

ESAMPLER: Boosting Sampling of Satisfying Assignments for Boolean Formulas via Derivation

Yongjie Xu^a, Fu Song^a, Taolue Chen^b

^a*School of Information Science and Technology, ShanghaiTech University, Shanghai, China*

^b*Department of Computer Science, Birkbeck, University of London, London, UK*

Abstract

Boolean satisfiability (SAT) plays a key role in diverse areas such as spanning planning, inference, data mining, testing and optimization. Apart from the classical problem of checking Boolean satisfiability, generating random satisfying assignments has attracted significant theoretical and practical interests over the past years. In practical applications, usually a large number of satisfying assignments for a given Boolean formula are needed, the generation of which turns out to be a computational hard problem in both theory and practice. In this work, we propose a novel approach to derive a large set of satisfying assignments from a given one in an efficient way. Our approach is based on an insight that flipping the truth values of properly chosen variables of a satisfying assignment could result in satisfying assignments without invoking computationally expensive SAT solving. We propose a derivation algorithm to discover such variables for each given satisfying assignment. Our approach is orthogonal to the previous techniques for generating satisfying assignments and could be integrated into the existing SAT samplers. We implement our approach as an open-source tool ESAMPLER using two representative state-of-the-art samplers (QUICKSAMPLER and UNIGEN3) as the underlying satisfying assignment generation engine. We conduct extensive experiments on various publicly available benchmarks and apply ESAMPLER to solve Bayesian inference. The results show that ESAMPLER can efficiently boost the sampling of satisfying assignments of both QUICKSAMPLER and UNIGEN3 on a large portion of the benchmarks and is at least comparable on the others. ESAMPLER performs considerably better than QUICKSAMPLER and UNIGEN3, as well as another state-of-the-art sampler SearchTreeSampler.

Keywords: Boolean satisfiability, Constraint-based sampling, SAT solving

*Corresponding author

Email address: songfu@shanghaitech.edu.cn (Fu Song)

1 Introduction

Boolean satisfiability, also known as SAT, concerns determining whether a given Boolean formula is satisfiable. There have been strong theoretical and practical interests in the SAT problem, which has played a key role in diverse areas spanning planning, inferencing, data mining, testing and optimization [1, 2]. Apart from the classical problem of checking Boolean satisfiability, generating random satisfying assignments has attracted significant theoretical and practical interests over the years [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]. In several practical applications, a large number of satisfying assignments for a given Boolean formula are needed. For instance, simulation-based verification is a commonly adopted technique to test hardware design. In this scenario, the simulated behavior is compared with the expected behavior where any mismatch is flagged as an indication of a bug [12, 13]. It is a common practice to generate a large number of stimuli satisfying a given set of constraints in the form of Boolean formulas. These constraints typically arise from various sources such as application-specific knowledge and environmental requirements. Another application scenario is the generation of adversarial examples for adversarial training [16, 17]. Adversarial training is a widely adopted technique to improve the robustness of neural networks against adversarial attacks where a large number of adversarial inputs (e.g., images) would be generated explicitly or implicitly. For instance, to adversarially train a binarized neural network [18, 19], adversarial images could be generated by encoding a binarized neural network as a Boolean formula based on which satisfying assignments are sampled [20, 21].

Sampling satisfying assignments for a given Boolean formula is, however, challenging. Cook has shown in 1971 that the SAT problem is **NP**-complete [22]. In recent years, we have seen a tremendous progress in SAT solving, supported by techniques such as conflict-driven clause learning (CDCL [23, 24, 25]), yielding powerful solvers such as CryptoMiniSAT [26]. However, generating a large number of satisfying assignments is still computationally prohibitive and often infeasible in practical settings [27, 28].

In this work, we develop ESAMPLER, aiming for boosting the generation of a large number of satisfying assignments efficiently for a given Boolean formula. The general strategy is to use an existing sampler to produce a seed sample as a satisfying assignment, from which we derive more satisfying assignments by flipping some variables of the given Boolean formula. Clearly, naively flipping variables may yield unsatisfying assignments. To tackle this problem, we propose a novel derivation procedure which explores the semantics of the Boolean formula under the seed sample, so that the resulting assignments can be guaranteed to satisfy the Boolean formula. The advantage of our approach lies in that it can be integrated with the existing SAT samplers, so would enjoy considerably wider applicability.

To demonstrate our approach, we implement a sampler ESAMPLER based on the two types of state-of-the-art sampler QUICKSAMPLER [28] and UNIGEN3 [29]. We carry out extensive experiments on the publicly available benchmarks from UNIGEN [30] which include hundreds of Boolean formulas from

46 real-world testing and verification applications, and apply ESAMPLER to solve
47 Bayesian inference of the plan recognition problems [31]. Our experimental re-
48 sults show that ESAMPLER is able to effectively boost the sampling of satisfying
49 assignments of both QUICKSAMPLER and UNIGEN3 on a large portion of the
50 benchmarks and is at least comparable on the others. Consequently, ESAMPLER
51 considerably performs better than QUICKSAMPLER and UNIGEN3, as well as
52 the another state-of-the-art sampler SEARCHTREESAMPLER (STS in short) [32].
53 The experimental results confirm the efficacy of our derivation approach.

54 Our main contributions can be summarized as follows.

- 55 • We introduce a novel approach for deriving a large set of satisfying as-
56 signments from a given seed. It is generic and could be integrated with
57 the existing samplers. To the best of our knowledge, it is the first work to
58 generate satisfying assignments from a given seed.
- 59 • We implement an integrated sampler ESAMPLER based on two state-of-
60 the-art samplers. Our tool is available at [https://github.com/ESampler/](https://github.com/ESampler/ESampler)
61 [ESampler](https://github.com/ESampler/ESampler).
- 62 • We conduct extensive experiments on hundreds of real-world benchmarks.
63 The results show that our derivation approach is effective and consequently
64 ESAMPLER performs considerably better than the three state-of-the-art
65 samplers QUICKSAMPLER, STS and UNIGEN3.

66 **Related Work.** Various techniques have been proposed to tackle the problem
67 of the satisfying assignment generation for Boolean formulas [33]. Binary de-
68 cision diagrams (BDD) and Markov Chain Monte Carlo (MCMC) algorithms
69 such as simulated annealing and Metropolis-Hastings are widely used for gen-
70 erating satisfying assignments [9, 34, 35]. These techniques usually provide
71 theoretical guarantees of uniformity but are limited in scalability and efficiency.
72 Therefore, heuristics are proposed to speed up at the cost of theoretical guar-
73 antees of uniformity [36, 37, 34]. Another class of satisfying assignment gen-
74 eration techniques with theoretical guarantees of uniformity is based on hash-
75 ing [38, 39, 40, 41, 42, 30, 43, 29]. Hashing-based techniques add hash functions
76 (e.g., XOR of a random subset of variables) to the Boolean formula in order
77 to partition the search space uniformly and then randomly pick a satisfying
78 assignment from a randomly chosen cell. These algorithms are also limited in
79 scalability and efficiency. In comparison, our approach primarily aims for effi-
80 ciency, using fewer solver calls to generate a large number of solutions. We also
81 provide a parameter to balance the uniformity of the generated samples and the
82 efficiency of the procedure. Although we do not provide a theoretical guarantee
83 of uniformity, the experimental results demonstrate that our approach is able
84 to produce solutions nearly uniformly when the maximal number of solutions
85 per seed is set in a reasonable range.

86 SAT samplers aiming to quickly generate a large number of assignments have
87 recently been proposed. Both QUICKSAMPLER [28] and STS [32] share the same
88 goal as our work, namely, fast generation of a larger number of assignments.

89 QUICKSAMPLER works as follows. Given a Boolean formula Φ , it first constructs
 90 a random assignment v and then uses the MaxSAT solver [44] to solve the
 91 MaxSAT problem with the hard constraint Φ and soft constraint Ψ , where Ψ
 92 is the conjunction of literals x if $v(x) = 1$ or $\neg x$ if $v(x) = 0$ for each Boolean
 93 variable x of Φ . Solving the MaxSAT problem yields a satisfying assignment
 94 v' of Φ that is close to the random assignment v . After that, QUICKSAMPLER
 95 iteratively flips the value of each variable x in the satisfying assignment v' to
 96 find another close satisfying assignment v'_x using the MaxSAT solver, where the
 97 soft constraint asserts the satisfying assignment v' except for the flipped variable
 98 x , and the original Boolean formula together with the flipped variable is used as
 99 the hard constraint. For each flipped variable x , the difference δ_x between two
 100 satisfying assignments v' and v'_x is computed. All such differences are combined
 101 and applied to mutate the satisfying assignment v' to generate a large number of
 102 assignments. However, the assignments generated by QUICKSAMPLER may not
 103 satisfy the Boolean formula, hence follow-up checkings are needed. In contrast,
 104 our approach only mutates proper variables by which the formula is guaranteed
 105 to be satisfied. STS explores the tree of variable assignments in a breadth-first
 106 way with the MiniSat SAT solver [45] as an oracle. During this procedure, it
 107 generates *pseudosolutions*, which are partial assignments to the variables that
 108 can be completed to full satisfying assignments. However, it has to invoke
 109 SAT solvers multiple times during the breadth-first exploration. In contrast,
 110 ESAMPLER does not require SAT solving when generating satisfying assignments
 111 from a seed.

112 Technically, our derivation procedure aims to generate a large set of sat-
 113 isfying assignments from a given seed, and is orthogonal to the existing SAT
 114 samplers. It can be integrated into the existing samplers to improve their effi-
 115 ciency as we demonstrated using QUICKSAMPLER and UNIGEN3.

116 Sampling satisfying assignments is also closely related to the model-counting
 117 problem which counts the number of satisfying assignments for a Boolean for-
 118 mula. Model-counting techniques have been used for sampling satisfying as-
 119 signments (e.g., SPUR [46]) while satisfying assignment sampling techniques
 120 can also be used for model-counting (e.g., STS [32] and APPROXCOUNT [35]).

121 This article is an extended version of [47], but with substantial new material.
 122 In particular, we apply ESAMPLER to boost another uniform sampler UNIGEN3
 123 and carry out more experiments (cf. Section 5.4), which show the generality and
 124 wide applicability of ESAMPLER to diverse seed generation samplers. We also
 125 apply ESAMPLER for inference of Bayesian networks and report experimental
 126 results on the real-world plan recognition problems (cf. Section 6), showing a
 127 significant improvement of our approach ESAMPLER over the samplers QUICK-
 128 SAMPLER, STS and UNIGEN3.

129 **Outline.** The remainder of this paper is organized as follows. In Section 2, we
 130 briefly revisit related concepts of Boolean formulas. We present our derivation
 131 procedure in Section 3, and show how to integrate it into existing SAT samplers
 132 in Section 4. We report evaluation results in Section 5. We apply ESAMPLER
 133 to Bayesian inference in Section 6 and conclude this work in Section 7.

