Bit-Vector Model Counting Using Statistical Estimation

Seonmo Kim(✉) and Stephen McCamant(✉)

University of Minnesota, Minneapolis, MN 55455, USA
{smkim,mccamant}@cs.umn.edu

Abstract. Approximate model counting for bit-vector SMT formulas (generalizing #SAT) has many applications such as probabilistic inference and quantitative information-flow security, but it is computationally difficult. Adding random parity constraints (XOR streamlining) and then checking satisfiability is an effective approximation technique, but it requires a prior hypothesis about the model count to produce useful results. We propose an approach inspired by statistical estimation to continually refine a probabilistic estimate of the model count for a formula, so that each XOR-streamlined query yields as much information as possible. We implement this approach, with an approximate probability model, as a wrapper around an off-the-shelf SMT solver or SAT solver. Experimental results show that the implementation is faster than the most similar previous approaches which used simpler refinement strategies. The technique also lets us model count formulas over floating-point constraints, which we demonstrate with an application to a vulnerability in differential privacy mechanisms.

Keywords: Model counting · Bit-vectors · Floating point · #SAT
Randomized algorithms

1 Introduction

Model counting is the task of determining the number of satisfying assignments of a given formula. Model counting for Boolean formulas, #SAT, is a standard model-counting problem, and it is a complete problem for the complexity class #P in the same way that SAT is complete for NP. #P is believed to be a much harder complexity class than NP, and exact #SAT solving is also practically much less scalable than SAT solving. #SAT solving can be implemented as a generalization of the DPLL algorithm [17]. and a number of systems such as Relsat [4], CDP [6], Cachet [38], sharpSAT [41], DSHARP [35] and countAntom [9] have demonstrated various optimization techniques. However, not surprisingly given the problem's theoretical hardness, such systems often perform poorly when formulas are large and/or have complex constraints.

Since many applications do not depend on the model count being exact, it is natural to consider approximation algorithms that can give an estimate of

D. Beyer and M. Huisman (Eds.): TACAS 2018, LNCS 10805, pp. 133–151, 2018.
https://doi.org/10.1007/978-3-319-89960-2_8

a model count with a probabilistic range and confidence. Some approximate model counters include ApproxCount [43], SampleCount [24], MiniCount [31], ApproxMC [12], ApproxMC-p [30] and ApproxMC2 [13]. In this paper we build on the approximation technique of XOR streamlining [25], which reduces the number of solutions of a formula by adding randomly-chosen XOR (parity) constraints. In expectation, adding one constraint reduces the model count by a factor of 2, and k independent constraints reduce the model count by 2^k. If a formula with extra constraints has $n > 0$ solutions, the original formula likely had about $n \cdot 2^k$. If the model count after constraints is small, it can be found with a few satisfiability queries, so XOR streamlining reduces approximate model counting to satisfiability. However to have an automated system, we need an approach to choose a value of k when the model count is not known in advance.

One application of approximate model counting is measuring the amount of information revealed by computer programs. For a deterministic computation, we say that the *influence* [36] is the base-two log of the number of distinct outputs that can be produced by varying the inputs, a measure of the information flow from inputs to outputs. Influence computation is related to model counting, but formulas arising from software are more naturally expressed as SMT (satisfiability modulo theories) formulas over bit-vectors than as plain CNF, and one wants to count values only of output variables instead of all variables. The theory of arithmetic and other common operations on bounded-size bit-vectors has the same theoretical expressiveness as SAT, since richer operations can be expanded ("bit-blasted") into circuits. But bit-vector SMT is much more convenient for expressing the computations performed by software, and SMT solvers incorporate additional optimizations. We build a system for this generalized version of the problem which takes as input an SMT formula with one bit-vector variable designated as the output, and a specification of the desired precision.

Our algorithm takes a statistical estimation approach. It maintains a probability distribution that reflects an estimate of possible influence values, using a particle filter consisting of weighted samples from the distribution. Intuitively the mean of the distribution corresponds to our tool's best estimate, while the standard deviation becomes smaller as its confidence grows. At each step, we refine this estimate by adding k XOR constraints to the input formula, and then enumerating solutions under those constraints, up to a maximum of c solutions (we call this enumeration process an *exhaust-up-to-c* query [36]). At a particular step, we choose k and c based on our previous estimate (prior), and then use the query result to update the estimate for the next step (posterior). The update from the query reweights the particle filter points according to a probability model of how many values are excluded by XOR constraints. We use a simple binomial-distribution model which would be exact if each XOR constraint were fully independent. Because this model is not exact, a technique based only on it does not provide probabilistic soundness, even though it performs well practically. So we also give a variant of our technique which does produce a sound bound, at the expense of requiring more queries to meet a given precision goal.

We implement our algorithm in a tool SearchMC that wraps either a bit-vector SMT solver compatible with the SMT-LIB 2 standard or a SAT solver, and report experimental results. SearchMC can be used to count solutions with respect to a subset of the variables in a formula, such as the outputs of a computation, the capability that Klebanov et al. call projected model counting [30], and Val et al. call subset model counting [42]. In our case the variables not counted need not be of bit-vector type. For instance this makes SearchMC to our knowledge the first tool that can be used to count models of constraints over floating-point numbers (counting the floating-point bit patterns individually, as contrasted with computing the measure of a subset of $\mathbb{R}^n$ [7,15]). We demonstrate the use of this capability with an application to a security problem that arises in differential privacy mechanisms because of the limited precision of floating-point values.

Compared to ApproxMC2 [13] and ApproxMC-p [30], concurrently-developed approximate #SAT tools also based on XOR streamlining, our technique gives results more quickly for the same requested confidence levels.

In summary, the key attributes of our approach are as follows:

- Our approximate counting approach gives a two-sided bound with user-specified confidence.
- Our tool inherits the expressiveness and optimizations of SMT solvers.
- Our tool gives a probabilistically sound estimate if requested, or can give a result more quickly if empirical precision is sufficient.

2 Background

XOR Streamlining. The main idea of XOR streamlining [25] is to add randomly chosen XOR constraints to a given input formula and feed the augmented formula to a satisfiability solver. One random XOR constraint will reduce the expected number of solutions in half. Consequently, if the formula is still satisfiable after the addition of s XOR constraints, the original formula likely has at least 2^s models. If not, the formula likely has at most 2^s models. Thus we can obtain a lower bound or an upper bound with this approach. There are some crucial parameters to determine the bounds and the probability of the bounds and they need to be carefully chosen in order to obtain good bounds. However, early systems [25] did not provide an algorithm to choose the parameters.

