (variable state at line 7) and a stack of feature variables accessed during the test run (variable stack at line 8). An object of type Assign encodes an assignment of boolean values to feature variables. It denotes a partial configuration (see Section II-A). S-SPLat supports different sampling heuristics (line 4).

The algorithm first initializes the values of feature variables (lines 13–16) using the feature model. Mandatory features are set to true (the only value they can hold) and optional features are initially set to false. We omit details of the feature model interface for brevity. It is important to note that the steps related to the feature model are optional. We did not show this variant for brevity. When present, feature models enable consistency checks on configurations, reducing the explored search space. After this initialization step, S-SPLat automatically instruments the code under test to track variable read accesses (line 19).

The main loop of the algorithm (lines 21–43) looks for a *partial configuration* (lines 30–42) and runs a test against that configuration (line 25). Whenever test execution is about to read a feature variable, it calls back the method notifyFeatureRead (line 47) and, if the variable has not been visited before, it is pushed onto the stack and a new value is assigned to it. When S-SPLat finishes the execution of a test on one configuration (line 25), it has effectively covered a set of complete configurations, extensions of the corresponding partial configuration. SPLat determines the next configuration to execute by backtracking on the stack (lines 30–42). If the last read feature has value true, then S-SPLat has explored both values of that feature, and it is popped off the stack (lines 32–36). If the last read feature has value false, then SPLat has explored only the false value, and the feature value should be set to true (lines 35–41). This process repeats until the stack becomes empty and all dynamically reachable (partial) configurations are covered.

```
1  /* data structures */
2  class FeatureVar {...}
3  class Assign { Map<FeatureVar, Boolean> state; }
4  enum Mode = {RANDOM, ONE_ENABLED, ..., PAIRWISE};
5
6  /* state */
7  Map<FeatureVar, Boolean> state;
8  Stack<FeatureVar> stack;
9  static boolean checkfm = false;
10 static Mode heuristic = Mode.RANDOM;
11
12 void S-SPLat(Test t, FeatureModel fm) {
13  // Initializes features
14  state = new Map();
15  for (FeatureVar f: fm.getFeatureVariables())
16   state.put(f, fm.isMandatory(f));
17
18  // Instruments the code under test
19  instrumentOptionalFeatureAccesses();
20
21  do { // Repeatedly run the test
22   stack = new Stack();
23
24   // run test with configuration in state (line 7)
25   t.runInstrumentedTest();
26   Assign pa = getPartialAssignment(state, stack);
27   print("configs covered: " + fm.getValid(pa));
28
29   // Look for next configuration to run test at line 25
30   while (!stack.isEmpty()) {
31    FeatureVar f = stack.top();
32    if (state.get(f)) {
33     state.put(f, false);  // Restore
34     stack.pop();
35    } else {
36     state.put(f, true);
37     pa = getPartialAssignment(state, stack);
38     // check if satisfying vector is reachable from stack
39     if (heuristicIsSatisfied(pa, checkfm))
40      break; // success!
41    }
42   }
43  } while (!stack.isEmpty());
44 }
45
46 // Called-back from test execution
47 void notifyFeatureRead(FeatureVar f, FeatureModel fm) {
48  if (!stack.contains(f)) {
49   stack.push(f);
50   state.put(f, fm.isMandatory(f));
51  }
52 }
53
54 boolean heuristicIsSatisfied(Assign pa, boolean checkfm){
55  boolean res = false;
56  switch(heuristic){
57   case ONE_ENABLED:
58    res = checkOneEnabled(pa, checkfm); break;   ...
59  }
60  return res;
61 }
```

Fig. 3. S-SPLat algorithm (SPLat algorithm modified).

B. Sampling

The method heuristicIsSatisfied (lines 54–61), called at line 39, checks if the partial configuration pa, passed as parameter, satisfies the selected sampling heuristic. This method delegates to a "checking function" the decision of whether or not the configuration should be sampled. If a relevant configuration is found, execution proceeds to that configuration (line 25). If not, S-SPLat keeps searching for a configuration on the stack. If no configuration can be found, the stack becomes empty, and SPLat terminates (line 43).

Checking functions. Test requirements vary with the sampling heuristic. A (boolean) checking function answers positively

```
1  // Checks if pa covers a one-enabled test requirement
2  boolean checkOneEnabled(Assign pa, boolean checkFM){
3   boolean shouldSample = false;
4   int oneEnabledCounter = 0;
5   for(boolean b : pa.state.values())
6    oneEnabledCounter += b?1:0;
7   if(oneEnabledCounter == 1)
8    shouldSample = true;
9   if (!shouldSample) return false;
10  if (checkFM) return isSAT(pa);
11  return true;
12 }
```

Fig. 4. Example of a checking function for S-SPLat Algorithm

whenever an input configuration satisfies a test requirement that has not been yet covered. Figure 4 shows an example checking function, `checkOneEnabled`, for deciding if the input configuration should be sampled according to *one-enabled*. This function is called at line 58 from Figure 3. For this heuristic, the configuration will be considered for sampling only if the number of variables set in the partial input configuration (`pa`) is exactly one. If the parameter `checkFM` is set, this function will also check if the configuration is satisfiable (line 10) according to the input feature model. Note that this check is only performed after a test execution completes, producing a partial configuration (line 25); not during test execution. Other sampling heuristics are implemented similarly. We chose this function for brevity.

