Scalable Uniform Sampling for Real-World Software Product Lines

Jeho Oh  
University of Texas at Austin  
USA  
jeho@cs.utexas.edu

Paul Gazzillo  
University of Central Florida  
USA  
paul.gazzillo@ucf.edu

Don Batory  
University of Texas at Austin  
USA  
batory@cs.utexas.edu

Marijn Heule  
Carnegie Mellon University  
USA  
marijn@cmu.edu

Margaret Myers  
University of Texas at Austin  
USA  
myers@cs.utexas.edu

Abstract

Software Product Lines (SPLs) often have huge numbers of configurations that are impossible to enumerate. This raises the need for uniform sampling, which yields samples that are representative of the configuration space and enables accurate estimates of configuration properties by classical statistical methods. Prior work on sampling SPLs either achieved uniform sampling or scaled to large configuration spaces but not both, limiting their applicability to real-world SPLs.

Smarch is a new uniform sampling algorithm that scales to real-world SPLs. It maintains a one-to-one correspondence between integers and configurations, converting uniformly sampled integers into uniformly sampled configurations. As Smarch only creates configurations that are used as samples, it has better scalability than other uniform sampling algorithms. Smarch can be optimized with respect to variable selection, parallelism, and caching to reduce its sampling time.

We evaluate Smarch on 39 real-world SPLs. Smarch is able to sample configuration spaces as large as $10^{417}$, which is $10^{405}$ times larger than existing uniform sampling algorithms can handle. Optimizations make Smarch sampling at least 12 times faster.

Keywords software product lines, uniform sampling

1 Introduction

Software product lines (SPLs) are highly configurable. They allow users to specify products declaratively by selecting desired features, which are increments of product functionality. The set of features that comprise a product is unique and is called a configuration. As the number of features increase, the set of all possible configurations, called a configuration space, grows exponentially. This allows users great freedom in combining features, but complicates finding the most desirable configuration [32] and finding configuration bugs due to faulty interactions among features [33].

Understanding properties of a configuration space is essential for many software engineering tasks: deriving a model to predict the performance of any given configuration [12, 18, 34, 36], finding an optimal configuration given user-specified constraints [13, 15, 32, 35], and finding variability bugs due to specific combinations of features [9, 14, 25, 28, 33]. Real-world SPLs have so many features that investigating each configuration is infeasible. Instead, researchers rely on configuration sampling to estimate the properties of configurations needed for their tasks.

The accuracy of these analyses depends on samples being representative of the configuration space. This requires uniform sampling, where each configuration has an equal chance of being chosen [38]. Classical statistical methods assume uniform sampling as a precondition for their use [38]; if this precondition is violated, the computed statistics can be far from the truth. This is born out in practice: Kaltenecker et al. showed that uniform sampling produces more accurate performance prediction models of a configuration space compared to other sampling techniques [18]. Maarenen et al. showed that the accuracy of evolutionary algorithms improved by 10% by starting with uniform sampling [23].

Prior work on sampling SPL configurations presents researchers with a dilemma: either sacrifice accuracy with non-uniform sampling that scales to real-world SPLs, or sacrifice scalability with a uniform sampling algorithm limited to small SPLs [18, 33]. State-of-the-art sampling approaches based on satisfiability (SAT) solvers [7, 9, 15] can generate configuration samples quickly for large SPLs, but their samples are not uniformly distributed [33]. Prior approaches that achieved uniform sampling [5, 32] do not scale to configuration spaces larger than $10^{48}$ [33].

Our paper resolves this dilemma: we present Smarch, an algorithm that can uniformly sample from configuration spaces of sizes up to $10^{417}$. Existing uniform sampling methods encode or generate configurations that are not used as samples, which is prohibitively expensive for large SPLs. Smarch, in contrast, generates only configurations that are being used as samples, which greatly improves scalability.

The key to Smarch is that it maintains a one-to-one correspondence between integers and configurations, converting uniformly sampled integers into uniformly sampled configurations. Unlike prior work that used SAT solvers to identify
configurations, Smarch uses #SAT solvers to count configurations efficiently without having to produce configurations.

We also present optimizations to reduce the sampling time of Smarch. We apply the cube-and-conquer algorithm of Heule et al. [16] to select features to partition a space that reduces search complexity as much as possible. We also parallelize the sampling process and cache the partition sizes so that they can be reused in later integer-to-configuration translations.

We evaluated Smarch and existing algorithms with 39 real-world SPLs. Our results confirm that prior algorithms are either scalable or uniform, while Smarch is both scalable and uniform.

Our work makes the following contributions:

- The Smarch algorithm (RQ1);
- A demonstration of Smarch’s ability to sample from configuration spaces up to size 10^117 (RQ2);
- Optimizations to Smarch that improve sampling time at least 12× using variable selection, parallelization, and caching (RQ3);
- An evaluation of other state-of-the-art uniform sampling algorithms that reveals Smarch can sample spaces 10^485 times larger (RQ4 and RQ5); and
- A public repository containing Smarch and its evaluation data.\(^1\)

2 Background and Motivation

We begin with basics on uniform sampling of SPL configurations, why it is both challenging and scientifically important.

2.1 Feature Model

The variability of an SPL is described by its feature model, which encodes the features of the SPL and their constraints. A feature model defines the valid configurations of a SPL.

Fig. 1a is a snippet of the feature model from the fiasco micro-kernel, simplified for illustration. Each node represents a feature; edges between nodes and the clauses beneath the graph encode constraints among features. Each edge can represent a different type of constraint where a feature is: mandatory (\(\land\)), optional (\(\lor\)), choose 1 (\(\lor\)) or choose 1 or more (\(\lor\)). For instance, fiasco requires the selection of an architecture (arch), either IA32 or ARM but not both.

2.2 Automated Solvers for SPL Configurations

Manually deriving a configuration that satisfies all constraints for even tiny feature models is error-prone and inefficient [3]. The use of automated solvers, like SAT solvers, for this purpose is standard today [21].

Automated solvers take propositional formula as an input. It is well-known that a feature model can be converted into a propositional formula \(\psi\) by a simple procedure [2, 3]. To derive \(\psi\), features are treated as variables while the constraints are converted into propositional clauses. \(\psi\) is formed by a conjunction of all the clauses [2], Fig. 1b is the propositional formula of the feature model of Fig. 1a. The top four lines of conjuncts are derived from the feature tree of Fig. 1a; the constraints below the graph in Fig. 1a are the last two lines.

Automated solvers find variable assignments that satisfy \(\psi\), which identifies a unique configuration. Fig. 1c shows solutions that can be obtained from a SAT solver. The set of all solutions to \(\psi\) is the configuration space of the SPL and the number of solutions is the size of the space.

2.3 Uniform Sampling of SPLs

Although automated solvers can produce configurations, this is insufficient for uniform sampling. Calling solvers multiple times can yield different configurations. But such configurations are biased as some features are never selected [10, 15, 30]. For example, a SAT solver is unlikely to select watchdog in Fig. 1b as clauses of the form \((\ldots \lor \text{watchdog} \Rightarrow \text{true})\) are always true. The BCP algorithm of a SAT solver concentrates on clauses that are not yet satisfied to make them satisfied [10], ignoring all other clauses.