Influence. Newsome *et al.* [36] introduced the terminology of "influence" for a specific application of model counting in quantitative information-flow measurement. This idea can capture the control of input variables over an output variable and distinguish true attacks and false positives in a scenario of malicious input to a network service. The influence of input variables over an output variable is the $\log_2$ of the number of possible output values.

Exhaust-up-to-c Query. Newsome *et al.* also introduced the terminology of an "exhaust-up-to-c query", which repeats a satisfiability query up to some number c of solutions, or until there are no satisfying values left. This is a good approach to find a model count if the number of solution is small.

Particle Filter. A particle filter [19] is an approach to the statistical estimation of a hidden state from noisy observations, in which a probability distribution over the state is represented non-parametrically by a collection of weighted samples referred to as particles. The weights evolve over time according to observations; they tend to become unbalanced, which is corrected by a resampling process which selects new particles with balanced weights. A particle filtering algorithm with periodic resampling takes the following form:

1. Sample a number of particles from a prior distribution.
2. Evaluate the importance weights for each particle and normalize the weights.
3. Resample particles (with replacement) according to the weights.
4. The posterior distribution represented by the resampled particles becomes the prior distribution to next round and go to step 2.

3 Design

This section describes the approach and algorithms used by SearchMC. It is implemented as a wrapper around an off-the-self bit-vector satisfiability solver that supports the SMT-LIB2 format [3]. It takes as input an SMT-LIB2 formula in a quantifier-free theory that includes bit-vectors (QF_BV, or an extension like QF_AUFBV or QF_FPBV) in which one bit-vector is designated as the output, i.e. the bits over which solutions should be counted. (For ease of comparison with #SAT solvers, SearchMC also has a mode that takes a Boolean formula in CNF, with a list of CNF variables designated as the output.) SearchMC repeatedly queries the SMT solver with variations of the supplied input which add XOR constraints and/or "blocking" constraints that exclude previously-found solutions; based on the results of these queries, it estimates the total number of values of the output bit-vector for which the formula has a satisfying assignment.

SearchMC chooses fruitful queries by keeping a running estimate of possible values of the model count. We model the influence ($\log_2$ of model count) as if it were a continuous quantity, and represent the estimate as a probability distribution over possible influence values. In each iteration we use the current estimate to choose a query, and then update the estimate based on the query's results. (At a given update, the most recent previous distribution is called the *prior*, and the new updated one is called the *posterior*.) As the algorithm runs, the confidence in the estimate goes up, and the best estimate changes less from query to query as it converges on the correct result. Each counting query SearchMC makes is parameterized by k, the number of random XOR constraints to add, and c, the maximum number of solutions to count. The result of the query is a number of satisfying assignments between 0 and c inclusive, where a result that stops at c means the real total is at least c. Generally a low result leads to the next estimate being lower than the current one and a high result leads to the estimate increasing. We will describe the process of updating the probability distribution, and then give the details of the algorithms that use it.

Updating Distribution and Confidence Interval. We here explain the idea of how we compute a posterior distribution over influence, where both the prior

and posterior are represented by particles. Suppose we have a formula f with a known influence $\log_2 N$, and add k XOR random constraints to the formula. If we simulate checking the satisfiability of this augmented formula f_k for different XOR constraints, we can estimate a probability of sat/unsat on f_k. We expand this idea by applying exhaust-up-to-c approach to f_k. We count the number of satisfying assignments n up to c and generate the distributions for each number of satisfying assignments (where $n = c$ means that the number of satisfying assignments is in fact c or more). Thus under an assumption on the true influence of a formula, we can estimate the probabilities of each number of satisfying assignments based on k. By collecting these probabilities across a range of influence, we obtain a probability distribution over influence for an unknown formula assumed to have less than a maximum bits of influence. Under the idealized assumption that each XOR constraint is completely independent, adding k XOR constraints will leave each satisfying assignment alive with probability $1/2^k$. For any particular set of $n \geq 0$ satisfying assignments remaining out of an original N, the probability that exactly those n solutions will remain is the product of $1/2^k$ for each n and $1-(1/2^k)$ for each of the other $N-n$. Summing the total number of such sets with a binomial coefficient, we can approximately model the probability of exactly n solutions remaining as:

$$Pr_{=n}(N,k) = \binom{N}{n}(\frac{1}{2^k})^n(1-\frac{1}{2^k})^{N-n} \tag{1}$$

For the case when the algorithm stops looking when there might still be more solutions, we also want an expression for the probability that the number of solutions is n or more. We compute this straightforwardly as one minus the sum of the probabilities for smaller values:

$$Pr_{\geq n}(N,k) = 1 - \sum_{i=0}^{n-1} Pr_{=i}(N,k) \tag{2}$$

We use XOR constraints that contain each counted bit with probability one half, and are negated with probability one half. (This is the same family of constraints used in other recent systems [12,13,30]. Earlier work [25] suggested using constraints over exactly half of the bits, which have the same expected size, but less desirable independence properties.) Our binomial probability model is not precise in general, because these XOR constraints are 3-independent, but not r-independent for $r \geq 4$. When $N \geq 4$, some patterns among solutions (such as a set of four bitvectors whose XOR is all zeros) lead to correlations in the satisfiability of XOR constraints, and in turn to higher variance in the probability distribution without changing the expectation. This effect is relatively small, but we lack an analytic model of it, so we compensate by slightly increasing the confidence level our tool targets compared to what the user originally requested.

This probability model lets us simulate the probability of various query results as a function of the unknown formula influence. We use this model as a weighting function for each particle and resample particles based on each particle's weight value. Then, we estimate a posterior distribution from sampled

particles that have all equal weights. For instance, given a prior distribution over the influence sampled at 0.1 bit intervals, we can compute a sampled posterior distribution by counting and re-normalizing just the probability weights that correspond to a given query result value n. From the estimated posterior distribution, the mean μ and the standard deviation σ are computed. Hence, the μ is our best possible answer as our algorithm iterates and σ shows how much we are close to the true answer. Sequentially, the posterior distribution will be the next round's prior distribution and for use in the very first step of the algorithm we also implement a case of the prior distribution as uniform over influence.

