Scalable Model Counting for Program Analysis

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Seonmo Kim

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Stephen McCamant, Advisor

March, 2022

Acknowledgements

First of all, I would like to express my deepest appreciation to my advisor, Prof. Stephen McCamant, who was always approachable and gave me valuable feedback during my graduate program. I learned so many great things from him and he broadened my perspective as a researcher. I also thank my committee members, Prof. Jie Ding, Prof. Nick Hopper and Dr. Mike Whalen, for their time and feedback to improve my dissertation. I am also thankful to Prof. Wonhong Nam for giving me the opportunity to do my first research project with him during my undergraduate degree program. On a personal note, I thank all of my friends and colleagues, Navid, Vaibhav, Qiuchen, Bowen and Soha, for sharing their ideas and graduate school life together. I am thankful to my wife, Jinhee Bok, for always being by my side and taking good care of her immature husband. I am also thankful to my son, Leon, for loving his busy dad and giving me great joy. I thank my mother-in-law, Chaulsoon Kim, for her prayer and encouragement. Last but not least, I thank my parents, Larkkyo Kim and Kyunghee Kim, and my brother, Hyungmo Kim, for their support and prayer.

Dedication

To my Lord, Jesus Christ.

Abstract

Various model counting techniques have been proposed to show their scalability for problem domains such as combinatorics, safety analysis, probabilistic inference and quantitative information-flow analysis of software. In particular, random hashing-based methods have shown to be highly successful in computing bounds on the model count of a propositional formula. These hashing-based algorithms repeatedly check the satisfiability of a formula subject to random parity constraints (XOR streamlining) and give an estimate of the model count with a probabilistic range and confidence. However, these approaches often perform poorly when the size of parity constraints becomes too large and/or the complexity of the formula increases because they are highly dependent on the performance of a decision procedure.

In this thesis, we focus on increasing scalability of hashing-based model counting techniques and applying the model counting techniques to analyze programs. This thesis is divided into three parts. We first describe two efficient approximate model counting tools: SearchMC and SMC. SearchMC is a hashing-based approximate model counter which uses fruitful SAT (or SMT) queries to compute a lower bound and an upper bound based on statistical estimation, and yields results more quickly than existing systems. SMC is a structural model counter which analyzes the structure of a formula to compute firm lower and upper bounds in polynomial time. Moreover, we can use SMC to compute a refined initial hypothesis without any solver calls for SearchMC. Secondly, we explain a divide-and-conquer algorithm, MultiSearchMC, to increase scalability of hashing-based model counting techniques using parallelization. We first split an input formula into small formulae and run the combination of SMC and SearchMC in parallel. Then, we conservatively combine all the results to compute an estimate of the model count. Lastly, we analyze realistic programs using this improved model counting technique. We show how we can apply model counting techniques to quantitative information flow analysis and uniform sampling for testing. Also, the experimental results illustrate that our approach is able to handle large-sized input/output data for quantitative information flow analysis and uniform sampling in certain domains.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

Model counting is the task of determining the number of satisfying assignments of a given formula. Boolean formula model counting, known as #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 [1] and a number of systems such as Relsat [2], CDP [3], Cachet [4], sharpSAT [5], DSHARP [6] and countAntom [7] 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 a model count with a probabilistic range and confidence. Some approximate model counters include ApproxCount [8], SampleCount [9], MiniCount [10] and so on. One effective category of approximate model counting techniques is hashing-based model counting [11, 12, 13, 14], which employs XOR constraints to reduce the solution space [15]. The main contribution of hashing-based model counting approaches is a good trade-off between computational performance and probabilistic guarantees, which means these approaches reduce the #P-complete model counting problem to a polynomial number of NP-complete problems. The basic concept of hashing-based model

counting techniques is that adding one XOR constraint (generated by universal hashing) reduces the model count by a factor of 2 in expectation, therefore k independent constraints are likely to 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 adding XOR constraints reduces approximate model counting to satisfiability. Existing hashing-based model counting techniques differ how to choose a value of k when the model count is not known in advance.

