Efficient Sampling of SAT Solutions for Testing

Rafael Dutra, Kevin Laeufer, Jonathan Bachrach and Koushik Sen
EECS Department
University of California, Berkeley, USA
{rtd,laeufer,jrb,ksen}@cs.berkeley.edu

ABSTRACT
In software and hardware testing, generating multiple inputs which satisfy a given set of constraints is an important problem with applications in fuzz testing and stimulus generation. However, it is a challenge to perform the sampling efficiently, while generating a diverse set of inputs which satisfy the constraints. We developed a new algorithm QuickSampler which requires a small number of solver calls to produce millions of samples which satisfy the constraints with high probability. We evaluate QuickSampler on large real-world benchmarks and show that it can produce unique valid solutions orders of magnitude faster than other state-of-the-art sampling tools, with a distribution which is reasonably close to uniform in practice.

CCS CONCEPTS
• Software and its engineering → Software testing and debugging; Formal software verification; • Hardware → Theorem proving and SAT solving;

KEYWORDS
sampling, stimulus generation, constraint-based testing, constrained-randowm verification

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden © 2018 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5638-1/18/05...$15.00
https://doi.org/10.1145/3180155.3180248

1 INTRODUCTION
Given a set of constraints, the problem of generating a set of random solutions to the constraints is important both in software and hardware testing and verification. For instance, conventional symbolic execution [13, 25] and dynamic symbolic execution techniques [1–4, 7, 8, 12, 18, 19, 24, 28, 35–40, 42] generate a path constraint for each prefix of feasible execution paths in a program and use a SMT-solver to generate a solution for each such constraint. However, in practice, these techniques face scalability problems because the number of paths for any reasonable program is astronomically large. Instead of generating a single solution for the path constraint of a path prefix, one could generate multiple solutions to randomly test multiple paths having the same prefix. We call this approach constraint-based fuzzing. If multiple solutions could be generated efficiently, this would significantly speed up symbolic execution and reap the benefits of random testing [6, 17, 21, 22, 34, 45, 46].

Similar ideas have been proposed and developed in hardware verification. For example, constrained-random verification (CRV) [33] has been proposed to generate high-quality inputs for hardware designs. In CRV, verification engineers specify preconditions required by the hardware and other constraints based on domain-specific knowledge [32, 47]. Multiple random inputs satisfying the constraints are then generated using a constraint solver that can sample random solutions from a constraint.

However, despite its importance, the problem of sampling a diverse set of solutions efficiently is still challenging today [9]. There are approaches which give strong guarantees of uniformity [11], but are expensive to run when a large number of samples is required. Other approaches use heuristics for faster sampling [43], but that can make the samples biased towards one portion of the sampling space.

In this work, we specifically focus on the goal of generating random samples to be used as inputs for testing. We assume that the constraints are given as Boolean satisfiability (SAT) problems, since constraints from higher level domains, such as bit-vectors or other satisfiability modulo theories (SMT) problems can be mapped into SAT. Our goal is to efficiently generate lots of random satisfying assignments to SAT formulas, also known as SAT witnesses.

In the testing domain, it is often acceptable to generate invalid solutions some of the time. For example, in constraint-based fuzzing, we may use constraints to direct the execution towards a certain portion of the program, but it is still fine if some samples don’t satisfy the constraints and end up executing other program parts. It is also important to notice that, in the testing domain, the most important metric is the number of unique valid solutions generated over time. That is because each unique valid input can help cover new portions of the program and find previously unseen bugs, while repeated samples do not increase coverage.

With that in mind, we have designed QuickSampler, a new technique for efficient sampling. QuickSampler uses a small number of constraint solver calls to generate a large number of samples. QuickSampler works as follows. First, it finds a random assignment to the variables of the Boolean formula (i.e. the constraint). Such an assignment may not satisfy the formula. QuickSampler then uses a MAX-SAT solver to find a solution of the formula that is close to the random satisfying assignment. It then flips the value of each variable in the solution and again uses MAX-SAT to find another close solution of the formula. The difference between the
original solution and the modified solution is called an atomic mutation. For each variable in the formula, this generates at most one atomic mutation. A small bounded number of such atomic mutations are then combined and applied to the original solution to generate a potentially new solution. We found that such combinations of small atomic mutations often results in new valid random solutions. This is because each atomic mutation identifies a small set of variables that are tightly coupled with each other. Whereas the variables from two different atomic mutations are often independent. Therefore, if two such atomic mutations are combined and applied to the original solution, then the resulting solution will often satisfy the formula. The entire process is repeated several times. Since QuickSampler creates lots of solutions by simply combining atomic mutations, it avoids making frequent solver calls (which is often the bottleneck). This in turn results in quick generation of lots of random solutions.

We have implemented QuickSampler as an open-source tool available at https://github.com/RafaelTupynamba/quicksampler/. We use Z3 [14] to solve MAX-SAT queries. The samples generated by QuickSampler are not guaranteed to satisfy a given formula, but our experiments show that they are valid solutions in our benchmarks with high probability (i.e. $\geq 0.75$). QuickSampler also produces unique valid solutions orders of magnitude (i.e. $\geq 1000\times$) faster than other state-of-the-art samplers, while generating a distribution of samples which is still close to uniform. For applications which require only valid solutions, it is also possible to use our technique, by simply checking the samples for validity and filtering out the invalid ones. Our evaluation shows that QuickSampler is still faster than the other samplers, even when including this additional checking.

