iTree: Efficiently Discovering High-Coverage Configurations Using Interaction Trees

Charles Song, Adam Porter, *Senior Member, IEEE,* and Jeffrey S. Foster

Abstract—Modern software systems are increasingly configurable. While this has many benefits, it also makes some software engineering tasks, such as software testing, much harder. This is because, in theory, unique errors could be hiding in any configuration, and, therefore, every configuration may need to undergo expensive testing. As this is generally infeasible, developers need cost-effective techniques for selecting which specific configurations they will test. One popular selection approach is combinatorial interaction testing (CIT), where the developer selects a strength t and then computes a covering array (a set of configurations) in which all t-way combinations of configuration option settings appear at least once. In prior work, we demonstrated several limitations of the CIT approach. In particular, we found that a given system's *effective configuration space*—the minimal set of configurations needed to achieve a specific goal— could comprise only a tiny subset of the system's full configuration space. We also found that effective configuration space may not be well approximated by t-way covering arrays. Based on these insights we have developed an algorithm called *interaction tree discovery (iTree)*. iTree is an iterative learning algorithm that efficiently searches for a small set of configurations that closely approximates a system's effective configuration space. On each iteration iTree tests the system on a small sample of carefully chosen configurations, monitors the system's behaviors, and then applies machine learning techniques to discover which combinations of option settings are potentially responsible for any newly observed behaviors. This information is used in the next iteration to pick a new sample of configurations that are likely to reveal further new behaviors. In prior work, we presented an initial version of iTree and performed an initial evaluation with promising results.

This article presents an improved iTree algorithm in greater detail. The key improvements are based on our use of *composite proto-interactions* – a construct that improves iTree's ability to correctly learn key configuration option combinations, which in turn significantly improves iTree's running time, without sacrificing effectiveness. Finally, the article presents a detailed evaluation of the improved iTree algorithm by comparing the coverage it achieves versus that of covering arrays and randomly generated configuration sets, including a significantly expanded scalability evaluation with the ∼1M-LOC MySQL. Our results strongly suggest that the improved iTree algorithm is highly scalable and can identify a high-coverage test set of configurations more effectively than existing methods.

Index Terms—Empirical Software Engineering, Software Configurations, Software Testing and Analysis

1 Introduction

As software systems continue to grow in size and complexity, they are increasingly designed to be configurable. This is desirable because it enables systems to be more portable, reusable, and extensible. At the same time, however, configurability can greatly complicate software development tasks, such as testing, because each configuration can contain unique faults, and therefore, each configuration may need to undergo expensive testing—something that is generally infeasible in practice.

To address this problem, researchers have proposed a variety of *combinatorial interaction testing* (CIT) techniques [7], [2], [24]. In the context of configurable systems, developers often apply CIT by manually modeling the system's *full configuration space*—all the ways in which it can be configured—and then by using the resulting model and model coverage criteria to guide the identification of a small test set of configurations under which to test.

For example, with one popular CIT approach, developers choose an *interaction strength*, t, and use it to compute a *covering array*, which is a set of configurations in which all possible t-tuples of option settings appear at least once. Research and experience indicate that covering arrays are relatively small in size and provide good coverage of a system's behavior [21], [11], [19], even though the cost to compute them grows quickly with t.[1] Nevertheless, compared to other alternatives, CIT approaches are widely believed to be a cost-effective way to find faults in configurable systems.

Our work, however, challenges this belief. Instead, we believe that, in practice, a system's *effective configuration space*—the minimal set of configurations needed to achieve a specific testing goal, given a specific test

- *C. Song is with Fraunhofer USA Center for Experimental Software Engineering, College Park, MD, 20740*
  E-mail: csong@fc-md.umd.edu
- *A. Porter and J. S. Foster are with Computer Science Department, University of Maryland, College Park, MD, 20742*
  E-mail: {aporter,jfoster}@cs.umd.edu

1. For instance, the $5-$way covering arrays we generated for MySQL in Section 5.2, took several days of computing time.

suite—typically comprises only a tiny subset of the full configuration space [31]. We also believe that t-way covering arrays poorly approximate a system's effective configuration space. Specifically, covering arrays contain many configurations that do not improve the covering array's ability to meet a particular testing goal, while at the same time lacking other configurations that would.

To test these conjectures, we used symbolic execution [18], [15], [6] to discover subject systems' *interactions*, where an interaction is defined as a minimal conjunction of option settings guaranteed to achieve a specific testing goal for a particular test suite. In our case, the testing goals were particular forms of program coverage (i.e., line, basic block, edge, and condition). Our results strongly supported our hypotheses [31].

In particular, we found that for our subject systems and their test suites, there indeed exist much smaller and more effective test sets than those created by current CIT methods. However, the method we used to study this issue, symbolic execution, is extremely expensive and cannot scale to large systems. Specifically, while symbolic execution gave us precise information about every possible configuration of two, roughly 10K LOC subject systems, the complete analysis required using 40+ computers working round the clock for several days. Thus, our findings, while revealing, could not be put to practical use. In addition, symbolic execution techniques are currently limited to systems written in specific programming languages, and do not handle compile-time configuration options.

Therefore, we developed a new algorithm, called *interaction tree discovery* (iTree), that quickly, cheaply, and effectively approximates a system's effective configuration space. iTree does this by following an iterative process of instrumenting the system under test, executing a small number of low-strength covering arrays, and applying machine learning (ML) techniques to incrementally learn the interactions that define a system's effective configuration space. We presented an initial version of this algorithm and an initial evaluation of it in Song. et al. [35].

While iTree's performance was quite promising, our evaluation identified key weaknesses that compromised its accuracy and scalability. This article presents an improved iTree algorithm that successfully addresses some of those limitations. This article also presents a new empirical evaluation of the improved algorithm, comparing it both to the previous iTree version and to other alternative approaches, including random selection and CIT. Finally, this paper presents a new evaluation of the improved algorithm's scalability to very large software systems.

This article and its major finding are organized and presented as follows. First, we present the improved iTree algorithm. The improvements stem from our introduction of *composite proto-interactions*. This concept allows iTree to merge and process the information it learns in a way that significantly improves the algorithm's running time. Specifically, compared to the old algorithm, the new one reduced testing effort by roughly a factor of 2 to 3 times in our experiments.

Next, we present a new empirical evaluation comparing the improved iTree algorithm to other existing configuration selection methods as applied to two medium-sized software systems. This evaluation showed that the improved algorithm was as effective as random selection and CIT, while requiring only 25–37% of the testing effort.

Finally, we present a new and expanded scalability evaluation, applying the improved iTree algorithm to the ∼1M-LOC MySQL database system, for which symbolic execution is infeasible. Compared to our previous study of the old algorithm, this new study increases the size of the configuration space considered by a factor of more than 10^6. This evaluation demonstrated that the improved iTree algorithm easily scaled up to this large MySQL configuration space, and that was again more efficient and effective than either CIT or random sampling.

Overall, these results suggest that the improved iTree algorithm is an important advance in the testing of highly configurable systems. We believe that it will enable developers to test their systems to higher levels of coverage at much lower cost than is currently possible with traditional CIT. We also believe the improved iTree algorithm is a foundational, practical technique for discovering a system's configuration-related structures, and can help support a wide range of software engineering tasks such as impact analysis, reverse engineering, bug isolation, and more.

2 Effective Configuration Spaces

As we mentioned earlier, the fundamental hypothesis underlying our work is that, in practice, a system's effective configuration space will typically be much smaller than its full configuration space. To illustrate why this might be true, consider a hypothetical program with four binary-valued configuration options: a, b, c, and d. Assume that all 16 possible configurations of these options are valid. Assume also that for the system's test suite and with the testing goal of 100% line coverage, this program has exactly three interactions: $a \wedge b \wedge \neg c$, $a \wedge b$, and $a \wedge d$. By definition, then, one or more lines of code will be covered whenever the test suite is run in a configuration that satisfies the following constraints: $a \wedge b \wedge \neg c$; $a \wedge b$, or $a \wedge d$. In addition, the union of the lines guaranteed to be covered by any of the three interactions is the maximal coverage achievable across all possible configurations.

In this example all three interactions can be satisfied by a single concrete configuration, namely $a \wedge b \wedge \neg c \wedge d$. Thus, for this system, coverage goal, and test suite, the effective configuration space contains only one

configuration, while the full configuration space has 16. Moreover, since at least one of the system's interactions involves three options, covering arrays of strength 2 or less would not be guaranteed to achieve maximal coverage, while covering arrays of strength 3 or higher will contain configurations that add nothing to overall coverage.