C. *Handling non-boolean variables*

Non-boolean variables are common in configurable systems. We describe in the following how `S-SPLat` handles variables with integer ranges, which are relatively common in GCC. One option to handle range types is to incorporate range values in the `S-SPLat` algorithm. With that, instead of making a binary choice where the algorithm assigns a (boolean) value to a variable, `S-SPLat` would make a multi-valued choice. We disconsidered that design choice as it would create too many very similar configurations by construction. Instead, we applied the Ostrand and Barcer's category partitioning method [43], treating range types as boolean types. More precisely, we partitioned the input domain in two categories, *enabled* and *disabled*, selecting one and only one element in the range to belong to the *disabled* category (0 if in the range).

V. Evaluation

We evaluated `S-SPLat` in two scenarios. In the first scenario we considered relatively small software product lines with the goal of identifying patterns of performance across different sampling heuristics. In the second scenario, we evaluated the techniques on a large configurable system (GCC [5]) manifesting different characteristics compared to SPLs. The goals of this experiment are to validate whether results obtained on GCC are consistent with those observed on SPLs, and to assess the ability of `S-SPLat`'s heuristics to find real faults. Furthermore, we evaluated the influence of feature constraints in our techniques.

We pose the following research questions:

- **RQ1.** Which heuristics maximize efficiency?
- **RQ2.** Which heuristics maximize efficacy?
- **RQ3.** Which heuristics (basic or combination) maximize efficiency and efficacy?

TABLE I. Software Product Lines used.

Subject	*#Tests*	*#Conf. variables*	*#Confs.*	*#Valid Confs.*	*LOC*
101Companies	38	10	1,024	192	2,059
DesktopSearcher	31	16	65,536	462	3,779
Email	4	8	256	40	1,233
GPL	24	13	8,192	73	1,713
JTopas	11	5	32	32	2,031
Notepad	43	17	131,072	256	2,074
Sudoku	5	6	64	20	853
ZipMe	32	13	8,192	24	3,650

A. *Software Product Lines (SPLs)*

Subjects. We selected eight SPLs previously used in other studies [13], [26], [32], [40]. All these SPLs provide tests and their feature variables are dynamically bound to code. Table I characterizes these subjects. Column "#Tests" and "#Features" show, respectively, number of tests and feature variables. Columns "#Confs." and "#Valid Confs." denote, respectively, the number of configurations and the number of configurations which are valid according to the feature model. Finally, column "LOC" shows code size.

Techniques. We considered six basic techniques: the baseline `SPLat` and the following heuristics for selecting configurations with `SPLat`: `random` (`ran`), `one-enabled` (`oe`), `one-disabled` (`od`), `most-enabled-disabled` (`med`), and `pairwise` (`pw`), as defined in Section II-B. We also considered combinations of these basic heuristics to answer RQ3. We set a global timeout of 48h on `SPLat` and `random` per subject as these techniques can potentially take too long to finish.

1) Answering RQ1 (Efficiency): We used the *average number of configurations explored per test* that a technique selects as metric for efficiency. Using wall clock time could bias results in favor of heuristics that select configurations exercising short paths. In addition, results are more vulnerable to measurement noise for small subjects and short-running tests. To note that previous studies used similar metrics for similar reasons [26], [38]. Figure 5 shows the distribution of results per technique as boxplots. A point in the boxplot indicates a measurement (i.e., the average number of configurations that a basic technique explores on a given test).

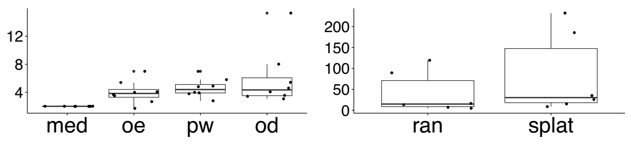


Fig. 5. Average number of samples per technique.

As expected the number of configurations that `SPLat` and `random` explore is much higher compared to alternative techniques. On average, `random` sampling (on `S-SPLat`) explores half of the configurations that `SPLat` explores, i.e., every configuration that `S-SPLat` reaches, when using random selection, has fifty percent chance of being selected. Intuitively, if `SPLat`'s exploration blows up, `S-SPLat` with `random` exploration should also blow up. We considered different rates of sampling as to assess its effects on results (see Section V-E).

`SPLat` and `random` selected sample sets with average sizes of 232 and 120, respectively. Recall that we configured `random` to explore, on average, half of the configurations that `SPLat` explores. Furthermore, it is important to note that the distributions of `SPLat` and `random` do not include results for *DesktopSearcher* and *Notepad*. These techniques timed out on these subjects because of the cost of exploring a high number of configurations on long-running (GUI/system) tests. This case sheds light to the inherent limitation of these techniques.

Considering all basic techniques but `SPLat` and `random`, the size of the sample sets (i.e., average number of configurations explored per test) ranged from 2 to 15.3. Overall, `most-enabled-disabled` explored the smallest sample sets whereas `one-disabled` explored the largest sample sets.

2) Answering RQ2 (Efficacy): In this experiment we used the *average number of failures per test* as metric to assess efficacy of each technique. We considered distinct failures (as opposed to faults) as proxy for efficacy because documented faults are scarce for these programs. We ignored repeated manifestations of failures, i.e. failures of a given type that have been already reported with a given pair of test and technique.

Figure 6 shows the distributions of number of distinct failures found per technique. We observed that `one-disabled` and `pairwise` found, on average, more failures than any other technique, except `SPLat`. Given the number of configurations that `random` selects, we found surprising that it performed poorly in several cases. This result indicates that the use of heuristics to explore the search space more uniformly is important. We also observed that `SPLat`'s distribution of results, ignoring outliers, is not very different compared with the distributions of other heuristics. Finally, considering the subject *Email*, we observed that all techniques revealed the same failures. That happened because the failure density ratio was high on this subject, indicating, perhaps as expected, that high failure density ratio favors more aggressive heuristics.