2 RELATED WORK

There are several different techniques used to tackle the problem of sampling solutions to Boolean constraints [29]. The problem of sampling SAT witnesses is also closely related to the problem of counting the number of solutions, which has $\#P$-complete complexity. Several sampling techniques can be applied to model counting or use some form of model counting internally [16, 30, 44].

One class of sampling methods is based on Markov Chain Monte Carlo (MCMC) algorithms [26, 27]. These include simulated annealing and Metropolis-Hastings which are used to generate samples from a probability space. Those MCMC methods are guaranteed to eventually converge to the desired distribution (such as uniform sampling). However, this convergence is slow in practice for real-world problems, so the algorithms typically employ heuristics which make the sampling more biased [27, 43]. For example, [43] combines Metropolis steps with random walk steps through the assignments to the variables of the formula. In comparison, QuickSampler does not need to wait for a convergence time and covers the search space by finding solutions closest to randomly selected points.

One similar line of work attempts to modify the SAT solver search heuristics in order to generate a more diverse set of solutions [31]. However, unlike QuickSampler, this diverse sampling has different goals and does not attempt to cover the whole search space nearly uniformly. QuickSampler also does not modify the inner search strategies of SAT solvers, but instead uses the SAT solvers as an oracle to answer MAX-SAT queries.

The closest technique to QuickSampler in literature appears to be SearchTreeSampler [16], which also uses a SAT solver as an oracle. However, SearchTreeSampler performs simple satisfiability queries instead of the MAX-SAT queries by QuickSampler. SearchTreeSampler works by exploring the tree of variable assignments in a breadth-first way, generating pseudosolutions, which are partial assignments to the variables that can be completed to a full solution. SearchTreeSampler uses a parameter $k$ which specifies the number of samples computed per level in the tree and can be used to trade-off uniformity and number of solver calls required. On the other hand, QuickSampler uses a different strategy to cover the search space, and also generates a vastly larger number of samples per solver call, by combining learned mutations.

A different class of algorithms is based on universal hashing [15, 30] and can provide strong guarantees of uniformity. These techniques work by adding additional constraints to the formula (known as hash functions) in order to partition the search space uniformly. Those hash functions are typically formed by computing the XOR of a random subset of variables [20]. UniGen [11] and UniGen2 [9] are examples of this class, with the latter also employing parallelism to improve performance. In comparison, QuickSampler does not attempt to be perfectly uniform, but only close to uniform in practice. QuickSampler primarily aims for efficiency, using solver calls which are much less expensive to solve than the XOR constraints of hash functions, and generating a large number of samples per solver call.

3 QUICKSAMPLER ALGORITHM

Given a Boolean formula $\phi$, the goal of QuickSampler is to generate unique solutions of $\phi$ efficiently. Another goal of QuickSampler is to make sure that solutions of $\phi$ are sampled almost uniformly at random. The key idea behind QuickSampler is to make a small number of solver calls to generate a large number of potentially unique solutions of $\phi$. The core algorithm behind QuickSampler works as follows. QuickSampler assumes that we are given an initial random solution $\sigma$ (i.e. a satisfying assignment to $\phi$), where $\sigma$ is a vector of 1s and 0s. Each location in the vector corresponds to a Boolean variable in $\phi$ and the value at that location in the vector denotes the value assigned to this variable in the solution $\sigma$. Let $V$ be the set of all Boolean variables in $\phi$. For example, in Figure 1 we
show a possible vector \( \sigma \), in a case where the number of variables is \(|V| = 12\).

For each variable \( v \in V \), \textsc{QuickSampler} finds a solution \( \sigma_v \), such that \( \sigma_v \) and \( \sigma \) are minimally different and \( \sigma_v(v) \neq \sigma(v) \), where \( \sigma(v) \) is the value of the variable \( v \) in the solution \( \sigma \). Note that such a solution may not exist for all variables in \( V \). The diff between \( \sigma \) and \( \sigma_v \), which we will denote using \( \delta \), and which is the XOR of \( \sigma \) and \( \sigma_v \), is called an atomic mutation of \( \sigma \). That is \( \delta_v = \sigma_v \oplus \sigma \).

In the example from Figure 1, if the first variable of the formula is \( v \), we might find a new solution \( \sigma_v \) which has the first bit flipped (corresponding to variable \( a \)) and additionally other two bits flipped. The corresponding atomic mutation \( \delta_v \) is also shown in Figure 1. Similarly, if the second variable of the formula is \( b \), we might find a new solution \( \sigma_b \) as shown in Figure 1, which has the second bit (corresponding to variable \( b \)) flipped, but also other 3 bits flipped. The corresponding atomic mutation \( \delta_b \) is again shown in Figure 1.

By definition, the atomic mutation \( \delta_v \) always ensures that at least one variable \( v \) and difference in the values of the remaining variables is minimal.

We will later explain how a MAX-SAT query to a SAT solver can be used to find \( \sigma_v \) given \( \phi \), \( \sigma \), and \( v \). Given \( \sigma \), \textsc{QuickSampler} first computes the set of all atomic mutations by going over all the variables \( v \in V \). Let us denote the set of all such atomic mutations by \( \Delta^k_\sigma \). Note that given \( \sigma \) and \( \delta_v \), we can compute \( \sigma_v \) as \( \delta_v \oplus \sigma \).