134 **2. Preliminaries**

135 We first recap some basic notions and notations which are used in this work.

136 **Boolean formulas.** Let us fix a set of Boolean variables \mathcal{V} . A literal l is either
 137 a Boolean variable $x \in \mathcal{V}$ or its negation $\neg x$. We denote by $\text{var}(l)$ the variable
 138 x used in the literal l , namely, $\text{var}(x) = \text{var}(\neg x) = x$.

139 A *Boolean formula* Φ is a Boolean combination of literals using logical-AND
 140 (\wedge) and logical-OR (\vee) operators. As a convention, we assume that Boolean
 141 formulas are given in the conjunctive normal form (CNF) $\bigwedge_{j=1}^m \bigvee_{i=1}^{n_j} l_i^j$, where
 142 for each $1 \leq j \leq m$ and $1 \leq i \leq n_j$, l_i^j is a literal, and $\bigvee_{i=1}^{n_j} l_i^j$ is referred to a
 143 *clause* for each $1 \leq j \leq m$. Given a Boolean formula Φ and a literal l , let Φ_l
 144 denote the set of clauses that contain the literal l . For each clause $\phi = \bigvee_{i=1}^{n_j} l_i^j$,
 145 we assume that all literals in ϕ are distinct, and denote by $|\phi|$ the number n_j of
 146 literals in the clause ϕ .

147 **Assignments.** An *assignment* is a function $v: \mathcal{V} \rightarrow \{0, 1\}$ which assigns a
 148 Boolean value to each Boolean variable $x \in \mathcal{V}$. Given a Boolean formula Φ and
 149 an assignment v , v is a *satisfying assignment* of Φ , denoted by $v \models \Phi$, if the
 150 Boolean formula Φ evaluates to 1 under the assignment v . A *partial assignment*
 151 is a partial function $v: \mathcal{V} \rightarrow \{0, 1\}$ such that for each $x \in \mathcal{V}$, $v(x)$ is a Boolean
 152 value if x is defined in v , otherwise x is undefined in v .

153 For each assignment v , variable $x \in \mathcal{V}$ and value $i \in \{0, 1\}$, we denote by
 154 $v[x \mapsto i]$ the assignment that agrees with v except for the variable x , i.e., for
 155 each variable $y \in \mathcal{V}$,

$$v[x \mapsto i](y) = \begin{cases} v(y), & \text{if } y \neq x; \\ i, & \text{otherwise.} \end{cases}$$

156 **Satisfiability and maximum satisfiability.** Given a Boolean formula Φ , the
 157 *satisfiability problem* (SAT) is to determine whether a satisfying assignment of
 158 Φ exists or not. If Φ is satisfied, then a solution is produced as a witness. It is
 159 well-known that the SAT problem is **NP**-complete [22].

160 Given a pair of Boolean formulas (Φ, Ψ) , the *maximum satisfiability problem*
 161 (MaxSAT) is to find a satisfying assignment that satisfies the Boolean formula
 162 Φ and meanwhile maximizes the number of satisfied clauses in Ψ . The clauses
 163 in Φ are usually called *hard* constraints, while the clauses in Ψ are called *soft*
 164 constraints. It is easy to see that the MaxSAT problem is at least **NP**-hard and
 165 can be solved by the state-of-the-art solvers such as Z3 [44].

166 In this work, by solvers we mean tools that are able to produce one satisfying
 167 assignment of the (Max)SAT problem whilst by samplers we mean those that
 168 are able to generate more than one satisfying assignments.

169 **Independent support.** Given a Boolean formula Φ , an *independent support*
 170 **Supp** of Φ [30], is a set of variables such that for each pair of satisfying as-
 171 signments (v, v') of Φ , if $v(x) = v'(x)$ holds for all variables $x \in \text{Supp}$, then
 172 $v(y) = v'(y)$ holds for all variables $y \in \mathcal{V} \setminus \text{Supp}$. Intuitively, the truth values of

173 the independent support Supp_Φ uniquely determine the truth values of the other
 174 variables. In other words, flipping the truth value of any variable $y \in \mathcal{V} \setminus \text{Supp}$ in
 175 the satisfying assignment v only will make the resulting assignment $v[y \mapsto \neg v(y)]$
 176 fail to satisfy Φ .

177 It is easy to see that any superset of an independent support of Φ is also an
 178 independent support. There are tools, such as MIS and SMIS [48], that are able
 179 to compute minimal and minimum independent supports for Boolean formulas,
 180 where *minimal* means removing any variable from the independent support X
 181 will lead to a non-independent support, and *minimum* means there does not
 182 exist any independent support whose size is smaller. Remark that the problem of
 183 deciding whether a set of variables is a minimal independent support of a
 184 Boolean formula Φ is **DP**-complete [49], where $\mathbf{DP} := \{A - B \mid A, B \in \mathbf{NP}\}$.

185 3. Derivation Procedure

186 In this section, we first present a motivating example which exemplifies the
 187 key insight behind our approach for efficiently generating a large number of
 188 satisfying assignments. We then provide a derivation procedure which is able
 189 to derive more satisfying assignments from a seed by flipping the truth values
 190 of properly chosen variables without invoking computationally expensive SAT
 191 solving. The derivation procedure is the basis for efficiently generating a large
 192 number of satisfying assignments, and can be integrated into other samplers.

193 3.1. Motivating Example

194 To exemplify the key insight behind our approach, let us consider the fol-
 195 lowing Boolean formula

$$\Phi_e \equiv (\neg a \vee b \vee c) \wedge (a \vee \neg c \vee \neg d) \wedge (\neg b \vee c) \wedge (b \vee d).$$

196 Suppose we have already obtained one satisfying assignment v (called *seed*) of
 197 Φ_e with $v(a) = v(b) = v(c) = v(d) = 1$. We can observe that the clause $\neg a \vee b \vee c$
 198 (resp. $b \vee d$) contains two literals b and c (resp. b and d) whose values are 1
 199 under the assignment v . Moreover, the common literal b does not appear in the
 200 other clauses, namely, $a \vee \neg c \vee \neg d$ and $\neg b \vee c$. By flipping the value $v(b)$ of the
 201 variable b in the assignment v , we can obtain a new assignment $v[b \mapsto \neg v(b)]$,
 202 which is also a satisfying assignment of Φ_e .

203 However, by flipping the value $v(c)$ of the variable c in the assignment v ,
 204 the new assignment $v[c \mapsto \neg v(c)]$ is not a satisfying assignment of Φ_e . This is
 205 because the clause $\neg b \vee c$ contains only one literal c whose value is 1 under the
 206 assignment v . After flipping the value $v(c)$ of the variable c in the assignment
 207 v , the clause $\neg b \vee c$ is no more satisfied.

208 This simple observation suggests that, for a seed v , we may identify proper
 209 variables (such as b but not c in the above example) so that when the value of
 210 one such variable is flipped it is still a satisfying assignment. Furthermore, the
 211 new satisfying assignments can be used as seeds to derive more satisfying assign-
 212 ments. This often allows generation of a larger number of satisfying assignments
 213 without invoking computationally expensive SAT solving.

Algorithm 1 Deriving satisfying assignments from a seed

```
1: procedure DERIVATION( $\Phi$ ,  $v$ , MaxNum, Supp)
2:   Derived =  $\{v\}$ ;
3:   Queue =  $[v]$ ;
4:   while Queue  $\neq \emptyset$   $\wedge$   $|\mathbf{Derived}| \leq \mathbf{MaxNum}$  do
5:      $v = \mathbf{Queue}.\mathbf{DEQUEUE}()$ ;
6:      $L = \{x \mid v(x) = 1\} \cup \{\neg x \mid v(x) = 0\}$ ;
7:     for all  $l \in L \wedge \mathbf{var}(l) \in \mathbf{Supp}$  do
8:       if  $\forall \bigvee_{i=1}^m l_i \in \Phi_l, \exists i. (1 \leq i \leq m \wedge l \neq l_i \wedge l_i \in L)$  then
9:          $x = \mathbf{var}(l)$ ;
10:         $v' = v[x \mapsto \neg v(x)]$ ;
11:        if  $v' \notin \mathbf{Derived}$  then
12:          Derived = Derived  $\cup \{v'\}$ ;
13:          Queue.ENQUEUE( $v'$ );
14:        end if
15:      end if
16:    end for
17:  end while
18:  return Derived;
19: end procedure
```

214 *3.2. Derivation Algorithm*

215 In this subsection, we present a derivation procedure for deriving new satisfy-
216 ing assignments from a given seed. Given a Boolean formula Φ , a seed v , an
217 independent support **Supp** of Φ , and the maximal number **MaxNum** of expected
218 satisfying assignments, the procedure DERIVATION in Algorithm 1 iteratively
219 derives new satisfying assignments from the seed v until no new satisfying as-
220 signment can be found or the number of generated satisfying assignments hits
221 the threshold **MaxNum**. It returns the set of generated satisfying assignments
222 including the original seed v .

223 To start, Algorithm 1 initializes the set **Derived** for recording all the gener-
224 ated satisfying assignments (Line 2) and the queue **Queue** for storing the seeds
225 (Line 3). It then repeats the following procedure until no new satisfying as-
226 signments can be found or the number of the generated satisfying assignments
227 exceeds the threshold **MaxNum** (While-loop).

228 For each seed v in **Queue** (Line 5), it first identifies all the literals whose
229 value is 1 under the assignment v (Line 6). After that, for each literal $l \in L$
230 whose variable $\mathbf{var}(l) \in \mathbf{Supp}$ (Line 7), it checks, for each clause $\bigvee_{i=j}^m l_j$ that
231 contains the literal l (i.e., $\bigvee_{i=j}^m l_j \in \Phi_l$), whether $\bigvee_{i=j}^m l_j$ contains a distinct
232 literal l_i whose value is also 1, i.e., $l_i \in L$ (Line 8). If this is the case, we can
233 deduce that the assignment $v[x \mapsto \neg v(x)]$ obtained from the assignment v by
234 flipping the variable $x = \mathbf{var}(l)$ is also a satisfying assignment of Φ . Therefore,
235 we extract the variable x from the literal l (Line 9) and construct the assignment
236 $v' = v[x \mapsto \neg v(x)]$ (Line 10). If the assignment v' has not been generated before,

237 it is inserted to `Derived` and `Queue` (Lines 12 and 13).

238 One may notice that only variables in `Supp` are considered for flipping
 239 (Line 7). In general, we can take all the variables into account for flipping.
 240 However, as mentioned before (cf. Section 2), flipping variables outside of `Supp`
 241 will definitely lead to unsatisfying assignments. Therefore, it suffices to consider
 242 variables from `Supp` for flipping. Due to this, the values of each variable outside
 243 of `Supp` are the same in all the generated satisfying assignments from a given
 244 seed.

245 We remark that the derivation procedure `DERIVATION` could alternatively
 246 be presented as a recursive procedure which invokes itself when a new satisfying
 247 assignment is generated, or equivalently, use a stack rather than a queue to store
 248 the generated seeds. Intuitively, using the queue `Queue` to store the seeds, our
 249 algorithm works in a breadth-first fashion, while the other two ways would follow
 250 a depth-first fashion. We adopt the current way because it is more efficient than
 251 the other two ways.