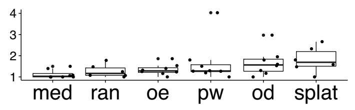


Fig. 6. Average number of failures per technique.

3) Answering RQ3 (Efficiency and Efficacy): The goal of this experiment is to better understand the relationship between efficiency and efficacy, which are conflicting optimization dimensions. A technique can optimize one dimension but perform poorly with respect to the other(s). `SPLat`, for example, optimizes efficacy but perform poorly, relative to others, with respect to efficiency. In addition to the basic heuristics discussed, we also considered combinations of heuristics in this experiment. The intuition is that unexpected combinations could provide a better overall balance between these dimensions. We implemented 11 combinations, referred with the following ids (symbol "+" indicates combination): `c1:oe+od`, `c2:oe+med`, `c3:oe+pw`, `c4:od+med`, `c5:od+pw`, `c6:med+pw`, `c7:oe+od+med`, `c8:oe+med+pw`, `c9:od+med+pw`, `c10:oe+od+pw`, `c11:oe+od+med+pw`. We obtained results for these combinations by taking, for each test, the union of the sample sets and failure sets generated by the corresponding basic heuristics. For example, let us consider technique `c6`, which combines `most-enabled-disabled` (`med`) with `pairwise` (`pw`). Suppose `med` selected two configurations on a given test, namely `11111?` and `00000?`, and found one failure whereas `pw` selected three configurations for the same test, namely `11111?`, `0110?1`, and `11??00`, and found two failures, with one of those also detected by `med`. For this case, `c6` produces four configurations and two failures; duplicates are ignored. Note that the set of combinations are partially ordered by inclusion (e.g., `c1` $\sqsubseteq$ `c10` $\sqsubseteq$ `c11`).

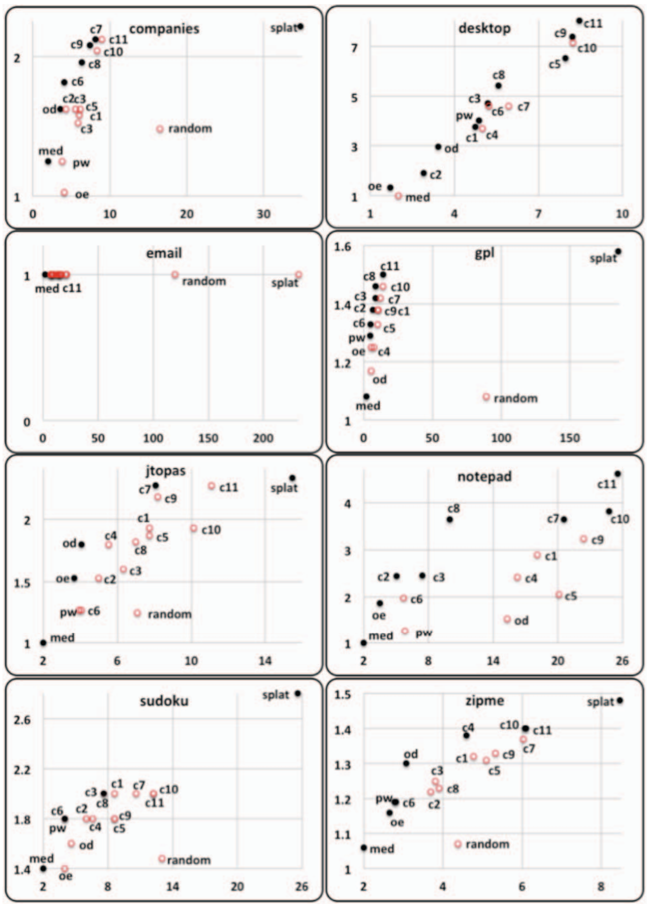


Fig. 7. Comparison of efficiency (x-axis) versus efficacy (y-axis). The Pareto front (shows non-dominated data points) is highlighted with filled circles.

Figure 7 shows results, with each plot focusing on one subject. The x-axis shows average number of configurations per test and the y-axis shows average number of failures detected. A point in the plot denotes the result of a heuristic (basic or combination) on a given subject. To facilitate visualization, the plots show the Pareto fronts [16] of measurements as solid circles. A point that belongs to the front is not dominated by any other point. The front helps one to make an informed decision on which technique should be selected provided she decides on the importance of one dimension over the other.

Note that, as a sound technique, `SPLat` appears in the front of all plots with the exception of *Desktop* and *Notepad* (due to timeouts). The plots also show that `SPLat` and `med` are, in almost all cases, at two opposing extremes in the optimization scale (see corner positions); these techniques optimize one dimension at the expense of the other dimension. Furthermore, we found that combination techniques appeared consistently at the front or close to it. Note that the fronts serve as reference of dominance, not to draw general conclusions about performance – the distance from a non-dominated point to dominated point (i.e., technique) can be minor in several cases. For example, combination `c7` performed consistently well in both dimensions but only appeared at the front in three cases.

Summary: Results indicate that the heuristics studied dramatically reduce the number of configurations explored by `SPLat` yet retaining their ability to reveal failures. In particular, the basic heuristic `one-disabled` and the combination heuristic `c11` performed consistently well with respect to efficiency and efficacy in all cases.