Our previous work [31] empirically studied whether the hypothetical situation just described actually occurs in two configurable systems, vsftpd and ngIRCd. The results of that study strongly suggest that, for a given test suite, maximal levels of coverage could be achieved with a very small number of carefully chosen configurations, i.e., the effective configuration spaces of these systems with respect to various coverage criteria were indeed small. For developers to exploit this insight, however, they need cost-effective and time-sensitive techniques for choosing the specific configurations to test. Unfortunately, we currently know of no such techniques; the approaches we used in our previous work are computationally very expensive, and symbolic execution cannot be run to exhaustion on large, practical systems.

Thus, our goal is to begin creating a practical method to identify or approximate a system's effective configuration space [35]. Our particular focus is on finding small sets of configurations for testing that yield a high degree of coverage, but we believe our approach generalizes to other software engineering tasks.

Toward this aim, we reexamined the specific interactions we found in our initial study, and made a number of observations that suggest the design of such a method. We illustrate our observations using the code in Figure 1, which contains a highly simplified snippet of vsftpd's source code. The code includes three traditional program variables, dsa_cert_file, one_process_mode, and vsftp_data_bufsize, which are initialized on lines 1, 2, and 3. In practice, these variables are program inputs whose values would come from a test case, but we have hard-coded them here for simplicity. The program also contains six binary configuration options and one integer configuration option, highlighted in bold, whose values depend on the system's runtime configuration. In this example, the values of the integer configuration option, trans_chunk_size, are limited to 0, 2048, 4096, and 65536. In the actual source code, vsftpd's configuration options are set in a configuration file called tunable.c, and each option's name is prefixed with tunable_, but we have omitted the prefixes to save space.

Figure 1 includes eight regions of code, marked */* R1–R8 */*, whose coverage we are interested in. The coverage of these regions depends on the values of the configuration options and program variables. For each region, we list the interaction that controls coverage of that region for this particular test case. For example, at the beginning of the program, the coverage of *R1–R3* depends on configuration variables **listen** and **accept_timeout**.

```
1  int* dsa_cert_file=NULL; /* test input */
2  int one_process_mode=1; /* test input */
3  int vsftp_data_buffsize=65536; /* test input */
4  if (listen} {
5      if (accept_timeout) {
6          /* R1: listen ∧ accept_timeout */
7      } else {
8          /* R2: listen ∧ ¬accept_timeout */
9      }
10 } else {
11     /* R3: ¬listen */
12 }
13 if (ssl_enable) {
14     if (!dsa_cert_file)
15         die();
16 }
17 /* R4: ¬ssl_enable */
18 if (one_process_mode) {
19     if (local_enable || ssl_enable)
20         die();
21 }
22 /* R5: ¬ssl_enable ∧ ¬local_enable */
23 if (!local_enable && !anonymous_enable)
24     die();
25 /* R6 (lots of code) : ¬ssl_enable∧
26     ¬local_enable ∧ anonymous_enable */
27 if (trans_chunk_size < vsftp_data_bufszie &&
28       trans_chunk_size > 0) {
29     if (dual_log_enable) {
30       /* R7: ¬ssl_enable ∧ ¬local_enable∧
31           anonymous_enable∧
32           trans_chunk_size = 2048 | 4096∧
33           dual_log_enable */
34     } else {
35       /* R8: ¬ssl_enable ∧ ¬local_enable∧
36           anonymous_enable∧
37           trans_chunk_size = 2048 | 4096∧
38           ¬dual_log_enable */
39     }
40 }
```

Fig. 1. An example program and its interactions.

More interestingly, for execution to reach the large amount of code in *R6*, several options must be set in specific ways. First, to reach *R4* and any code thereafter, **ssl_enable** must be set to 0, because this test case sets **dsa_cert_file** to be NULL. Next, consider reaching *R5*. Since **one_process_mode** is set to 1, to reach *R5* the condition on line 19 must be false; and since as just discussed **ssl_enable** is 0 if we reach this line, **local_enable** must also be 0.

To continue on to reach *R6*, we need the condition on line 23 to be false, and since **local_enable** $= 0$ if we reach that line, we must have **anonymous_enable** $= 1$. Putting this together, any configuration that reaches *R6* for this test case needs at least **ssl_enable** $= 0$, **local_enable** $= 0$, and anonymous_enable $= 1$.

To continue on to reach the rest of the lines, the value of **trans_chunk_size** can be either 2048 or 4096 to make the condition on line 27 true. Finally, the

coverage of *R7* and *R8* also depends on the value of **dual_log_enable**.

Note that although in this example we were able to reach all code regions, and coverage of each was guaranteed by a distinct interaction, in practice this is not usually the case. In actual systems some regions are unreachable with the given test suite, and some regions have more than one interaction that guarantees their coverage.

We found that the configuration option patterns just described are common in vsftpd and ngIRCd. From these patterns, we make three observations:

Observation 1: *Interactions are relatively rare.* Only a handful of specific options setting combinations had to be exercised to maximize coverage, even under a comprehensive criterion, such as path coverage. This suggests that CIT's insistence on testing every t-way combination of option settings may be unnecessarily expensive.

The code shown in Figure 1 illustrates this observation. It includes six binary options, so in the worst case there could be 639 different interactions[2]. In the example code, however, there are only eight interactions. Since some of these interactions can be simultaneously satisfied in a single configuration, only three configurations are needed to cover all eight regions. For vsftpd and ngIRCd respectively, there were only 43 and 435 interactions.

Observation 2: *Most coverage was achieved by lower-strength interactions, but higher-strength interactions are needed for maximum coverage.*

For the systems and test suites we examined, over 94% of the achievable coverage could be achieved with four or fewer interactions. Full coverage, however, required a handful of higher-strength interactions (up to strength seven).

This suggests that while using CIT at low strength can yield significant coverage, it will often fail to achieve the maximum possible coverage. This finding has great practical significance, because CIT is most commonly applied at strength 2 (i.e., pairwise testing).

In the example, five of the eight interactions involve only one or two option settings, one interaction involves three settings, and the remaining two involve four settings each. While the example is highly simplified, we found the same trend in the actual systems.

Observation 3: *Higher-strength interactions tend to be built on lower-strength ones.*

We observed that, in implementation terms, interactions generally arose because control-flow guards effectively stacked up on each other, i.e., higher-strength interactions add additional constraints to existing lower-strength ones. For example, the higher strength interactions guaranteeing coverage of *R7* and *R8* are refinements of the interaction at *R6*, which is itself a refinement of *R5*. This again suggests that CIT's systematic approach to include every t-way combination of option settings without discrimination, especially when t gets large, maybe tremendously wasteful.

2. Computed as $1 + \sum_{i=1}^{6} C(6,i) \cdot 2^i$, i.e., the sum of all ways of picking option subsets times the number of settings, plus the interaction *true*.

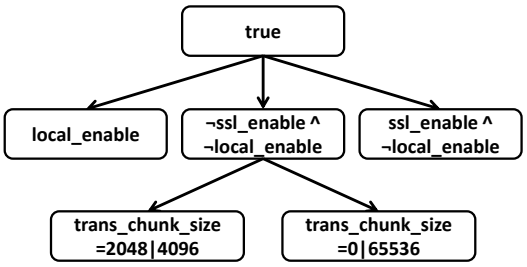


Fig. 2. The interaction tree for the example program.

3 Interaction Tree Discovery

Based on the insights discussed in Section 2, we developed the interaction tree discovery algorithm (iTree). iTree's goal is to automatically discover and then test a small set of high-coverage configurations. iTree works as follows. First, it instruments the system under test to measure some desired type of coverage. This paper focuses on line coverage, but the algorithm should apply to any type of coverage. The only requirement is that coverage of a configuration can be expressed as a mapping between a program entity and a boolean indicating whether it has been covered.

Next, iTree repeats the following steps until particular stopping criteria are met. First, it computes a small sample of configurations under which to test the system. As we shall see later, the sample is chosen to select configurations that are likely to exercise previously uncovered entities. Next, iTree runs the system's test suite on each of the sampled configurations and records coverage information. Using this coverage data, iTree then attempts to learn *proto-interactions*—conjunctions of option settings—that cause the new coverage and that may warrant further exploration in the next iteration of iTree.

We represent iTree's behavior as an *interaction tree*, which is a hierarchical representation of proto-interactions. The nodes of the tree represent proto-interactions rather than interactions because they may not, in fact, be full-fledged interactions; because iTree is heuristic in nature, some nodes may represent only portions of interactions, or may represent full interactions with additional constraints.

In this work, we improve upon the initial iTree algorithm by adding support for *composite proto-interactions*. In the improved iTree, the nodes in the interaction tree can be composite proto-interactions. Composite proto-interactions, like the proto-interactions, are also conjunctions of option settings, but with one major difference. In a composite

proto-interaction, instead of fixing every option to a single setting, each option can take on one or more settings, where each setting is derived from at least one discovered proto-interaction.