One way to achieve uniform sampling is to enumerate all configurations and uniformly sample from them. But this is feasible for only the smallest of configuration spaces [18].

State-of-the-art approaches tried to overcome these challenges. Some derived samples by focusing on the difference of feature selections among samples [9, 18]. Samples are generated incrementally, so that a new set of samples have different feature selections from previous samples as much as possible. This makes sampling fast, but does not guarantee uniform sampling.

Algorithms that achieve uniform sampling work by partitioning the configuration space so that one of the partitions can be randomly selected for sampling [5, 32]. However, they may encode or enumerate configurations that were not being sampled as well, preventing them from scaling to large configuration spaces [18, 33].

2.4 Why is Uniform Sampling Important?

Researchers take samples from SPL configuration spaces and apply classical statistical methods to yield means with accuracy bounds (e.g., 15.7 ± 3.1 with 95% confidence). So why is uniform sampling important? Answer: Classical statistical methods assume uniform sampling as a precondition for their use; if this precondition is violated, computed statistics can be far from the truth.

Let \(\mu\) be the population mean for a property of a configuration space. By taking \(n\) samples, building the configuration for each sample, and benchmarking that property’s value, it is trivial to produce a sample mean \(m\), a standard sample deviation \(\sigma\), and a 95% confidence interval for \(\mu\) following\(^1\)
the rote procedure in [4]:

\[
(m - 1.96 \cdot \frac{\sigma}{\sqrt{n}}) \leq \mu \leq (m + 1.96 \cdot \frac{\sigma}{\sqrt{n}})
\]  

That is, \( \mu \) is within the given bounds with 95% confidence.

**Example:** Let \( \mu \) be the average number of features that are ‘true’ in a configuration. Fig. 2 shows estimates of \( \mu \) for two SPLs, using different sampling methods and sample sizes. The X-axis is \( n \) and the Y-axis shows \( \mu \) estimates. The straight line (−) is the true \( \mu \) derived from enumeration, as these SPLs are small enough to enumerate and compute the correct answer. The dashed lines indicate the 95% confidence interval for \( \mu \) estimates. ♦ plots estimates by uniform sampling. ♣ ♦ plot estimates by state-of-the-art sampling methods that do not guarantee uniform sampling. (The identities of these sampling methods are given in footnote 9).

**Figure 2.** Estimating \( \mu \) by Different Sampling Methods.

Observe:
- All seem to be converging to an answer with increasing \( n \);
- Uniform sampling correctly estimates \( \mu \) with increasing accuracy, while other sampling methods converge to different incorrect answers; and
- Method ♦ selects different sample sets each time for Fig. 2b, but oddly the same number of features is selected for all samples. We know ♦ produces unique samples, but selecting the same number of features may be a kink in its algorithm that we simply stumbled across. Regardless, the estimate by ♦ is suspicious because it has no variability.

Setting the above aside, uniform sampling has other known benefits:
- Population statistics (like \( \mu \)) can be predicted by classical probability analyses. Uniform sampling can then confirm the correctness of these predictions. See RQ1 and RQ4 for an example.
- When analytical predictions are unavailable, uniform sampling can estimate population statistics that a correct analysis should return.
- Additional observations by others are listed in Section 5.

## 3 The Smarch Algorithms

Suppose each configuration can be given a unique integer as its index. By randomly selecting an integer, we are randomly selecting the index of a configuration where all configurations have an equal likelihood of being selected. If we can associate an integer to a configuration without enumerating configurations, sampling can be done without enumeration as well.

Fig. 3 illustrates how Smarch samples a configuration in this way. Given a propositional formula \( \phi \), a #SAT solver can count the number of solutions, \(|\phi|\). Then, a *pseudo-random number generator* (PRNG), eg., Mersenne Twister algorithm [24], selects an integer \( j \) in range \( 1 \) to \(|\phi|\). Smarch uses \( \phi \) to derive a unique configuration that corresponds to \( j \).

The essence of Smarch is its one-to-one mapping between integers and configurations without having to enumerate configurations. Smarch achieves this by recursively partitioning a configuration space based on feature selections, until the partition has one configuration.

### 3.1 Basic Smarch

Smarch derives a configuration \( c_j \) corresponding to a given integer \( j \) by recursively partitioning a configuration space. Each recursion reveals whether or not a specific feature belongs to \( c_j \). This step is performed for all features in a feature model.

Let \( C \) be a configuration space and \(|C|\) its size (cardinality). Smarch starts by computing \(|C|\) given \( \phi \), the propositional formula of the SPL’s feature model. (Recall \(|\phi| = |C| \) from Section 2.2). We use sharpSAT [37], a state-of-the-art #SAT solver, to compute \(|C|\).
Let \( \mathcal{L} = (f_1, f_2, \ldots) \) be an arbitrary but fixed list of all features in a feature model. To map a randomly selected integer \( j \in [1..|\phi|] \) to a unique solution \( c_j \), Smarch does the following:

1. Feature \( f_1 \) partitions \( \phi \) into disjoint spaces \((\phi \land \neg f_1)\) and \((\phi \land f_1)\). sharpSAT counts the size of each partition as \(|\phi \land \neg f_1|\) and \(|\phi \land f_1|\).
2. If \( j \in (\phi \land \neg f_1) \) then \( f_1 \) belongs to \( c_j \) and the \((\phi \land \neg f_1)\) space is selected for recursive partitioning. Otherwise \( f_1 \) belongs to \( c_j \), \((\phi \land f_1)\) is selected for recursive partitioning and \(|\phi \land \neg f_1|\) is subtracted from \( j \) to adjust the search in \((\phi \land f_1)\).
3. Steps 1 and 2 are repeated using the remaining variables in \( \mathcal{L} \), until all features of \( c_j \) are determined. In this case, the size of the partition becomes 1.

This algorithm ensures a one-to-one mapping between integers and solutions of \( \phi \). The returned solution is always valid, because any invalid assignments result in an empty partition that cannot be selected. Assigning all variables results in a partition of size 1. In addition, two distinct random numbers cannot return the same solution, because they will end up selecting different partitions. Coupled with a uniform PRNG, this one-to-one mapping guarantees uniform sampling.

### 3.2 Smarch Optimizations

We now describe optimizations that shorten the sampling time of Smarch.

**Free Variables.** A special case makes further partitioning unnecessary: all the variables left for partitioning are free—variables without constraints. In this case, the size of the selected partition is \(2^u\), where \( u \) is the number of variables left for partitioning. Assigning any value to a free variable does not affect the validity of a solution. When this situation arises, the integer for selecting a partition is converted into a binary number of \( u \) bits. Each bit determines the value of a free variable.

**Selecting Variables to Recurse.** The run-time of Smarch is affected by two factors:

1. The number of recursions, as each recursion involves at least one sharpSAT call which is expensive; and
2. The complexity of the formula for each partition, as a more complex formula takes sharpSAT more time to count.

When a space is partitioned, the formula of a sub-space is less complicated and has a smaller number of solutions. Some features are more constraining than others. Partitioning on these features reduces the complexity of the formula which, in turn, reduces the number of subsequent recursions and sharpSAT solving time.

Smarch uses the *cube-and-conquer* (CC) algorithm of Heule et al. [16] to optimize partitioning. CC finds variables that most increase the number of reduced-but-not-satisfied clauses. Free variables, just discussed, are deferred to the end.