After computing \( \Delta^k_\sigma \), \textsc{QuickSampler} computes sets of composite mutations \( \Delta^k_\sigma \) for \( k > 1 \), where \( \Delta^k_\sigma \) contains the union of all \( k \) distinct mutations in \( \Delta^k_\sigma \). For example, if \( \delta_a \) and \( \delta_b \) are two mutations in \( \Delta^1_\sigma \) such that \( a \neq b \), then \( \delta_a \lor \delta_b \) is a mutation present in \( \Delta^2_\sigma \). (Since each of \( \delta_a \) and \( \delta_b \) is a bit-vector, \( \delta_a \lor \delta_b \) is computed by taking bitwise OR of the two bit-vectors.) For example, after computing the atomic mutations \( \delta_a \), \( \delta_b \in \Delta^1_\sigma \) from Figure 1, the combined mutation \( \delta_a \lor \delta_b \) is added to \( \Delta^2_\sigma \). If we apply the combined mutation to \( \sigma \), by computing \( \sigma \oplus (\delta_a \lor \delta_b) \) we obtain a new assignment \( \tilde{\sigma} \), as in Figure 1. Note that \( \tilde{\sigma} \) differs from \( \sigma \) on all the bits set in either of the two atomic mutations \( \delta_a \) and \( \delta_b \).

This new assignment \( \tilde{\sigma} \) is not guaranteed to be a valid solution, but we have found that it has a high probability of being valid in real benchmarks\(^1\). This is because the differences \( \delta_a \) and \( \delta_b \) consist of a minimal set of bits which can be flipped while still preserving the satisfiability of the formula. So the bits in \( \delta_a \) are likely to be closely related to each other by some clauses in the formula. It is likely that those clauses would still be satisfied in \( \sigma \oplus (\delta_a \lor \delta_b) \), where we flip all the bits from \( \delta_a \) in addition to the bits from \( \delta_b \).

In general, each mutation \( \delta \) present in \( \Delta^k_\sigma \) denotes a composite mutation and can be XORed with \( \sigma \) to get an assignment \( \tilde{\sigma} \) to the variables in \( \phi \). Such an assignment may or may not be a solution of \( \phi \). Surprisingly, in our experiments we found that for small values of \( k \) (i.e. \( k \leq 6 \)), more than 73% of such assignments obtained by XORing are solutions of \( \phi \). Let us denote the assignments obtained by applying all the mutations present in \( \Delta^k_\sigma \) to \( \sigma \) by \( \Sigma^k_\sigma \), i.e.

\[
\Sigma^k_\sigma = \{ \delta \oplus \sigma \mid \delta \in \Delta^k_\sigma \}
\]

\(^1\)Our heuristic to generate samples exploits the clause structure found in real-world benchmarks. We expect it to perform poorly if applied to a randomly-generated SAT formula.

We let \( \Sigma_\sigma = \bigcup_{1 \leq k \leq |V|} \Sigma^k_\sigma \). We found experimentally that over all benchmarks, 75% of the assignments in \( \Sigma_\sigma \) are solutions of \( \phi \).

We now make a few interesting and important observations about the set of assignments \( \Sigma_\sigma \). \textsc{QuickSampler} needs to make solver calls only to compute \( \Delta^1_\sigma \). Moreover, it is not always necessary to make a solver call while computing the elements of \( \Delta^k_\sigma \)—if \textsc{QuickSampler} flips the bit corresponding to the variable \( v \) in \( \sigma \) and discovers that the resulting bit-vector is a satisfying assignment to \( \phi \), then \textsc{QuickSampler} can skip the solver call for \( \delta_v \). For the computation of all other \( \Sigma^k_\sigma \), \textsc{QuickSampler} needs no solver calls because each element in \( \Sigma^k_\sigma \) is obtained by applying at most \( k \) bitwise Boolean operations. An assignment in \( \Sigma^k_\sigma \) may or may not be a valid solution, however checking its validity is fast and takes linear time in the size of \( \phi \). In summary, \textsc{QuickSampler} can potentially make solver calls for the computation of \( \Sigma^1_\sigma \), but it makes no solver calls to compute the remaining sets \( \Sigma^k_\sigma \). Another observation is that size of \( \Sigma^k_\sigma \) could grow exponentially with \( k \). This observation combined with the facts that a significant number of assignments in \( \Sigma_\sigma \) have been empirically found to be solutions of \( \phi \) and that we make at most \(|V| \) solver calls suggests that given \( \sigma \), \textsc{QuickSampler} can rapidly generate lots of unique solutions of \( \phi \) by making very few solver calls. This forms the crux of \textsc{QuickSampler}'s core algorithm for sampling.

Given a random solution \( \sigma \), we described how \textsc{QuickSampler} generates lots of solutions that are small mutations of \( \sigma \). We next describe how we generate a random solution \( \sigma \). \textsc{QuickSampler} first chooses a random assignment \( \sigma' \) by picking the values of variables in \( V \) uniformly at random. Then it uses a MAX-SAT query to find a closest solution \( \sigma \) to the random assignment \( \sigma' \). We picked this strategy to make sampling of solutions more uniform. Overall, \textsc{QuickSampler} execution is divided into epochs. In each epoch, \textsc{QuickSampler} generates a random solution \( \sigma \) using the method described above. Then it computes \( \Sigma_\sigma \) and outputs the elements of \( \Sigma_\sigma \) that are solutions of \( \phi \). \textsc{QuickSampler} repeats this process in a loop until it has run out of time budget or it has finished generating a user-specified number of solutions.