Figure 2 shows an interaction tree for Figure 1. Each node is labeled with a set of option settings, with *true* at the root node (corresponding to the empty setting). A node represents the proto-interaction that is the conjunction of settings along the path from the root to the node. Notice that the t strength of represented interactions increases from the root to the leaves of the tree. For example, the interaction $\neg\mathbf{local\_enable} \wedge \neg\mathbf{ssl\_enable} \wedge \mathbf{trans\_chunk\_size} = 2048 \mid 4096$ is represented by the left node on the lowest level of the tree. This interaction is, in fact, a composite proto-interaction that actually represents two proto-interactions; the final option setting in the conjunction states that **trans_chunk_size**'s value can be set to either 2048 or 4096.

We also see that $\neg\mathbf{local\_enable} \wedge \mathbf{ssl\_enable}$ is in the interaction tree, but does not correspond to an actual interaction that guarantees coverage of particular code regions. Thus, in this case, iTree has created a proto-interaction that will not lead to useful higher-strength interactions.

Improved iTree Algorithm

Figure 3 gives the pseudocode for the improved iTree algorithm. Like the initial iTree algorithm, the improved algorithm runs in a loop, iterating until developer-supplied stopping criteria are met (e.g., no more coverage is achieved or a time limit has expired). The algorithm begins with an interaction tree iTree containing just one node, *true*. As iTree progresses, it also records in runs the set of all configurations executed so far and their corresponding coverage information. At the beginning of each iteration, findBestLeafNode uses various heuristics to pick a leaf node to explore next. (This heuristic is important because we do not expect to fully explore the interaction tree, as that would be too expensive.)

Next, the proto-interaction represented by the path to the selected node is passed to generateConfigSet. In the improved iTree, this can be a composite proto-interaction.

The generateConfigSet method creates a sample set of configurations that are consistent with the proto-interaction represented by the selected node, while the set of configurations broadly samples all options not participating in the proto-interaction. Currently, iTree leverages CIT for this step, but other sampling techniques could be substituted.

Next, executeConfigSet compiles, instruments, and executes the system's test suite under each configuration in the sample. The data from the resulting executions is then added to runs. Then runs and node.proto_interaction, the proto-interaction represented by node, are passed to discoverProtoInters,

```
1  iTree = /* tree containing root 'true' */
2  runs = ∅ /* (config × coverage) set */
3  do {
4      node = findBestLeafNode(iTree, runs);
5
6      /* generate configurations from composite proto−interactions */
7      configSet = generateConfigSet(node.proto_interaction);
8
9      newruns = executeConfigSet(configSet);
10     if cov(newruns) ⊆ cov(runs)
11       continue;
12     runs = runs ∪ newruns
13
14     interactions = discoverProtoInters(node.proto_interaction, runs);
15
16     if (!interactions.empty())
17       /* merge equivalent proto−interactions */
18       /* to form composite proto−interactions */
19       comp_inters = mergeProtoInters(interactions);
20
21       /* add newly discovered interactions to tree */
22       updateTree(iTree, node, comp_inters);
23 } while (!stoppingCriteriaMet());
```

Fig. 3. Pseudocode for the improved iTree algorithm with composite proto-interactions.

which uses machine learning to discover additional proto-interactions that account for any newly covered entities. Note that, by design, any proto-interactions discovered at this step must include the settings in node.proto_interaction.

Finally, in the improved iTree algorithm, the mergeProtoInters method combines any proto-interactions into composite proto-interactions before updateTree adds the newly discovered proto-interactions to the interaction tree as children of the current node.

We now discuss each step of the algorithm in more detail.

findBestLeafNode. Since iTree aims to find high-coverage configurations, findBestLeafNode prioritizes nodes by the amount of coverage achieved by configurations containing the node's proto-interaction. The assumption is that proto-interactions corresponding to high-coverage configurations are more likely to lead to uncovered code with further exploration. iTree computes a node's priority as follows. First, let $\mathit{Conf}(\mathsf{runs}, \mathsf{node})$ be the subset of runs whose configurations are consistent with node's proto-interaction. For a run $r \in \mathit{Conf}(\mathsf{runs}, \mathsf{node})$, define $\mathit{cov}(r)$ as the number of entities covered by r. Then node's priority is given by

$$\mathit{priority}(\mathsf{node}) = \frac{\sum_{r \in \mathit{Conf}(\mathsf{runs},\mathsf{node})} \mathit{cov}(r)}{|\mathit{Conf}(\mathsf{runs}, \mathsf{node})| + 1}$$

and the highest-priority node is chosen. The formula simply computes a slightly biased average coverage for all configurations that are consistent with the node's proto-interaction. The bias of one added in

	ssl	loc	lis	acc	anon	chunk	dual
C1	0	0	1	1	0	2048	1
C2	0	0	0	0	1	65536	1
C3	1	1	1	1	0	65536	0
C4	0	0	0	0	0	0	0
C5	1	1	0	0	1	2048	0
C6	1	1	1	1	1	0	1
C7	0	1	1	0	0	4096	0
C8	1	0	0	1	1	4096	1

(a) Initial covering array

	ssl	**loc**	lis	acc	anon	chunk	dual
C9	**0**	**0**	1	1	1	65536	1
C10	**0**	**0**	0	1	1	0	0
C11	**0**	**0**	0	1	0	4096	0
C12	**0**	**0**	1	0	1	4096	1
C13	**0**	**0**	0	0	0	65536	0
C14	**0**	**0**	1	0	0	0	1
C15	**0**	**0**	1	0	0	2048	0
C16	**0**	**0**	0	1	1	2048	1

(b) Covering array with ssl = loc = 0

ssl=ssl_enable	loc=local_enable	lis=listen
acc=accept_timeout	anon=anonymous_enable	
chunk=trans_chunk_size	dual=dual_log_enable	

Fig. 4. Example 2-way covering arrays.

the denominator means that nodes corresponding to few runs will have lower priority than their average, but has little effect on nodes corresponding to many runs (since then $|Conf(\mathsf{runs}, \mathsf{node})|$ is high). We found this adjustment useful in that it leads to a slight, but beneficial, preference for nodes that correspond to multiple, high-coverage configurations, over nodes that correspond to fewer, high-coverage configurations.

generateConfigSet. This function generates a sample set of configurations, each of which is consistent with its parameter node.proto_interaction. To do this we use a well-known CIT tool called CASA [8] to generate a low-strength covering array over only the remaining options. We then combine those partial configurations with the settings from node.proto_interaction. In our experiments, we used both 2- and 3-way covering arrays in this step, and found the performance was not sensitive to this choice.

Figure 4 shows two covering arrays created by generateConfigSet as iTree discovered the interaction tree in Figure 2. In this case we chose to generate 2-way covering arrays. Figure 4(a) gives the covering array picked in the first iteration of iTree. Interestingly, our 2-way covering array happened to include the 3-way interaction (see Figure 1) ¬ssl_enable ∧ ¬local_enable ∧ anonymous_enable (in C2) needed to reach *R6* and beyond. However, none of the configurations included the interactions needed to execute *R7* or *R8*. After the data from these configurations was analyzed, iTree added the three children of *true* shown in Figure 2.

The next iteration of iTree expanded the middle of the three nodes (since this node covered *R6*, which contains many lines), and generateConfigSet created the covering array shown in Figure 4(b). Note that in this covering array, ssl_enable and local_enable are fixed. As a result, the 2-way covering array of the remaining options is more effective, and includes the 5-way interaction needed to reach *R7* in C12. This time iTree added two children to the middle node that represents the interaction ¬ssl_enable ∧ ¬local_enable. The new children are composite proto-interactions; we will discuss the generation of composite proto-interactions shortly.

iTree continues, and the next iteration of iTree expanded the left of the two newly added nodes. This node contains a composite proto-interaction, $\neg\mathbf{local\_enable} \wedge \neg\mathbf{ssl\_enable} \wedge \mathbf{trans\_chunk\_size} = 2048 \mid 4096$, that actually represents two equivalent proto-interactions: $\neg\mathbf{local\_enable} \wedge \neg\mathbf{ssl\_enable} \wedge \mathbf{trans\_chunk\_size} = 2048$ and $\neg\mathbf{local\_enable} \wedge \neg\mathbf{ssl\_enable} \wedge \mathbf{trans\_chunk\_size} = 4096$. Two proto-interactions are equivalent if they constrain the same set of options (with different values) and guarantee the exact same coverage.

During covering array generation, instead of creating two covering arrays for each equivalent proto-interaction in our example, generateConfigSet creates only one covering array that includes all t-way combinations of the equivalent option settings and option settings not participating in the composite proto-interaction.

Figure 5(a) shows the covering arrays that would have been created with the two equivalent proto-interactions in the initial iTree algorithm. Figure 5(b) shows the covering array created with the single composite proto-interaction in the improved iTree algorithm.

Both configuration sets were very effective and included the two 5-way interactions needed to cover *R7* and *R8*. However, without composite proto-interactions, initial iTree reached full coverage after analyzing ten configurations. On the other hand, with the composite proto-interaction, the improved iTree algorithm only needed to analyze six configurations.