252 **Theorem 1.** *Given a Boolean formula Φ , a seed v and an independent support*
 253 *`Supp` of Φ , the set `Derived` returned by Algorithm 1 contains only satisfying*
 254 *assignments of Φ . Moreover, these assignments agree on the variables outside*
 255 *of `Supp`.*

256 **PROOF.** We show that the set `Derived` returned by Algorithm 1 contains only
 257 satisfying assignments of Φ by applying induction on the sequence $v_0 v_1 \dots$ of
 258 the assignments added into `Derived`. The base case is trivial as the seed v_0
 259 satisfies the Boolean formula Φ . We prove the inductive step below.

260 Suppose $v_0, v_1 \dots v_{k-1}$ have been added into the set `Derived` and the in-
 261 ductive step adds the assignment v_k into the set `Derived`. Then, v_k must be
 262 added due to one v of the previously added satisfying assignments $v_0, v_1 \dots v_{k-1}$.
 263 There necessarily exists a literal l such that $x = \text{var}(l)$ and $v_k = v[x \mapsto \neg v(x)]$.

264 To show that v_k satisfies Φ , it is sufficient to prove that v_k satisfies all the
 265 clauses of Φ . Let us consider a clause $\bigvee_{i=j}^m l_j$ of Φ ,

- 266 • If $\bigvee_{i=j}^m l_j$ does not contain the literal l , then by applying induction hy-
 267 pothesis, v satisfies the Boolean formula Φ and hence v satisfies the clause
 268 $\bigvee_{i=j}^m l_j$. Since $v_k = v[x \mapsto \neg v(x)]$ and $x = \text{var}(l)$, the truth of the clause
 269 $\bigvee_{i=j}^m l_j$ does not change when the value of x in v is flipped. Therefore, we
 270 get that the assignment v_k satisfies the clause $\bigvee_{i=j}^m l_j$.
- 271 • If $\bigvee_{i=1}^m l_i$ contains the literal l , then there exists another literal $l_i \in$
 272 $\{l_1, \dots, l_m\}$ such that $l_i \neq l$ and $l_i \in L = \{x \mid v(x) = 1\} \cup \{\neg x \mid v(x) = 0\}$.
 273 From $l_i \in L = \{x \mid v(x) = 1\} \cup \{\neg x \mid v(x) = 0\}$, we deduce that the literal
 274 l_i , hence the clause $\bigvee_{i=1}^m l_i$, holds under the assignment v_k .

275 **Example 1.** *Recall the motivating example Φ_e . Suppose the input seed is v_1*
 276 *with $v_1(a) = v_1(b) = v_1(c) = v_1(d) = 1$ and the independent support `Supp` =*
 277 *$\{a, b, d\}$. The derivation steps are shown in Figure 1. At the beginning of the*
 278 *first iteration of the while-loop, $v = v_1$ and $L = \{a, b, c, d\}$.*

$$\begin{array}{l}
\Phi_e : \quad (\neg a \vee b \vee c) \quad \wedge \quad (a \vee \neg c \vee \neg d) \quad \wedge \quad (\neg b \vee c) \quad \wedge \quad (b \vee d) \\
v_1 : \quad (0 \vee 1 \vee 1) \quad \wedge \quad (1 \vee 0 \vee 0) \quad \wedge \quad (0 \vee 1) \quad \wedge \quad (1 \vee 1) \\
\quad \text{flip } b \text{ and } d \text{ respectively } \Downarrow \\
v_2 : \quad (0 \vee \mathbf{0} \vee 1) \quad \wedge \quad (1 \vee 0 \vee 0) \quad \wedge \quad (\mathbf{1} \vee 1) \quad \wedge \quad (\mathbf{0} \vee 1) \\
v_3 : \quad (0 \vee 1 \vee 1) \quad \wedge \quad (1 \vee 0 \vee \mathbf{1}) \quad \wedge \quad (0 \vee 1) \quad \wedge \quad (1 \vee \mathbf{0}) \\
\quad \text{flip } a \Downarrow \\
v_4 : \quad (\mathbf{1} \vee 1 \vee 1) \quad \wedge \quad (\mathbf{0} \vee 0 \vee 1) \quad \wedge \quad (0 \vee 1) \quad \wedge \quad (1 \vee 0)
\end{array}$$

Figure 1: Derivation steps of the motivating example

- 279 1. Suppose the variable a is chosen for flipping (Line 7). Since the clause $a \vee$
280 $\neg c \vee \neg d$ does not have any literals other than a that occur in L , Algorithm 1
281 will not flip the variable a .
- 282 2. Next, the variable b is chosen for flipping (Line 7). Since the clause $\neg a \vee$
283 $b \vee c$ contains the literal c , the clause $b \vee d$ contains the literal d , and both
284 literals c and d occur in L , Algorithm 1 will flip the variable b (Line 9)
285 and produce a new satisfying assignment $v_2 = v_1[b \mapsto 0]$ (Line 10).
- 286 3. Finally, the variable d is chosen for flipping (Line 7). Since the clause
287 $b \vee d$ contains literal b that occurs in L , Algorithm 1 will flip the vari-
288 able d (Line 9) and produce a new satisfying assignment $v_3 = v_1[d \mapsto 0]$
289 (Line 10).

290 At the end of the first iteration of the while-loop, $\text{Derived} = \{v_1, v_2, v_3\}$ and
291 $\text{Queue} = [v_2, v_3]$. After entering the second iteration of the while-loop, $v = v_2$,
292 Queue (resp. L) becomes $[v_3]$ (resp. $\{a, \neg b, c, d\}$). By applying similar steps
293 as above, the satisfying assignment v_2 is regenerated but will not be inserted to
294 Derived or Queue .

295 At the end of the second iteration of the while-loop, $\text{Derived} = \{v_1, v_2, v_3\}$
296 and $\text{Queue} = [v_3]$. After entering the third iteration of the while-loop, $v = v_3$,
297 Queue (resp. L) becomes \emptyset (resp. $\{a, b, c, \neg d\}$). By applying similar steps as
298 above, Algorithm 1 will flip the variable a and produce a new satisfying assign-
299 ment $v_4 = v_3[a \mapsto 0]$. In the end, no more new satisfying assignments can be
300 generated and Algorithm 1 returns the set $\{v_1, v_2, v_3, v_4\}$.

301 4. ESAMPLER

302 In this section, we show that our derivation procedure is of generic nature
303 in the sense that it can be integrated with other samplers. The basic idea is to
304 generate seeds by invoking an existing sampler as an iterator, which returns a
305 unique satisfying assignment each time. For each seed, we derive more satisfy-
306 ing assignments by invoking our derivation procedure. However, our derivation
307 procedure may not be effective on some Boolean formulas. Therefore, we propo-
308 se a heuristic to determine whether our derivation procedure is able to derive
309 a large number of satisfying assignments or not. If it can derive a large number

Algorithm 2 Integrated our derivation procedure into an existing sampler

```
1: procedure INTEGRATEDSAMPLER(Sampler,  $\Phi$ , T, MaxPerSeed, Supp, RT, DT)
2:   Solutions =  $\emptyset$ ;
3:   Derivable = false;
4:   Round = 0;
5:   Iterator = Sampler( $\Phi$ , Supp);
6:   repeat
7:     v = Iterator.next();
8:     if v == Null then
9:       break;
10:    end if
11:    if v  $\in$  Solutions then
12:      continue;
13:    end if
14:    if Derivable == true  $\vee$  Round < RT then
15:      Derived = DERIVATION( $\Phi$ , v, MaxPerSeed, Supp);
16:      Solutions = Solutions  $\cup$  Derived;
17:      if  $|\text{Derived}| \geq \text{DT}$  then
18:        Derivable = true;
19:      else
20:        Round = Round + 1;
21:      end if
22:    else
23:      Solutions = Solutions  $\cup$  {v};
24:    end if
25:  until T is satisfied
26:  return Solutions;
27: end procedure
```

310 of satisfying assignments, we apply the derivation procedure for each satisfying
311 assignment generated by the sampler, otherwise we disable it.

312 Our idea is formalized as the procedure INTEGRATEDSAMPLER in Algo-
313 rithm 2, which takes, as input, an off-the-shelf sampler *Sampler*, a Boolean
314 formula Φ , a threshold *T* as the termination condition, the maximum number
315 *MaxPerSeed* of satisfying assignments per seed, an independent support *Supp*
316 of the Boolean formula Φ , two thresholds *RT* and *DT* to determine whether our
317 derivation procedure is able to derive a large number of satisfying assignments,
318 and returns a set *Solutions* of satisfying assignments of the formula Φ .

319 The procedure INTEGRATEDSAMPLER first initializes the set *Solutions*, the
320 Boolean flag *Derivable*, the counter *Round* and the iterator *Iterator* of the
321 sampler using the independent support *Supp* and Boolean formula Φ (Lines 2–
322 5), where the Boolean flag *Derivable* and counter *Round* are used to determine
323 if our derivation procedure is able to derive a large number of satisfying as-
324 signments. Then, it repeats the following procedure until the threshold *T* is

325 hit.

326 During each iteration, INTEGRATEDSAMPLER first invokes the iterator to get
327 a satisfying assignment v , where v is `Null` if Φ is unsatisfiable or the iterator
328 cannot find new satisfying assignments. If v is `Null`, it breaks the loop (Line 9).
329 If v already exists in `Solutions`, it skips this loop (Line 12). Otherwise it checks
330 if the Boolean flag `Derivable` is true or the number `Round` of iterations is less
331 than the threshold `RT`.

- 332 • If neither holds, the derivation procedure is considered to be not able to
333 derive a large number of satisfying assignments and will be skipped;
- 334 • Otherwise, the derivation procedure is invoked to generate more satisfying
335 assignments which are added to the set `Solutions` (Lines 15–16). If the
336 number of satisfying assignments generated by the derivation procedure
337 exceeds the threshold `DT`, we consider that the derivation procedure is able
338 to derive a large number of satisfying assignments and set the Boolean flag
339 `Derivable` to true (Line 18). Otherwise, we increase the counter `Round` by
340 one. In general, we probe the effectiveness of the derivation procedure by
341 checking the number of satisfying assignments generated by the derivation
342 procedure in the first `RT` iterations. In our experiments, we found few
343 rounds are sufficient to detect for each benchmark whether a large number
344 of satisfying assignments can be derived from a seed. In fact, on some
345 benchmarks, the derivation algorithm can derive a few satisfying solutions
346 from a seed in the beginning, but no solution could be derived afterwards.
347 Thus, a small `DT` value can be used to avoid `Derivable` being set to true on
348 these benchmarks, while it will not change on other benchmarks. Based
349 on these observations, we set `RT` = 3 and `DT` = 16.