Note that more than one variable can be selected for partitioning, where \( n \) variables yields \( 2^n \) partitions. Selecting more variables lessens the number of recursions, but it may also waste more time on counting partitions as only one partition among them is selected for recursion. We found from experiments that two variables per recursion was optimal in general.

**Parallelization and Caching.** A key property of Smarch is that it can execute in parallel to reduce the sampling time. Once random integers are generated, translating one integer to a configuration does not affect the translation of others. This allows a set of random integers to be processed more efficiently when batched.

Sampling a configuration may revisit the partitions that were used by the prior samples. By caching the intermediate partition sizes, the expense of calling #SAT can be amortized as Smarch samples more configurations. Smarch only caches a result of #SAT if recomputing it exceeds a certain amount of time, which we tuned to 50ms based on empirical evaluation.

With parallelization, caching may not be effective as counting partitions are distributed across different processors. To mitigate this, Smarch keeps the integer assigned to each processor as close to each other as possible. Since a random integer is used to traverse recursive partitions, integers that are close to each other are more likely traverse the same partitions.

### 3.3 Optimized Smarch

Alg. 1 is the Smarch algorithm with optimizations. Smarch starts by counting the number of solutions of \( \phi \) and generating random integers (lines 2–3). These integers are then allocated to processes, so that each process can find matching configurations in parallel (lines 6–9). Each process uses the given random integers to derive matching solutions (lines 12–18).

A solution is derived for each random integer by recursive partitioning (lines 20–40). At each recursion, the algorithm checks if all unassigned variables are free (line 22). If so, their values are assigned without further partitioning (lines 23–27). If there are remaining variables that are not free (line 28), CC selects two variables (or one if there is only one left) to partition the formula (line 29). This creates a maximum of 4 partitions. Then, the size of each partition is measured

---

2 With CC, different partitions may choose different variables to create their child partitions, which means that the order of variables can be different between paths. This, however, does not hamper uniform sampling as: 1) all variables appear only once in a path, 2) CC always results in the same variables for a given partition, and 3) the order of child partitions to select based on the random number is always the same regardless of which random number is used for sampling.
(lines 30–31) and cached if needed (lines 32-33). Among the partitions, one is selected to recurse based on $j$ (lines 34–38). The recursion continues with the selected partition (line 39).

Algorithm 1: Smarch Algorithm

```plaintext
Procedure SMARCH(n, φ, p):
  Input : n (number of samples)
          φ (propositional formula)
          p (number of processes)
  Output : pSet ← set of p processes;
          total ← number of solutions for φ from sharpSAT with CC;
          rSet ← n distinct random integers from [1, total];
          samples ← new empty set;
          for each φ in pSet do
            rPart ← get ⌈n/p⌉ smallest integers from rSet;
            sSet ← p.start(SAMPLE_PROCESS(rPart, φ));
            add sSet to samples;
          return samples;
```

```plaintext
Procedure SAMPLE_PROCESS(rSet, φ):
  Input : rSet (set of random numbers)
          φ (propositional formula)
  Output : sSet (set of n samples)
  sSet ← new empty set;
  for each r in rSet do
    s ← new empty set;
    s ← SAMPLE_ONE(r, φ, s);
    add s to sSet;
  return sSet;
```

```plaintext
Procedure SAMPLE_ONE(r, φ, s):
  Input : r (random integers)
          φ (propositional formula)
  Output : s (sample = set of variable selections)
  if BCP(φ, s) is empty then
    for all variables unassigned in BCP(φ, s) do
      i ← ⌈r/2⌉; r ← r/2;
      if (i == 0) then
        add v to s;
      else
        add v to s;
  else if unassigned variables remain then
    cubes ← variable assignment sets from CC(φ ∧ s);
    for each cube in cubes do
      cs ← sharpSAT(φ ∧ s ∧ cube);
      if sharpSAT took more than 50ms then
        cache cube and cs pair;
      if (r ≤ cs) then
        add cube to s;
      break;
    else
      r ← r - cs;
    s ← SAMPLE_ONE(r, φ, s);
  return s;
```

4 Evaluation

We evaluate Smarch by sampling large, real-world SPLs and compare it with four state-of-the-art sampling algorithms. This evaluation is driven by the following research questions:

- **RQ1**: Does Smarch perform uniform sampling?
- **RQ2**: Can Smarch sample large SPLs?
- **RQ3**: Do optimizations reduce sampling time?
- **RQ4**: Which algorithms can uniformly sample SPLs?
- **RQ5**: Which sampling algorithms are the fastest?

**RQ1** through **RQ3** evaluate Smarch for both sampling performance and whether samples are uniformly distributed. To do so, we use 39 feature models that were used in prior work to evaluate SPL sampling algorithms, algorithms that are now state-of-the-art. The feature models are represented in DIMACS format, which is the standard format for SAT solvers [8]. We describe those feature models in more detail at Section 4.3. **RQ4** and **RQ5** compare Smarch with four state-of-the-art SPL sampling algorithms, which aimed to generate samples that are representative of a configuration space. To evaluate these algorithms, we applied the same evaluation method as Smarch for comparison. We describe each algorithm in more detail at Section 4.2.

To evaluate the performance of a sampling algorithm, we sampled 100 configurations from each feature model and computed the average sampling time per configuration. If sampling a single configuration took more than 5 minutes, we declared a *timeout*. To confirm a timeout, we gave each algorithm an hour to generate at least 12 samples. If not, it "timed-out".

Evaluating whether samples are uniformly distributed is less straightforward. The de-facto standard metric of uniformity from prior work requires an enumeration of the configuration space [5, 9], which is infeasible for the large SPLs we consider. Instead, we use a theoretical statistical metric that estimates the uniformity of a sample set, extending a metric used in prior work [17, 33]. We define this metric in Section 4.1.

All experiments were performed on an Intel i7-6700@3.4Ghz with 16GB of RAM running Ubuntu 16.04. All data and source code for the evaluation can be found in our repository\(^3\).

4.1 The Uniformity Index (UI)

To evaluate whether samples are uniformly distributed in a large configuration space, we propose a new metric called the *Uniformity Index (UI)*, inspired by [33].\(^4\) UI measures how accurately a sample set estimates the configuration space by comparing the distribution of feature assignments. UI represents how often a sample’s estimates are within the *margin-of-error* (ME) for each feature. If a sample set comes

---

\(^3\)Omitted for blind review

\(^4\)Our definition of UI is different than [33], but the base equation (2) is the same.
from a uniform sampling algorithm, UI will be approximately \(\approx 95\%\). Here is how the UI is computed.

Again, \(\phi\) is the propositional formula of an SPL’s feature model. Let \(v\) denote a variable (feature) from the set of all variables (features) \(V\) in \(\phi\), and let \(r_v\) denote the fraction of all valid configurations where \(v=\text{true}\). \(r_v\) is computed by two calls to a \#SAT solver:

\[
r_v = \frac{|\phi \land v|}{|\phi|} \quad (2)
\]

where \(|\phi|\) and \(|\phi \land v|\) are the number of solutions to \(\phi\) and \((\phi \land v)\).