At this point, iTree has covered all the marked regions of the program.

executeConfigSet. In this step we instrument the system under test and execute its test suite on each configuration in the sample. We compute line coverage with gcov. In our experiments, we ran the instrumented systems on Skoll, a distributed, continuous quality assurance system running on a grid comprising 120 CPUs [26]. As we will see in Section 5.2, Skoll allowed us to scale up iTree, running and analyzing many jobs at once.

discoverProtoInters. Next, we use a two step process to discover proto-interactions to add to the interaction tree: first, we statistically cluster configurations

	ssl	**loc**	lis	acc	anon	**chunk**	dual
C17	**0**	**0**	0	0	0	**2048**	0
C18	**0**	**0**	1	0	1	**2048**	1
C19	**0**	**0**	1	1	0	**2048**	1
C20	**0**	**0**	0	1	1	**2048**	1
C21	**0**	**0**	1	1	1	**2048**	0
C22	**0**	**0**	0	0	0	**4096**	0
C23	**0**	**0**	1	0	1	**4096**	1
C24	**0**	**0**	1	1	0	**4096**	1
C25	**0**	**0**	0	1	1	**4096**	1
C26	**0**	**0**	1	1	1	**4096**	0

(a) Covering arrays w/ proto-interactions

C27	**0**	**0**	0	1	0	**2048**	1
C28	**0**	**0**	0	0	1	**2048**	1
C29	**0**	**0**	1	1	1	**4096**	1
C30	**0**	**0**	0	0	1	**4096**	0
C31	**0**	**0**	1	1	0	**4096**	0
C32	**0**	**0**	1	0	0	**2048**	0

(b) w/ composite proto-interactions

ssl=ssl_enable	loc=local_enable	lis=listen
acc=accept_timeout	anon=anonymous_enable	
chunk=trans_chunk_size	dual=dual_log_enable	

Fig. 5. Comparing covering arrays with and without composite proto-interaction.

	R1	R2	R3	R4	R5	R6	R7	R8	Cluster
C1	x			x	x				0
C2			x	x	x	x			0
C4			x	x	x				0
C7		x		x					1
C3	x								2
C5			x						2
C6	x								2
C8			x						2

(a) clustering of configurations by coverage

Decision Tree Model
ssl_enable = 0
| local_enable = 0: cluster0
| local_enable = 1: cluster1
ssl_enable = 1: cluster2

Discovered Proto-interactions	
cluster0	¬ssl_enable ∧ ¬local_enable
cluster1	¬ssl_enable ∧ local_enable
cluster2	ssl_enable

(b) discovering proto-interactions via decision tree

Fig. 6. Two step process to discover proto-interactions.

according to their coverage data, and second, we try to find proto-interactions responsible for differences in execution.

In the first step, we find all runs involving configurations consistent with the proto-interaction we are exploring. Note that we extract this subset from all of runs, not just those newly explored in the current iteration—this way we get better information as iTree progresses. We then cluster these runs using Weka's [16] implementation of CLOPE [37], a clustering algorithm that groups together similar transactional data records with high dimensionality. As input to CLOPE, we represent each line as a boolean attribute set to *true* if covered in a run and *false* otherwise. Then we use CLOPE to cluster together configurations that execute many of the same lines.

Figure 6(a) shows the line coverage (R1 − R8) of the first covering array from Figure 4 and the clusters formed by the CLOPE algorithm. Configurations C1, C2, and C4 formed a cluster since they had relatively high coverage, including R4 and R5; C7 formed a cluster by itself because it covered two lines; and the rest of the configurations covered one line each and formed the third cluster.

In the second step, we use decision trees [29] to discover commonalities among configurations in each of the clusters. In our implementation, each configuration option is an attribute, and the cluster that a configuration belongs to is the classification. The decision tree algorithm then builds a model for classifying the cluster to which a configuration belongs based on the configuration's option settings. If the resulting model identifies specific option settings that predict cluster membership, then we treat them as new proto-interactions and append them to the interaction tree to form higher-strength proto-interactions. Otherwise no new proto-interactions are added, and exploration of this path stops. In our experiments we evaluated several decision tree algorithms and found each to be adequate for this task.

Figure 6(b) shows a C4.5 decision tree model built for the clusters from Figure 6(a). This decision tree first branches on the values of ssl_enable to distinguish clusters 0 and 1 from cluster2. The decision tree branches again, if $\mathbf{ssl\_enable = 0}$, on the values of **local_enable** to further distinguish cluster0 from cluster1. Using this model, a configuration can be classified to one of the three clusters based on its settings of **ssl_enable** and **local_enable**. iTree parses this decision tree model and extracts each path as a proto-interaction. In this example, the three proto-interactions $\neg$**ssl_enable**$\wedge\neg$**local_enable**, $\neg$**ssl_enable** $\wedge$ **local_enable**, and **ssl_enable** were extracted.

We should mention that CLOPE requires a special parameter called repulsion, which ranges from 0.5 to 4.0, and which controls the ease with which clusters form. To make iTree completely automated, when discoverProtoInters is called, CLOPE is run several times, once for each repulsion value from 0.5 to 4.0 in increments of 0.5. We then perform a separate interaction discovery process for each repulsion value. At the end, we keep the most frequently occurring proto-interaction sets under the range of repulsion values.

mergeProtoInters. The base iTree implementation explores every unique proto-interaction further. However, it is evident from our previous results that some proto-interactions can be redundant. For instance, in our vsftpd example, to reach *R7* or *R8*, trans_chunk_size can be set to either 2048 or 4096. Thus, the two option settings are redundant. Exploring these proto-interactions one at a time can dramatically increase the number of iTree iterations and configurations tested, as illustrated in the example in Figure 5.

In an attempt to handle multiple, equivalent proto-interactions in an efficient way, the improved iTree attempts to merge and simplify any equivalent proto-interactions before adding them to the interaction tree. For example, during the second iteration of our example, iTree discovered two equivalent proto-interactions: $\neg\mathbf{local\_enable} \wedge \neg\mathbf{ssl\_enable} \wedge \mathbf{trans\_chunk\_size} = 2048$ and $\neg\mathbf{local\_enable} \wedge \neg\mathbf{ssl\_enable} \wedge \mathbf{trans\_chunk\_size} = 4096$. After analyzing the data in runs, iTree determines that both proto-interactions guarantee the same coverage, and therefore iTree can create a single composite proto-interaction by merging trans_chunk_size's settings. We note that a composite proto-interaction is a heuristic that might not actually correspond to the original set of equivalent proto-interactions. However, this does not affect iTree's correctness because the algorithm still observes actual coverage reached at runtime.

The improved iTree algorithm merges the equivalent proto-interactions, as opposed to simply pick one and discard the rest, because every proto-interaction must be independently explored. Even though the equivalent proto-interactions yield the same coverage at the current t strength, some of their option settings, in combination with other settings of other options, may lead to higher strength interactions. Therefore, not exploring some proto-interactions may cause iTree to improperly abandon an important path and miss some coverage.

stoppingCriteriaMet. iTree allows its users to plug in their own criteria for determining when to halt execution. Our default is to halt execution when the interaction tree has no more unexplored proto-interactions. The experiments in the following sections include other criteria as well, e.g., in some experiments, we stop execution when a maximum number of configurations have already been tested. Another possibility is to use wall clock time as a stopping criteria, e.g., when doing nightly testing.

4 Evaluating iTree Parameters

We explored the impact of iTree parameters on its cost-effectiveness in a series of three experiments. The first experiment aims to determine two key algorithmic parameters: the covering array strength to use in generateConfigSet, and the decision tree implementation to use in discoverProtoInters. The second experiment explores whether composite proto-interactions can improve the cost-effectiveness of an iTree run. Finally, the third experiment explores modifying iTree to adaptively select covering array strength and use multiple decision tree approaches simultaneously.

	vsftpd	ngIRCd
Version	2.0.7	0.12.0
# Lines (sloccount)	10,482	13,601
# Run-time opts.	30	13
Boolean/Integer	20/10	5/8
Full config space	2.1×10^9	2.9×10^5
# Test cases	64	141
# Max coverage	2,549	3,193

Fig. 7. Program statistics for vsftpd and ngIRCd. Note that we removed some unreachable code before measuring lines of code.

Experimental Setup

Subject Systems. In this first series of experiments, we used vsftpd, a widely used secure FTP daemon, and ngIRCd, the "next generation IRC daemon," as subjects. We studied these systems extensively in prior work [31]. From that work, we have test suites for these systems and detailed information about the systems' configuration spaces with respect to those test suites. Figure 7 gives descriptive statistics for each system. They have roughly 10–13KLOC and are written in C. The figure details the total number of configuration options we analyzed, broken down by type (boolean or integer). This is the same set of options and settings we used in our prior work. The values we used for the integer options also came from our previous work, and were chosen to maximize path coverage for these subject systems and test cases. Finally, the last rows list the size of the full configuration space for the options (the total number of different possible configurations); the number of test cases in our test suite; and the maximum possible number of lines covered if we execute every test case under every possible configuration.