350 By Theorem 1, we obtain that

351 **Theorem 2.** *The set `Solutions` returned by Algorithm 2 contains only satis-*
352 *fying assignments of Φ .*

353 5. Implementation and Evaluation

354 We implement Algorithms 1–2 as an open-source tool `ESAMPLER` in C++,
355 with `QUICKSAMPLER` as the underlying seed generator. `QUICKSAMPLER` takes
356 a Boolean formula and its independent support as inputs, and outputs a set of
357 assignments. However, as mentioned above, assignments produced by `QUICK-`
358 `SAMPLER` may be duplicated or not satisfy the formula. As we focus on satisfy-
359 ing assignments of each Boolean formula in this work, we modify it so that du-
360 plicated and unsatisfying assignments are omitted. To demonstrate the generic
361 nature of Algorithm 1 for deriving satisfying assignments from a seed, we also
362 implement Algorithm 2 with `UNIGEN3` as the underlying seed generator in our
363 tool `ESAMPLER`. In contrast to `QUICKSAMPLER`, `UNIGEN3` only produces satisfy-
364 ing assignments for each given Boolean formula and the satisfying assignments
365 are sampled uniformly at random with theoretical guarantees.

366 ESAMPLER takes a Boolean formula in the DIMACS [50] format and other
367 required options as inputs, and outputs a set of satisfying assignments for the
368 given Boolean formula. To reduce the memory usage of storing the satisfy-
369 ing assignments, we only store and output the satisfying assignments for the
370 variables in the given independent support. Indeed, the truth values of the in-
371 dependent support determine those of the other variables, thereby the satisfying
372 assignments can be easily completed.

373 In the rest of this work, we denote by ESAMPLER+QS (resp. ESAMPLER+UG)
374 our tool ESAMPLER using QUICKSAMPLER (resp. UNIGEN3) as the underlying
375 seed generator.

376 We mainly compare ESAMPLER+QS with three state-of-the-art tools QUICK-
377 SAMPLER, STS and UNIGEN3 [29]. As done by [28], for a fair comparison, we
378 modify STS so that the additional independent support information can be
379 used by STS. To show the generic nature of Algorithm 1, we also compare
380 ESAMPLER+UG with UNIGEN3.

381 **Benchmarks.** To evaluate the performance, we conducted extensive exper-
382 iments. Industrial testing and verification instances are typically proprietary
383 and unavailable for published research. Therefore, we conducted experiments
384 on the publicly available benchmarks from UNIGEN [30], which consist of 370
385 Boolean formulas in the DIMACS format and the independent supports thereof.
386 Indeed, the independent supports of most Boolean formulas could be computed
387 using MIS [48] in few seconds. These benchmarks come from four classes of
388 problem instances:

- 389 1. ISCAS89: constraints arising from ISCAS89 circuits with parity conditions
390 on randomly chosen subsets of outputs and next-state variables;
- 391 2. SMTLib: bit-blasted versions of SMTLib benchmarks;
- 392 3. ProgSyn: constraints arising from automated program synthesis; and
- 393 4. BMC: constraints arising in bounded model checking of circuits.

394 Note that the accompanied independent supports of these benchmarks may con-
395 tain variables that are not involved in the corresponding Boolean formulas; such
396 variables are removed from the independent supports in our experiments. We
397 remark that our approach also works without the given independent supports,
398 in which case the independent support of a Boolean formula contains all the
399 involved variables.

400 Since it does not make any sense to compute solutions for unsatisfiable
401 Boolean formulas or the satisfiability cannot be solved, we checked the satis-
402 fiability of all these Boolean formulas with a timeout of one hour per Boolean
403 formula using Z3 [51]. There are two unsatisfiable formulas (79.sk_4_40 and
404 36.sk_3_77), and four unsolvable formulas (logcount.sk_16_86, log2.sk_72_391,
405 xpose.sk_6_134, and listReverse.sk_11_43). These formulas are not considered
406 here, leaving 364 Boolean formulas.

407 **Experiment setup.** In our experiments, for each sampler and each Boolean
408 formula, we run the sampler once on the Boolean formula. Though samplers

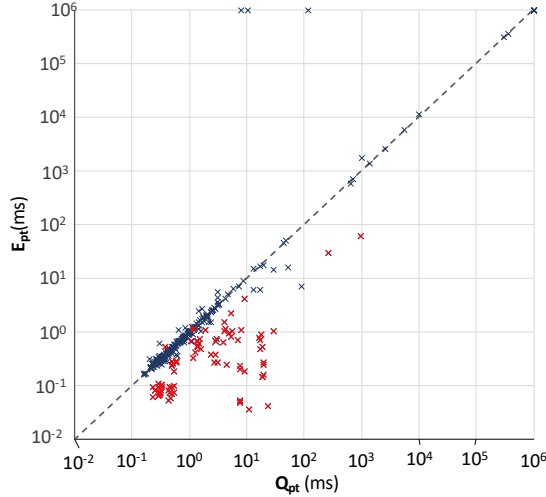


Figure 2: ESAMPLER+QS vs. QUICKSAMPLER

409 are non-deterministic, results on a large number of Boolean formulas are suf-
 410 ficient to demonstrate the performance of ESAMPLER. The maximal number
 411 **MaxPerSeed** of satisfying assignments per seed is 10,000 and the maximal num-
 412 ber **T** of satisfying assignments to compute is 1,000,000, unless the recent 10
 413 assignments/pseudosolutions already exist. As aforementioned, we set $RT = 3$
 414 and $DT = 16$ for ESAMPLER. For STS and QUICKSAMPLER, we use their default
 415 parameter settings. All the experiments were conducted on Intel Xeon E5-2620
 416 v4 2.10GHz CPU with 256 RAM GB and the one-hour timeout.

417 5.1. Comparison of ESAMPLER+QS and QUICKSAMPLER

418 Figure 2 shows the scatter plot comparing the average execution time per
 419 satisfying assignment between ESAMPLER+QS and QUICKSAMPLER on all the
 420 364 formulas. Timeout occurred along the top or right border; the red color indi-
 421 cates that **Derivable** is set true by Algorithm 2, namely, it determines that our
 422 derivation procedure is able to derive a large number of satisfying assignments.
 423 Points below (resp. above) the diagonal line indicate that ESAMPLER+QS per-
 424 forms better (resp. worse) than QUICKSAMPLER.

425 The comparison of QUICKSAMPLER and ESAMPLER+QS for a representa-
 426 tive subset of the benchmarks is reported in Table 1. Columns **benchmark**,
 427 **#Vars** and **#Cls** respectively show the name, numbers of variables and clauses
 428 in each Boolean formula. Columns Q_t and E_t (resp. Q_{pt} and E_{pt}) give the total
 429 execution time in thousand seconds (ks) (resp. execution time per satisfying as-
 430 signment in milliseconds (ms)) of QUICKSAMPLER and ESAMPLER+QS, respec-
 431 tively. Columns Q_n and E_n show the total numbers of satisfying assignments
 432 generated by QUICKSAMPLER and ESAMPLER+QS, respectively. Column E_{dn}
 433 gives the numbers of satisfying assignments generated by our derivation proce-
 434 dure. The last column provides the ratio of execution time per satisfying as-

Table 1: Comparison of QUICKSAMPLER and ESAMPLER+QS

Benchmark	#Vars	#Cls	$Q_t(ks)$	Q_n	$Q_{pt}(ms)$	$E_t(ks)$	E_n	E_{dn}	$E_{pt}(ms)$	$\frac{Q_{pt}}{E_{pt}}$
s27_new_15_7	17	43	0.00	48	1.39	0.00	48	42	0.54	2.56
blasted_case_54	203	725	0.20	691,127	0.30	0.20	664,548	0	0.30	0.99
20.sk_1_51	15,475	60,994	3.94	491,074	8.02	1.67	1,520,152	~1,520k	1.10	7.31
s35932_7_4	17,849	44,425	4.22	245,506	17.17	0.63	1,270,247	~1,270k	0.50	34
blasted_case_126	302	1,129	0.34	1,007,411	0.34	0.34	1,022,991	0	0.33	1.03
blasted_case_40	245	650	0.41	1,149,017	0.35	0.41	1,149,017	0	0.36	0.99
s349_3_2	198	469	0.24	1,008,386	0.24	0.07	1,142,757	~1,088k	0.06	3.81
56.sk_6_38	4,842	17,828	1.97	1,004,037	1.96	1.18	1,093,080	~1,092k	1.08	1.81
blasted_case_107	618	1,661	0.82	1,149,017	0.72	0.84	1,149,017	0	0.73	0.98
s832a_15_7	693	2,017	0.53	1,001,732	0.53	0.52	1,000,093	4	0.52	1.01
s420_new_7_4	312	770	0.35	1,117,085	0.31	0.08	1,048,576	~1,043k	0.07	4.18
blasted_case_124	133	386	0.23	1,039,563	0.22	0.22	1,008,715	0	0.22	1.02
s35932_15_7	17,918	44,709	4.29	145,499	29.46	1.34	1,270,247	~1,270k	1.06	27
blasted_case_207	824	2,128	1.02	1,149,017	0.89	0.98	1,149,017	0	0.86	1.04
blasted_case_120	284	851	0.41	1,113,780	0.37	0.40	1,044,731	0	0.38	0.97
63.sk_3_64	7,242	24,379	4.04	917,681	4.41	0.30	1,200,120	~1,200k	0.25	17
s420_7_4	312	770	0.32	1,058,100	0.31	0.10	1,366,784	~1,363k	0.07	4.14

435 signment between QUICKSAMPLER and ESAMPLER+QS, depicting the speedup
436 of ESAMPLER+QS. We can observe when our derivation procedure works, it
437 can produce more satisfying assignments (e.g., 20.sk_1_51 and s35932_7.4) than
438 QUICKSAMPLER in the same time budget, while when it does not work well,

439 it often does not produce any satisfying assignments (e.g., `blasted_case.54` and
 440 `blasted_case.40`). Note that, since QUICKSAMPLER is a randomized approach,
 441 QUICKSAMPLER and ESAMPLER+QS may produce different satisfying assign-
 442 ments when our derivation procedure does not work, although ESAMPLER+QS
 443 is built on QUICKSAMPLER.

444 **Summary.** ESAMPLER+QS and QUICKSAMPLER respectively failed on 11 and
 445 7 benchmarks due to the failures of MaxSAT solving. The difference between the
 446 numbers of the failed benchmarks indicates that the soft constraints generated
 447 randomly slightly affect MaxSAT solving. When ESAMPLER+QS determined
 448 that the derivation procedure can generate a large number of satisfying assign-
 449 ments, ESAMPLER+QS performed better than QUICKSAMPLER on almost all
 450 the benchmarks. While ESAMPLER+QS determined that our derivation proce-
 451 dure was not able to generate a large number of satisfying assignments, ESAM-
 452 PLER+QS was comparable to QUICKSAMPLER. Specifically, ESAMPLER+QS
 453 was faster than QUICKSAMPLER on 227 benchmarks. It was $1.66\times$ faster on av-
 454 erage and more than $5\times$ faster on 41 benchmarks, while it was 1.2 times slower
 455 on 16 benchmarks.