Next we compute a confidence interval (lower bound and upper bound) symmetrically from the mean of the posterior distribution even though the distribution is not likely to be symmetrical. There are several ways to compute the confidence interval but the difference of the results is negligible as the posterior distribution gets narrower. Therefore, we used a simple way to compute the confidence interval: a half interval from the left side of the mean and another half from the right side.

Algorithm. We present our main algorithm `SearchMC` that runs automatically and always gives an answer with a given confidence interval. The pseudocode for algorithm `SearchMC` is given as Algorithm 1. Our algorithm takes as input a formula f, a desired confidence level CL ($0 < CL < 1$), a confidence level adjustment α ($0 \leq \alpha < 1$), a desired range size $thres$ and an initial prior distribution $InitDist$. f contains a set of bit-vector variables and bit-vector operators. We can obtain a confidence interval at a confidence level for a given mean and standard deviation. A confidence level CL is a fraction parameter specifying the probability with which the interval should contain the true answer, for example, 0.95 (95%) or 0.99 (99%). As we described above, the binomial probability model does not exactly capture the full behavior of XOR constraints, which could lead to our results being over-confident. We introduce a confidence level adjustment factor α to internally target a higher confidence level than the user requested, making it more likely that the requested confidence can be met. If $\alpha = 0$, we do not adjust the input confidence level. We have currently set the value for α empirically from $\frac{1}{2}$ to $\frac{1}{4}$. However this single factor may not ideally capture the control of the confidence level. Further investigation of the confidence gap will be future work. Our algorithm terminates when the length of our confidence interval is less than or equal to a given non-negative parameter $thres$. This parameter determines the amount of running time and there is a trade-off. If $thres$ value is small, it gives a narrow confidence interval, but the running time would be longer. If the value is large, it gives a wide confidence interval, but a shorter running time. Our tool can choose any initial prior distribution $InitDist$ represented by particles. For example, a generic strategy is to start with a uniform distribution over 0 to a number of output variables. If we have a better prior bound on the true influence (for instance 64 bits), a uniform distribution from 0 to that bound will generally perform better.

Variables. There are several variables: $prior$, $post$, $width$, k, c, $nSat$, UB, LB and δ. $prior$ represents a prior distribution by sampled particles with

Algorithm 1. `SearchMC`(f, $thres$, CL, α, $InitDist$)

```
1: CL ← CL + (1 − CL) × α                          ▷ Confidence level adjustment
2: width ← getWidth(f)                              ▷ The width of the output bit-vector of f
3: prior ← InitDist                                 ▷ Initial distribution
4: δ ← width
5: while δ > thres do
6:     c, k ← ComputeCandK(prior, width)
7:     nSat ← MBoundExhaustUpToC(f, width, k, c)
8:     post, UB, LB ← Update(prior, c, k, nSat, CL)
9:     δ ← UB − LB
10:    if k == 0 then
11:        output "Exact Count: ", nSat
12:    else
13:        prior ← post
14:        output "Lower: ", LB, "Upper: ", UB
15:    end if
16: end while
```

Algorithm 2. `ComputeCandK`($prior$, $width$)

```
1: μ, σ ←getMuSigma(prior)
2: c ← ⌈((2^σ + 1)/(2^σ − 1))^2⌉
3: k ← ⌊μ − (1/2) log2 c⌋
4: if k ≤ 0 then
5:     c ← 2^width + 1                              ▷ In this case, c is effectively infinite
6:     k ← 0                                        ▷ No constraints
7: end if
8: return c, k
```

corresponding weights. In one iteration, we obtain the updated posterior distribution *post* with resampled particles based on our probabilistic model as described above. The posterior becomes the prior distribution for the next iteration. While our algorithm is in the loop, it keeps updating *post*. *width* is the width of the output bit-vector of an input formula f, which is an initial upper bound for the influence since the influence cannot be more than the width of the output bit-vector. k is a number of random XOR constraints and c specifies the maximum number of solutions for the exhaust-up-to-c query. We obtain c and k using the `ComputeCandK` function shown as Algorithm 2 and discussed below. $nSat$ is a number of solutions from the exhaust-up-to-c query. `MBoundExhaustUpToC` runs until it finds the model count exactly or c solutions from formula f with k random XOR constraints. UB and LB are variables to store an upper bound and a lower bound of the current model count approximation with a given confidence level as we describe above. δ is the distance between the upper bound and lower bound. This parameter determines whether our algorithm terminates or not. If δ is less than or equal to our input value $thres$, our algorithm terminates. If not, it runs again with updated *post* until δ reaches the desired range size $thres$. An extreme case $k = 0$ denotes that our

guess is equivalent to the true model count. In this case, we print out the exact count and terminate the algorithm.

Functions. To motivate the definition of the function `ComputeCandK`, we view an exhaust-up-to-c query as analogous to measuring influence with a bounded-length "ruler." Suppose that we reduce the expected value of the model count by adding k XOR constraints to f. Then, we can use the "length-$(\log_2 c)$ ruler" to measure the influence starting at k and this measurement corresponds to the result of an exhaust-up-to-c query: the length-$(\log_2 c)$ ruler has c markings spaced logarithmically as illustrated in Fig. 1. Each iteration of the algorithm chooses a location (k) and length (c) for the ruler, and gets a noisy reading on the influence as one mark on the ruler. Over time, we want to converge on the true influence value, but we also wish to lengthen the rule so that the finer marks give more precise readings. Based on this idea, we have the `ComputeCandK` function to choose the length of and starting point of the ruler from a prior distribution. Then, we run an exhaust-up-to-c query and call `Update` to update the distribution based on the result of the query.

The pseudocode for algorithm `update` is described as Algorithm 3. A prior distribution *prior* and a posterior distribution *post* are represented as a set of sampled particles (influences). We sampled 500 particles for each `update` function call. Once we have the updated distribution, we can find out the interval of a given confidence level.