Covering Array Strengths. Each iTree iteration begins by creating a sample of configurations, derived from a t-way covering array. The value of t determines the size of each sample, and may also influence the speed with which iTree terminates. In this study, we use either $t = 2$ or $t = 3$.

Decision Trees. Many different decision tree classifiers have been proposed in the machine learning literature. We used two algorithms in our experiment: C4.5 and CART, both as implemented in Weka [16]. We picked C4.5 (the open source implementation of J48) and CART because they are the most popular decision tree implementations and because they are designed to produce compact classifications, which

```
Conf_ListenIPv4 = 1
| Conf_PongTimeout = 1: cluster0
| Conf_PongTimeout = 20: cluster1
| Conf_PongTimeout = 3600
| | Conf_MaxNickLength = 0: cluster0
| | Conf_MaxNickLength = 4: cluster0
| | Conf_MaxNickLength = 5: cluster0
| | Conf_MaxNickLength = 6: cluster1
| | Conf_MaxNickLength = 8: cluster0
| | Conf_MaxNickLength = 9: cluster1
| | Conf_MaxNickLength = 10: cluster0
| | Conf_MaxNickLength = 100: cluster0
Conf_ListenIPv4 = 0: cluster2
```

(a) C4.5 Decision Tree

```
Conf_ListenIPv4=(0): cluster2
Conf_ListenIPv4!=(0)
| Conf_MaxNickLength=(4)|(5)|(0): cluster0
| Conf_MaxNickLength!=(4)|(5)|(0)
| | Conf_PongTimeout=(3600)|(20): cluster1
| | Conf_PongTimeout!=(3600)|(20): cluster0
```

(b) CART Decision Tree

Fig. 8. Classification models generated using C4.5 and CART decision trees.

may be well-suited to iTree's incremental search approach.

The classification models generated by these two decision trees have important differences that affect the proto-interactions generated as well as the process of extracting them. Figure 8 shows the C4.5 and CART models generated for an ngIRCd iteration. As we can see, the C4.5 model creates a unique path for every combination of option setting, even if some combinations lead to the same classification. The CART model, on the other hand, branches on rules for the options' settings rather than the individual values of the options.

From the C4.5 models, we simply parse for distinct paths and each path is treated as one proto-interaction. For the CART models, however, we must first reference the system's configuration space model to solve the inequalities in the branching rules. For example, $\mathsf{Conf\_MaxNickLength!} = (4)|(5)|(0)$ would transform to $\mathsf{Conf\_MaxNickLength} = (6)|(8)|(9)|(10)|(100)$ using ngIRCd's configuration space model. We then generate one proto-interaction for every unique combination of option settings in a decision tree path. We can also see that, because of the differences in decision tree, the cluster to which a proto-interaction belongs can be different as well. However, once the proto-interaction extraction is done, iTree will not need the clustering information.

Iteration Retries. An iTree iteration may fail to discover proto-interactions for a number of reasons. For instance, the decision tree algorithm may fail to generate a classification model. Failing to discover useful proto-interactions will cause iTree to improperly abandon the current path. This can delay or even prevent the coverage of some necessary higher strength interactions.

The obvious solution to this problem is to perform retries, that is, to generate more configuration samples and performed the classification again. However, not all failing iterations should be retried either, since some proto-interactions do not lead to actual interactions. Retrying every iteration can dramatically increase the number of configurations tested.

For our first two experiments, our iTree implementation only allows for retries on the very first iteration, because without any proto-interactions the algorithm cannot proceed. In the third experiment, we explore other conditions under which retries should be performed.

Experimental Design. We ran iTree 30 times on both subject systems under each possible combination of decision tree and covering array strength. For each run, we continued execution until we reached the maximum possible coverage (as determined from our prior work [31]). The number of configurations executed is our metric for finding the best parameter settings—the lower the number, the faster the algorithm achieves full coverage. Note that in these experiments, rather than actually run the executeConfigSet step we instead used the code coverage data we had already computed in our prior work (which gave us a mapping from configurations to their coverage).

Experiment 1 Data and Analysis

Figure 9 shows boxplots of our experimental results for vsftpd and ngIRCd. The left half of each chart shows the results of the decision trees for $t = 2$, and the right half shows the results for $t = 3$. The y-axis reports the number of configurations required to achieve full coverage. The number in parentheses under each boxplot indicates the number of iTree runs, out of 30, in which full coverage was achieved. c4.5 and cart are the results of the base iTree algorithm. We defer discussion of c4.5_c, cart_c and vote_adapt.

Covering Array Strengths. We see in Figure 9 that increasing the t strength of the covering arrays did not greatly change the cost of running iTree for vsftpd. It did have some effect for ngIRCd, where the average size of the configuration sets increased across all decision tree algorithms. However, we also see that the number of runs in which the iTree algorithm reached maximal coverage is substantially higher when $t = 3$ than when $t = 2$, for both subject systems. We

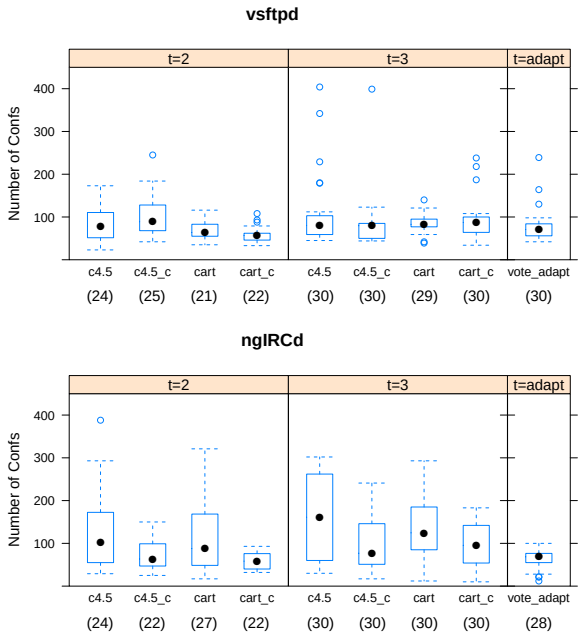


Fig. 9. Interaction tree effort for different iTree parameters.

looked in more detail at the individual runs and observed that both the likelihood of discovering proto-interactions and the accuracy of the discovered proto-interactions at each iteration dramatically improved as t increased. However, this resulted in a trade off. While increased sample size means more cost at each iteration, it also resulted in fewer overall iterations for our subject systems. In the end, the total cost did increase for ngIRCd.

We note that variance in the number of configurations tested appears unrelated to covering array strength. Instead, it seems more tied to the system tested. In particular, for vsftpd, the range of the number of configurations tested is fairly stable, while for ngIRCd, it fluctuates considerably. Based on further analysis, we believe this occurs because the configuration space model for ngIRCd, taken from our previous analysis results, contained many redundant option settings from the perspective of line coverage. This resulted in many equivalent proto-interactions being used to redundantly explore the same part of its effective configuration space and can delay iTree from reaching complete coverage. On the other hand, this also means ngIRCd's 2-way covering arrays experiments already enjoyed the benefits of larger configuration samples.

Decision Trees. In Figure 9, the c4.5 and cart columns for each t-value show the effect of the C4.5 and CART decision trees on the base iTree algorithm. The data shows no systematic differences in performance across the two classifiers. Looking at the individual iterations of iTree, however, we did find some differences: CART fails to discover any proto-interactions in the configuration samples more often than C4.5 does. Fortunately, this situation occurs mostly during the very first iteration, and our retry heuristic guarantees that some proto-interactions will be discovered eventually. Therefore, it does not impact the performance of iTree greatly.

In some of the runs, we find that both C4.5 and CART can discover proto-interactions that are not quite accurate. This usually results in poor iTree performance. We explore ways to improve the situation in Section 4.

Experiment 2 Data and Analysis

Columns c4.5_c and cart_c in Figure 9 show the performance of C4.5 and CART decision trees under each t-value using the improved iTree with composite proto-interactions. For vsftpd, using composite proto-interaction did not significantly change the performance of the iTree runs; this is understandable since most vsftpd's configuration options are boolean.

But for ngIRCd, we see that both the number of iterations and the number of configurations needed to reach complete coverage decreased substantially under both $t = 2$ and $t = 3$. In addition, the decrease in variance among the runs is quite significant, especially for $t = 2$. These results show that using composite proto-interactions increased the likelihood of iTree covering all the interactions needed to reach complete coverage during earlier iterations. In particular, on average, using composite proto-interactions decreased the number of configurations needed for complete coverage by a factor of 2–3 under both $t = 2$ and $t = 3$; for C4.5 from 101 and 160 down to 53 and 96, respectively, and for CART from 112 and 125 down to 43 and 95.5 for CART, respectively.