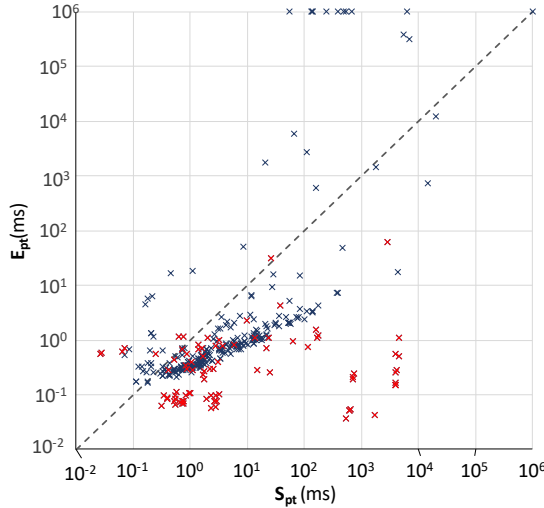


Figure 3: ESAMPLER+QS vs. STS

456 *5.2. Comparison of ESAMPLER+QS and STS*

457 Figure 3 shows the scatter plot comparing the average execution time per
 458 satisfying assignment between ESAMPLER+QS and STS on all the 364 formu-
 459 las. Recall that timeout occurred along the top or right border, the red color
 460 indicates that `Derivable` is set true by Algorithm 2, and points below the diag-
 461 onal line indicate that ESAMPLER+QS performs better than QUICKSAMPLER,
 462 and vice versa.

Table 2: Comparison of STS and ESAMPLER+QS

Benchmark	#Vars	#Cls	$S_t(ks)$	S_n	$S_{pt}(ms)$	$E_t(ks)$	E_n	E_{tn}	$E_{pt}(ms)$	$\frac{S_{pt}}{E_{pt}}$
s27_new_15_7	17	43	0.00	48	0.85	0.00	48	42	0.54	1.57
blasted_case_54	203	725	1.45	961,782	1.51	0.20	664,548	0	0.30	5.06
20_sk_1_51	15,475	60,994	3.60	151,948	23.69	1.67	1,520,152	~1,520k	1.10	21
s35932_7_4	17,849	44,425	3.49	800	4.361	0.63	1,270,247	~1,270k	0.50	8.757
blasted_case_126	302	1,129	0.92	1,000,006	0.92	0.34	1,022,991	0	0.33	2.78
blasted_case_40	245	650	1.53	1,000,000	1.53	0.41	1,149,017	0	0.36	4.30
s349_3_2	198	469	0.31	1,000,028	0.31	0.07	1,142,757	~1,088k	0.06	4.94
56_sk_6_38	4,842	17,828	1.99	1,000,048	1.99	1.18	1,093,080	~1,092k	1.08	1.84
blasted_case_107	618	1,661	3.60	558,950	6.44	0.84	1,149,017	0	0.73	8.82
s832a_15_7	693	2,017	1.55	1,000,018	1.55	0.52	1,000,093	4	0.52	2.97
s420_new_7_4	312	770	0.72	1,000,001	0.72	0.08	1,048,576	~1,043k	0.07	9.68
blasted_case_124	133	386	0.32	1,000,013	0.32	0.22	1,008,715	0	0.22	1.47
s35932_15_7	17,918	44,709	3.50	800	4.380	1.34	1,270,247	~1,270k	1.06	4.140
blasted_case_207	824	2,128	3.60	276,250	13.03	0.98	1,149,017	0	0.86	15
blasted_case_120	284	851	1.59	1,000,000	1.59	0.40	1,044,731	0	0.38	4.13
63_sk_3_64	7,242	24,379	3.60	148,050	24.31	0.30	1,200,120	~1,200k	0.25	97
s420_7_4	312	770	0.74	1,000,038	0.74	0.10	1,366,784	~1,363k	0.07	9.93

463 Table 2 reports the performance of STS and ESAMPLER+QS for the same
464 representative subset of the benchmarks. Column S_t (resp. S_{pt}) gives the total
465 execution time in thousand seconds (ks) (resp. execution time per satisfying
466 assignment in milliseconds (ms)) of STS. Column S_n shows the total number

467 of satisfying assignments generated by STS for each Boolean formula. The last
 468 column provides the ratio of execution time per satisfying assignment between
 469 STS and ESAMPLER+QS, depicting the speedup of ESAMPLER+QS.

470 **Summary.** STS failed on 1 benchmark because the underlying SAT solver
 471 Minisat failed to solve the Boolean formula, while ESAMPLER+QS failed on 11
 472 benchmarks. In general, ESAMPLER+QS performed better than STS on most
 473 benchmarks. It was faster on 316 benchmarks ($5.47\times$ faster on average and
 474 more than $10\times$ faster on 93 benchmarks), while it was 1.2 times slower on only
 475 45 benchmarks.

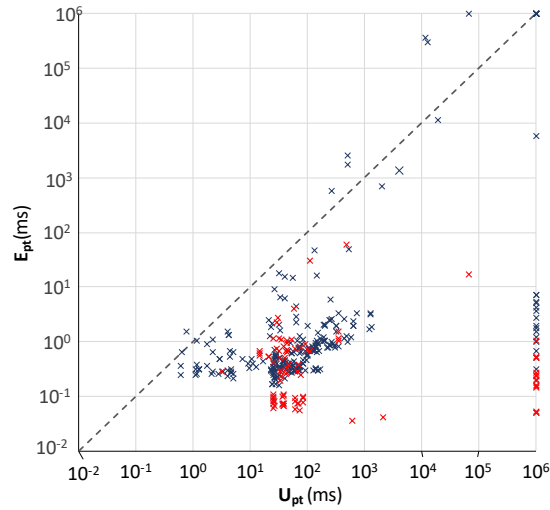


Figure 4: ESAMPLER+QS vs. UNIGEN3

476 *5.3. Comparison of ESAMPLER+QS and UNIGEN3*

477 Figure 4 shows the scatter plot comparing the average execution time per
 478 satisfying assignment between ESAMPLER+QS and UNIGEN3 on all the 364
 479 formulas. Almost all the points are below the diagonal line, indicating ESAM-
 480 PLER+QS significantly outperforms UNIGEN3.

481 Table 3 reports the performance of UNIGEN3 and ESAMPLER+QS on the
 482 same representative subset of benchmarks. Column U_t (resp. U_{pt}) gives the
 483 total execution time in thousand seconds (ks) (resp. execution time per sat-
 484 isfying assignment in milliseconds (ms)) of UNIGEN3. Column U_n shows the
 485 total number of satisfying assignments generated by UNIGEN3 for each Boolean
 486 formula. The last column provides the ratio of execution time per satisfying
 487 assignment between UNIGEN3 and ESAMPLER+QS, depicting the speedup of
 488 ESAMPLER+QS.

489 **Summary.** UNIGEN3 failed on 40 benchmarks. Recall that ESAMPLER+QS
 490 failed on 11 benchmarks. No matter whether or not ESAMPLER+QS determined

Table 3: Comparison of UNIGEN3 and ESAMPLER+QS

Benchmark	#Vars	#Cls	$U_t(ks)$	U_n	$U_{\mu}(ms)$	$E_t(ks)$	E_n	E_{dn}	$E_{\mu}(ms)$	$\frac{U_{\mu}}{E_{\mu}}$
s27_new_15_7	17	43	0.00	48	20.83	0.00	48	42	0.54	38.57
blasted_case_54	203	725	3.60	158,168	22.76	0.20	664,548	0	0.30	75.87
20_sk_1_51	15,475	60,994	3.60	70,312	51.21	1.67	1,520,152	~1,520k	1.10	46.56
s35932_7_4	17,849	44,425	3.60	0	-	0.63	1,270,247	~1,270k	0.50	-
blasted_case_126	302	1,129	3.60	77,185	46.67	0.34	1,022,991	0	0.33	141.4
blasted_case_40	245	650	3.60	50,380	71.46	0.41	1,149,017	0	0.36	198.5
s349_3_2	198	469	3.60	144,279	24.95	0.07	1,142,757	~1,088k	0.06	415.8
56_sk_6_38	4,842	17,828	3.60	104,149	34.57	1.18	1,093,080	~1,092k	1.08	32.01
blasted_case_107	618	1,661	3.60	0	-	0.84	1,149,017	0	0.73	-
s832a_15_7	693	2,017	3.60	132,705	27.13	0.52	1,000,093	4	0.52	52.17
s420_new_7_4	312	770	3.60	98,934	36.39	0.08	1,048,576	~1,043k	0.07	519.9
blasted_case_124	133	386	3.60	89,376	40.28	0.22	1,008,715	0	0.22	182.1
s35932_15_7	17,918	44,709	3.60	0	-	1.34	1,270,247	~1,270k	1.06	-
blasted_case_207	824	2,128	3.60	15,026	239.92	0.98	1,149,017	0	0.86	279
blasted_case_120	284	851	3.60	51,799	69.5	0.40	1,044,731	0	0.38	182.9
63_sk_3_64	7,242	24,379	3.60	48,004	75.01	0.30	1,200,120	~1,200k	0.25	300
s420_7_4	312	770	3.60	95,260	37.79	0.10	1,366,784	~1,363k	0.07	539.9

491 that the derivation procedure was able to generate a large number of satisfying
492 assignments, ESAMPLER+QS performed significantly better than UNIGEN3 on
493 almost all the benchmarks. Specifically, ESAMPLER+QS was faster than UNI-
494 GEN3 on 348 benchmarks. It was $69.8\times$ faster on average and more than $100\times$
495 faster on 194 benchmarks, while it was 1.2 times slower on only 7 benchmarks.

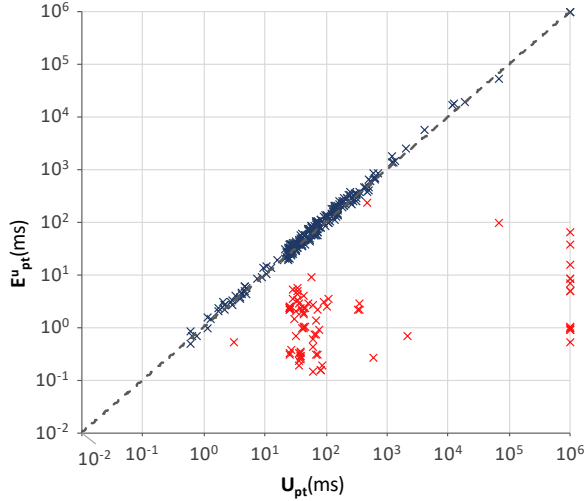


Figure 5: ESAMPLER+UG vs. UNIGEN3

496 *5.4. Comparison of ESAMPLER+UG and UNIGEN3*

497 Figure 5 shows the scatter plot comparing the average execution time per
 498 satisfying assignment between UNIGEN3 and ESAMPLER+UG on all the 364
 499 formulas. Almost all the red points are below the diagonal line while almost all
 500 the blue points are close to the diagonal line. indicating that ESAMPLER+UG
 501 significantly outperforms UNIGEN3 on the benchmarks on which `Derivable` was
 502 set to true by Algorithm 2, while it was still comparable on other benchmarks.