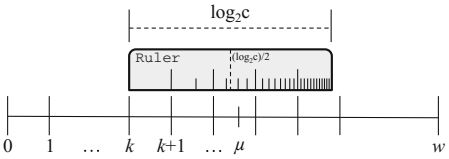


Fig. 1. Ruler intuition

Since we observe that our running σ represents how much we are close to the true answer, we use a rational function to satisfy the condition that c increases as σ decreases (i.e., we get more accurate result as c increases).

The k value denotes where to put the ruler. We want to place the ruler where the expected value of the prior distribution lies near the middle of the ruler hence our expected value is in the range of the ruler with high probability. Therefore, we subtract the half length of the ruler ($\frac{1}{2}\log_2 c$) from the expected value μ and then use the floor function to the value because k has to be an nonnegative integer value. The expected value always lies in the right-half side of the ruler by using the floor function. However, it is not essential which rounding function is used. Note that there might be a case where k becomes negative. If this happens, we set $k = 0$ and $c = \infty$, because our expected value is so small that we can run the solver exhaustively to give the exact model count. The formula for c is motivated by the intuition that the spacing between two marks near the middle of the ruler

Algorithm 3. `Update`($prior$, c, k, $nSat$, CL)

1: **for** t `from` 1 `to` $nParticles$ **do**
2: $x_t \leftarrow prior_t$ ▷ x is a list of sampled particles (influences)
3: **if** $nSat < c$ **then** ▷ Updating each weight of each particle
4: $w_t = Pr_{=nSat}(2^{x_t}, k)$
5: **else**
6: $w_t = Pr_{\geq nSat}(2^{x_t}, k)$
7: **end if**
8: **end for**
9: $w \leftarrow$ `normalize`() ▷ Normalizing the weights
10: $post \leftarrow$ `sample`$(x, w, nParticles)$ ▷ Resampling based on the weights
11: $UB, LB \leftarrow$ `getBounds`$(post, CL)$
12: **return** $post, UB, LB$

should be proportional to the standard deviation of the probability distribution, to ensure that a few different results of the query are possible with relatively high probability; the spacing between the two marks closest to $\frac{1}{2}\log_2 c = \log_2 \sqrt{c}$ will be about $\log_2(\sqrt{c}+\frac{1}{2}) - \log_2(\sqrt{c}-\frac{1}{2})$. Setting this equal to σ, solving for c, and taking the ceiling gives line 3 of Algorithm 2.

Probabilistic Sound Bounds. The binomial model performs well for choosing a series of queries, and it yields an estimate of the remaining uncertainty in the tool's results, but because the binomial model differs in an hard-to-quantify way from the true probability distributions, the bounds derived from it do not have any associated formal guarantee. In this section we explain how to use our tool's same query results, together with a sound bounding formula, to compute a probabilistically sound lower and upper bound on the true influence. As a trade-off, these bounds are usually not as tight as our tool's primary results.

The idea is based on Theorem 2 from Chakraborty *et al.*'s work [12], which in turn is a variant on Theorem 5 by Schmidt *et al.* [39]. For convenience we substitute our own terminology.

Lemma 1. *Let nSat be the return value from* `MBoundExhaustUpToC`*. Then,*

$$Pr\left[0 < nSat < c \text{ and } k \leq log_2|f| \text{ and } (1+\epsilon)^{-1}|f| \leq 2^k|f_h| \leq (1+\epsilon)|f|\right] > 0.6$$

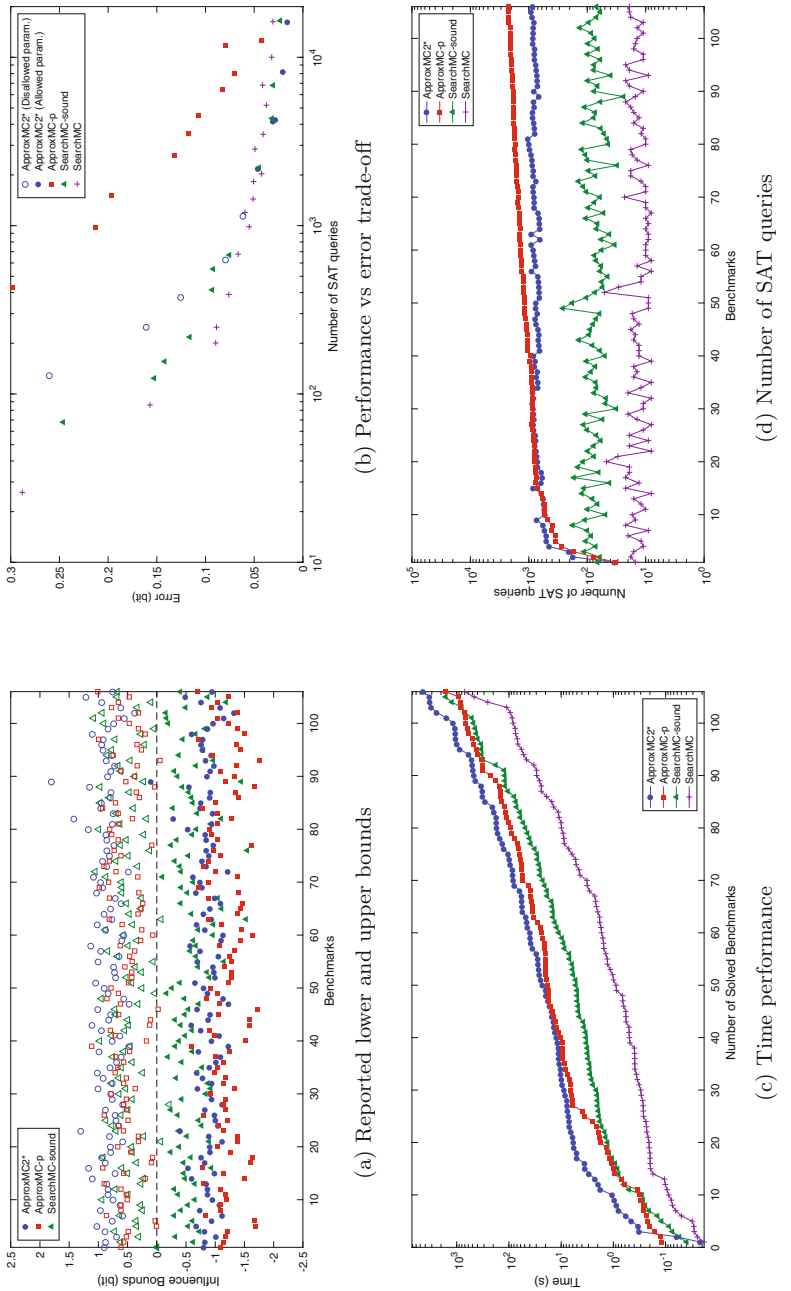
Chakraborty *et al.* use *pivot* for what we call c from an exhaust-up-to-c query and $pivot = 2\lceil 3e^{1/2}(1+\frac{1}{\epsilon})^2\rceil$. Since $0 < \epsilon < 1$, c ($pivot$) should be always greater than 40 to make the lemma true with a probability of at least 0.6. (The constant 0.6 comes from $(1-e^{-3/2})^2 \approx 0.6035$.)