However, we also see that when $t = 2$, slightly fewer runs reached complete coverage for ngIRCd. We found the cause to be that a covering array generated (with the CASA tool) in a single iteration using a composite proto-interaction contains fewer configurations than several covering arrays generated in numerous iterations using several equivalent proto-interactions. In the case of $t = 2$, the smaller configuration samples were not as effective for executing new coverage and discovering proto-interactions. The $t = 3$ runs, with larger samples, were not affected.

Overall, we find that using composite proto-interactions substantially reduced the running time and configuration set size, without compromising accuracy.

Hybrid Approaches

When we examined the worst-performing runs from the previous experiments, we found that they suffered from inaccurate proto-interaction discovery. Both C4.5 and CART sometimes produce inaccurate classifications—they include option settings in the proto-interactions that are not part of the actual

interactions. This situation can have great negative consequences, especially during the early stages of the iTree discovery, because the inaccuracies can propagate through an iTree path. This would effectively send iTree on a wild goose chase. Specifically, inaccurate proto-interactions may restrict configuration sampling to unimportant parts of the configuration space, thus preventing iTree from covering the interactions needed for complete coverage.

There are two main causes to the inaccurate proto-interaction discovery. The first cause is insufficient training examples provided to the decision tree classifiers. From our previous experiments we see that increasing the t-value improved the accuracy of the discovered proto-interactions. However, increasing the t-value also increased testing efforts. So, to increase the size of the configuration samples without dramatically increasing the test obligations, we decided to use two 2-way covering arrays for every iteration of iTree (with the CASA tool, 2-way covering arrays are about one third the size of 3-way covering arrays).

The second cause is inherent in the design of the decision tree classifiers. These classifiers are designed to minimize errors in classifying the training examples, and this can cause the decision tree models to be overfitted for the configurations used to discover the proto-interactions. We noticed, however, that C4.5 and CART use different heuristics to reduce the error rate, and that the inaccuracies often involved different option settings. Thus, we developed an *aggregation classifier*, similar to bagging [1], [30], that creates an ensemble classifier out of C4.5 and CART decision trees. This aggregation classifier filters out option settings from proto-interactions unless both C4.5 and CART produce them as classifiers. The assumption is that option settings that appear in both models are more likely to be part of actual interactions.

However, this aggregation classifier is also more likely to fail to discover proto-interactions; if the decision trees do not agree on any option setting during an iteration then iTree would be forced to abandon the current path. To alleviate this problem, iTree performs a retry of the current iteration if new coverage was executed during this iteration but no proto-interactions were discovered.

With the combination of aggregation classifier and the retry condition, iTree essentially produces either lower strength proto-interactions with option settings it is confident about, or it increases the configuration sample size to produce more accurate classifications.

Experiment 3 Data and Analysis

The results using the adaptive approach along with composite proto-interactions are shown in Figure 9's vote_adapt column. We can see from the figure that vote_adapt is an attractive choice overall — its average cost is lower or only slightly worse than the best of the other algorithms, and it yields full coverage on every or almost every run. In addition, the variance in the number of configurations tested decreased for both subject systems, making the approach's performance more stable.

5 Comparing iTree to Other Approaches

Our remaining experiments compare the coverage achieved by iTree, random sampling, and a single, high-strength covering array.

As mentioned earlier, researchers and practitioners have developed several strategies to generate configuration samples for software testing. Both CIT and random sampling are popular approaches for generating configuration samples and produce relatively good results in practice. To better understand how iTree compares with these existing approaches, we conducted a series of experiments.

5.1 Evaluating Performance

For the first experiment, we again used vsftpd and ngIRCd and ran each technique 30 times. One problem with CIT and random sampling is that developers cannot know *a priori* how large a sample is necessary. For CIT, developers must pick a t value, and for random sampling developers must guess a sample size.

In this experiment, we created covering arrays using a range of different t strengths. For each strength, testing ran until either the maximum possible coverage was achieved or until no more configurations remained. Using 5- and 4-way covering arrays for vsftpd and ngIRCd, respectively, often produced maximal coverage, so we used those as our largest sample sizes. We next tested the systems with random samples sized equal to the average size of these largest covering arrays.

We also tested the systems using improved iTree. For this experiment, we used vote_adapt as described in the previous section, and set iTree to stop when either no new coverage is achieved on any path or the number of configurations tested exceeded the size of the random configuration samples. We measured performance using two criteria: (1) whether complete coverage was reached and (2) if so, the number of configurations needed to reach the complete coverage.

Data and Analysis

Figure 10 shows the results of these experiments. The x-axis is the number of configurations tested so far in each run, and the y-axis is the median number of lines covered at that point across all runs. Here we are assuming that configurations are tested in the order they are generated, although in practice the work can be done in parallel across multiple CPUs. The 10 data

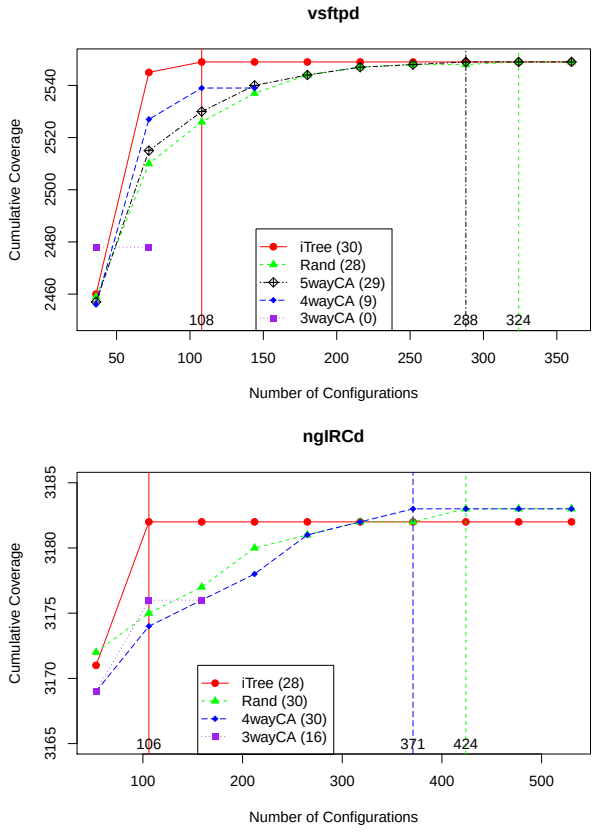


Fig. 10. Comparing iTree against covering arrays and random sampling.

points plotted in each figure divide the time line into equal epochs, corresponding to 36 or 53 configurations tested for vsftpd and ngIRCd, respectively. Note that iTree runs can terminate before executing the number of configurations of the other methods. In that case, we simply treat subsequent time points as unchanged from the previous time point. The figures also include a vertical line indicating the epoch in which 90% of the runs achieved maximal coverage. The numbers in parentheses in the legend indicate the total number of runs, out of 30 each, that reached full coverage for each approach.

In the top portion of Figure 10 showing the vsftpd results, we see that iTree, 5-way covering arrays, and random sampling eventually reached full coverage in almost all runs (30, 29, and 28, respectively), but 4-way only reached full coverage in a third of the runs, and 3-way never reached full coverage. Moreover, looking at the vertical lines, we see that 90% of the iTree runs reached full coverage with around one third the number of configurations, on average, of the 5-way covering arrays, which themselves did noticeably better than random sampling. We see a similar trend for ngIRCd in the bottom figure, for which iTree, 4-way covering arrays, and random sampling achieved full coverage in all or almost all runs (28, 30, and 30, respectively), but 3-way covering arrays only reached full coverage in just over half the runs (16). Again iTree performed significantly better: 90% of the iTree runs reached full coverage with less than one third the number of configurations, on average, of the 4-way covering arrays, 90% of which reached full coverage faster than random sampling. With vote_adapt, iTree's benefit is quite an impressive improvement over the other approaches.

It is important to mention that our analyses ignore the time needed to generate the configurations to be tested. For iTree, this time is dramatically smaller than the time required to run the test cases because our algorithm uses multiple low strength covering arrays during testing. For random sampling, the generation time is also quite small. However, for high strength covering arrays, the generation time can actually exceed the test case execution time. For instance, vsftpd's 4-way and 5-way covering arrays each took several hours to several days to generate, respectively.

In addition to coverage, we also attempted to measure the effectiveness of composite proto-interactions. Since a single composite proto-interaction can represent multiple equivalent proto-interactions, we wanted to measure how many more proto-interactions the improved iTree algorithm was able analyze over the base algorithm. For the vsftpd experiments, each run performed, on average, 11 iterations of analysis. None of the interactions analyzed were composite proto-interactions. This observation is in line with the results from 4. For the ngIRCd experiments, each run performed, on average, 7 iterations of analysis. In the base algorithm, this would translate to 7 proto-interactions that were explored further. In the improved algorithm, however, 7 composite proto-interactions were explored, which translated to 26 proto-interactions; almost 4 times more proto-interactions were analyzed.