We can estimate \(r_v\) by sampling. Take \(N\) samples and let \(k\) be the number of samples where \(v=\text{true}\). \(s_v\) is the estimation of \(r_v\):

\[
s_v = \frac{k}{N} \quad (3)
\]

If the samples are uniformly distributed and 
\(N\) is large enough, the Central Limit Theorem tells us that \(s_v\) is approximately normally distributed. This allows us to derive the 95% margin of error (ME) for the sample mean \(s_v\) [4]:

\[
|s_v - r_v| \leq ME_v, \quad \text{where } ME_v = 1.96 \times \sqrt{\frac{r_v(1-r_v)}{N}} \quad (4)
\]

Here is what (4) says: with 95% probability, the difference between \(s_v\) and \(r_v\) is smaller than \(ME_v\). If we measure \(s_v\) for all variables in \(V\), we expect 95% of them to satisfy (4), regardless of \(\phi\).

Here is how we used \(ME_v\) to check if samples uniformly distributed. We sampled \(N=100\) configurations from each \(\phi\) and used those samples to compute \(s_v\) for all \(v \in V\). Then, we derived UI (the Uniformity Index) as the percentage of \(v \in V\) that satisfies (4):

\[
\text{UI} = \frac{\sum_{v \in V} \begin{cases} 1 & \text{if } |s_v - r_v| \leq ME_v; \\ 0 & \text{otherwise} \end{cases}}{|V|} \times 100 \quad (5)
\]

UI will be very close to 95% when samples are uniformly distributed.

### 4.2 Evaluated Algorithms

We evaluated Smarch with 4 state-of-the-art SPL sampling algorithms whose aim is to generate samples that are representative for a given configuration space.

**Smarch**. We evaluated both the unoptimized and optimized versions of Smarch, **Smarch_base** and **Smarch_opt** respectively. Smarch_opt was given seven of our machine’s eight cores for parallel execution.

**Counting Binary Decision Diagram (CBDD)**. [32] is a uniform sampling algorithm based on Binary Decision Diagrams (BDDs), which represents the feature model in a tree structure that encodes all valid configurations. CBDD, like Smarch, creates a one-to-one mapping from numbers to configurations. Unlike Smarch, CBDD requires a BDD to be created prior to sampling.

**Unigen2**. [5] is a uniform sampling algorithm. Unigen2 partitions a solution space evenly as possible using a hashing function and an approximate \#SAT solver [6]. When partitions are small enough, a partition is randomly selected and solutions in the partition are enumerated for random selection. Unigen2 also supports parallelism on sampling, and we give it seven threads as with Smarch_opt.

Unigen2 samples configurations in multiples of 11, so we took 99 samples instead of 100 and divided the Unigen2 run time by 99.

**Diversified Distance-based Sampling (DDbS)**. [18] is a sampling algorithm for deriving performance models of SPLs. Its authors considered uniform sampling infeasible and proposed DDbS as an alternative. DDbS treats configurations as vectors where their elements are feature selections, and derives samples with a maximum distance from previous samples.

DDbS requires its input to be in SPLConqueror\(^6\) feature model format (.xml) instead of DIMACS format. We used the functionality of SPLConqueror to convert a DIMACS file into the required feature model format; and

**QuickSampler (QS)**. [9] is a sampling algorithm to generate a large number of samples for testing SPLs. First, it randomly selects features and generates configurations that are close to the selection using a MaxSAT solver.\(^7\) Then, the generated configurations are mutated with each other to create new samples by calling MaxSAT solver on them.

QS does not have a function to generate a requested number of valid configurations, as we do not know how many samples are valid among them. Even when QS generates a large number of samples, the number of valid samples among them can be less than the requested number. To compensate, we generated samples so that the number of valid samples is \(\geq 100\). The time per sample was the total run time divided by the number of valid samples.

### 4.3 Subject Feature Models

We used 39 feature models that were used to evaluate the algorithms in Section 4.2, represented in DIMACS format [17, 18, 33]:

---

\(^5\)Equation (4) is closely related to (1) from Section 2.4, with an important distinction. Equation (1) is based on the sample mean and standard deviation, while Equation (4) is the population (configuration space) mean and standard deviation. The sample mean and standard deviation are estimations of the population mean and standard deviation when the complete population is not or cannot be known. UI is able to use the population statistics \(r_v\), enabling a more exacting test of uniformity [4].

\(^6\)https://github.com/se-passau/SPLConqueror

\(^7\)A MaxSAT solver differs from \#SAT in that it finds a set of variable assignments that satisfies as many clauses as possible, even though those assignments may not lead to a legal configuration.
• 10 feature models from Kaltenecker et al.’s evaluation of their 
DdbS algorithm [18]. Three among them were also used to evaluate the CBDD algorithm;
• 24 feature models from Plazar et al.’s evaluation of the
Unigen2 and QS [33]; and
• 5 feature models of Kconfig-based systems [17].
These feature models are from a range of software, including
systems software (kernels, compilers, etc.) and applications
(databases, compression, etc.).
While Plazar et al. provides 119 feature models, we se-
lected a subset of models with maximal variety, as models
with similar number of variables and clauses yielded simi-
lar results. To select such subset, we grouped formulas into
those with less than a 5% difference in the number of vari-
able and clauses. Then we selected the one with the most
variables and clauses from each group.

4.4 Evaluation Results
Table 2 (next page) shows the UI values from 100 samples for
all 39 formulas. The rows represent the 39 feature models,
sorted by size of the configuration space (|C|) from smallest
to largest. The first column is the name of the feature model,
while columns two through four are the numbers of variables
(|V|), clauses (|C1|), and (|C|), respectively. The remainder of
columns show the UI values for each algorithm.
Formula and sampling method pairs that timed out were
not used for evaluation and were marked as not available
(N/A). Formula embtoolkit and formulas at the bottom 5
rows of Table 2 were also not evaluated. We could not derive
r_v for them as sharpSAT could not count them or it took too
much time. Unigen2 resulted in an error when partitioning
LLVM’s formula.
Table 3 shows the average time to sample a configuration
for all 39 formulas by each sampling algorithm, measured in
milliseconds. The rows and columns are the same as Table 2,
extcept it measures the sampling time instead of the UI value.

RQ1: Does Smarch perform uniform sampling?
The two right-most columns of Table 2 show that UI values
from Smarch_base and Smarch_opt samples are close to 95%
for all formulas that didn’t time-out. These results indicate
that Smarch_base and Smarch_opt samples are uniformly
distributed.
For a deeper evaluation, we checked if the results are
consistent for different sample sizes as well. To do so, we
derived UI values from 100, 500, and 1000 samples. Further,
we were also interested in whether s_v becomes more accurate
as the number of samples increases, which is expected by
the Law of Large Numbers [4]. We computed the sample
mean error e_v for each variable v, which is a component of
UI (Equation (4)), and computed its average e_\phi:

\[ e_v = |s_v - r_v| = \frac{k}{N} - r_v \]  \hspace{1cm} (6)
\[ e_\phi = \frac{1}{|V|} \sum_{v \in V} e_v \]  \hspace{1cm} (7)

Table 1 shows the result from six feature models with varying
|V|, |C1|, and |C|. From these experiments, we observed:
• UI was close to 95% for all formulas and sample sizes;
and
• e_\phi decreased with increasing sample sizes for all formu-
las.