Now we describe how MAX-SAT queries can be used to obtain the random solution \( \sigma \) and also to obtain the solutions \( \sigma_v \) for each variable \( v \). The maximum satisfiability problem, or MAX-SAT, is defined as follows: given a set of hard constraints and a set of soft constraints, find a solution which satisfies all the hard constraints and additionally satisfies the maximum possible number of soft constraints. In order to compute the random solution \( \sigma \), we just need to specify one hard constraint that the formula \( \phi \) must be satisfied and \(|V| \) soft constraints indicating that the values of each variable \( v \) should preferably be equal to their respective values in the random assignment \( \sigma' \), i.e. \( \forall v \in V : \sigma(u) = \sigma'(u) \). In order to compute each solution \( \sigma_v \), we specify two hard constraints and \(|V| - 1 \) soft constraints. The hard constraints are that the formula \( \phi \) must be satisfied and that the value of variable \( v \) must be flipped, i.e. \( \sigma_v(v) \neq \sigma(v) \). The soft constraints are that the values of other variables should preferably remain the same, or \( \forall v \in V \setminus \{v\} : \sigma_v(u) = \sigma(u) \).

This completes the general description of the \textsc{QuickSampler} algorithm. However, we have also made some adaptations to this initial algorithm in order to improve its performance. We discuss
in §3.1 the eager generation of samples, in §3.2 the use of the independent support of the formula, and in §3.3 the removal of unsatisfiable variables.

3.1 Eager Generation of Samples
As we have seen, QuickSampler only requires solver calls in order to produce the atomic mutations in $\Delta^1$. After that, the computation of the samples in $\Delta^k$ can be performed with simple bitwise operations. However, the solver calls can be expensive for the largest benchmarks. So ideally we would like to leverage each solver call to generate samples as soon as the call is completed, so that we do not need to wait for all solver calls before generating samples.

We have adapted the basic QuickSampler algorithm to generate samples as soon as each solver call is completed. Figure 2 presents an example of this eager generation. Each circle represents one mutation and inside it we indicate the number of atomic mutations used to generate it. When solution $\sigma_a$ is returned by the solver, we learn one atomic mutation $\delta_a$, represented by the first circle in the figure. Then, as soon as solution $\sigma_b$ becomes available, we learn the atomic mutation $\delta_b$ and also combine it with $\delta_a$ to generate a mutation in $\Delta^2$. Then, as soon as solution $\sigma_c$ becomes available, we learn the mutation $\delta_c$ and combine it with the three previously known mutations in order to generate three new mutations.

In conjunction with this eager generation of mutations, we also eliminate duplicate mutations in the current sampling epoch. We found it important to avoid duplicates within one epoch, otherwise we would output too many repeated samples and perform unnecessary work computing them.

We maintain a collection $\Delta$ of currently known mutations generated in the current epoch. Whenever we learn a new mutation from a solver call, we combine it with the mutations in $\Delta$. The new generated mutations are added to $\Delta$ only if they are previously unseen mutations in this epoch. Also, we only combine the new atomic mutations with mutations from $\Delta^k$ for $k < 6$. We do not want to generate mutations composed of more than 6 atomic mutations, because they would have a lower chance of generating valid solutions, according to our experiments.

3.2 Independent Support
Similarly to UniGen2 [9], we can restrict our sampler to only operate over the variables in an independent support $S$ of the formula, instead of generating assignments to all the variables in $V$. The independent support is a subset of variables which completely determines all the assignments to a formula. More specifically, given an assignment of values to the variables in the independent support $S$, there is at most one completion of this assignment to the remaining variables which satisfies the formula. So we can think of all other variables being dependent on the variables in the independent support. Knowing an independent support is helpful in reducing the number of variables for which we need to assign values.

In many cases, an independent support arises naturally from the application. For example, when the Tseytin transformation is used to transform a combinatorial logic circuit into a Boolean formula in conjunctive normal form (CNF), auxiliary variables are introduced for all intermediate wires in the circuit. All of those auxiliary variables can be uniquely determined given the inputs to the circuit, so the inputs form an independent support. In cases when an independent support is not known for a formula, there are also methods to compute a minimal independent support for it [23].

3.3 Unsatisfiable variables
If one MAX-SAT query for variable $v$ returns no solutions, we learn that $v$ can only have one truth value in this formula. When this happens in the first epoch, we record the variable $v$ in a set $U$ of unsatisfiable variables. Then, we do not try to flip the value of $v$ again in other epochs. We found that, over all benchmarks, on average 6% of the variables from the independent support were added to the set $U$. This means that, after the first epoch, all subsequent epochs can work over a reduced sampling set and avoid unnecessary solver calls.

4 IMPLEMENTATION
We have implemented the technique in C++, using Z3 [14] as the underlying solver. QuickSampler uses the Z3 optimization subsystem $iZ$ [5] in order to solve the MAX-SAT queries. We also use the push() and pop() interfaces to efficiently add and remove constraints from a single solving context.

QuickSampler takes as input a SAT formula in conjunctive normal form (CNF), represented in the DIMACS file format. The formula includes a list of variables which compose its independent support.

Our implementation outputs the samples generated to a file without checking if they are valid solutions. If desirable, it is possible to add a posterior check which verifies if the samples are valid or not (and possibly filters out the invalid ones). We also do not check for duplicates, which can appear between different epochs in the sampling algorithm. This global check for uniqueness could also be added, but it would require an additional time and memory

2The source code is available at https://github.com/RafaelTupynamba/quickSampler.
overhead. Some applications might prefer not to keep all generated samples in memory, and allow the generation of repeated samples instead.

We have implemented an offline analysis to check if the samples are valid and generate histograms that count how many times each solution has been sampled. We record the time taken by the sample generation and also the time taken by the checking phase. The checking phase is not heavily optimized and for most benchmarks it was more expensive than the sampling phase. We believe there is still room for improvement in the checking phase, since all it needs to do is to propagate the values of the independent support to the remaining variables and check if all clauses are satisfied.