We first introduce a hashing-based model counting tool, SearchMC [12], which takes a statistical estimation approach to compute the most appropriate k for each step. It maintains a probability distribution that reflects an estimate of possible model counts: 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 use a particle filter to refine this estimate by adding k XOR constraints to the input formula, and then enumerating solutions under those constraints. Hashing-based model counting techniques can be more scalable than exact model counting techniques by reducing the number of solver calls. However these approaches, including SearchMC, still have limited scalability in practice: they require a large number of satisfiability queries to achieve tight bounds, and hashing can make individual queries much more expensive, given the complex interactions between hashing constraints and solver optimizations.

Structural model counting techniques are valuable but less developed approximation algorithms which achieve guaranteed efficiency and firm bounds without a solver call. These techniques analyze the syntactic structure of a formula to compute conservative lower and upper bounds that are always correct. We introduce a new structural model counting tool called SMC [16] for quantifier-free SMT formulas over the theory of bit vectors (SMT-LIB QF_BV), one of the most common theories used to model bounded arithmetic and software semantics. The results of this structural approximate model counting technique can be used as a prior hypothesis to produce more useful results in SearchMC. Even if we combine structural model counting and hashing-based model counting techniques, we still face the scalability issue since this reduces only a small number of satisfiability queries. The performance of hashing-based model counters highly depends on the performance of decision procedures and there are limiting factors to improve the performance of solvers practically and theoretically if an input formula is too large or complex.

In order to increase scalability of hashing-based model counting techniques, we introduce MultiSearchMC to improve performance by extending the idea of computing a prior hypothesis. The key idea is that we split an input formula into multiple sub-formulae and compute a model count of each sub-formula with a hashing-based model counter in parallel. This is similar to a "divide-and-conquer" style approach where we solve multiple sub-problems first to solve the original problem. We first split an input formula using projected model counting [17] which computes a number of satisfying assignments over a subset of original variables. The length of each XOR constraint is proportional to the total number of original variables and we observe that reasoning with long XOR constraints is computationally more expensive than short XOR constraints. The length of XOR constraints gets decreased as the number of projected variables gets decreased, and this helps SAT solvers to solve a formula faster. We then run SMC and SearchMC for each sub-formula in parallel and combine the results. Note that we need to combine the results conservatively since each result is likely dependent on other results. All the results are combined to generate a more useful hypothesis than SMC itself and we can use this result as an initial hypothesis to run SearchMC with the original formula. Moreover, there are some special cases where applying this idea significantly improves performance. When the tool can tell that each result is independent, we can apply the multiplication rule to combine the lower bounds and this makes a more precise result. Consequently, if the combined result is precise enough, we do not need to run SearchMC to compute the final answer and this leads to improved runtime performance.

In this thesis, we focus on using this scalable model counting technique to analyze programs. One application of approximate model counting techniques is to measure the amount of information revealed by computer programs. Quantitative information flow (QIF) analysis is a powerful approach to measure the amount of sensitive information leakage. Early information flow research focused on enforcing observable outputs to be totally independent of the sensitive inputs. This is highly desirable if it can be applied to any systems that need to guarantee it. But it is more realistic and typical to weaken this property because the outputs can be dependent on the sensitive inputs. By measuring the information leakage using model counting techniques, it is more applicable to provide how much information can be leaked by a program or a function. Specifically, we use symbolic execution tools to analyze realistic binary executables and apply quantitative information-flow analysis.

Another application of approximate model counting techniques is uniform sampling for

testing [18, 19, 20, 21]. Uniform sample generation for SAT/SMT formulas is widely used in various areas such as probabilistic reasoning in AI systems, functional verification and so on. Generating independent uniformly distributed samples over a set of satisfying assignments is a challenging problem both theoretically and practically. Our work is inspired by Unigen [22], which uses hashing-based techniques to compute an estimate model count and generate near-uniform samples. Basically, we apply our approximate model counting approach to increase its scalability and performance and follow the same sampling process as Unigen. Our experiments show that our version achieves a greater speed-up than Unigen3 [23], which is the most recent version of Unigen.