503 Table 4 reports the performance of UNIGEN3 and ESAMPLER+UG on the
 504 same representative subset of benchmarks. Column E_t^u gives the total execution
 505 time in thousand seconds (ks) of ESAMPLER+UG, while column E_{pt}^u gives the
 506 execution time per satisfying assignment in milliseconds (ms) . Column E_n^u
 507 shows the total number of satisfying assignments generated by ESAMPLER+UG
 508 for each Boolean formula, while column E_{dn}^u gives the numbers of satisfying
 509 assignments generated by the derivation procedure. The last column provides
 510 the ratio of execution time per satisfying assignment between UNIGEN3 and
 511 ESAMPLER+UG, depicting the speedup of our algorithm.

512 **Summary.** UNIGEN3 failed on 40 benchmarks while ESAMPLER+UG failed
 513 on 25 benchmarks. Specifically, ESAMPLER+UG was faster than UNIGEN3 on
 514 207 benchmarks. It was $2.19\times$ faster on average and more than $10\times$ faster on
 515 85 benchmarks, while there were only 12 benchmarks on which it was at least
 516 $1.2\times$ slower. The results demonstrate the generic nature of Algorithm 1 for
 517 deriving satisfying assignments from a seed using UNIGEN3 as the underlying
 518 seed generator in our tool ESAMPLER.

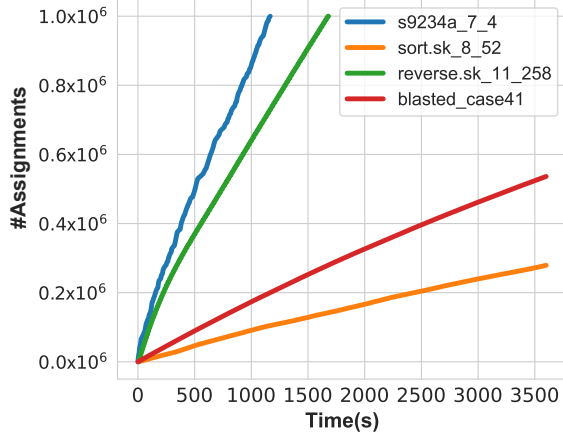
Table 4: Comparison of UNIGEN3 and ESAMPLER+UG

Benchmark	#Vars	#Cls	$U_l(ks)$	U_n	$U_{pl}(ms)$	$E_l^u(ks)$	E_n^u	E_{dn}^u	$E_{pl}^u(ms)$	$\frac{U_{pl}^u}{E_n^u}$
s27_new_15_7	17	43	0.00	48	0.02	0.00	48	42	0.01	2.20
blasted_case_54	203	725	3.60	158,168	22.8	3.60	197,693	0	18.21	1.25
20_sk_1_51	15,475	60,994	3.60	70,312	51.2	3.87	1,320,000	~1,319k	2.93	17.46
s35932_7_4	17,849	44,425	3.60	0	-	4.03	60,005	~59k	67.16	-
blasted_case_126	302	1,129	3.60	77,185	46.7	3.60	74,111	0	56.01	0.96
blasted_case_40	245	650	3.60	50,380	71.46	3.60	46,327	0	48.59	0.92
s349_3_2	198	469	3.60	144,279	24.95	2.42	1,001,811	~974k	2.41	10.35
56_sk_6_38	4,842	17,828	3.60	104,149	34.57	3.82	1,194,765	~1,194k	3.2	10.81
blasted_case_107	618	1,661	3.60	0	-	3.60	0	0	-	-
s832a_15_7	693	2,017	3.60	132,705	27.13	3.60	130,075	3	33.4	0.98
s420_new_7_4	312	770	3.60	98934	36.39	0.26	1,086,720	~1,083k	0.236	154.47
blasted_case_124	133	386	3.60	89,376	40.27	3.60	94,080	0	38.27	1.05
s35932_15_7	17,918	44,709	3.60	55	65.455	3.66	230,032	~229k	15.92	4109
blasted_case_207	824	2,128	3.61	15,026	239.9	3.61	15,054	0	239.9	1.00
blasted_case_120	284	851	3.60	51,799	69.5	3.60	64,745	0	55.6	1.25
63_sk_3_64	7,242	24,379	3.60	48,004	75.02	3.70	3,960,000	~3,959k	0.934	80.39
s420_7_4	312	770	3.60	95,260	37.79	0.32	1,096,960	~1,093k	0.292	129.95

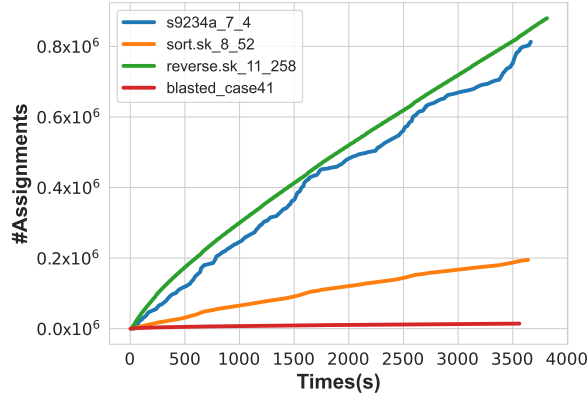
519 5.5. Execution Time vs Number of Satisfying Assignments

520 To see the relation between the execution time and the number of satisfying
521 assignments, we evaluate ESAMPLER on four randomly chosen benchmarks by
522 varying the execution time and counting the number of satisfying assignments.

523 Figure 6(a) and Figure 6(b) respectively show the plots of results on the



(a) ESAMPLER+QS



(b) ESAMPLER+UG

Figure 6: Time vs. #assignments of ESAMPLER

524 four randomly chosen benchmarks using ESAMPLER+QS and ESAMPLER+UG,
 525 where the x-axis is the execution time (in seconds) and the y-axis is number
 526 of satisfying assignments (#assignments). We can observe that the number
 527 of satisfying assignments for each benchmark is almost linear in the execution
 528 time. These results demonstrate the effectiveness of our derivation procedure.

529 5.6. Testing Uniformity

530 Similar to QUICKSAMPLER, ESAMPLER does not provide a guarantee of
 531 uniformity. Remark that UNIGEN3 provides a theoretical guarantee of unifor-
 532 mity based on hashing, at the cost of sampling efficiency. We empirically show

533 that the uniformity of the solutions can be controlled by adjusting the maximal
 534 number of solutions per seed, i.e., the parameter `MaxNumPerSeed`. We run both
 535 ESAMPLER+QS and ESAMPLER+UG on a randomly selected benchmark (i.e.,
 536 27.sk_3_32) on which our derivation procedure works, where duplicated solutions
 537 are recorded to measure uniformity and the mutation phase of QUICKSAMPLER
 538 is disabled to be more precise.

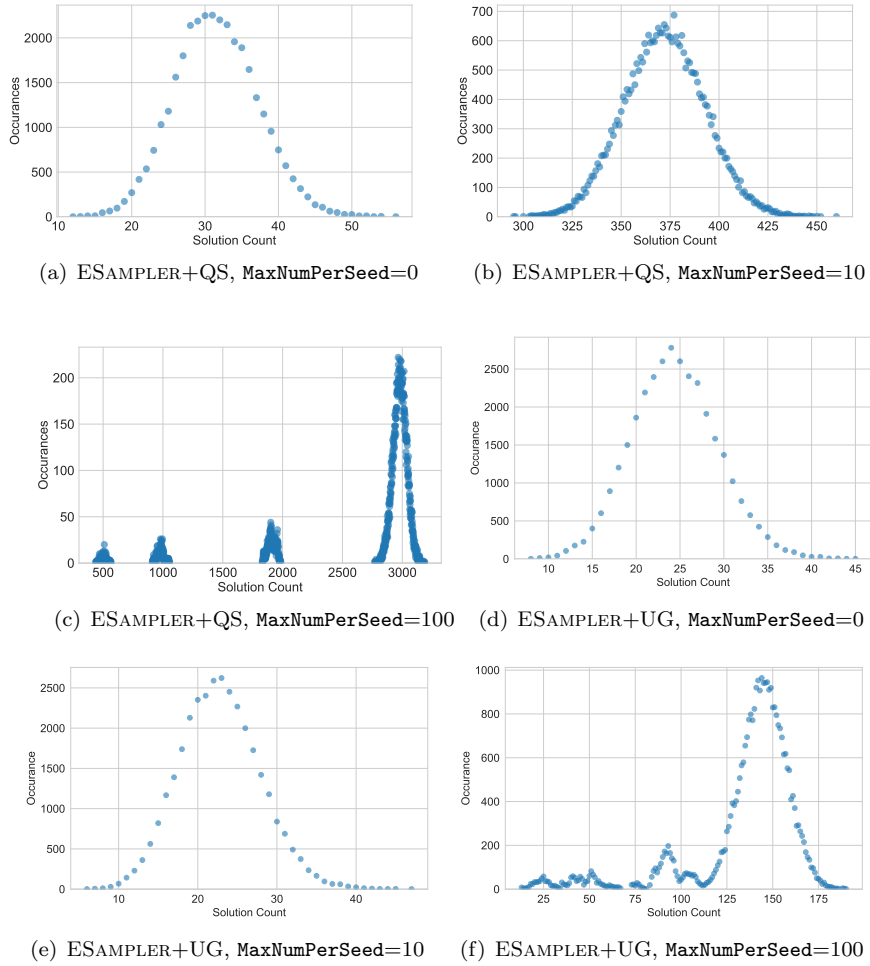


Figure 7: Distributions of solutions

539 Figure 7 depicts the distributions of solutions ESAMPLER+QS and ESAM-
 540 PLER+UG when `MaxNumPerSeed` is set to 0, 10 and 100, where (x, y) denotes
 541 that there are y unique solutions each of which occurs x times. We can observe
 542 that the smaller the parameter `MaxNumPerSeed` is, the closer the distribution

543 is to the normal distribution, meaning that the solutions generated by our tool
544 are actually close to uniform when `MaxNumPerSeed` is chosen properly.

545 6. Application to Bayesian Inference

546 In this section, to further show the effectiveness and efficiency of ESAMPLER
547 in real-world applications, we apply ESAMPLER to Bayesian inference, namely,
548 computing the posterior probability of a query given evidence in a Bayesian
549 network [52, 53].