5 EVALUATION

We evaluate QUICKSAMPLER by comparing against two state-of-the-art samplers UniGen2 [9] and SearchTreeSampler [16]. UniGen2 provides strong uniformity guarantees, by using hash functions composed of XOR constraints in order to partition the search space into similar-sized bins.

SearchTreeSampler, on the other hand, uses the SAT solver to find pseudosolutions (partial assignments to the few variables) and progressively augments the pseudosolutions into real solutions. SearchTreeSampler is only approximately uniform. The uniformity can be increased with a higher number of samples per level $k$, but at a cost of also increasing the number of solver calls required. In our experiments, we used the default value of $k = 50$.

Both QUICKSAMPLER and UniGen2 can leverage the knowledge of an independent support of the formula to improve sampling performance. So in order to make for a fair comparison, we modified SearchTreeSampler to use this additional information. We reorder the variables of the formula in order to place first the ones which are part of the independent support. And we additionally tell SearchTreeSampler to finish sampling after processing those variables and output pseudosolutions (assignments to the variables of the independent support) that it has produced so far. Since an assignment to the independent support can only be completed to one solution, there is no need to find assignments to the remaining variables.

For the evaluation, we use the set of benchmarks from the UniGen2 paper [9] available online. From the benchmarks listed in [9], we found 173 on the online repo. The benchmarks include bit-blasted versions of SMTLib benchmarks, ISCAS89 circuits augmented with parity conditions on randomly chosen subsets of outputs and next-state variables, problems arising from automated program synthesis and constraints arising in bounded model checking. Thus, they are representative of the kinds of constraints that might appear in SMT formulas for software testing or circuit constraints for hardware.

On 10 benchmarks, UniGen2 reported an error because the specified independent support is not really an independent support for the formula. In all those benchmarks, we verified that the number of solutions computed by the exact model counter sharpSat [41] is larger than $2^{38}$, which should not happen if $S$ is a real independent support for the formula. So we decided to exclude those benchmarks from our results. The results in this paper include the remaining 163 benchmarks.

On 3 benchmarks, UniGen2 could not estimate the number of solutions: on parity.sk_11_11, UniGen2 raised a floating point exception and on isolateRightmost.sk_7_481 and listReverse.sk_11_43, the ApproxMC [10] model counter used by UniGen2 couldn’t finish even in 40 hours. On 2 benchmarks, UniGen2 estimated the number of solutions but couldn’t produce any samples: on doublyLinkedList.sk_8_37 it timed out and on diagStencilClean.sk_41_36 it ran out of memory.

The experiments were conducted on a 12-core, 3.50GHz Intel Core i7-5930K CPU. For each benchmark, each of the algorithms was given one core and 1.5 GB of memory. For QUICKSAMPLER and SearchTreeSampler, we allowed a maximum timeout of 1 hour, or 2 hours on the hardest benchmarks. We also stopped the sampling after a large number of samples had been produced (at least 10 million samples).

For UniGen2, we requested the generation of 1000 samples for most benchmarks, allowing up to 20 hours to produce them. For the hardest benchmarks, we reduced the number of requested samples to 500. For all the benchmarks in which UniGen2 failed to produce any samples, it times out after 20 hours even when the number of requested samples was 1.

5.1 Correctness of Samples

On Table 1, we list the average number of samples produced and how many of those were valid, on one epoch of the sampling algorithm. The results were averaged across all 163 benchmarks. They are classified according to the number of individual atomic mutations which compose the mutation. The base solution used in the epoch is the one with 0 atomic mutations and the neighbors of the base solution obtained when flipping each bit correspond to 1 atomic mutation. Those are always valid solutions to the formula, since they are obtained as the result of solver calls.

From 2 to 6 atomic mutations, we see that the fraction of valid solutions decreases from 96% to 73%. And overall, 75% of all samples produced were valid, when we allow a maximum of 6 atomic mutations. Table 1 shows that, by adjusting this maximum, we can

<table>
<thead>
<tr>
<th>Atomic mutations</th>
<th>Total</th>
<th>Valid</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>100%</td>
</tr>
<tr>
<td>1</td>
<td>32</td>
<td>32</td>
<td>100%</td>
</tr>
<tr>
<td>2</td>
<td>511</td>
<td>492</td>
<td>96%</td>
</tr>
<tr>
<td>3</td>
<td>5619</td>
<td>5208</td>
<td>93%</td>
</tr>
<tr>
<td>4</td>
<td>47493</td>
<td>42179</td>
<td>89%</td>
</tr>
<tr>
<td>5</td>
<td>346367</td>
<td>282860</td>
<td>82%</td>
</tr>
<tr>
<td>6</td>
<td>2143385</td>
<td>1571553</td>
<td>73%</td>
</tr>
<tr>
<td>Total</td>
<td>2543409</td>
<td>1902325</td>
<td>75%</td>
</tr>
</tbody>
</table>

Table 1: Correctness statistics for the samples produced in one epoch of QUICKSAMPLER (average among all benchmarks)
change the accuracy of the sampling. For example, with a maximum of 5 atomic mutations instead of 6, the fraction of valid samples would increase to 83%. However, there would be a substantial decrease in the quantity of samples produced. We have chosen to use the maximum number of 6 atomic mutations to allow the generation of millions of samples, while still having a reasonably good accuracy of 75%.

If $n$ is the number of atomic mutations, then the number of mutations of level 6 can go up to $\binom{n}{6}$, a sixth-degree polynomial in $n$. This explains why we can generate millions of samples from only tens of atomic mutations.