In SearchMC's iterations, given c and k, we can compute ϵ value to estimate the bounds. Therefore, when c is greater than 40 from our tool's iteration, we can compute a lower and upper bound such that the true influence is within the bounds with a probability of at least 0.6.

4 Experimental Results

In this section, we present our experimental results. All our experiments were performed on a machine with an Intel Core i7 3.40 Ghz CPU and 16 GB memory. Our main algorithm is implemented with a Perl script and `Update` function is implemented in a C program called by the main script. Our algorithm can be applied to both SMT formulas and CNF formulas. We have tested a variety of SAT solvers and SMT solvers, and our current implementation specifically supports Cryptominisat2 [40] for CNF formulas and Z3 [18] and MathSAT5 [16] for SMT formulas. For pure bit-vector SMT formulas, our tool also supports eagerly converting the formula to CNF first and then using CNF mode. (We implement the conversion using the first phase of the STP solver [2,23] with optimizations disabled and a patch to output the SMT-to-CNF variable mapping.) Performing CNF translation eagerly gives up the benefit of some (e.g., word-level) optimizations performed by SMT solvers, but it can sometimes be profitable because it avoids repeating bit-blasting, and allows the tool to use a specialized multiple-solutions mode of Cryptominisat.

We run our algorithm with a set of DQMR (Deterministic Quick Medical Reference) benchmarks [1] and ISCAS89 benchmarks [8] converted to CNF files by TG-Pro [14] and compare the results of the benchmarks with ApproxMC2 [13] and ApproxMC-p [30]. ApproxMC2 and ApproxMC-p are state-of-the-art approximate #SAT solvers which we describe in more detail in Sect. 5. We used Cryptominisat2 as the back-end solver with all the tools for fair comparison. For the parameters for the tools, we set a 60% confidence level, a confidence level adjustment $\alpha = 0.25$ and a desired interval length of 1.7. As described above, SearchMC-sound gives correct bounds with a probability of at least 0.6. Since the desired confidence level for ApproxMC2 is $1 - \delta$, it can achieve a 60% confidence level by setting a parameter $\delta = 0.4$ which corresponds to our parameter $CL = 0.6$. Using the same confidence level for ApproxMC-p avoids an apparent mistake in the calculation of its base confidence pointed out by Biondi *et al.* [5]. The length of the interval for ApproxMC2 is computed as $\log_2(|f| \times (1+\epsilon)) - \log_2(|f| \times (1/(1+\epsilon))) = 1.7$ hence we can obtain the interval length 1.7 by setting a parameter $\epsilon = 0.8$, corresponding to our parameter $thres = 1.7$. Computing the interval for ApproxMC-p is a little different. The length of the interval for ApproxMC-p is $\log_2(|f| \times (1+\epsilon)) - \log_2(|f| \times (1-\epsilon)) = 1.7$ hence we can obtain the interval length 1.7 by setting a parameter $\epsilon = 0.53$. Note that SearchMC increases the c value of an exhaust-up-to-c query as it iterates while the corresponding ApproxMC2 and ApproxMC-p parameters are fixed as a function of ϵ (72 and 46, respectively) in this experiment. Also, we set an initial prior to be a uniform distribution over 0 to 64 bits for SearchMC. We tested 122 benchmarks (83 DQMRs and 39 ISCAS89s). All the tools were able to solve a set of 106 benchmarks (83 DQMRs and 23 ISCAS89s) within 2 h.



(a) Reported lower and upper bounds

(b) Performance vs error trade-off

(c) Time performance

(d) Number of SAT queries

Fig. 2. Comparison between SearchMC, ApproxMC2, and ApproxMC-p

Figure 2a compares the quality of lower bounds and upper bounds computed by SearchMC-sound, ApproxMC-p and ApproxMC2*[1]. Note that the benchmarks are arranged in increasing order of the true influence in Fig. 2a and d. The influence bounds are the computed bounds minus the true influence. Filled markers and empty markers represent reported lower bounds and upper bounds, respectively. SearchMC-sound, ApproxMC-p and ApproxMC2* out-perform the requested 60% confidence level. The incorrect bounds are visible as empty markers below the dotted line and filled markers above the line.

SearchMC-sound tends to give tighter bounds than the ApproxMC algorithms since it stops when the interval length becomes less than $thres$, while the interval lengths for the ApproxMCs are fixed by a parameter ϵ. We do not include the result of SearchMC in this figure to limit clutter, but the full results are available in the longer version of our paper [27]. In brief, SearchMC reported 65 correct bounds out of 106 benchmarks, which is slightly higher than the requested 60% confidence level.

Figure 2b shows another perspective on the trade-off between performance and error. We selected a single benchmark and varied the parameter settings of each algorithm, measuring the absolute difference between the returned answer and the known exact result. We include results from running ApproxMC2* with parameter settings outside the range of its soundness proofs (shown as "disallowed" in the plot), since these settings are still empirically useful, and SearchMC makes no such distinction. From this perspective the tools are complementary depending on one's desired performance-error trade-off. The results from all the tools improve with configurations that use more queries, but SearchMC performs best at getting more precise results from a small number of queries.