**Conclusion:** Both Smarch_base and Smarch_opt perform
uniform sampling, and increasing the number of samples
increases statistical accuracy as predicted.

| Prop. Formula (|V| / |C1| / |C|) | N | UI   | e_\phi |
|-----------------------------|---|------|-------|
| VP9                        | 100 | 95.2%| 0.026 |
| (42 / 104 / 2.16E+05)      | 500 | 95.2%| 0.011 |
|                            | 1000| 95.2%| 0.009 |
| fiasco_17_10               | 100 | 97.9%| 0.014 |
| (234 / 1178 / 1.00E+10)    | 500 | 98.2%| 0.006 |
|                            | 1000| 95.7%| 0.005 |
| uClibc-ng_cache-8          | 100 | 97.0%| 0.022 |
| (269 / 1403 / 8.00E+26)    | 500 | 97.0%| 0.011 |
|                            | 1000| 95.5%| 0.009 |
| toybox_0_7_5               | 100 | 95.9%| 0.036 |
| (316 / 106 / 1.40E+81)     | 500 | 95.9%| 0.016 |
|                            | 1000| 96.2%| 0.011 |
| pati                       | 100 | 97.7%| 0.020 |
| (1248 / 3266 / 7.90E+126)  | 500 | 96.9%| 0.010 |
|                            | 1000| 96.4%| 0.007 |
| busybox_1_28_0             | 100 | 95.5%| 0.037 |
| (998 / 962 / 1.30E+248)    | 500 | 95.6%| 0.016 |
|                            | 1000| 95.1%| 0.012 |

RQ2: Can Smarch sample large SPLs?
The last (right-most) column of Table 3 (next page) shows
the average time taken to sample a configuration by Smarch_opt,
in milliseconds. Fig. 4 (below) plots the time taken to sample
a configuration from different formulas by |V|, |C1|, and |C|.
The dotted line indicates the linear regression for each graph.
Table 2. Uniformity Indices (UI) for each each sampling algorithm using 100 samples from each propositional formula.

<table>
<thead>
<tr>
<th>Prop. Formula</th>
<th></th>
<th></th>
<th>CBDD</th>
<th>Unigen2</th>
<th>DDBS</th>
<th>QS</th>
<th>Smarch_base</th>
<th>Smarch_opt</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip</td>
<td>20</td>
<td>63</td>
<td>1.44E+02</td>
<td>100%</td>
<td>100%</td>
<td>45.0%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>LLVM</td>
<td>11</td>
<td>1</td>
<td>1.02E+03</td>
<td>100%</td>
<td>N/A</td>
<td>90.9%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>X264</td>
<td>16</td>
<td>11</td>
<td>1.15E+03</td>
<td>100%</td>
<td>100%</td>
<td>87.5%</td>
<td>56.3%</td>
<td>100%</td>
</tr>
<tr>
<td>Dune</td>
<td>17</td>
<td>16</td>
<td>2.34E+03</td>
<td>100%</td>
<td>100%</td>
<td>88.2%</td>
<td>70.6%</td>
<td>100%</td>
</tr>
<tr>
<td>BerkeleyDBC</td>
<td>18</td>
<td>29</td>
<td>2.56E+03</td>
<td>100%</td>
<td>100%</td>
<td>77.8%</td>
<td>61.1%</td>
<td>100%</td>
</tr>
<tr>
<td>HiAcc</td>
<td>31</td>
<td>104</td>
<td>1.35E+02</td>
<td>69.8%</td>
<td>100%</td>
<td>61.3%</td>
<td>25.8%</td>
<td>100%</td>
</tr>
<tr>
<td>JHipster</td>
<td>45</td>
<td>104</td>
<td>2.63E+02</td>
<td>97.8%</td>
<td>95.6%</td>
<td>46.7%</td>
<td>46.7%</td>
<td>95.6%</td>
</tr>
<tr>
<td>Polly</td>
<td>40</td>
<td>100</td>
<td>4.00E+02</td>
<td>100%</td>
<td>97.5%</td>
<td>35.0%</td>
<td>57.5%</td>
<td>95.0%</td>
</tr>
<tr>
<td>7z</td>
<td>55</td>
<td>210</td>
<td>4.49E+02</td>
<td>95.5%</td>
<td>100%</td>
<td>38.6%</td>
<td>36.4%</td>
<td>97.7%</td>
</tr>
<tr>
<td>JavaGC</td>
<td>39</td>
<td>105</td>
<td>7.15E+02</td>
<td>100%</td>
<td>97.5%</td>
<td>35.9%</td>
<td>53.8%</td>
<td>97.4%</td>
</tr>
<tr>
<td>7z</td>
<td>44</td>
<td>210</td>
<td>6.86E+02</td>
<td>95.5%</td>
<td>100%</td>
<td>38.6%</td>
<td>36.4%</td>
<td>97.7%</td>
</tr>
<tr>
<td>LLVM</td>
<td>11</td>
<td>1</td>
<td>1.02E+03</td>
<td>100%</td>
<td>N/A</td>
<td>90.9%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>X264</td>
<td>16</td>
<td>11</td>
<td>1.15E+03</td>
<td>100%</td>
<td>100%</td>
<td>87.5%</td>
<td>56.3%</td>
<td>100%</td>
</tr>
<tr>
<td>Dune</td>
<td>17</td>
<td>16</td>
<td>2.34E+03</td>
<td>100%</td>
<td>100%</td>
<td>88.2%</td>
<td>70.6%</td>
<td>100%</td>
</tr>
<tr>
<td>BerkeleyDBC</td>
<td>18</td>
<td>29</td>
<td>2.56E+03</td>
<td>100%</td>
<td>100%</td>
<td>77.8%</td>
<td>61.1%</td>
<td>100%</td>
</tr>
<tr>
<td>HiAcc</td>
<td>31</td>
<td>104</td>
<td>1.35E+02</td>
<td>69.8%</td>
<td>100%</td>
<td>61.3%</td>
<td>25.8%</td>
<td>100%</td>
</tr>
<tr>
<td>JHipster</td>
<td>45</td>
<td>104</td>
<td>2.63E+02</td>
<td>97.8%</td>
<td>95.6%</td>
<td>46.7%</td>
<td>46.7%</td>
<td>95.6%</td>
</tr>
<tr>
<td>Polly</td>
<td>40</td>
<td>100</td>
<td>4.00E+02</td>
<td>100%</td>
<td>97.5%</td>
<td>35.0%</td>
<td>57.5%</td>
<td>95.0%</td>
</tr>
<tr>
<td>7z</td>
<td>55</td>
<td>210</td>
<td>4.49E+02</td>
<td>95.5%</td>
<td>100%</td>
<td>38.6%</td>
<td>36.4%</td>
<td>97.7%</td>
</tr>
<tr>
<td>JavaGC</td>
<td>39</td>
<td>105</td>
<td>7.15E+02</td>
<td>100%</td>
<td>97.5%</td>
<td>35.9%</td>
<td>53.8%</td>
<td>97.4%</td>
</tr>
<tr>
<td>7z</td>
<td>44</td>
<td>210</td>
<td>6.86E+02</td>
<td>95.5%</td>
<td>100%</td>
<td>38.6%</td>
<td>36.4%</td>
<td>97.7%</td>
</tr>
</tbody>
</table>

We observed:

- The largest configuration space Smarch_opt could sample was uClinix-config whose \(|C| = 10^{417}\), taking approximately 42 seconds per sample;
- Smarch_opt was able to sample all formulas that sharpSAT could count;
- Except embtoolkit which had the largest \(|V|\) and \(|Cl|\), Smarch_opt was able to sample a configuration in under 42 seconds;
- Fig. 4 suggests that linearly increasing \(|V|\), \(|Cl|\), and \(\log_{10}|C|\) yields an approximately linear increase in sampling time.