### 5.2 Performance Comparison

For the performance comparison, we define $t_q, t_s, t_u$ to be the average time taken by QuickSampler, SearchTreeSampler and UniGen2, respectively to produce a valid sample. $t_q$ was computed as $t_q = T_q/(s_q \cdot p)$, where $T_q$ is the total execution time, $s_q$ is the total number of samples produced and $p$ is the fraction of samples which are valid for QuickSampler. We additionally define $t_q^*$ to be the estimated time per valid sample that QuickSampler would require if it also checked all samples for validity. $t_q^*$ was computed as $t_q^* = (T_q + T_c)/(s_q \cdot p)$, where $T_c$ is the total time taken to check the validity of all $s_q$ produced samples.

Table 2 shows the performance comparison among a selected set of benchmarks. We have included the largest benchmarks (more than 4000 variables), the benchmarks which were listed as representative benchmarks in [9] and the benchmarks used for uniformity plots in §5.3. This includes the benchmarks which QuickSampler or SearchTreeSampler found hard.

The first group of columns in Table 2 shows basic information about the benchmarks: size of the independent support, number of variables, clauses and solutions. The number of solutions was obtained from UniGen2. On most benchmarks, an exact number of solutions is known, while for some we only know an approximation (represented with $\approx$) and on some UniGen2 failed completely to compute the number of solutions.

The second group of columns shows results for QuickSampler: the number of epochs completed, number of MAX-SAT solver calls, number of samples generated, fraction of samples which are valid and the average times per valid sample $t_q$ and $t_q^*$ in microseconds. The third and fourth group of columns present results for SearchTreeSampler and UniGen2: the number of samples produced and the average time per sample, taken in comparison with the QuickSampler time $t_q$.

The mean value for some ratios of interest is shown on Table 3. For example, $t_s/t_q \approx 10^{2.240.3}$. This was computed by taking the average and the standard deviation of $\log_{10}(t_s/t_q)$ across all benchmarks.

Figure 3 shows a comparison of the average time per valid sample, against SearchTreeSampler and UniGen2. As reported in Table 3, QuickSampler was on average 2.5 orders of magnitude faster than SearchTreeSampler and 4.7 orders of magnitude faster than UniGen2. Overall, QuickSampler was only slower than SearchTreeSampler on the benchmark diagStencilClean.sk_41_36, with $t_s/t_q = 6.6 \cdot 10^{-5}$. We believe QuickSampler did not do well on diagStencilClean.sk_41_36 because the Z3 solver used in our implementation did not perform well on this formula. In comparison, MiniSAT, the solver used by SearchTreeSampler, was much faster on this benchmark. The opposite effect can be seen, for example, on parity.sk_11_11, where MiniSAT was only able to complete a small number of solver calls.

Next we present graphs of the same metrics, but now also taking into account the time that would be required for QuickSampler to check if the samples are valid. This should only be needed if the application cannot deal with invalid samples. Figures 4a and 4b show the comparison with SearchTreeSampler and UniGen2, respectively. We see that QuickSampler is still 1 order of magnitude faster than SearchTreeSampler and 3.2 orders of magnitude faster than UniGen2, even when including this checking time. QuickSampler was only slower than SearchTreeSampler on three benchmarks,
5.3 Uniformity of Coverage

The results from §5.2 show that QUICKSAMPLER can produce unique valid solutions very fast, which was our primary goal. But we would still like to check if the distribution of samples produced is similar to uniform, because we don’t want to be missing a large portion of the solution space, while focusing on a very biased subset of solutions. We have designed our main sampling function to start from a random point in the space of possible variable assignments in order to make our coverage more uniform. This also guarantees that any solution has a positive probability of being output by our algorithm.

In order to empirically evaluate the uniformity of QUICKSAMPLER, we compare its distribution of solutions with the ones from the

Table 2: Mean ratio comparisons across all benchmarks

<table>
<thead>
<tr>
<th>Ratio</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_s/t_q$</td>
<td>$10^{0.5-0.8}$</td>
</tr>
<tr>
<td>$t_u/t_q$</td>
<td>$10^{1.7+1.0}$</td>
</tr>
<tr>
<td>$t_s^3/t_q$</td>
<td>$10^{-0.6+0.5}$</td>
</tr>
<tr>
<td>$t_u^3/t_q$</td>
<td>$10^{-2.0+0.7}$</td>
</tr>
<tr>
<td>$u_q/t_s$</td>
<td>$10^2,3.0+0.7$</td>
</tr>
<tr>
<td>$u_q/t_u$</td>
<td>$10^{-4.1+1.1}$</td>
</tr>
</tbody>
</table>

where the ratio $t_s/t_q$ was 0.95 for 17, sk_3_45, 0.71 for 70, sk_3_40 and 6.6 · 10^{-3} for diagStencilClean.sk_41_36.

Those results show clearly that QUICKSAMPLER is capable of generating valid solutions orders of magnitude faster than the other techniques. However, we believe that an even more important metric is the number of unique valid solutions generated over time, since repeated solutions do not help uncover new behavior in the test program. So we performed an experiment to evaluate the number of unique valid solutions generated.

All three algorithms were allowed to run until they produced 10 million samples or reached 1 hour of execution. If their execution times are $T_s, T_u, T_q$, we define $T = \min(T_s, T_u, T_q)$ and look at the number of unique valid solutions that each algorithm could produce in time $T$ and represent those numbers as $u_q, u_s, u_u$. We found out that on most benchmarks QUICKSAMPLER was able to produce 10 million samples before 1 hour and it was the fastest algorithm to finish. So the uniqueness comparison is performed at 1 hour. The names of those benchmarks are marked with an asterisk in Table 2.