1.2 Background

1.2.1 Boolean Formula

A Boolean formula is a finite expression constructed from a combination of Boolean constants, variables, operators and parentheses and produces a Boolean value (true or false) when evaluated. Let F be a Boolean formula where the formula is expressed in Boolean variables x_i and Boolean operations such as $\wedge$ (AND), $\vee$ (OR), $\oplus$ (XOR) or $\neg$ (NOT). For example, we have a Boolean formula F_1 as:

$$F_1 = x_1 \wedge x_2 \vee \neg x_3 \tag{1.1}$$

where it has three Boolean variables: $\{x_1, x_2, x_3\}$. In a Boolean formula, a *literal* is a Boolean variable or its negation such as x_1, x_2 or $\neg x_3$. A *clause* is an expression generated from a finite collection of literals that are combined by disjunction. We can evaluate F by assigning either true or false to each Boolean variable x_i. A *satisfying assignment* or *solution* of F is an assignment of all the variables in F such that F evaluates to true under the variable assignment. For example, a satisfying assignment of F_1 is $(x_1, x_2, x_3) = (1, 0, 0)$ where 1 and 0 denote true and false, respectively. We say a formula F is *satisfiable* if there is at least one satisfying assignment to F and *unsatisfiable* if there is no satisfying assignment to F.

A formula F is in Conjunctive Normal Form (CNF) when F is expressed as:

$$F = C_1 \wedge C_2 \wedge \cdots \wedge C_n \tag{1.2}$$

where F is a conjunction of one or more clauses where each clause C_i is a disjunction of literals such as $C_i = (l_{i,1} \vee l_{i,2} \vee \cdots)$.

1.2.2 Bit-Vector Formulas

A bit-vector is an array of Boolean variables (bits). Here we only consider fixed-width bit-vectors which have a constant size. We assume that each Boolean variable of a bit-vector is indexed from 0 through $n-1$ where 0 is the rightmost bit of a n-width bit-vector. The first-order logic of bit-vector formulas has been widely studied [24, 25] and formulas arising from software are more naturally expressed as SMT (satisfiability modulo theories) formulas over bit-vectors than as plain CNF. 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 larger Boolean formulae. But bit-vector SMT is much more convenient for expressing the computations performed by software.

1.2.3 SAT Solving and Model Counting

The Boolean satisfiability problem (abbreviated SAT) is the problem of determining if there exists a satisfying assignment for a given formula. SAT is proven to be a NP-complete problem [26] which can be solved by a nondeterministic polynomial time algorithm. However, SAT instances that arise in practice can often be solved in better than exponential time. SAT solvers compute whether a given formula is either *satisfiable* or *unsatisfiable* and gives an example satisfying assignment when satisfiable. Since any Boolean formula can be translated into an equisatisfiable CNF formula with only a linear size increase, many SAT solvers take CNF as input. In addition, satisfiability modulo theories (SMT) problem is a decision problem for logical formulas with combinations of background theories such as floating point arithmetic, arrays, bit-vectors and so on. An SMT formula over finite theories can be converted to a SAT formula which has an equivalent satisfiability and we call this conversion "*bit-blasting*".

Model counting is the problem of computing the number of satisfying assignments of a given formula and the model counting problem for a Boolean formula is known as #SAT. Computing the exact model count is a complete problem for the complexity class #P and believed to be a much harder complexity class than NP. Practically, exact #SAT solving is also much less scalable than SAT solving. The model count is the number of all satisfying assignments of F and we denote a model count of F by $MC(F)$. Also, we denote that a model count count of F over a subset of (or projected) variables P by $MC(F, P)$. The principal and practical

way to solve #SAT (and SAT as well) is the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [1]. The DPLL algorithm is a backtracking-based search algorithm for deciding the SAT problem where as it stops after finding the first satisfying assignment. If the DPLL algorithm searches the entire solution space to find all the satisfying assignments, it can be used for the #SAT problem. Many approaches have been motivated by the DPLL algorithm and introduced various optimizations. Birnbaum and Lozinskii [3] formalized this DPLL idea and introduced their model counter CDP for counting models of propositional formulas. Based on this idea, Relsat [2], Cachet [4] sharpSAT [5] and DSHARP [6] showed improvements by using several optimizations such as component caching, clause learning, etc. Although exact model counters perform well in small-sized problems, scaling to the larger problem instances remains as a significant challenge. In order to overcome the limitation of exact model counting techniques, approximate model counting techniques have been proposed since many applications do not depend on the model count being exact. We will discuss more about approximate model counting techniques in Chapter 2.