We believe that increasing the number of processes for parallelization can reduce sampling time further. Sampling more configurations can also reduce sample time, as more partitions can be cached for reuse.

**Conclusion:** Smarch_opt is scalable up to \(|C| = 10^{417}\), which took less than 42 seconds per sample (uClinix-config of Table 3).

**RQ3:** Do optimizations reduce sampling time?

The ninth column (or second-from-right) of Table 3 shows the time taken per sample by Smarch_base. The last (right-most) column is the average time taken per sample by Smarch_opt. By comparing their sampling times, we observed:

- For the formulas that both Smarch_base and Smarch_opt could sample without timeout, Smarch_opt was 2050× faster than Smarch_base on average and minimally 12× faster (HiAcc);
- An extreme case was uLinux, where Smarch_opt was 53671× faster. Smarch_opt did not partition the formula as all variables of uLinux are either free, fixed as true,
or fixed as false. Ignoring uClinux, Smarch_opt was 64× faster than Smarch_base on average;

- For the formulas that Smarch_opt could sample, Smarch_base timed-out on the 7 formulas with the largest |V| and |C1|; and
- The largest |C| that Smarch_opt could sample without timeout was $10^{165}$× larger than what Smarch_base could sample without timeout.

As Smarch_opt sampled with 7 processes in parallel, we can expect Smarch_opt to be at least 7× faster than Smarch_base by parallelization in theory. However, the minimum observed speedup was greater than 7× (i.e., it was 12×), indicating that the selection of variables and caching also contributed to reducing sampling time.

**Conclusion:** Optimizations made sampling at least >12× faster.

### RQ4: Which algorithms can uniformly sample SPLs?

Columns 5-8 of Table 2 (previous page) shows the UI values from 100 samples collected for 4 state-of-the-art sampling approaches. We observed:

- Neither CBDD, Unigen2, nor DDbS were able to sample a formula with |C| > $10^{13}$;
- UI of CBDD and Unigen2 were close to 95% for the formulas that they could sample in time; and
- For DDbS, UI ranged from 35.0% to 90.9%, with an average of 59.1%. Among them, only LLVM exceeded 90%.
- For QS, UI was 100% for the two smallest SPLs, lzip and LLVM. For the remaining, UI ranged widely from 5.1% to 70.6%, with an average of 42.5%.

<table>
<thead>
<tr>
<th>Prop. Formula</th>
<th></th>
<th></th>
<th></th>
<th>CBDD</th>
<th>Unigen2</th>
<th>DDbS</th>
<th>QS</th>
<th>Smarch_base</th>
<th>Smarch_opt</th>
</tr>
</thead>
<tbody>
<tr>
<td>lzip</td>
<td>20</td>
<td>63</td>
<td>1.44E+02</td>
<td>&lt; 1</td>
<td>&lt; 1</td>
<td>4</td>
<td>&lt; 1</td>
<td>181</td>
<td>10</td>
</tr>
<tr>
<td>LLVM</td>
<td>11</td>
<td>1</td>
<td>1.02E+03</td>
<td>&lt; 1</td>
<td>N/A</td>
<td>2</td>
<td>&lt; 1</td>
<td>105</td>
<td>&lt; 1</td>
</tr>
<tr>
<td>X64</td>
<td>16</td>
<td>11</td>
<td>1.15E+03</td>
<td>&lt; 1</td>
<td>&lt; 1</td>
<td>4</td>
<td>&lt; 1</td>
<td>141</td>
<td>&lt; 1</td>
</tr>
<tr>
<td>Dune</td>
<td>17</td>
<td>16</td>
<td>2.34E+03</td>
<td>&lt; 1</td>
<td>&lt; 1</td>
<td>4</td>
<td>&lt; 1</td>
<td>162</td>
<td>11</td>
</tr>
<tr>
<td>BerkeleyDBC</td>
<td>18</td>
<td>29</td>
<td>2.56E+03</td>
<td>&lt; 1</td>
<td>&lt; 1</td>
<td>5</td>
<td>&lt; 1</td>
<td>158</td>
<td>4</td>
</tr>
<tr>
<td>HiPAcc</td>
<td>31</td>
<td>104</td>
<td>1.35E+03</td>
<td>1</td>
<td>&lt; 1</td>
<td>9</td>
<td>&lt; 1</td>
<td>327</td>
<td>28</td>
</tr>
<tr>
<td>JHipster</td>
<td>45</td>
<td>104</td>
<td>2.63E+03</td>
<td>&lt; 1</td>
<td>&lt; 1</td>
<td>443</td>
<td>&lt; 1</td>
<td>384</td>
<td>13</td>
</tr>
<tr>
<td>Polly</td>
<td>40</td>
<td>100</td>
<td>4.00E+04</td>
<td>1</td>
<td>1</td>
<td>259</td>
<td>&lt; 1</td>
<td>368</td>
<td>24</td>
</tr>
<tr>
<td>7z</td>
<td>44</td>
<td>210</td>
<td>6.86E+04</td>
<td>1</td>
<td>1</td>
<td>180</td>
<td>&lt; 1</td>
<td>471</td>
<td>38</td>
</tr>
<tr>
<td>JavaGC</td>
<td>39</td>
<td>105</td>
<td>1.93E+05</td>
<td>1</td>
<td>1</td>
<td>1321</td>
<td>1</td>
<td>373</td>
<td>26</td>
</tr>
<tr>
<td>VP9</td>
<td>42</td>
<td>104</td>
<td>2.16E+05</td>
<td>3</td>
<td>2</td>
<td>2724</td>
<td>&lt; 1</td>
<td>586</td>
<td>32</td>
</tr>
<tr>
<td>fiasco_17_10</td>
<td>254</td>
<td>1178</td>
<td>1.00E+10</td>
<td>timeout</td>
<td>108</td>
<td>timeout</td>
<td>2</td>
<td>3474</td>
<td>38</td>
</tr>
<tr>
<td>axTLS_2_1_4</td>
<td>94</td>
<td>190</td>
<td>2.00E+12</td>
<td>timeout</td>
<td>62541</td>
<td>timeout</td>
<td>&lt; 1</td>
<td>889</td>
<td>62</td>
</tr>
<tr>
<td>fiasco</td>
<td>1638</td>
<td>5228</td>
<td>3.58E+14</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>26</td>
<td>70069</td>
<td>302</td>
</tr>
<tr>
<td>toybox</td>
<td>544</td>
<td>1020</td>
<td>1.45E+17</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>1</td>
<td>7627</td>
<td>152</td>
</tr>
<tr>
<td>axtls</td>
<td>684</td>
<td>2155</td>
<td>4.29E+20</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>3</td>
<td>15078</td>
<td>254</td>
</tr>
<tr>
<td>uClibc-ng_1_0_29</td>
<td>269</td>
<td>1403</td>
<td>8.00E+26</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>1</td>
<td>4548</td>
<td>233</td>
</tr>
<tr>
<td>toybox_0_7_5</td>
<td>316</td>
<td>106</td>
<td>1.40E+81</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>&lt; 1</td>
<td>3480</td>
<td>88</td>
</tr>
<tr>
<td>uClinux</td>
<td>1850</td>
<td>2468</td>
<td>1.63E+91</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>1</td>
<td>46599</td>
<td>1</td>
</tr>
<tr>
<td>ref955</td>
<td>1218</td>
<td>3099</td>
<td>8.20E+123</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>4</td>
<td>44020</td>
<td>1149</td>
</tr>
<tr>
<td>adderII</td>
<td>1276</td>
<td>3206</td>
<td>3.57E+125</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>4</td>
<td>56073</td>
<td>1356</td>
</tr>
<tr>
<td>ecos-icse11</td>
<td>1244</td>
<td>3146</td>
<td>4.97E+125</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>6</td>
<td>1999</td>
<td></td>
</tr>
<tr>
<td>m5272c3</td>
<td>1323</td>
<td>3297</td>
<td>2.37E+125</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>4</td>
<td>52481</td>
<td>1399</td>
</tr>
<tr>
<td>pati</td>
<td>1248</td>
<td>3266</td>
<td>7.90E+125</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>3</td>
<td>44542</td>
<td>1162</td>
</tr>
<tr>
<td>olpecz2294</td>
<td>1274</td>
<td>3881</td>
<td>8.19E+126</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>4</td>
<td>62845</td>
<td>2167</td>
</tr>
<tr>
<td>integrator_arm9</td>
<td>1267</td>
<td>50606</td>
<td>4.06E+129</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>8</td>
<td>20169</td>
<td></td>
</tr>
<tr>
<td>at91sam7sek</td>
<td>1319</td>
<td>3963</td>
<td>9.45E+131</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>4</td>
<td>53257</td>
<td>1987</td>
</tr>
<tr>
<td>se77x9</td>
<td>1319</td>
<td>49937</td>
<td>1.20E+135</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>11</td>
<td>13321</td>
<td></td>
</tr>
<tr>
<td>phycore229x</td>
<td>1360</td>
<td>4026</td>
<td>1.77E+136</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>4</td>
<td>63972</td>
<td>2376</td>
</tr>
<tr>
<td>busybox-1.18.0</td>
<td>6796</td>
<td>17836</td>
<td>8.50E+216</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>20</td>
<td>12391</td>
<td></td>
</tr>
<tr>
<td>busybox_1_28_0</td>
<td>998</td>
<td>962</td>
<td>1.30E+248</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>1</td>
<td>18365</td>
<td>982</td>
</tr>
<tr>
<td>embtoolkit</td>
<td>23516</td>
<td>180511</td>
<td>2.10E+252</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>2677</td>
<td>157077</td>
<td></td>
</tr>
<tr>
<td>freebsd-icse11</td>
<td>1396</td>
<td>62163</td>
<td>8.39E+313</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>12</td>
<td>33366</td>
<td></td>
</tr>
<tr>
<td>uClinux-config</td>
<td>11254</td>
<td>31637</td>
<td>7.78E+417</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>47</td>
<td>41557</td>
<td></td>
</tr>
<tr>
<td>buildroot</td>
<td>14910</td>
<td>45603</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>355</td>
<td>timeout</td>
<td></td>
</tr>
<tr>
<td>freetz</td>
<td>31012</td>
<td>102705</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>695</td>
<td>timeout</td>
<td></td>
</tr>
<tr>
<td>2.6.28.6-icse11</td>
<td>6888</td>
<td>343944</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>108</td>
<td>timeout</td>
<td></td>
</tr>
<tr>
<td>2.6.32-2var</td>
<td>60072</td>
<td>268223</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.6.33.3-2var</td>
<td>62482</td>
<td>273799</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td>timeout</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
UI of LLVM was high for DDbS and QS as LLVM has a very
simple feature model with only 10 free variables and 1 var-
iable fixed as true. UI of lrizp from QS was 100% as it
sampled all configurations in the configuration space.