Figure 5a compares unique solutions produced by QUICKSAMPLER and SearchTreeSampler. On average, the number of unique solutions produced by quickSampler was 2.3 orders of magnitude larger, as seen in Table 3. On only one benchmark it was lower (karatsuba.sk_7_41, with $u_q/u_u = 0.76$).

In Figure 5b, we present the ratio of unique solutions between QUICKSAMPLER and UniGen2. Again, the ratio was lower only on karatsuba.sk_7_41, with $u_q/u_u = 0.08$. On average, $u_q$ was 4.4 orders of magnitude higher than $u_u$. We found that QUICKSAMPLER performed poorly on karatsuba.sk_7_41 because it had not completed one sampling epoch within the first hour of execution, and most of the samples are generated towards the end of the sampling epoch. However, within 2 hours, QUICKSAMPLER was able to complete 2 sampling epochs, generating a vastly larger amount of samples, as reported in Table 2.

Efficient Sampling of SAT Solutions for Testing

ICSE ’18, May 27-June 3, 2018, Gothenburg, Sweden
two other samplers SearchTreeSampler, UniGen2 as well as a distribution from a perfect uniform sampler. Only the valid samples are considered in this analysis. We compare on the benchmarks for which the number of samples generated by UniGen2 in a time limit of 10 hours was at least five times the total number of solutions. It is important for statistical significance that each solution be sampled on average at least five times. For each of the benchmarks, let $s_q, s_s, s_u$ be the number of valid samples generated by each algorithm and $s = \min\{s_q, s_s, s_u\}$. We subsample uniformly $s$ samples from the valid samples produced by each algorithm and we also generate $s$ samples from a perfectly uniform distribution, using the total number of solutions provided by UniGen2.

Figures 6 to 10 show the results of the comparison on all benchmarks for which the number of generated samples $s$ can be at least
five times the number of solutions before the timeout is reached. The x axis represents the number of times each solution has been sampled and the y axis represents the quantity of solutions which have been sampled x times. We can see that SearchTreeSampler and UniGen2 are usually indistinguishable from uniform, but QuickSampler is still close to uniform on most benchmarks. However, this result should be taken with care, since the uniformity test is not very reliable on benchmarks where QuickSampler completed a small number of epochs or when the number of produced samples is too low.

Besides analyzing the uniformity of the distribution, we also measured the number of unique valid solutions generated. This is arguably more important than the histograms of solution counts, because we want unique solutions to increase coverage in testing. We computed the number \( u \) of unique solutions that should be generated if the sampling was perfectly uniform. We record the ratio \( u/\bar{u} \) for all benchmarks for which we have an estimate of the number of solutions. We measured the number of unique valid solutions generated by QuickSampler and also the number \( \bar{u} \) of unique solutions that should be generated if the sampling was perfectly uniform. We reject the null hypothesis that the distribution is uniform if the p-value is lower than the confidence level of 0.05. This gives a bound on the type I error rate (i.e., the probability that a uniform distribution is mistakenly rejected as non-uniform). Table 4 show the results of applying this test to the 160 benchmarks for which we know an estimate of the number of solutions. We can see that SearchTreeSampler and UniGen2 are more uniform, but QuickSampler is still close to uniform on most benchmarks. However, this result should be taken with care, since the uniformity test is not very reliable on benchmarks where QuickSampler completed a small number of epochs or when the number of produced samples is too low.

Besides analyzing the uniformity of the distribution, we also measured the number of unique valid solutions generated. This is arguably more important than the histograms of solution counts, because we want unique solutions to increase coverage in testing.

We computed the number \( u \) of unique solutions generated by QuickSampler and also the number \( \bar{u} \) of unique solutions that should be generated if the sampling was perfectly uniform. We record the ratio \( u/\bar{u} \) for all benchmarks for which we have an estimate of the number of solutions. We measured the number of unique valid solutions generated by QuickSampler and also the number \( \bar{u} \) of unique solutions that should be generated if the sampling was perfectly uniform. We reject the null hypothesis that the distribution is uniform if the p-value is lower than the confidence level of 0.05. This gives a bound on the type I error rate (i.e., the probability that a uniform distribution is mistakenly rejected as non-uniform). Table 4 show the results of applying this test to the 160 benchmarks for which we know an estimate of the number of solutions. We can see that SearchTreeSampler and UniGen2 are more uniform, but QuickSampler is still close to uniform on most benchmarks. However, this result should be taken with care, since the uniformity test is not very reliable on benchmarks where QuickSampler completed a small number of epochs or when the number of produced samples is too low.

Table 4: Chi-squared Uniformity Test

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>QuickSampler</th>
<th>SearchTreeSampler</th>
<th>UniGen2</th>
</tr>
</thead>
<tbody>
<tr>
<td>_blasted_case110</td>
<td>Not Rejected</td>
<td>Rejected</td>
<td></td>
</tr>
<tr>
<td>s820a_7_4</td>
<td>Not Rejected</td>
<td>Rejected</td>
<td></td>
</tr>
<tr>
<td>s820a_15_7</td>
<td>Not Rejected</td>
<td>Rejected</td>
<td></td>
</tr>
</tbody>
</table>

\[ \chi^2 = \sum \frac{(O - E)^2}{E} \]

\[ \text{p-value} = P(\chi^2 > \chi^2_{crit}) \]

We computed the number \( u \) of unique solutions generated by QuickSampler and also the number \( \bar{u} \) of unique solutions that should be generated if the sampling was perfectly uniform. We record the ratio \( u/\bar{u} \) for all benchmarks for which we have an estimate of the number of solutions. We measured the number of unique valid solutions generated by QuickSampler and also the number \( \bar{u} \) of unique solutions that should be generated if the sampling was perfectly uniform. We reject the null hypothesis that the distribution is uniform if the p-value is lower than the confidence level of 0.05. This gives a bound on the type I error rate (i.e., the probability that a uniform distribution is mistakenly rejected as non-uniform). Table 4 show the results of applying this test to the 160 benchmarks for which we know an estimate of the number of solutions. We can see that SearchTreeSampler and UniGen2 are more uniform, but QuickSampler is still close to uniform on most benchmarks. However, this result should be taken with care, since the uniformity test is not very reliable on benchmarks where QuickSampler completed a small number of epochs or when the number of produced samples is too low.