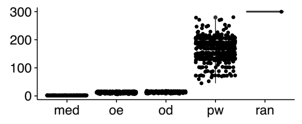
B. GCC

We also evaluated `S-SPLat` against the GNU Compiler Collection (GCC) [5], a large configurable system with hundreds of options [6]. The primary goal of this experiment is to assess how the heuristics perform on a large system with hundreds of input options. We considered three different setups in this experiment. In one setup we measured ability to find test failures in the GCC release 6.1 [7] (Section V-B1). Then, we evaluated how the basic heuristics perform to find unknown crashes, also using release 6.1 (Section V-B2). Finally, we used an older release of GCC, 4.8.2, to evaluate ability of the basic heuristics to find known crashes (Section V-B3). We focused on crashes that the authors found in a previous study [49].

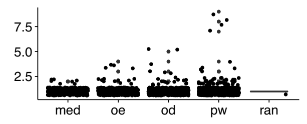
Tests analyzed. GCC uses DejaGnu [2] as testing framework. The test artifact typically includes some code fragment to be compiled, a number of compilation tasks, and required options. For example, compilation tasks include preprocessing, compiling, assembling, linking, and running code [20]. We analyzed a total of 3,557 tests from the "gcc-dg" test suite. We focused on that suite because previous study has shown a higher incidence of bugs found with it [49]. A GCC test runs by default against a single (default) configuration. `SPLat` runs the same test on multiple configurations, but respecting mandatory test options.

Options analyzed. We limited the number of options to analyze (see [6]) given the long execution times found in some test cases. We used the 50 most frequently cited options in the GCC bug reports from the period of April 27, 2016 to May 27, 2016. The rationale was that more bugs could be found close to where existing bugs have been recently reported. We noticed that the top options do not change much across months.

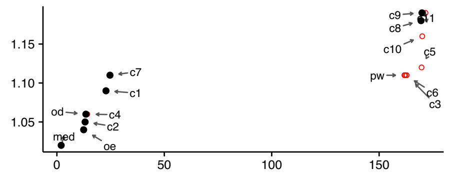
Techniques. This experiment considered all basic and combination heuristics of `SPLat`. We did not consider `SPLat` itself because of its high execution cost. For example, considering this setup, `SPLat` could finish execution of only 202 tests ($\sim 5.7\%$) after one week of execution. Recall that this inherent limitation of `SPLat` motivated this work. For `random`, which samples configurations reachable from `SPLat`'s execution, we used a bound of 300 configurations per test. This number is twice the average number of configurations explored by `pairwise`, which is the heuristic that explores more configurations amongst the basic heuristics we considered.



(a) Efficiency. Distributions of number of configurations each technique explores per test.



(b) Efficacy. Distributions of number of failures revealed by each technique per test.



(c) Efficiency (x-axis) versus Efficacy (y-axis).

Fig. 8. Failures in GCC version 6.1.

1) Finding failures in the GCC version 6.1: This experiment uses test failures to measure efficacy of the techniques. These failures are not necessarily fault-revealing; they may be manifestations of undocumented preconditions in DejaGnu tests (which are still important to report). We used the GCC release 6.1, available since April 2016, in this experiment. Figures 8(a) and 8(b) show, respectively, efficiency and efficacy results. Figure 8(c) shows the relationship between these two metrics.

Considering each dimension in separate, results indicate that the heuristic `pairwise` found more failures than any other technique in absolute numbers, however, it was also one of the most expensive techniques, only behind `random`. More importantly, the heuristics `one-enabled` and `one-disabled` found almost as many failures as `pairwise` but required much fewer configurations. This observation indicates that `pairwise` performs relatively worse in GCC compared to SPLs. One hypothesis for this is that the number of variables accessed in GCC was higher compared to SPLs. The intuition is that pairwise is more sensible to that factor than other techniques. Figure 9 shows a histogram of variables accessed per test in GCC. Indeed, note that most of the tests in GCC often access more variables than the total number of variables in SPLs.

Figure 8(c) relates efficiency and efficacy. The arrangement of two groups of techniques in opposing corners of the plot is noticeable. Observe that, although the range of the x-axis is wide, the range of the y-axis is relatively narrow, going from $\sim$1.0 to $\sim$1.2 failures detected per test. Conceptually, it means that it is preferable to pick the best performing heuristics in the leftmost group as cost savings are high and failure loss is low. Within that group, the combinations `c1` and `c7`, which builds on `one-enabled` and `one-disabled`, reported more failures per test. This observation is consistent with the results reported in Figure 8 and the results involving SPLs.

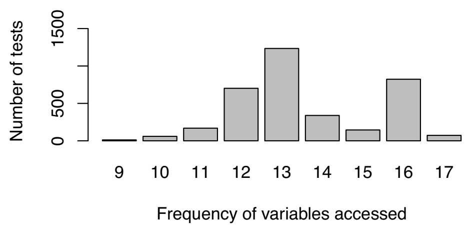


Fig. 9. Frequency of variables accessed per test. We obtained data by randomly sampling 300 configurations over all 3,557 tests. A total of 24 variables model the 50 options considered in this study.

Summary: Results show that `one-enabled` and `one-disabled` found the higher number of failures per test, exploring a relatively low number of configurations.