1.2.4 Quantitative Information-flow Analysis

Quantitative information-flow (QIF) measures information leakage of a program. The basic concept of information-flow analysis was first introduced by Denning [27] and Gray [28] and many recent works have been proposed in many area such as secure information flow analysis [29], anonymity protocols [30] and side-channel analysis [31]. In this thesis, we are mostly interested in the problem whether a program/function can leak sensitive information from secret inputs into public outputs and how to measure the the leakage.

In order to totally protect the confidentiality of sensitive inputs, a standard approach is to enforce *noninterference*, which means public outputs are independent of secret inputs hence there is no way to reveal some information about the secret inputs from observing the outputs. However, the majority of complex systems depend on the secret inputs and reveal some amount of input information by the public outputs. For example, the secret inputs (password, credit card number, etc.) need to be entered into the systems and the systems reveal whether the input information is correct or not. As an adversary, incorrect information can be gathered mostly and the probability of finding out the correct information may be slightly increased.

Therefore, it is unavoidable to reveal some information about the secret inputs. The most

important question is that how much information is leaked. For example, image data anonymization systems hide someone's identity mostly by blurring his/her visible face, but some information such as gender, height or clothes still can be revealed if only the face is blurred. Also, recent study shows that image recognition methods based on artificial neural networks can reveal hidden information from this blurred image [32]. This increases the need of relaxing the noninterference and developing a quantitative theory of information flow that focuses on how much information is being leaked and calculating tolerable leakage bounds. We focus on measuring how much information is leaked and apply model counting techniques to measure the leakage. One of the QIF analysis techniques is the combination of model counting and symbolic execution. We use a symbolic execution tool to analyze binary executables and find a specific path which might leak some private data. Once the execution path is generated from the symbolic execution tool, we measure the leakage using model counting techniques.

1.2.5 Uniform Sampling

In the area of testing, the main goal is to generate various inputs for a program and check possible behaviours of the program. We expect to generate program testing inputs such that satisfy the program's requirements for completeness and correctness. Also, we want to reveal any fault from the testing inputs to properly fix defects. The key success of testing is based on the quality of input samples with which a program is executed. It is still a challenging problem to discover hidden faults from the generation of input samples.

There has been a growing interest in generating high-quality samples and this has motivated *constrained random sampling* [18, 33, 20, 21], which is designed to generate random samples satisfying given constraints. The constraints can be added to represent some behaviours of the program or to describe problematic input areas. In general, the constrained random sampling problem is to sample randomly from the set of solutions of input constraints and a constraint solver is called to generate random samples satisfying the constraints. Also, it is desirable to generate uniformly random samples when we do not have any knowledge about the program because we want to avoid only generating tests in a way that has a complementary bias that keeps us from generating fault-revealing inputs.

A good trade-off between scalability and uniformity has been always challenging for uniform sampling. Strengthening uniformity eventually loses scalability and increasing scalability provides weak theoretical guarantees. There have been many efforts to compromise these two

extreme problems. In this thesis, we focus on increasing scalability of random sampling and achieve a near-uniform distribution over samples.

1.3 Overview

The main contribution of this thesis is to increase scalability of hashing-based model counting techniques and the precision of QIF analysis and uniform sampling. Previous research applied QIF analysis and uniform sampling on small-sized programs that worked well with existing approaches since both symbolic execution and model counting are very expensive. We show more variety of programs to which we can apply QIF analysis and uniform sampling and improve existing model counting techniques to handle larger and more realistic inputs in this thesis.