These results indicate that DDbS and QS do not return
uniform samples. Both generate samples without considering
the configuration space and focus on the difference of feature
selections among the samples. This result is also consistent
with their analysis, where they showed that the samples may
not be uniformly distributed [9, 18].

CBDD and Unigen2 samples were uniformly distributed.
However, they had limited scalability, which we further ex-
plain in RQ5.

**Conclusion:** DDbS and QS do not guarantee uniform sam-
pling, while CBDD and Unigen2 do produce uniform sam-
pies but have limited scalability.8

**RQ5: Which sampling algorithms are the fastest?**

Columns 5–8 of Table 3 show the time to sample a config-
uration for the 4 state-of-the-art approaches in milliseconds.
We observed:

- **CBDD** was faster than Smarch_opt for formulas with |C|<10^{10};
  but could not sample formulas with |C|>10^{10};
- **Unigen2** was faster than Smarch_opt for formulas with
  |C|<10^{10} but could not sample formulas with |C|>10^{13};
- **DDbS** was faster than Smarch_opt for formulas with |V|<31,
  but could not sample formulas with |C|>10^{18};
- **QS** was the fastest overall, and could sample formulas
  that sharpSAT could not count;
- For the formulas that CBDD, Unigen2, and DDbS timed-
  out, they were unable to sample a single configuration
  within an hour. This is different from the timeout of
  Smarch_base, which was able to sample at least one con-
  figuration as long as sharpSAT could count; and
- **Formulas 2.6.32-2var and 2.6.33.3-2var** were infeasible
  for all sampling methods.

Sampling by CBDD was fast once a BDD is generated, as
sampling requires a mere traversal of the BDD without further
computation. However, creating a BDD has limited scalability
[1], which was not able to finish in time for formulas with
|C|>10^{10}.

For Unigen2, partitioning was faster than Smarch_opt as
it relies on an approximate #SAT solver which ran faster
than sharpSAT. However, as it enumerates the partitions to
sample configurations from them, sampling time increased
greatly as |C| increased, and became unusable with |C|>10^{13}.

The sampling time of DDbS increases as the number of
variables and clauses increase overall, since more variables
require more candidate samples to be considered for measur-
ing their distance. For formulas with |C|>10^{10}, we believe

that the large number of |V| and |C| makes formulas too
complicated for DDbS to sample.

QS was faster and more scalable than all other algorithms
as it involves neither counting the configurations as CBDD,
Unigen2, and Smarch_opt do, or checking the difference of
samples as DDbS does.