550 **Bayesian inference.** A *Bayesian network* is a tuple (V, E, T) , where $V =$
551 $\{X_1, \dots, X_n\}$ is a finite set of nodes each of which represents a discrete random
552 variable, $E \subseteq V \times V$ is a finite set of edges each of which represents dependence
553 between two random variables. (V, E) forms a directed acyclic graph (DAG),
554 and T is a finite set of conditional probability tables (CPTs) each of which
555 encodes the conditional probability distribution of a random variable. Given a
556 random variable X , value a , and a partial assignment v of some other random
557 variables, the *Bayesian inference* is to compute the posterior probability $\Pr(X =$
558 $a \mid v)$.

559 Bayesian inference is a well-known $\#\mathbf{P}$ -complete problem [54]. Sang *et al.* [31]
560 *al.* [31] proposed an encoding from the Bayesian inference problem to the model-
561 counting problem of Boolean formulas ($\#\text{SAT}$), which we leverage to solve
562 Bayesian inference.

563 **Bayesian inference to $\#\text{SAT}$.** Given a Bayesian network (V, E, T) and a
564 Bayesian inference query $\Pr(X = a \mid v)$, Sang *et al.* [31] use *chance* variables to
565 encode entries in CPTs and *state* variables to encode the values of the nodes,
566 based on which two Boolean formulas $\Phi_1 \wedge \Phi_2 \wedge \Phi_3$ and $\Phi_1 \wedge \Phi_2$ can be con-
567 structed, where Φ_1 encodes the Bayesian network (V, E, T) , Φ_2 encodes the
568 partial assignment v and Φ_3 encodes $X = a$. With satisfying assignments of
569 $\Phi_1 \wedge \Phi_2 \wedge \Phi_3$ and $\Phi_1 \wedge \Phi_2$, one can calculate (or approximate) the posterior
570 probability in the Bayesian network. Suppose n_1 (resp. n_2) denotes the number
571 of the discovered satisfying assignments of $\Phi_1 \wedge \Phi_2 \wedge \Phi_3$ (resp. $\Phi_1 \wedge \Phi_2$), the
572 approximate posterior probability is $\frac{n_1}{n_2}$. Furthermore, if n_1 is exact, then $\frac{n_1}{n_2}$
573 gives an upper-bound of the posterior probability; on the other hand, if n_2 is
574 exact, then $\frac{n_1}{n_2}$ gives a lower-bound.

575 **Performance of Bayesian inference.** To evaluate ESAMPLER for Bayesian
576 inference, we use the QUICKSAMPLER sampler as the seed generation engine
577 to solve Bayesian inference of the plan recognition problems provided by [31].
578 There are 11 plan recognition problems given as Bayesian networks on which we
579 compute the posterior probability for each random variable, resulting in 11,326
580 Bayesian inference queries. For each query, we sample satisfying assignments
581 until the 10 recently generated assignments already exist. Solved by ESAMPLER
582 in 65,122 seconds, the calculated probabilities of variables are shown in Table 5,
583 where the columns (Var ID) show the indices of the random variables, and the
584 columns (Prob) show the calculated posterior probabilities of the random vari-
585 able. For the sake of brevity, we show first hundred variables in problem tire-3,

Table 5: Calculated probabilities of variables in problem tire-3.

Var ID	Prob	Var	Prob	Var ID	Prob	Var ID	Prob
5	0.9391	43	0.0335	64	0.4827	83	0.0000
12	0.0000	44	0.2407	66	0.0000	85	0.5720
19	0.0000	45	0.0000	67	0.0000	86	0.0000
20	0.6307	50	0.6962	68	0.2452	87	0.0000
21	0.0000	51	0.0000	69	0.0000	88	0.9085
25	0.8177	52	0.5173	71	0.0000	89	0.0000
26	0.4827	53	0.9645	72	0.2470	90	0.2159
27	0.9997	54	0.2358	74	0.1287	91	0.5366
31	0.9997	55	0.7548	75	0.5021	92	0.1862
33	0.5172	56	0.1253	76	0.2502	94	0.2452
35	0.0000	58	0.9837	77	0.0000	95	0.9342
36	0.0000	59	0.9977	78	0.0499	96	0.9903
38	0.5028	61	0.2670	79	0.1691	98	0.2981
40	0.3693	62	0.0000	80	0.0253
41	0.4972	63	0.0000	82	0.0000		

586 and the results are omitted if the posterior probability is 1. We notice that
587 the reported probabilities in Table 5 are approximation of the exact posterior
588 probabilities when the sampler fails to generate all the possible satisfying assign-
589 ments of a Bayesian inference query. Remark that computing exact posterior
590 probabilities is computational hard ($\#P$ -completeness).

591 **Comparison of samplers on Bayesian inference.** To compare the efficiency
592 of ESAMPLER (i.e., ESAMPLER+QUICKSAMPLER), STS, QUICKSAMPLER and
593 UNIGEN3 in solving Bayesian inference problems, we test them on 11 randomly
594 chosen formulas from the plan recognition problems, each of which is aimed to
595 compute 100,000 satisfying assignments within 10 minutes. The other settings
596 are the same as in Section 5.

597 The results are reported in Table 6, where the last three columns provide the
598 ratio of the execution time per satisfying assignment for QUICKSAMPLER, STS,
599 UNIGEN3 to ESAMPLER respectively, measuring the speedup of ESAMPLER. All
600 the samplers are able to generate satisfying assignments except that UNIGEN3
601 failed on 4 benchmarks (log-1, log-4, log-5 and tire-1). For the sake of brevity, we
602 only report the number of satisfying assignments generated by ESAMPLER. We
603 can observe that ESAMPLER outperforms the other three samples on Bayesian
604 inference. On average, ESAMPLER is 13.8, 18.7 and 556.3 times faster than
605 QUICKSAMPLER, STS and UNIGEN3, respectively.

606 7. Conclusion

607 We have proposed a novel approach to derive a large set of satisfying as-
608 signments from a seed assignment without invoking computationally expensive
609 SAT solving. Our approach is orthogonal to the previous techniques and could
610 be integrated into the existing SAT samplers. We have also developed a new
611 tool ESAMPLER, based on the recent samplers QUICKSAMPLER and UNIGEN3

Table 6: Comparison of ESAMPLER, QUICKSAMPLER, STS and UNIGEN3 on benchmarks derived from Bayesian inferences of plan recognition problems

Benchmark	#Vars	#Cls	$E_t(s)$	E_n	E_{dn}	$E_{pn}(ms)$	$\frac{Q_{pt}}{E_{pt}}$	$\frac{S_{pt}}{E_{pt}}$	$\frac{U_{pt}}{E_{pt}}$
4step	165	418	17.8	66,935	0	0.27	1.01	1.78	36.47
5step	177	475	3.01	80,033	~80k	0.04	7.69	11.52	273.8
log-1	939	3,785	10.4	160,016	160k	0.07	16.31	72.99	-
log-2	1,377	24,777	53.9	110,011	110k	0.49	12.67	50.01	251.8
log-3	1,413	29,487	166	170,017	170k	0.98	4.78	24.14	511.3
log-4	2,303	20,963	18.2	120,012	120k	0.15	32.74	423.8	-
log-5	2,701	29,534	958	10,001	100k	95.81	4,355	2.99	-
tire-1	352	1,038	9.4	130,347	~130k	0.07	6.81	17.01	-
tire-2	550	2,001	15.8	160,016	160k	0.10	6.63	10.72	731.7
tire-3	578	2,004	13.7	140,014	140k	0.10	11.31	16.65	3,496
tire-4	812	3,222	20.1	120,012	120k	0.17	5.92	13.64	5,009

612 as the seed generator. The extensive experiments on publicly available bench-
613 marks and application on Bayesian inference confirmed the effectiveness and
614 efficiency of our approach.

615 In future, we plan to further improve the performance of the tool ESAMPLER
616 and extend our derivation approach to SMT formulas, as well as their practical
617 applications.

618 Acknowledgement

619 This work is supported by the National Natural Science Foundation of China
620 (NSFC) under Grants No. 62072309 and No. 61872340, an oversea grant from
621 the State Key Laboratory of Novel Software Technology, Nanjing University
622 (KFKT2018A16), and Birkbeck BEI School Project (EFFECT).

623 References

- 624 [1] A. Biere, M. Heule, H. van Maaren, T. Walsh (Eds.), Handbook of Satisfi-
625 ability, Vol. 185 of Frontiers in Artificial Intelligence and Applications, IOS
626 Press, 2009.
- 627 [2] S. Abed, A. A. Abdelaal, M. H. Alshayji, I. Ahmad, SAT-based and CP-
628 based declarative approaches for top-rank-k closed frequent itemset mining,
629 Int. J. Intell. Syst. 36 (1) (2021) 112–151.
- 630 [3] F. Bacchus, S. Dalmao, T. Pitassi, Algorithms and complexity results for
631 #SAT and bayesian inference, Proceedings of the 44th Symposium on
632 Foundations of Computer Science, 11-14 October 2003, Cambridge, MA,
633 USA, 2003, pp. 340–351.

- 634 [4] D. Roth, On the hardness of approximate reasoning, *Artificial Intelligence*
635 82 (1-2) (1996) 273–302.
- 636 [5] L. G. Valiant, The complexity of enumeration and reliability problems,
637 *SIAM Journal on Computing* 8 (3) (1979) 410–421.
- 638 [6] D. Angluin, On counting problems and the polynomial-time hierarchy, *The-*
639 *oretical Computer Science* 12 (1980) 161–173.
- 640 [7] Y. Naveh, M. Rimon, I. Jaeger, Y. Katz, M. Vinov, E. Marcus, G. Shurek,
641 Constraint-based random stimuli generation for hardware verification, *AI*
642 *magazine* 28 (3) (2007) 13–13.
- 643 [8] Y. Naveh, M. Rimon, I. Jaeger, Y. Katz, M. Vinov, E. Marcus, G. Shurek,
644 Constraint-based random stimuli generation for hardware verification, *Pro-*
645 *ceedings of the 21st National Conference on Artificial Intelligence and the*
646 *18th Innovative Applications of Artificial Intelligence Conference*, 2006, pp.
647 1720–1727.
- 648 [9] J. Yuan, A. Aziz, C. Pixley, K. Albin, Simplifying boolean constraint
649 solving for random simulation-vector generation, *IEEE Transactions on*
650 *Computer-Aided Design of Integrated Circuits and Systems* 23 (3) (2004)
651 412–420.
- 652 [10] E. Guralnik, M. Aharoni, A. J. Birnbaum, A. Koyfman, Simulation-based
653 verification of floating-point division, *IEEE Transactions on Computers*
654 60 (2) (2011) 176–188.
- 655 [11] K. Vorobyov, P. Krishnan, Combining static analysis and constraint solving
656 for automatic test case generation, *Proceedings of the 5th IEEE Interna-*
657 *tional Conference on Software Testing, Verification and Validation*, 2012,
658 pp. 915–920.
- 659 [12] R. Naveh, A. Metodi, Beyond feasibility: CP usage in constrained-
660 random functional hardware verification, *Proceedings of the 19th Interna-*
661 *tional Conference on Principles and Practice of Constraint Programming*,
662 Springer, 2013, pp. 823–831.
- 663 [13] Y. Zhao, J. Bian, S. Deng, Z. Kong, Random stimulus generation with self-
664 tuning, *Proceedings of the 13th International Conference on Computers*
665 *Supported Cooperative Work in Design*, IEEE, 2009, pp. 62–65.
- 666 [14] Y. Zhang, M. Zhang, G. Pu, F. Song, J. Li, Towards backbone computing:
667 A greedy-whitening based approach, *AI Communications* 31 (3) (2018)
668 267–280.
- 669 [15] Y. Zhang, J. Li, M. Zhang, G. Pu, F. Song, Optimizing backbone filter-
670 ing, in: *Proceedings of the 11th International Symposium on Theoretical*
671 *Aspects of Software Engineering*, 2017, pp. 1–8.