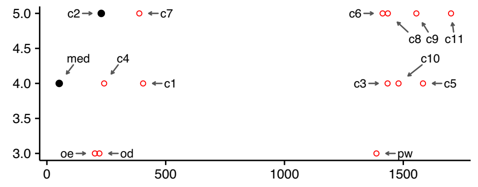
2) Finding crashes in the GCC version 6.1: In this experiment we looked for crashes in the test suites `gcc-dg` and `dg-torture`. Considering the suite `gcc-dg`, `most-enabled-disabled` did not find crashes, `one-enabled` found two crashing configurations on different tests (`pr44545.c` and `pr47086.c`), `one-disabled` found two crashes but on the same test (`pr47086.c`), and `pairwise` found multiple crashing configurations but on the same two tests that `one-enabled` found crashes. We filed one bug report[1] for all these cases, given the similarity of the configurations and the similarity of the crash reports. The GCC team confirmed and fixed the bug. Considering the suite `dg-torture`, only `one-enabled` found a crash in one configuration and on a single test (`pr48953.c`). We also filed a bug report[2] to for this case; the GCC team confirmed it was a bug that had been recently fixed and marked the report as duplicate.

3) Finding crashes in the GCC version 4.8.2: Next we evaluated the techniques on bugs found in a study, previously conducted by some of the authors of this paper, on release 4.8.2 of GCC [49]. The primary goal of this experiment is to evaluate the techniques in finding real known bugs[3] that `SPLat` was able to find when given a large time budget. In this setup, we used 29 tests that we knew a priori would reveal crashes in specific configurations. Overall, these crashes exposed five distinct bugs. On a crash, the GCC testing infrastructure reports an *"Internal Compiler Error (ICE)"* message followed by a specific error description which includes the statement that manifested the crash. We used these messages to identify the crash as to avoid counting multiple times the same one.

Considering *efficiency*, we observed that the heuristic `most-enabled-disabled` was the most efficient, followed by `one-enabled`, `one-disabled`, and `pairwise`. The same pattern was observed in the experiment with GCC release 6.1 (see Section V-B1). Considering *efficacy*, at least one technique detected each bug, with all five bugs detected. Furthermore, all four basic techniques found the first and second bugs, both `one-enabled` and `pairwise` found the third bug, only `most-enabled-disabled` found the fourth bug, and `most-enabled-disabled` and `one-disabled` found the fifth bug. Figure 10(a) summarizes these results. Overall, we observed that `most-enabled-disabled` found four of the five bugs, and each other basic technique found three bugs. We found surprising that `most-enabled-disabled` revealed more crashing while exploring few configurations per test. In particular, it was the only technique that found crash number four. It is also worth noting that `pairwise` explored many more configurations but could only reveal three crashes.

Basic	*Crash Id*				
Technique	1	2	3	4	5
`med`	✓	✓		✓	✓
`oe`	✓	✓	✓		
`od`	✓	✓			✓
`pw`	✓	✓	✓		

(a) Distribution of crashes found.



(b) Number of samples (x-axis) vs. number of bugs (y-axis).

Fig. 10. Crashes in GCC version 4.8.2.

Figure 10(b) shows results relating number of bugs found and number of samples required to find those bugs. Similar to the plot from Figure 8(c), techniques form clusters based on the number of configurations explored per test. In addition, the clusters are very similar across these two experiments. This plot shows that `most-enabled-disabled` and `c2` are not dominated by any other technique (filled circles indicate the Pareto front). We also observed that `c2` found all bugs with a relatively small number of samples.

Finally, analyzing configurations associated with each bug we observed that crashes were manifested with only few options enabled. This can explain why `one-enabled` and `most-enabled-disabled` performed well. These results confirm previous observations that configuration-related errors are often manifested in configurations involving a small number of input options [11], [19], [31], [39].

Summary: Results shows that all five bugs were captured by at least one basic heuristic and each heuristic captured at least three of the bugs. Considering combination techniques, the combination of `oe` and `med` found all bugs with a relatively small number of configurations.

C. Handling constraints

Feature constraints add a new dimension to this study. An error report manifested by a test on an invalid configuration is considered a false positive. For space limitations, we summarized here results we obtained validating constraints.

[1]https://gcc.gnu.org/bugzilla/show_bug.cgi?id=71512

[2]https://gcc.gnu.org/bugzilla/show_bug.cgi?id=77320

[3]All bugs were confirmed by the GCC team: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=<x>, where x=61980, 62069, 62070, 62140, 62141

Table I provides an indication of the complexity of the feature models of SPLs from the proportion of configurations which are valid, per subject. For those subjects, we observed that most configurations selected without constraint validation are invalid, reflecting the complexity of the models. For example, considering `one-disabled`, 54% of the selected configurations are invalid and 43% of failures are false positives. Despite these numbers, we observed that techniques perform consistently with and without feature models (results discussed in previous sections). Considering the crash scenarios of GCC, we observed that all crashes found manifested in valid configurations, indicating that the use of validation is not beneficial. (We used GCC constraints documented in a previous study we conducted [49].) Considering the scenarios of failures of GCC, we also observed that the techniques performed consistently with and without feature constraints.

D. Threats to Validity and Limitations

The main threats to validity are as follows. *External Validity*: The selection of subjects we used may not generalize to other cases. To mitigate this threat, we used subjects from a variety of sources, including a large configurable system with hundreds of options. *Internal Validity*: Eventual errors in our implementation could invalidate results. To mitigate this threat, we thoroughly checked our implementation and our experimental results, looking for discrepancies that would signal potential errors. Our datasets and implementations are publicly available [48]. One *limitation* of the study relates to the fact that `SPLat` currently only supports systems with dynamically bound feature variables (e.g., Groupon web [8], [29], and GCC [5]). It remains to investigate how `SPLat` and `S-SPLat` would perform on systems with `#ifdef` variability.