Discussion

Overall, these results showed the improved iTree algorithm performing better than t-way covering arrays and random sampling, at substantially less cost. This conclusion, of course, depends on how those approaches are actually used. For example, if developers used high strength covering arrays or large random samples, they would be likely to get most of the available coverage, but would do so at a large cost. As we know from our previous research, this is not a very efficient approach, because few of those configurations are really necessary to achieve specific types of coverage, such as line coverage. For instance, it would require a 7-way covering array with thousands of configurations to guarantee complete line coverage for vsftpd and ngIRCd. If developers instead used a lower-strength, 2-way covering array, the cost would be much lower, but so would the coverage.

5.2 Evaluating Scalability

Using the improved iTree algorithm, we were able to achieve maximal coverage while executing on av-

	MySQL
Version	5.1
# Lines (sloccount)	939,842
# Compile-time opts.	18
Boolean/Integer	18/0
# Run-time opts.	18
Boolean/Integer	11/7
Full config space	4.9×10^{12}
# Test cases	1244

Fig. 11. MySQL program statistics.

erage about 100 configurations for both vsftpd and ngIRCd. This is encouraging, but after all, we had already solved this problem using symbolic execution, albeit at a far higher cost. Ultimately our goal is to handle much larger systems, written in a variety of languages, with compile-time as well as run-time configuration options. None of these issues can currently be addressed using symbolic execution, but we believe that iTree may be the right tool for this problem.

To better understand this issue, we evaluated the scalability of the improved iTree algorithm by running it on MySQL, a popular open source database. We are not aware of any current symbolic execution system that can fully handle this system. MySQL has more than 900K lines of code. It is written in a combination of C and C++, and its configuration space includes a large number of run-time as well as compile-time configuration options. As in our previous experiments in Section 5.1, our evaluation compared iTree against covering arrays and randomly sampled configurations.

Subject System

Figure 11 gives descriptive statistics for MySQL. The top two rows list the version we used and the lines of code it contains as computed by sloccount [36]. Next, the figure lists the number and types of configuration options we selected for our experiment. We give the numbers of compile-time and run-time configuration options separately, and each number is also broken down by type (boolean or enumeration). All told, we are focusing on 36 configuration options. We selected configuration options and settings that enabled the test suite to exercise the major configurable features of MySQL, such as default storage engines, SQL modes, and transaction isolation modes. All other MySQL options were left with their default values.

The next row in Figure 11 lists the number of unique configurations that can be generated given the number of distinct settings of configuration option. All told, the full configuration space given the subset of MySQL options we are considering includes roughly five trillion configurations. This configuration space is dramatically larger compared to the one we analyzed in our previous work [35], which consisted of only 16 configuration options and 600K configurations.

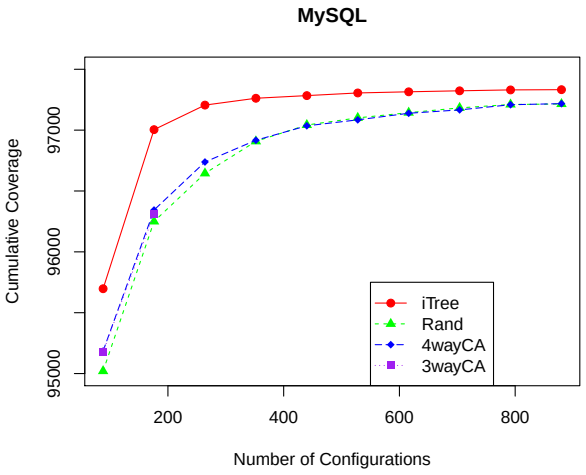


Fig. 12. Comparing the number of configurations and coverage achieved using different testing approaches.

Finally, the last row in the figure lists the number of test cases comprising the regression test suite we used in our experiment. Although this is the actual test suite that comes with MySQL, we note that it is not comprehensive or high-coverage. For example, in the default configuration it achieves coverage of only ~90K LOC. We also note that not every test case runs in every configuration.

Experimental Design

Our experimental design is similar to that of Section 5.1. Specifically, we compare 3-way covering arrays, 4-way covering arrays, random sampling, and iTree. On average, 3-way coverings contained 159 configurations, 4-way covering arrays contained 809 configurations, and random sampling also selected 809 configurations. We executed each approach 30 times and computed how much line coverage was achieved under each. For iTree we again used the vote_adapt approach. One key difference between this experiment and the last is that we cannot know the maximal possible coverage achievable by the test suite, and so we only discuss observed line coverage.

We executed the experiment on the Skoll cluster using up to 90 CPUs at a time. Executing the MySQL test suite on one configuration takes approximately 1.5 hours. The process involves downloading the MySQL source tree from the source code repository; compiling an instance according to the compile-time option settings for the configuration to be tested; instrumenting the instances with gcov; starting the instance with the run-time option settings dictated by the configuration to be tested; running the test suite; and collecting the execution data.

Data and Analysis

Figure 12 summarizes the experimental results. The figure shows the growth in median coverage over time under each of the four approaches used, measured at 10 equally spaced intervals. The y-axis is the

number of covered lines, and the x-axis indicates the number of configurations tested.

We can see from these results that iTree covered more lines of code on average than the other methods after running the same number of configurations. Interestingly, the traditional methods have very similar performance profiles. Thus, with respect to this data, it appears that at every level of effort, iTree-selected samples that included configurations with unique coverage patterns that were not found by more traditional approaches. As a result, iTree required substantially fewer configurations to reach the maximum coverage achieved by the best of the other approaches. For example, iTree achieved the same coverage as 4-way covering arrays (which used 809 configurations) with only 528 configurations, and the same coverage as random sampling (again using 809 configurations) with only 352 configurations. Taken together, these results show that iTree can achieve better results with less testing effort.

The absolute difference in line coverage ranges from a high of around 0.7% ($\sim$ 700 LOC) early on down to about 0.1% ($\sim$ 115 LOC) near the end of the experiment. To better understand why these lines were found by iTree, but not by the other methods, we manually inspected MySQL's source code. We observed that the extra lines covered with iTree involved many small pockets of code scattered across numerous files, methods, and blocks and are apparently only executed in very specific circumstances.

We further attempted to determine what interactions control the lines that are covered by iTree and not the other approaches, but were unable to decide this because of MySQL's size and complexity. However, we generated a 5-way covering array and executed its 3140 configurations, and found that it also failed to cover about half of the lines in question as well. This implies that at least some of these lines are controlled by interactions of strength 6 or higher. Almost all of the iTree runs from this experiment outperformed this 5-way covering array, even though they included only a fraction of the configurations.

In addition to coverage, we also attempted to measure the effectiveness of composite proto-interactions. During the iTree experiments, each run performed, on average, 17 iterations of analysis, which in the base algorithm, this would translate to 17 proto-interactions explored. In the improved algorithm, however, the 17 composite proto-interactions translated to, on average, 48 proto-interactions; almost 3 times more proto-interactions were explored.

Once again, we showed that iTree outperformed both t-way covering arrays and random sampling, at far less cost. But more importantly, these results showed that iTree is scalable to large industrial systems with substantial configuration spaces and the composite proto-interaction is an important improvement to the iTree algorithm.

6 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 widely used subject systems. Two are medium-sized; one is quite large. To balance greater external validity against higher costs and lessened experimental control, we focused on subsets of configuration options that we determined to be important. The size of these sets was substantial, but did not include every possible configuration option to keep our analysis tractable. The structural coverage criteria was line coverage. Other program behaviors such as data flows or fault detection might lead to different performance tradeoffs. Our test suites taken together have reasonable, but not complete, coverage. Individually, the test cases tend to focus 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.

7 Related Work

Combinatorial Interaction Testing. Covering array-based sampling for software testing is a specification-based technique that was originally proposed as a way to ensure even coverage of combinations of input parameters to programs [7], [2], [9], [5]. In more recent work, covering arrays have been used to model configurations that should be selected for testing [13], [28], [38], where the covering array defines a *test schedule* and each configuration is tested with an entire test suite. Covering arrays have also been used to test graphical user interfaces [40] and in model based testing [4].

Empirical research suggests testing with covering arrays with $t < 6$ can potentially find a large proportion of interaction faults [21]. Further studies suggest covering arrays can be effective in practice and can yield good structural coverage during testing [13], [27], [28], [38], [7], [12], [20].