Besides analyzing the uniformity of the distribution, we also measured the number of unique valid solutions generated. This is arguably more important than the histograms of solution counts, because we want unique solutions to increase coverage in testing.

We computed the number \( u \) of unique solutions generated by QuickSampler and also the number \( \bar{u} \) of unique solutions that should be generated if the sampling was perfectly uniform. We record the ratio \( u/\bar{u} \) for all benchmarks for which we have an estimate of the number of solutions. We measured the number of unique valid solutions generated by QuickSampler and also the number \( \bar{u} \) of unique solutions that should be generated if the sampling was perfectly uniform. We reject the null hypothesis that the distribution is uniform if the p-value is lower than the confidence level of 0.05. This gives a bound on the type I error rate (i.e., the probability that a uniform distribution is mistakenly rejected as non-uniform). Table 4 show the results of applying this test to the 160 benchmarks for which we know an estimate of the number of solutions. We can see that SearchTreeSampler and UniGen2 are more uniform, but QuickSampler is still close to uniform on most benchmarks. However, this result should be taken with care, since the uniformity test is not very reliable on benchmarks where QuickSampler completed a small number of epochs or when the number of produced samples is too low.

Besides analyzing the uniformity of the distribution, we also measured the number of unique valid solutions generated. This is arguably more important than the histograms of solution counts, because we want unique solutions to increase coverage in testing.

We computed the number \( u \) of unique solutions generated by QuickSampler and also the number \( \bar{u} \) of unique solutions that should be generated if the sampling was perfectly uniform. We record the ratio \( u/\bar{u} \) for all benchmarks for which we have an estimate of the number of solutions. We measured the number of unique valid solutions generated by QuickSampler and also the number \( \bar{u} \) of unique solutions that should be generated if the sampling was perfectly uniform. We reject the null hypothesis that the distribution is uniform if the p-value is lower than the confidence level of 0.05. This gives a bound on the type I error rate (i.e., the probability that a uniform distribution is mistakenly rejected as non-uniform). Table 4 show the results of applying this test to the 160 benchmarks for which we know an estimate of the number of solutions. We can see that SearchTreeSampler and UniGen2 are more uniform, but QuickSampler is still close to uniform on most benchmarks. However, this result should be taken with care, since the uniformity test is not very reliable on benchmarks where QuickSampler completed a small number of epochs or when the number of produced samples is too low.
We also present plots of the number of unique solutions produced over time, for two representative benchmarks. In Figure 11 we show the graph for benchmark s820a_7_4, where the number of samples produced is larger than the total number of solutions. We see that the number of unique solutions grows very fast initially, and then stabilizes as we approach complete coverage of all solutions. SEARCHTreeSampler and UniGen2, on the other hand, produce solutions at a much slower rate. In Figure 12 we show benchmark enqueueSeqSK.sk_10_42, where the number of valid samples produced is much smaller than the total number of solutions. We can see that QuickSampler is able to generate unique solutions orders of magnitude faster than SEARCHTreeSampler and UniGen2. We also notice a distinctive step pattern in the graph. This happens because we produce the largest number of samples at the end of each sampling epoch, when the collection of known mutations is the largest.

In summary, we see that SEARCHTreeSampler and UniGen2 are a bit closer to uniform sampling, but QuickSampler is still very close. In almost all cases the number of unique solutions generated was very close to the number that would be expected if the sampling was uniform and we are able to produce new unique solutions at a faster rate than the other techniques.

6 CONCLUSION

We have developed a new technique to sample solutions to Boolean constraints, with the goal of applying it to constrained-random verification and fuzz testing. For those applications, it is typically acceptable to produce a small number of invalid inputs, so we allow our technique to output samples which are not guaranteed to be valid. By leveraging a small number of MAX-SAT solver calls, QuickSampler can generate millions of samples.

Our experiments show that the produced samples are valid with an average probability of 75% on a set of large, real-world benchmarks. Moreover, QuickSampler is more than 2 orders of magnitude faster at producing valid samples, when compared to other state-of-the-art samplers. It is also more than 2 orders of magnitude faster at producing unique valid samples, which is specially important to increase testing coverage. We have also verified that QuickSampler is still 1 order of magnitude faster even when it takes the additional time to verify that the generated solutions are valid. Finally, the distribution of samples produced is close to uniform on most of the benchmarks.

ACKNOWLEDGMENTS

Research partially funded by Brazilian Science Without Borders CAPES 13245/13-9; NSF grants CCF-1409872 and CCF-1423645; DARPA CRAFT HR0011-16-C-0052; Intel Science and Technology Center for Agile Design; and ASPIRE Lab industrial sponsors and affiliates Intel, Google, HP, Huawei, LGE, Nokia, NVIDIA, Oracle, and Samsung. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and does not necessarily reflect the position or the policy of the sponsors.

REFERENCES