**Conclusion:** Smarch_opt achieves scalable uniform sam-
pling at the cost of more computation time.

**4.5 Summary**

Table 4 summarizes our evaluation. 'DNF' means 'did not
finish'.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Uniform Sampling</th>
<th>Speed Rank for</th>
<th>Speed Rank for</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBDD</td>
<td>Yes</td>
<td>2nd</td>
<td>DNF</td>
</tr>
<tr>
<td>Unigen2</td>
<td>Yes</td>
<td>2nd</td>
<td>DNF</td>
</tr>
<tr>
<td>DDbS</td>
<td>No</td>
<td>5th</td>
<td>DNF</td>
</tr>
<tr>
<td>QS</td>
<td>No</td>
<td>1st</td>
<td>1st</td>
</tr>
<tr>
<td>Smarch_opt</td>
<td>Yes</td>
<td>4th</td>
<td>2nd</td>
</tr>
</tbody>
</table>

CBDD and Unigen2 performed uniform sampling and were
faster than Smarch_opt for formulas with |C|<10^{10}. They
were unusable for formulas where |C|>10^{13}.

Neither DDbS and QS delivered uniform sampling, although
QS was very fast and scalable. In contrast, Smarch_opt achieves
uniform sampling at the cost of longer sampling time.

**Conclusion:** Smarch_opt is the most scalable uniform
sampling algorithm that currently exists for SPLs.9

**4.6 Threats to Validity**

**Internal Validity.** On evaluating the time taken to sample
a configuration, we averaged the time to sample a config-
uration from 100 samples, which provides approximately
10% margin of error on the average. We used the time mea-
surement from the sampling tool if the tool provided such
functionality. For all other cases, we used the Linux 'time'
tool.

When evaluating uniform sampling, we used different
formulas with varying number of samples to control the
randomness. Although deriving UI from a single set of 100
samples may not guarantee whether uniform sampling was
achieved, it is sufficient to show whether the uniform sam-
pling is not achieved.

**External Validity.** We used 39 real-world feature models
from different domains and different |V|, |C|, and |C|, used
by others to evaluate their sampling algorithm. We are aware
that the trend of sampling time in Fig. 4 and the uniform dis-
stribution of samples may not generalize to all SPLs. At least,
our evaluation gives strong evidence that our conclusion should be consistent for many SPLs.

5 Related Work

SPL Approaches that use Random Sampling. Sampling configurations is a fundamental part of many recent SPL analyses.

On deriving a performance model to predict the performance of configurations, Sarkar et al. [34] used Classification and Regression Tree as a performance model, which is derived from uniform sampling. Siegmund et al. [36] derived a performance model using linear regression from samples. Note that, as they considered uniform sampling infeasible, they sampled configurations by pre-selecting the features to analyze. Kaltenecker et al. [18] extended their work, proposing DBbS as an alternative to uniform sampling.

On finding optimal configurations, Oh et al. [32] used CBDD to find near-optimal configurations by recursively uniform sampling a configuration space with feature selections. Henard et al. [15], Sayyad et al. [35], and Guo et al. [13] used evolutionary algorithms to find optimal configurations with competing objectives, where random sampling was used to gather initial set of configurations to evolve. Chen et al. [7] recursively partitioned a large set of random samples to find an optimal configuration among them with minimal evaluation of samples as possible.

On finding variability bugs, Liebig et al. [22] compared different program analysis approaches for SPLs. They tried to sample configurations by randomly selecting features, which did not yield a valid configuration. Medeiros et al. [25] compared different sampling approaches for finding bugs, where uniform sampling was used as the baseline. Melo et al. [26] sampled configurations of Linux kernel and analyzed warnings aggregated from those samples. Mordahl et al. [27] analyzed the distribution of variability bugs in several Kconfig-based systems software, using Smarch to generate uniform samples for statistical accuracy.

Random Sampling of SPLs. In addition to the state-of-the-art sampling approaches described in Section 4, we present other sampling approaches that were used by the prior SPL analyses.

A number of papers [7, 13, 15] used automated solvers to generate configurations. Since automated solvers may yield biased samples, they tried to randomize the variable assignment within the solver or randomize the order of the clauses that the solver process. Medeiros et al. [25] randomly selected features, and discarded the sample if it is invalid. Liebig et al. [22] also tried to sample by randomly selecting the features, but sampling one million configurations from Linux kernel in this manner could not yield a valid configuration. Melo et al. [26] used randconfig, a configuration tool provided for Linux kernels, which works in a similar manner as automated solvers [17]. Sarkar et al. [34] achieved uniform sampling by enumerating the configuration space and randomly sampling configurations among them.

These approaches either could not guarantee uniform sampling or achieved it with limited scalability [18]. Plazar et al. [33] evaluated Unigen2 and QS regarding the sampling time and distribution of samples. In their analysis, Unigen2 was not able to sample from configuration spaces larger than $10^{18}$, although Unigen2 was able to sample from a propositional formula $\phi$ that was not of an SPL feature model, where $\phi=10^{48}$. QS was scalable, but its samples were not uniformly distributed. Our evaluations confirm their work.

Nevertheless, uniform sampling has shown its utility on these analyses. Kaltenecker et al. [18] demonstrated that uniform sampling yields the most accurate performance model among other SPL sampling methods. Oh et al. demonstrated that uniform sampling allows statistical bounds on the accuracy of the search for a given sample size [32]. Search results of evolutionary algorithms improved by 10% when the initial samples were from uniform sampling [23]. For testing SPLs, uniform sampling was considered as the baseline for comparing other sampling algorithms [14, 25, 33].

Automated Solvers for SPLs. Different solvers have different capabilities. SAT solvers can find a valid configuration for a feature model with Boolean features [21]; Satisfiability Modulo Theory (SMT) solvers and Constraint Programming (CP) solvers can find configurations for feature models with numerical features [13, 29]. BDDs compile a propositional formula into a tree structure which encodes all solutions of $\phi$. #SAT solvers are dedicated to count the number of solutions of a propositional formula more efficiently than SAT, SMT, or CP solvers [30, 32].

Uniform Sampling Solutions of Propositional Formulas. Besides Unigen2, Gogate et al. [11] used Markov Chain to select the variables of a sample based on its estimated probability. Kitchen et al. [19, 20] used Markov Chain Monte Carlo algorithms to progressively generate random solutions. Nadel et al. [31] tried to modify the heuristics of an SAT solver to generate more random solutions. These approaches, however, were not able to guarantee uniform distribution of the samples [5].

6 Conclusions

Analyzing SPL configuration spaces can greatly benefit from classical statistical methods. The number of SPL configurations is often too large to enumerate, while the properties of each configuration are often unknown before actually executing the configuration, i.e., building and benchmarking configurations are needed. In this situation, statistical analyses enable cost-efficient approaches to manage and explore large configuration spaces. Uniform sampling is a proven scientific way to accomplish this.
In this paper we presented Smarch, a new algorithm that can uniformly sample configurations of an SPL configuration space with greater scalability than existing methods. Smarch relies on a one-to-one mapping between integers and configurations, so that uniform sampling of an integer results in uniform sampling of a configuration. Smarch is an improvement over existing algorithms as it does not generate or encode configurations that are not used as samples. Further, optimizations increase its sampling speed by at least 12x, and Smarch may get faster with further parallelization.

In short, Smarch was able to uniform sample configuration spaces as large as $10^{417}$, which is $10^{605}$ times larger than what previous uniform sampling algorithms could sample.

We believe Smarch opens up new opportunities for classical statistical and probabilistic analyses on real-world SPLs. Future work includes improving Smarch’s sampling time even further and applying Smarch’s uniform sampling to increase the accuracy of other SPL analyses.

References