We also compare the running-time performance with ApproxMCs and show the running-time performance comparison on our 106 benchmarks in Fig. 2c. In this figure the benchmarks are sorted separately by running time for each tool, which makes each curve non-decreasing; but points with the same x position are not the same benchmark. Since ApproxMC-p refined the formulas of ApproxMC, it used a smaller number of queries than ApproxMC2. SearchMC can solve all the benchmarks faster than ApproxMCs with 60% confidence level. SearchMC-sound performs faster than ApproxMC-p even SearchMC-sound computes its confidence interval similarly to ApproxMC-p. The SearchMC's and SearchMC-sound's average running times are 24.59 and 108.24 s, compared to an average of 125.48 for ApproxMC-p. ApproxMC2* requires an average of 298.11 s just for the subset of benchmarks all the tools can complete. We also compare the number of SAT queries on the benchmarks for all the tools in Fig. 2d. For this figure the benchmarks are sorted consistently by increasing true model count for all tools.

[1] ApproxMC2* refers to our own re-implementation of the ApproxMC2 algorithm. With the latest version of ApproxMC2 we encountered problems (which we are still investigating) in which the SAT solver would sometimes fail to perform Gaussian elimination, which unfairly hurt the tool's performance. Our implementation also makes it easy to control the random seed for experiment repeatability.

Table 1. Results and performance of model counting ($\log_2$ shown) of naive Laplacian noise in IEEE floating point

Problem size	All noise			Intersection	
	Expected	SearchMC	Time	SearchMC	Time
15e7, 2^8	7.994	[7.374, 8.069]	164 s	1.000	312 s
16e7, 2^9	8.997	[8.566, 9.073]	470 s	3.322	1585 s
16e8, 2^{10}	9.999	[10.076, 10.844]	279 s	4.754	5279 s
18e8, 2^{10}	9.999	[9.675, 10.099]	583 s	1.000	1137 s
19e8, 2^{11}	10.999	[10.825, 11.404]	757 s	3.585	9848 s

The average number of SAT queries for SearchMC, SearchMC-sound ApproxMC-p and ApproxMC2* is about 14.7, 83.73, 1256.96 and 733.81 queries, respectively.

Floating Point/Differential Privacy Case Study. As an example of model counting with floating point constraints, we measure the security of a mechanism for differential privacy which can be undermined by unexpected floating-point behavior. The Laplace mechanism achieves differential privacy [22] by adding exponentially-distributed noise to a statistic to obscure its exact value. For instance, suppose we wish to release a statistic counting the number of patients in a population with a rare disease, without releasing information that confirms any single patient's status. In the worst case, an adversary might know the disease status of all patients other than the victim; for instance the attacker might know that the true count is either 10 or 11. If we add random noise from a Laplace distribution to the statistic before releasing it, we can leave the adversary relatively unsure about whether the true count was 10 or 11, while preserving the utility of an approximate result. A naive implementation of such a simple differentially private mechanism using standard floating-point techniques can be insecure because of a problem pointed out by Mironov [34]. For instance if we generate noise by dividing a random number in $[1, 2^{31}]$ by 2^{31} and taking the logarithm, the relative probability of particular floating point results will be quantized compared to the ideal probability, and many values will not be possible at all. If a particular floating point number could have been generated as $10 + \mathit{noise}$ but not as $11 + \mathit{noise}$ in our scenario, its release completely compromises the victim's privacy.

To measure this danger using model counting, we translated the standard approach for generating Laplacian noise, including an implementation of the natural logarithm, into SMT-LIB 2 floating point and bit-vector constraints. (We followed the `log` function originally by SunSoft taken from the `musl` C library, which uses integer operations to reduce the argument to $[\sqrt{2}/2, \sqrt{2})$, followed by a polynomial approximation.) A typical implementation might use double-precision floats with an 11-bit exponent and 53-bit fraction, and 32 bits of randomness, which we abbreviate "$53e11, 2^{32}$", but we tried a range of increasing sizes. We measured the total number of distinct values taken by $10 + \mathit{noise}$ as well as the size of the intersection of this set with the $11 + \mathit{noise}$ set.

The results and running time are shown in Table 1. (For space reasons we omit results for some smaller formats, which can be found in the extended version of the paper [27].) We ran SearchMC with a confidence level of 80%, a confidence level adjustment of 0.5 and a threshold of 1.0; the SMT solver was MathSAT 5.3.13 with settings recommended for floating-point constraints by its authors. We use one random bit to choose the sign of the noise, and the rest to choose its magnitude. The sign is irrelevant when the magnitude is 0, so the expected influence for n bits of randomness is $\log_2(2^n - 1)$. SearchMC's 80% confidence interval included the correct result in 4 out of 5 cases. The size of the intersections is small enough that SearchMC usually reports an exact result (always here). The size of the intersection is also always much less than the total set of noise values, confirming that using this algorithm and parameter setting for privacy protection would be ill-advised. The running time increases steeply as the problem size increases, which matches the conventional wisdom that reasoning about floating-point is challenging. But because floating-point SMT solving is a young area, there is future solvers may significantly improve the technique's performance.

Description of Archival Artifact. To facilitate reproduction of our experiments and future research, we have created an artifact archive containing code and data for performing the experiments described in this paper. This archive is a `zip` file containing data, instructions, source code, and binaries pre-compiled for Ubuntu 14.04 x86-64, which we have tested for compatibility with the virtual machine used during the artifact evaluation process [26]. The archive includes SearchMC itself and the modified version of STP it uses for bit-blasting, as well as scripts specific to the differential-privacy experiment, and the benchmark input files we used for performance evaluation. Information about accessing this artifact is found at the end of the paper.

5 Related Work

Exact Model Counting. Some of the earliest Boolean model counters used the DPLL algorithm [17] for counting the exact number of solutions. Birnbaum *et al.* [6] formalized this idea and introduced an algorithm for counting models of propositional formulas. Based on this idea, Relsat [4], Cachet [38] sharpSAT [43] and DSHARP [35] showed improvements by using several optimizations. The major contribution of countAntom [9] is techniques for parallelization, but it provides state-of-the-art performance even in single-threaded mode.

Phan *et al.* [37] encode a full binary search for feasible outputs in a bounded model checker. This approach is precise, but requires more than one call to the underlying solver for each feasible output. Klebanov *et al.* [29] perform exact model counting for quantitative information-flow measurement, with an approach that converts C code to a CNF formula with bounded model checking and then uses exact #SAT solving. Val *et al.* [42] integrate a symbolic execution tool more closely with a SAT solver by using techniques from SAT solving to prune the symbolic execution search space, and then perform exact model counting restricted to an output variable.