E. Discussion

While combination `c2` was the only technique that found all crashes with a relatively low number of configurations, it seems that the combination `c7` performs even better considering all the experiments we analyzed, including those with SPLs. This combination detected most failures and crashes through a relatively small number of configurations. Recall that this is the combination that includes all basic techniques but `pairwise` and `random`. This confirms results obtained by Medeiros et al. [38] showing the superior performance of this combination. Overall, the results we obtained suggest that it is preferable to combine different simple heuristics instead of using one that entails a larger number of test requirements. We remain to explore alternative heuristics not considered in this study. However, our initial results, indicate that the hybrid search proposed by `S-SPLat` is promising to reveal errors in potentially large configuration spaces.

In addition to the experiment documented above, we run two other experiments, whose results we summarize below. We (1) compared `S-SPLat` to `Regular Sampling` and we (2) evaluated `S-SPLat` with `random` sampling using different sampling rates (10% and 30% in addition to the default rate of 50%). For the first experiment, `Regular Sampling` detected the same bugs as `S-SPLat` but it required more configurations. Recall that `Regular Sampling` techniques are black-box, they explore precomputed configurations regardless if they are reachable by test executions or not whereas `S-SPLat` only explores configurations that both satisfy sampling heuristics and are reachable from tests. In the second experiment, we found that the results obtained are proportional to the change in the sampling rates of `random`. Additional results can be found on our website [48].

VI. Related Work

Sound Techniques. Sound techniques can be divided in two categories according to their execution mode. Multi-execution approaches, such as `DeltaExecution` [18], `SharedExecution` [28] and `Varex` [41], execute a given test simultaneously against sets of configurations; they leverage the similarities that exist across configurations to reduce the total number of paths explored in a test and the overall amount of computation. One multi-execution is typically more expensive than a regular execution but the overall execution compensates provided that enough similarity exist across each regular execution. Single-execution approaches, such as `SPLat` [29], [49], in contrast, execute a test once for each reachable configuration that they discover while building a decision tree from configuration variables accessed during execution. Several aspects need to be taken into consideration for choosing one approach or the other. For example, a tension between engineering-related limitations and computational cost exists between these approaches. On the one hand, building (and maintaining) such interpreters is challenging, especially for statically-typed languages [18], [28], [45]. Implementations for dynamically-typed languages exist but still suffer from important limitations (e.g. [41, § 3.2.4]). On the other hand, single-execution approaches can be expensive as they are unable to detect potential redundancies in computations across configurations. This paper focused on the single-execution technique `SPLat` but, in principle, sampling could also be combined with multi-execution approaches.

Sampling. Abal et al. [11] analyze the Linux kernel software repository to study configuration-related faults fixed by developers. They manually analyze the characteristics of 42 configuration-related faults and suggest using the `one-disabled` sampling algorithm to find bugs. Perrouin et al. [44] and Marijan et al. [37] use the *t-wise* sampling algorithm to cover all t configuration option combinations. Tartler et al. [50] propose the statement-coverage sampling algorithm and applied a per-file analysis to detect bugs in the Linux kernel. However, none of them perform a comparative study to understand the fault-detection capability and effort (size of sample set) of sampling algorithms different from our work using `S-SPLat`. Liebig et al. [33] perform studies to detect the strengths and weaknesses of variability-aware and sampling-based analyses (single configuration, pair-wise and code-coverage). They considered two analysis implementations (type checking and liveness analysis) and applied them to a number of subject systems, such as BusyBox and the Linux kernel. The variability-aware analysis outperforms most sampling heuristics with respect to analysis time while preserving completeness. We do not evaluate runtime of our algorithms. Song et al. [46] propose interaction tree discovery algorithm (iTree) to support the testing of highly configurable systems. iTree selects a subset of configurations in which the execution of the system's test suite will achieve high coverage. They found that iTree can identify high-coverage sets of configurations more effectively than traditional combinatorial interaction testing or random sampling. Apel et al. [14] have developed a model-checking tool for C and Java product lines. They compare sample-based and family-based strategies with regard to verification performance and the ability to find defects. They found that `triple-wise` outperformed `pairwise` sampling and that the family-based strategy outperformed all sampling-based strategies in terms of detection efficiency. Later, Medeiros et al. [38] conducted an extensive comparative study of 10 sampling algorithms (5 variations of *t-wise*, `statement-coverage`, `random`, `one-disabled`, `one-enabled`, and `most-enabled-disabled`) regarding their fault-detection ability and size of sample sets in the Linux kernel, Apache, and other real C program families. They also analyzed combinations of these algorithms. They found that, in most cases, `most-enabled-disabled` is the most efficient. In our work, we observed that, considering SPLs, `one-disabled` and the combination heuristic `c11` offered the best balance between efficiency and effectiveness in most cases. Considering GCC, `most-enabled-disabled` and the combinations of `one-enabled` and `one-disabled` (e.g., `c7`) offered the best balance.

Static Analysis. Kim et al. [26] previously developed a static analysis to determine which features are relevant to the outcome of a test. Conceptually, it enables one to run a test only on (all valid) combinations that involve relevant configuration variables. Despite the positive results reported by Kim et al. (caveat: evaluation involved large subjects but tests cover only a small fraction of the code), it is important to note that obtaining reachability information for these systems per test is challenging as: (i) the analysis needs to (re)run for each test, (ii) the analysis needs to run whenever the program changes, and (iii) often tests are designed to statically reach the entire codebase (e.g., system tests) – i.e., only the test input data can discriminate which parts of the code will be actually executed [47]. It is important to note that obtaining dynamic reachability information efficiently is the key feature of `SPLat` [29], which built upon the ideas of Kim et al. [26]. Note also that the use of static or dynamic reachability analysis is an orthogonal aspect of this work. In the future, we plan to empirically evaluate other methods to compute reachability information.