Machine Learning in Software Engineering. Many researchers have proposed using dynamic analysis with machine learning techniques to analyze program executions. Haran et al. [17] developed three techniques—association trees, random forests, and adaptive sampling association trees—to automatically classify fielded software system executions. Podgurski and colleagues [10], [25], [14] used tree-based strategies and random sampling to classify program faults in order to prioritize software failure reports. Brun and Ernst [3] use machine learning to classify program invariants that manifest themselves in failing program runs. We are not aware of any other work that applies machine learning to testing software configurations.

Test Case Prioritization. The iTree approach is similar to some test prioritization techniques which try to

find effective orderings of test cases that reveal faults earlier in the testing process. Many such techniques also utilize structural coverage as the surrogate criteria for prioritization [33], [34], [32]. Leon et al. [22] evaluated distribution-based cluster filtering on the execution profiles as prioritization scheme and found that it detects different faults than coverage-based prioritization. Yoo et al. [39] also applied clustering of test case dynamic runtime behavior to aid test prioritization. Li et al. [23] evaluated greedy, metaheuristic and evolutionary search algorithms for prioritization.

8 Conclusions and Future Work

We have presented in detail a scalable technique called iTree to support the testing of highly configurable systems. iTree's goal is to select a small set of configurations in which the execution of the system's test suite will achieve high coverage. This technique is based upon insights gained from our previous empirical studies in which we precisely quantified the relationships between software configuration and program execution behaviors. These insights led us to create a heuristic process that effectively searches out configurations in which high coverage is likely.

To evaluate the improved iTree algorithms, we conducted several sets of experiments. Keeping existing threats to validity in mind, we tentatively drew several conclusions. All of these conclusions are specific to our subject systems, test suites, and configuration spaces; further work is needed to establish more general trends. The first set of studies evaluated the basic iTree approach and its parameters. Based on these efforts we developed several optimizations, such as composite proto-interactions and adaptive voting, that improve robustness while also removing many issues that must be handled manually with current techniques. The second set of studies compared iTree with t-way covering arrays and random sampling, both existing techniques. The studies suggested that iTree produced higher coverage then the other techniques while testing fewer configurations. The final set of studies focused on scalability. This study applied iTree to MySQL, a large and popular database system. The results strongly suggested that iTree achieved higher coverage at lower cost than existing techniques. Taken together, our results strongly suggest that iTree is a promising technique that can scale to practical industrial systems.

Based on this initial work, we plan to pursue several research directions. First, we will extend our studies to include more systems with larger and more complex configuration spaces. Second, we plan to enhance iTree to incorporate new kinds of coverage and program behavior. We will also examine how information gained as iTree operates might be incorporated into iTree's heuristics. Finally, we will explore post-processing the information and artifacts that iTree creates to support other software engineering tasks such as impact analysis, reverse engineering, and automatic architecture documentation.

Acknowledgments

This research was supported in part by NSF CCF-1116740 and NSF-CCF-0811284.

References

[1] L. Breiman. Bagging predictors. *Machine Learning*, 24:123–140, 1996. 10.1007/BF00058655.

[2] 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.

[3] Y. Brun and M. D. Ernst. Finding Latent Code Errors via Machine Learning over Program Executions. In *ICSE*, pages 480–490, 2004.

[4] R. Bryce and C. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Journal of Information and Software Technology*, pages 960–970, 2006.

[5] K. Burr and W. Young. Combinatorial test techniques: Table-based automation, test generation and code coverage. In *ICSE Analysis & Review*, 1998.

[6] 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.

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

[8] M. B. Cohen. Combinatorial interaction testing portal: Casa, 2009.

[9] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *ICSE*, pages 285–294, 1999.

[10] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *ICSE*, pages 339–348, 2001.

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

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

[13] S. Fouché, M. B. Cohen, and A. Porter. Incremental covering array failure characterization in large configuration spaces. In *ISSTA*, pages 177–188, 2009.

[14] P. Francis, D. Leon, M. Minch, and A. Podgurski. Tree-Based Methods for Classifying Software Failures. In *ISSRE*, pages 451–462, 2004.

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

[16] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *SIGKDD Explor. Newsl.*, 11:10–18, November 2009.

[17] M. Haran, A. Karr, M. Last, A. Orso, A. Porter, A. Sanil, and S. Fouche. Techniques for Classifying Executions of Deployed Software to Support Software Engineering Tasks. *Trans. Soft. Eng.*, 33(5):287–304, May 2007.

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

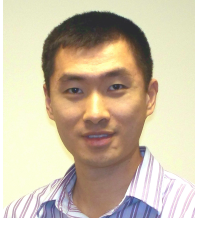
[19] D. Kuhn and M. Reilly. An investigation of the applicability of design of experiments to software testing. In *NASA Goddard/IEEE Software Engineering Workshop*, pages 91–95, 2002.

[20] 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.

[21] D. R. Kuhn, D. R. Wallace, and A. M. Gallo. Software fault interactions and implications for software testing. *Trans. Soft. Eng.*, 30:418–421, 2004.

[22] D. Leon and A. Podgurski. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In *ISSRE*, pages 442–453, 2003.

[23] Z. Li, M. Harman, and R. Hierons. Search algorithms for regression test case prioritization. *Tarns. Soft. Eng.*, 33(4):225–237, 2007.
[24] R. Mandl. Orthogonal Latin squares: an application of experiment design to compiler testing. *Commun. ACM*, 28(10):1054–1058, 1985.
[25] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated Support for Classifying Software Failure Reports. In *ISSRE*, page 465, 2003.
[26] A. Porter, C. Yilmaz, A. M. Memon, D. C. Schmidt, and B. Natarajan. Skoll: A Process and Infrastructure for Distributed Continuous Quality Assurance. *Trans. Soft. Eng.*, 33(8):510–525, August, 2007.
[27] X. Qu, M. Cohen, and K. Woolf. Combinatorial Interaction Regression Testing: A Study of Test Case Generation and Prioritization. In *ICSM*, pages 255 –264, 2007.
[28] X. Qu, M. B. Cohen, and G. Rothermel. Configuration-aware regression testing: an empirical study of sampling and prioritization. In *ISSTA*, pages 75–86, 2008.
[29] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986. 10.1007/BF00116251.
[30] J. R. Quinlan. Bagging, boosting, and c4.5. In *Proceedings of the thirteenth national conference on Artificial intelligence - Volume 1*, AAAI'96, pages 725–730. AAAI Press, 1996.
[31] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *ICSE*, pages 445–454, 2010.
[32] G. Rothermel, S. Elbaum, A. Malishevsky, P. Kallakuri, and B. Davia. The impact of test suite granularity on the cost-effectiveness of regression testing. In *ICSE*, pages 130–140, 2002.
[33] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Test case prioritization: An empirical study. In *ICSM*, pages 179–188, 1999.
[34] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Prioritizing test cases for regression testing. *Trans. Soft. Eng.*, 27(10):929–948, 2001.
[35] C. Song, A. Porter, and J. S. Foster. itree: efficiently discovering high-coverage configurations using interaction trees. In *ICSE*, pages 903–913, 2012.
[36] D. A. Wheeler. Sloccount. http://www.dwheeler.com/sloccount/.
[37] Y. Yang, X. Guan, and J. You. CLOPE: a fast and effective clustering algorithm for transactional data. In *KDD*, pages 682–687, 2002.
[38] C. Yilmaz, M. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *Trans. Soft. Eng.*, 31(1):20–34, 2006.
[39] S. Yoo, M. Harman, P. Tonella, and A. Susi. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In *ISSTA*, pages 201–212, 2009.
[40] X. Yuan, M. Cohen, and A. M. Memon. Covering Array Sampling of Input Event Sequences for Automated GUI Testing. In *ASE*, 2007.



Charles Song Charles Song received his Ph.D. in computer science from University of Maryland, College Park in 2011. He is currently a research scientist at Fraunhofer USA Center for Experimental Software Engineering. He aims to apply empirical research methods to real world problems and transfer practical solutions to industry and government collaborators. His research interests include applications of static and dynamic analysis, machine learning techniques and distributed computing to tackle software engineering problems.

Adam Porter Adam A. Porter earned his B.S. degree summa cum laude in Computer Science from the California State University at Dominguez Hills, Carson, California in 1986. In 1988 and 1991 he earned his M.S. and Ph.D. degrees from the University of California at Irvine.

Since 1991, Dr. Porter has been a professor of computer science at the University of Maryland and the University of Maryland Institute for Advanced Studies (UMIACS). His research focuses on empirical methods for identifying and eliminating bottlenecks in industrial development processes, experimental evaluation of fundamental software engineering hypotheses, and development of tools that demonstrably improve fundamental software development processes.

Jeffery S. Foster Jeffrey S. Foster is associate professor in the Department of Computer Science and the University of Maryland Institute for Advanced Computer Studies (UMIACS) at the University of Maryland, College Park. He is also associate chair for Graduate Education in the Department of Computer Science. His research aims to give programmers practical new tools to help improve the quality and security of their programs. His research interests include programming languages, program analysis, constraint-based analysis, and type systems.