Randomized Approximate Model Counting. Randomized approximate model counting techniques perform well on many kinds of a formula for which finding single solutions is efficient. Wei and Selman [43] introduced ApproxCount which uses near-uniform sampling to estimate the true model count but it can significantly over-estimate or underestimate if the sampling is biased. SampleCount [24] improves this sampling idea and gives a lower bound with high probability by using a heuristic sampler. MiniCount [31] computes an upper bound under statistical assumptions by counting branching decisions during SAT solving. MBound [25] is an approximate model counting tool that gives probabilistic bounds on the model counts by adding randomly-chosen parity constraints as XOR streamlining. Chakraborty *et al.* [12] introduced ApproxMC, an approximate model counter for CNF formulas, which automated the choice of XOR streamlining parameters. The ApproxMC algorithm, in our terminology, starts by fixing c and a total number of iterations based on the desired precision and confidence of the results. In each iteration ApproxMC searches for an appropriate k value, adds k XOR random constraints, and then performs an exhaust-up-to-c query on the streamlined formula and multiplies the result by 2^k. It stores all the individual estimates as a multiset and computes its final estimate as the median of the values. The original ApproxMC sequentially increases k in each iteration until it finds an appropriate k value. An improved algorithm ApproxMC2 [13] uses galloping binary search and saves a starting k value between iterations to make the selection of k more efficient. Other recent systems that build on ApproxMC include SMTApproxMC [11] and ApproxMC-p [30]. ApproxMC-p implements projection (counting over only a subset of variables), which we also require.

ApproxMC2, whose initial development was concurrent with our first work on SearchMC, is the system most similar to SearchMC: its binary search for k plays a similar role to our converging μ value. However SearchMC also updates the c parameter over the course of the search, leading to fewer total queries. ApproxMC, ApproxMC2, and related systems choose the parameters of the search at the outset, and make each iteration either fully independent (ApproxMC) or dependent in a very simple way (ApproxMC2) on previous ones. These choices make it easier to prove the tool's probabilistic results are sound, but they require a conservative choice of parameters. SearchMC's approach of maintaining a probabilistic estimate at runtime means that its iterations are not at all independent: instead our approach is to extract the maximum guidance for future iterations from previous ones, to allow the search to converge more aggressively.

The runtime performance of SearchMC, like that of ApproxMC(2), is highly dependent on the behavior of SAT solvers on CNF-XOR formulas. Some roots of the difficulty of this problem have been investigated by Dudek et al. [20,21].

Non-randomized Approximate Model Counting. Non-randomized approximate model counting using techniques similar to static program analysis is generally faster than randomized approximate model counting techniques, and such systems can give good approximations for some problem classes. However, they cannot provide a precision guarantee for arbitrary problems, and it is not possible to give more effort to have more refined results.

Castro *et al.* [10] compute an upper bound on the number of bits about an input that are revealed by an error report. Meng and Smith [33] use two-bit-pattern SMT entailment queries to calculate a propositional overapproximation and count its instances with a model counter from the computer algebra system Mathematica. Luu *et al.* [32] propose a model counting technique over an expressive string constraint language.

Applications: Security and Privacy. Various applications of model counting have been proposed for security and privacy purposes. Castro *et al.* [10] use model counting and symbolic execution approaches to measure leaking private information from bug reports. They compute an upper bound on the amount of private information leaked by a bug report and allow users to decide on whether to submit the report or not. Newsome *et al.* [36] show how an untrusted input affects a program and introduce a family of techniques for measuring influence which can be applicable to x86 binaries. Biondi *et al.* [5] use CBMC and ApproxMC2 to quantify information flow on a set of benchmarks and evaluate the leakage incurred by a small instance of the Heartbleed OpenSSL bug.

6 Future Work and Conclusion

Closing the gap between the performance of SearchMC and SearchMC-sound is one natural direction for future research. On one hand, we would like to explore techniques for asserting sound probabilistic bounds which can take advantage of the results of all of SearchMC's queries. At the same time, we would like to find a model of the number of solutions remaining after XOR streamlining that is more accurate than our current binomial model, which should improve the performance of SearchMC. Another future direction made possible by the particle filter implementation is to explore different prior distributions, including unbounded ones. For instance, using a negative exponential distribution over influence as a prior would avoid the any need to estimate a maximum influence in advance, while still starting the search process with low-k queries which are typically faster to solve.

In sum, we have presented a new model counting approach SearchMC using XOR streamlining for SMT formulas with bit-vectors and other theories. We demonstrate our algorithm that adaptively maintains a probabilistic model count estimate based on the results of queries. Our tool computes a lower bound and an upper bound with a requested confidence level, and yields results more quickly than previous systems.

Data Availability Statement and Acknowledgements. An archival snapshot of the tools and datasets analyzed in this work is available in the conference `figshare` repository at https://doi.org/10.6084/m9.figshare.5928604.v1 [28]. Updates will also be available via the project's GitHub page at https://github.com/seonmokim/SearchMC. We would like to thank the anonymous conference and artifact reviewers for suggestions which have helped us to improve our system and the paper's presentation. This research is supported by the National Science Foundation under grant no. 1526319.