- 672 [16] Z. Zhao, G. Chen, J. Wang, Y. Yang, F. Song, J. Sun, Attack as defense:
673 characterizing adversarial examples using robustness, in: Proceedings of
674 the 30th ACM SIGSOFT International Symposium on Software Testing
675 and Analysis, 2021, pp. 42–55.
- 676 [17] G. Chen, Z. Zhao, F. Song, S. Chen, L. Fan, Y. Liu, SEC4SR: A security
677 analysis platform for speaker recognition, CoRR abs/2109.01766.
- 678 [18] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, Y. Bengio, Binarized
679 neural networks, Proceedings of the Annual Conference on Neural Infor-
680 mation Processing Systems, 2016, pp. 4107–4115.
- 681 [19] Y. Zhang, Z. Zhao, G. Chen, F. Song, T. Chen, BDD4BNN: A BDD-based
682 quantitative analysis framework for binarized neural networks, in: Proceed-
683 ings of the 33rd International Conference on Computer Aided Verification,
684 2021, pp. 175–200.
- 685 [20] S. Korneev, N. Narodytska, L. Pulina, A. Tacchella, N. Bjørner, M. Sa-
686 giv, Constrained image generation using binarized neural networks with
687 decision procedures, Proceedings of the 21st International Conference on
688 Theory and Applications of Satisfiability Testing, 2018, pp. 438–449.
- 689 [21] N. Narodytska, Formal analysis of deep binarized neural networks, Proceed-
690 ings of the 27th International Joint Conference on Artificial Intelligence,
691 2018, pp. 5692–5696.
- 692 [22] S. A. Cook, The complexity of theorem-proving procedures, Proceedings
693 of the 3rd Annual ACM Symposium on Theory of Computing, 1971, pp.
694 151–158.
- 695 [23] J. P. M. Silva, K. A. Sakallah, Grasp-a new search algorithm for satisfia-
696 bility, The Best of ICCAD, Springer, 2003, pp. 73–89.
- 697 [24] J. P. M. Silva, K. A. Sakallah, Grasp: A search algorithm for propositional
698 satisfiability, IEEE Transactions on Computers 48 (5) (1999) 506–521.
- 699 [25] R. J. B. Jr., R. C. Schrag, Using CSP look-back techniques to solve real-
700 world SAT instances, in: Proceedings of the Fourteenth National Confer-
701 ence on Artificial Intelligence and Ninth Innovative Applications of Artifi-
702 cial Intelligence Conference, 1997, pp. 203–208.
- 703 [26] M. Soos, K. Nohl, C. Castelluccia, Extending SAT solvers to cryptographic
704 problems, Proceedings of the 12th International Conference on Theory and
705 Applications of Satisfiability Testing, 2009, pp. 244–257.
- 706 [27] N. Kitchen, A. Kuehlmann, Stimulus generation for constrained random
707 simulation, Proceedings of the 2007 International Conference on Computer-
708 Aided Design, 2007, pp. 258–265.

- 709 [28] R. Dutra, K. Laeuffer, J. Bachrach, K. Sen, Efficient sampling of SAT so-
710 lutions for testing, Proceedings of the 40th International Conference on
711 Software Engineering, 2018, pp. 549–559.
- 712 [29] M. Soos, S. Gocht, K. S. Meel, Tinted, detached, and lazy CNF-XOR
713 solving and its applications to counting and sampling, Proceedings of the
714 32nd International Conference on Computer Aided Verification, 2020, pp.
715 463–484.
- 716 [30] S. Chakraborty, K. S. Meel, M. Y. Vardi, Balancing scalability and uni-
717 formity in SAT witness generator, Proceedings of the 51st Annual Design
718 Automation Conference, 2014, pp. 60:1–60:6.
- 719 [31] T. Sang, P. Beame, H. A. Kautz, Performing bayesian inference by weighted
720 model counting, in: Proceedings, The Twentieth National Conference on
721 Artificial Intelligence and the Seventeenth Innovative Applications of Arti-
722 ficial Intelligence Conference, 2005, pp. 475–482.
- 723 [32] S. Ermon, C. P. Gomes, B. Selman, Uniform solution sampling using a con-
724 straint solver as an oracle, Proceedings of the Twenty-Eighth Conference
725 on Uncertainty in Artificial Intelligence, 2012, pp. 255–264.
- 726 [33] K. S. Meel, Constrained counting and sampling: Bridging the gap between
727 theory and practice, CoRR abs/1806.02239.
- 728 [34] N. Kitchen, Markov chain monte carlo stimulus generation for constrained
729 random simulation, Ph.D. thesis, University of California, Berkeley, USA
730 (2010).
- 731 [35] W. Wei, B. Selman, A new approach to model counting, Proceedings of the
732 8th International Conference on Theory and Applications of Satisfiability
733 Testing, 2005, pp. 324–339.
- 734 [36] J. H. Kukula, T. R. Shiple, Building circuits from relations, Proceedings of
735 the 12th International Conference on Computer Aided Verification, 2000,
736 pp. 113–123.
- 737 [37] W. Wei, J. Erenrich, B. Selman, Towards efficient sampling: Exploiting
738 random walk strategies, Proceedings of the 19th National Conference on
739 Artificial Intelligence, 16th Conference on Innovative Applications of Arti-
740 ficial Intelligence, 2004, pp. 670–676.
- 741 [38] M. Sipser, A complexity theoretic approach to randomness, Proceedings
742 of the 15th Annual ACM Symposium on Theory of Computing, 1983, pp.
743 330–335.
- 744 [39] M. Bellare, O. Goldreich, E. Petrank, Uniform generation of np-witnesses
745 using an np-oracle, Inf. Comput. 163 (2) (2000) 510–526.

- 746 [40] C. P. Gomes, A. Sabharwal, B. Selman, Near-uniform sampling of combinatorial spaces using XOR constraints, Proceedings of the 2th Annual
747 Conference on Neural Information Processing Systems, 2006, pp. 481–488.
748
- 749 [41] S. Chakraborty, K. S. Meel, M. Y. Vardi, A scalable and nearly uniform
750 generator of SAT witnesses, Proceedings of the 25th International Conference on Computer Aided Verification, 2013, pp. 608–623.
751
- 752 [42] S. Ermon, C. P. Gomes, A. Sabharwal, B. Selman, Embed and project: Discrete sampling with universal hashing, Proceedings of the 27th Annual
753 Conference on Neural Information Processing Systems, 2013, pp. 2085–
754 2093.
755
- 756 [43] S. Chakraborty, D. J. Fremont, K. S. Meel, S. A. Seshia, M. Y. Vardi, On parallel scalable uniform SAT witness generation, Proceedings of the 21st
757 International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Held as Part of the European Joint Conferences on
758 Theory and Practice of Software, Springer, 2015, pp. 304–319.
759
- 761 [44] N. Bjørner, A. Phan, νz - maximal satisfaction with Z3, Proceedings of the 6th International Symposium on Symbolic Computation in Software
762 Science, 2014, pp. 1–9.
763
- 764 [45] N. Sörensson, N. Eén, MiniSat: A SAT solver with conflict-clause minimization, Solver Description.
765
- 766 [46] D. Achlioptas, Z. S. Hammoudeh, P. Theodoropoulos, Fast sampling of perfectly uniform satisfying assignments, Proceedings of the 21st International
767 Conference on Theory and Applications of Satisfiability Testing, Springer, 2018, pp. 135–147.
768
769
- 770 [47] Y. Xu, F. Song, T. Chen, Esampler: Efficient sampling of satisfying assignments for boolean formulas, in: International Symposium on Dependable
771 Software Engineering: Theories, Tools, and Applications, Springer, 2021, pp. 279–298.
772
773
- 774 [48] A. Ivrii, S. Malik, K. S. Meel, M. Y. Vardi, On computing minimal independent support and its applications to sampling and counting, Constraints
775 21 (1) (2016) 41–58.
776
- 777 [49] C. H. Papadimitriou, Computational complexity, Addison-Wesley, 1994.
- 778 [50] DIMACS, Clique and coloring problems graph format, [http://archive.
779 dimacs.rutgers.edu/pub/challenge/graph/doc/ccformat.tex](http://archive.dimacs.rutgers.edu/pub/challenge/graph/doc/ccformat.tex) Accessed September 16, 2021 (1993).
780
- 781 [51] L. M. de Moura, N. Bjørner, Z3: an efficient SMT solver, Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, 2008, pp. 337–340.
782
783

- 784 [52] F. Y. Bois, Bayesian inference, *Computational toxicology* (2013) 597–636.
- 785 [53] D. Heckerman, A tutorial on learning with bayesian networks, in: *Learning*
786 *in Graphical Models*, 1998, pp. 301–354.
- 787 [54] D. Roth, On the hardness of approximate reasoning, *Artificial Intelligence*
788 82 (1-2) (1996) 273–302.

789
790
791



792
793
794
795
796
797
798
799

Yongjie Xu is a M.S. student in ShanghaiTech University, supervised by Prof. Fu Song. He received the B.S. degree in Computer Science from ShanghaiTech University in 2019. His research interests are in SAT solving, program analysis and AI security.



800
801
802
803
804
805
806
807
808
809
810
811

Fu Song received the B.S. degree from Ningbo University, Ningbo, China, in 2006, the M.S. degree from East China Normal University, Shanghai, China, in 2009, and the Ph.D. degree in computer science from University Paris-Diderot, Paris, France, in 2013. From 2013 to 2016, he was a Lecturer and Associate Research Professor at East China Normal University. From August 2016 to July 2021, he is an Assistant Professor with ShanghaiTech University, Shanghai, China. Since July 2021, he is an Associate Professor with ShanghaiTech University. His research interests include formal methods and computer/AI security.

812
813
814
815



816
817
818
819
820
821
822
823
824

Taolue Chen received the B.S. and M.S. degrees from the Nanjing University, China, both in Computer Science. He was a junior researcher at the Centrum Wiskunde & Informatica (CWI) and acquired the Ph.D. degree from the Vrije Universiteit Amsterdam, The Netherlands. He is currently a lecturer at the Department of Computer Science and Information Systems, Birkbeck, University of London, United Kingdom. His research interests include formal verification and synthesis, program analysis, software security, software engineering and machine learning.

825
826