Other. Recent empirical study found evidence that practical configuration complexity is often much lower compared to theoretical configuration complexity [40]. The intuition is that, even if multiple variables are accessed in a given test, configuration complexity may not be unacceptably high. For example, variables may not interact with each other or may interact according to specific patterns. The observation of this phenomenon enables techniques that leverage the similarities across executions of a test to reduce dimensionality of the search space and test execution to scale. The contribution of variability-aware execution is orthogonal to ours: it is possible to combine sampling even with variability-aware execution (sound by definition) as to explore more exhaustively only certain parts of the decision tree. Hybrid concolic execution [21], [36] is a variant of concolic execution that aims to explore the state space more broadly and deeply compared to a regular concolic execution, which conceptually can get stuck (e.g., as observed with saturation in coverage) in dense branches of the symbolic tree. The principle of seeking more uniform search has various manifestations in software testing; it is motivated by the assumption that bugs are uniformly distributed in the state space. Note that the goal of (hybrid) concolic execution is test input data generation whereas our goal is configuration selection. Furthermore, the heuristic component of their hybrid search only uses randomization (as to make jumps during symbolic execution) whereas we evaluated different sampling heuristics. Despite these difference both techniques approach the problem of searching a large search space by dropping guarantees of a systematic search for the sake of practicality.

VII. Conclusion

We presented `S-SPLat`, a technique for testing configurable systems that blends heuristic sampling and symbolic search to obtain both breadth and depth in the exploration of the configuration space. `S-SPLat` builds on `SPLat`, our previously developed technique, that explores all reachable configurations from tests. Results obtained on several subjects are encouraging. For example, `S-SPLat` could find all the bugs that `SPLat` previously found on GCC, release 4.8.2, but faster. Furthermore, it found new bugs in a newer release of GCC. Implementation and experimental infrastructure can be found on our website [48].

Acknowledgment

We would like to thank the anonymous reviewers and Leopoldo Teixeira who provided insightful comments on prior versions of this paper. This material is based upon work partially supported by the CNPq and CAPES Brazilian research agencies, and by a Microsoft SEIF'13 award.

References

[1] Configuration error brings down the Azure cloud platform. http://www.evolven.com/blog/configuration-error-brings-down-the-azure-cloud-platform.html, Accessed 18-Jan-2017.

[2] DejaGnu. http://www.gnu.org/software/dejagnu/, Accessed 18-Jan-2017.

[3] DNS misconfiguration. http://www.circleid.com/posts/misconfiguration_brings_down_entire_se_domain_in_sweden, Accessed 18-Jan-2017.

[4] Firefox web browser. http://hg.mozilla.org, Accessed 18-Jan-2017.

[5] GCC compiler infrastructure. http://gcc.gnu.org, Accessed 18-Jan-2017.

[6] GCC Options. https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html, Accessed 18-Jan-2017.

[7] GCC releases. https://gcc.gnu.org/releases.html, Accessed 18-Jan-2017.

[8] Groupon. http://groupon.com, Accessed 18-Jan-2017.

[9] Linux kernel. http://www.kernel.org, Accessed 18-Jan-2017.

[10] Soundiness webpage. http://soundiness.org/, Accessed 18-Jan-2017.

[11] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 42 variability bugs in the Linux kernel: A qualitative analysis. In *Proceedings of the Automated Software Engineering*, pages 421–432. ACM, 2014.

[12] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer-Verlag, 2013.

[13] Sven Apel and Dirk Beyer. Feature cohesion in software product lines: An exploratory study. In *Proceedings of the International Conference on Software Engineering*, pages 421–430. ACM, 2011.

[14] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Groblinger, and Dirk Beyer. Strategies for product-line verification: case studies and experiments. In *Proceedings of the International Conference on Software Engineering*, pages 482–491. ACM, 2013.

[15] Don Batory, Clay Johnson, Bob MacDonald, and Dale von Heeder. Achieving extensibility through product-lines and domain-specific languages: A case study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):191–214, 2002.

[16] Abu Syed Md Masud Ching-Lai Hwang. *Multiple Objective Decision Making – Methods and Applications: a state-of-the-art survey*, volume 164. Springer-Verlag, 1979.

[17] Myra B. Cohen, Peter B. Gibbons, Warwick B. Mugridge, and Charles J. Colbourn. Constructing test suites for interaction testing. In *Proceedings of the International Conference on Software Engineering*, pages 38–48. IEEE, 2003.

[18] Marcelo d'Amorim, Steven Lauterburg, and Darko Marinov. Delta execution for efficient state-space exploration of object-oriented programs. *IEEE Transactions on Software Engineering*, 34(5):597–613, 2008.

[19] Brady J. Garvin and Myra B. Cohen. Feature interaction faults revisited: An exploratory study. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 90–99. IEEE, 2011.

[20] GCC. Preparing Testcases. http://gcc.gnu.org/wiki/HowToPrepareATestcase, Accessed 18-Jan-2017.

[21] GrammaTech. Hybrid concolic execution - part 1. http://blogs.grammatech.com/hybrid-concolic-execution-part-1, Accessed 18-Jan-2017.

[22] Michaela Greiler, Arie van Deursen, and Margaret-Anne Storey. Test confessions: A study of testing practices for plug-in systems. In *Proceedings of the International Conference on Software Engineering*, pages 244–254. IEEE, 2012.

[23] Robert Johnson. More details on today's outage. https://www.facebook.com/notes/facebook-engineering/more-details-on-todaysoutage/431441338919, Accessed 18-Jan-2017.