References

1. Bayesian-inference as model-counting benchmarks. http://www.cs.rochester.edu/users/faculty/kautz/Cachet/Model_Counting_Benchmarks/index.htm
2. STP. http://stp.github.io/
3. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB standard: Version 2.0. Technical report (2010)
4. Bayardo Jr., R.J., Schrag, R.C.: Using CSP look-back techniques to solve real-world SAT instances. In: Proceedings of AAAI, pp. 203–208 (1997)
5. Biondi, F., Enescu, M.A., Heuser, A., Legay, A., Meel, K.S., Quilbeuf, J.: Scalable approximation of quantitative information flow in programs. Verification, Model Checking, and Abstract Interpretation. LNCS, vol. 10747, pp. 71–93. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-73721-8_4
6. Birnbaum, E., Lozinskii, E.L.: The good old Davis-Putnam procedure helps counting models. J. Artif. Intell. Res. (JAIR) **10**, 457–477 (1999)
7. Borges, M., Filieri, A., d'Amorim, M., Pasareanu, C.S., Visser, W.: Compositional solution space quantification for probabilistic software analysis. In: Proceedings of PLDI, pp. 123–132 (2014)
8. Brglez, F., Bryan, D., Kozminski, K.: Combinational profiles of sequential benchmark circuits. In: Proceedings of ISCAS, vol. 3, pp. 1929–1934 (1989)
9. Burchard, J., Schubert, T., Becker, B.: Laissez-Faire caching for parallel #SAT solving. In: Heule, M., Weaver, S. (eds.) SAT 2015. LNCS, vol. 9340, pp. 46–61. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24318-4_5
10. Castro, M., Costa, M., Martin, J.-P.: Better bug reporting with better privacy. In: Proceedings of ASPLOS, pp. 319–328 (2008)
11. Chakraborty, S., Meel, K.S., Mistry, R., Vardi, M.Y.: Approximate probabilistic inference via word-level counting. In: Proceedings of AAAI, pp. 3218–3224 (2016)
12. Chakraborty, S., Meel, K.S., Vardi, M.Y.: A scalable approximate model counter. In: Schulte, C. (ed.) CP 2013. LNCS, vol. 8124, pp. 200–216. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40627-0_18
13. Chakraborty, S., Meel, K.S., Vardi, M.Y.: Improving approximate counting for probabilistic inference: from linear to logarithmic SAT solver calls. In: Proceedings of IJCAI, pp. 3569–3576 (2016)
14. Chen, H., Marques-Silva, J.: TG-Pro: a SAT-based ATPG system. J. Satisf. Boolean Model. Comput. **8**(1–2), 83–88 (2012)
15. Chistikov, D., Dimitrova, R., Majumdar, R.: Approximate counting in SMT and value estimation for probabilistic programs. In: Proceedings of TACAS, pp. 320–334 (2015)
16. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 93–107. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_7
17. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM **5**(7), 394–397 (1962)
18. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
19. Del Moral, P.: Nonlinear filtering: interacting particle solution. Markov Process. Relat. Fields **2**(4), 555–580 (1996)
20. Dudek, J., Meel, K.S., Vardi, M.Y.: Combining the k-CNF and XOR phase-transitions. In: Proceedings of IJCAI, pp. 727–734 (2016)

21. Dudek, J., Meel, K.S., Vardi, M.Y.: The hard problems are almost everywhere for random CNF-XOR formulas. In: Proceedings of IJCAI, pp. 600–606 (2017)
22. Dwork, C.: Differential privacy. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4052, pp. 1–12. Springer, Heidelberg (2006). https://doi.org/10.1007/11787006_1
23. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_52
24. Gomes, C.P., Hoffmann, J., Sabharwal, A., Selman, B.: From sampling to model counting. In: Proceedings of IJCAI, pp. 2293–2299 (2007)
25. Gomes, C.P., Sabharwal, A., Selman, B.: Model counting: a new strategy for obtaining good bounds. In: Proceedings of AAAI, pp. 54–61 (2006)
26. Hartmanns, A., Wendler, P.: figshare (2018). https://doi.org/10.6084/m9.figshare.5896615
27. Kim, S., McCamant, S.: Bit-vector model counting using statistical estimation. CoRR, abs/1712.07770 (2017)
28. Kim, S., McCamant, S.: SearchMC: an approximate model counter using XOR streamlining techniques. figshare (2018). https://doi.org/10.6084/m9.figshare.5928604.v1
29. Klebanov, V., Manthey, N., Muise, C.: SAT-based analysis and quantification of information flow in programs. In: Joshi, K., Siegle, M., Stoelinga, M., D'Argenio, P.R. (eds.) QEST 2013. LNCS, vol. 8054, pp. 177–192. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40196-1_16
30. Klebanov, V., Weigl, A., Weisbarth, J.: Sound probabilistic #SAT with projection. In: Workshop on QAPL (2016)
31. Kroc, L., Sabharwal, A., Selman, B.: Leveraging belief propagation, backtrack search, and statistics for model counting. In: Perron, L., Trick, M.A. (eds.) CPAIOR 2008. LNCS, vol. 5015, pp. 127–141. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68155-7_12
32. Luu, L., Shinde, S., Saxena, P., Demsky, B.: A model counter for constraints over unbounded strings. In: Proceedings of PLDI, pp. 565–576 (2014)
33. Meng, Z., Smith, G.: Calculating bounds on information leakage using two-bit patterns. In: Proceedings of PLAS, pp. 1:1–1:12 (2011)
34. Mironov, I.: On significance of the least significant bits for differential privacy. In: Proceedings of CCS, pp. 650–661 (2012)
35. Muise, C., McIlraith, S.A., Beck, J.C., Hsu, E.I.: Dsharp: fast d-DNNF compilation with sharpSAT. In: Kosseim, L., Inkpen, D. (eds.) AI 2012. LNCS (LNAI), vol. 7310, pp. 356–361. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30353-1_36
36. Newsome, J., McCamant, S., Song, D.: Measuring channel capacity to distinguish undue influence. In: Proceedings of PLAS, pp. 73–85 (2009)
37. Phan, Q., Malacaria, P., Tkachuk, O., Păsăreanu, C.S.: Symbolic quantitative information flow. SIGSOFT Softw. Eng. Notes **37**(6), 1–5 (2012)
38. Sang, T., Bacchus, F., Beame, P., Kautz, H.A., Pitassi, T.: Combining component caching and clause learning for effective model counting. In: Proceedings of SAT (2004)
39. Schmidt, J.P., Siegel, A., Srinivasan, A.: Chernoff-hoeffding bounds for applications with limited independence. SIAM J. Discret. Math. **8**(2), 223–250 (1995)
40. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 244–257. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02777-2_24

41. Thurley, M.: sharpSAT – counting models with advanced component caching and implicit BCP. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 424–429. Springer, Heidelberg (2006). https://doi.org/10.1007/11814948_38
42. Val, C.G., Enescu, M.A., Bayless, S., Aiello, W., Hu, A.J.: Precisely measuring quantitative information flow: 10K lines of code and beyond. In: Proceeding of Euro S&P, pp. 31–46 (2016)
43. Wei, W., Selman, B.: A new approach to model counting. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 324–339. Springer, Heidelberg (2005). https://doi.org/10.1007/11499107_24