[24] Kyo Kang, Sholom Cohen, James Hess, William Nowak, and Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, United States, 1990.

[25] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the Object-Oriented Programing, Systems, Languages, and Applications*, pages 805–824. ACM, 2011.

[26] Chang Hwan Peter Kim, Don S. Batory, and Sarfraz Khurshid. Reducing combinatorics in testing product lines. In *Proceedings of the Aspect-Oriented Software Development*, pages 57–68. ACM, 2011.

[27] Chang Hwan Peter Kim, Eric Bodden, Don S. Batory, and Sarfraz Khurshid. Reducing configurations to monitor in a software product line. In *Proceedings of the Runtime Verification*, pages 285–299. Springer-Verlag, 2010.

[28] Chang Hwan Peter Kim, Sarfraz Khurshid, and Don Batory. Shared execution for efficiently testing product lines. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 221–230. IEEE, 2012.

[29] Chang Hwan Peter Kim, Darko Marinov, Sarfraz Khurshid, Don Batory, Sabrina Souto, Paulo Barros, and Marcelo d'Amorim. SPLat: Lightweight dynamic analysis for reducing combinatorics in testing configurable systems. In *Proceedings of the Foundations of Software Engineering*, pages 257–267. ACM, 2013.

[30] D. Richard Kuhn, Raghu N. Kacker, and Yu Lei. Practical combinatorial testing. Technical Report SP 800-142, National Institute of Standards & Technology, Gaithersburg, MD, United States, 2010.

[31] D. Richard Kuhn, Dolores R. Wallace, and Albert M. Gallo Jr. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 30(6):418–421, 2004.

[32] Philipp Lengauer, Verena Bitto, Florian Angerer, Paul Grünbacher, and Hanspeter Mössenböck. Where has all my memory gone?: Determining memory characteristics of product variants using virtual-machine-level monitoring. In *Proceedings of the Variability Modelling of Software-Intensive Systems*, pages 13:1–13:8. ACM, 2013.

[33] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. Scalable analysis of variable software. In *Proceedings of the Foundations of Software Engineering*, pages 81–91. ACM, 2013.

[34] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Communications of the ACM*, 58(2):44–46, 2015.

[35] Ivan do Carmo Machado, John D. Mcgregor, Yguaratã Cerqueira Cavalcanti, and Eduardo Santana de Almeida. On strategies for testing software product lines: A systematic literature review. *Information and Software Technology*, 56(10):1183–1199, 2014.

[36] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *Proceedings of the International Conference on Software Engineering*, pages 416–426. IEEE, 2007.

[37] Dusica Marijan, Arnaud Gotlieb, Sagar Sen, and Aymeric Hervieu. Practical pairwise testing for software product lines. In *Proceedings of the Software Product Line Conference*, pages 227–235. ACM, 2013.

[38] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. A comparison of 10 sampling algorithms for configurable systems. In *Proceedings of the International Conference on Software Engineering*, pages 643–654. ACM, 2016.

[39] Flávio Medeiros, Márcio Ribeiro, and Rohit Gheyi. Investigating preprocessor-based syntax errors. In *Proceedings of the Generative Programming: Concepts and Experiences*, pages 75–84. ACM, 2013.

[40] Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. On essential configuration complexity: Measuring interactions in highly-configurable systems. In *Proceedings of the Automated Software Engineering*, pages 483–494. ACM, 2016.

[41] Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. Exploring variability-aware execution for testing plugin-based web applications. In *Proceedings of the International Conference on Software Engineering*, pages 907–918. ACM, 2014.

[42] Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Computing Surveys*, 43(2):11:1–11:29, 2011.

[43] Thomas J. Ostrand and Marc J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of ACM*, 31(6):676–686, 1988.

[44] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves le Traon. Automated and scalable t-wise test case generation strategies for software product lines. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, pages 459–468. IEEE, 2010.

[45] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. MultiSE: Multi-path symbolic execution using value summaries. In *Proceedings of the Foundations of Software Engineering*, pages 842–853. ACM, 2015.

[46] Charles Song, Adam Porter, and Jeffrey S. Foster. iTree: Efficiently discovering high-coverage configurations using interaction trees. In *Proceedings of the International Conference on Software Engineering*, pages 903–913. IEEE, 2012.

[47] Sabrina Souto and Marcelo d'Amorim. Time-Space Efficient Regression Testing for Configurable Systems, February 2017. https://doi.org/10.5281/zenodo.291369, Accessed 13-Feb-2017.

[48] Sabrina Souto, Marcelo d'Amorim, and Rohit Gheyi. S-SPLat: Balancing soundness and efficiency for practical testing of configurable systems (Artifact). https://sabrinadfs.github.io/s-splat/, Accessed 18-Jan-2017.

[49] Sabrina Souto, Divya Gopinath, Marcelo d'Amorim, Darko Marinov, Sarfraz Khurshid, and Don Batory. Faster bug detection for software product lines with incomplete feature models. In *Proceedings of the Software Product Line Conference*, pages 151–160. ACM, 2015.

[50] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. Configuration coverage in the analysis of large-scale system software. In *Proceedings of the Programming Languages and Operating Systems*, pages 2:1–2:5. ACM, 2011.

[51] Lincoln University of Nebraska. Combinatorial interaction testing (CIT) portal. http://cse.unl.edu/~citportal/, Accessed 18-Jan-2017.

[52] Alan W. Williams and Robert L. Probert. A practical strategy for testing pair-wise coverage of network interfaces. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 246–254. IEEE, 1996.