This thesis is organized as follows. Chapter 2 introduces a hashing-based model counting tool, SearchMC, which computes a lower bound and an upper bound with a requested confidence level, and yields results more quickly than existing systems. Chapter 3 introduces a new structural approximate model counting tool, SMC, for quantifier-free SMT formulas over the theory of bit vectors (SMT-LIB QF_BV). This approach achieves guaranteed efficiency and sound bounds, at the expense of not providing an accuracy guarantee. In other words, it is a fast polynomial algorithm which gives firm lower and upper bounds that are always correct but the precision of results (the distance between lower and upper bounds) are not guaranteed. Chapter 4 explains a method called MultiSearchMC to increase scalability of hashing-based model counting techniques using parallelization. We first split an input formula into small formulae and run the combination of SMC and SearchMC in parallel. We conservatively combine all the results to compute an estimate of the model count. Chapter 5 shows applications of MultiSearchMC to software for program analysis and testing. We introduce quantitative information flow analysis using model counting techniques. We are interested in two basic scenarios of QIF analysis: measuring input and output. First, we select a program input which reaches to an interesting state such as a crash and then use symbolic execution to collect the path conditions while the program executes on that input. We measure the amount of information in the input to tell how much information can be leaked from the execution path. Secondly, if there is a program that receives some private input and produces some public output, we want to measure the number of outputs to quantify the amount of information leakage in the output. We also present case studies of quantitative information flow analysis with realistic programs. Uniform

sampling is another area which is applicable for model counting techniques. Uniform sampling takes an important role in functional verification of digital systems. When the distribution of errors is unknown, uniform sampling is desired to discover a bug. The main challenge is a trade-off between performance and uniformity. Our method, which uses MultiSearchMC, generates a large number of random samples efficiently and almost uniformly. Chapter 6 discusses some future directions and Chapter 7 concludes this thesis.

Chapter 2

SearchMC: A Hashing-based Model Counter

2.1 Introduction

Approximate model counting techniques have been proposed to increase scalability compared to exact model counting techniques. Specifically, hashing-based model counting techniques have been successful in giving an estimate of model count with a probabilistic range and confidence. Many hashing-based model counting techniques are based on the approximation technique of XOR streamlining [15], 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* [34] 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 [34]). 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[1] 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 [35], and Val et al. call subset model counting [36]. This capability is also used by Irvii et al. [37] to accelerate #SAT by not counting over redundant variables. 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

[1] The source code is available at `https://github.com/seonmokim/SearchMC`

numbers (counting the floating-point bit patterns individually, as contrasted with computing the measure of a subset of $\mathbb{R}^n$ [38, 39]). 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 [40] and ApproxMC-p [35], 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.

2.2 Background

2.2.1 XOR Streamlining

The main idea of XOR streamlining [15] 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 k XOR constraints, the original formula likely has at least 2^k models. If not, the formula likely has at most 2^k models. Thus we can obtain a lower bound or an upper bound with this approach. A random XOR constraint is proven to be a 3-universal hash function [41], which maps $\{0,1\}^n$ to $\{0,1\}^k$, and defined as follows:

$$\{h|h(y)\,[i] = a_{i,0} \oplus (\bigoplus_{j=1}^{n} a_{i,j} \cdot y[j]), a_{i,j} \in \{0,1\}, 1 \leq i \leq k\} \tag{2.1}$$

$h(y)\,[i]$ denotes the ith components of the vector $h(y)$ and $\oplus$ denotes the XOR operation. Assigning values of $a_{i,j}$ from $\{0,1\}$ randomly and independently leads to a random hash function.

In sum, adding k randomly-chosen parity constraints to input formula F reduces the number of solutions $MC(F)$ to $MC(F)/2^k$ in expectation. Moreover, if this augmented formula is satisfiable, then $MC(F)$ is likely greater than 2^k. If the augmented formula is unsatisfiable, then $MC(F)$ is likely less than 2^k. 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 [15] did not provide an algorithm to choose the parameters.

2.2.2 Influence

Newsome *et al.* [34] 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.

2.2.3 Exhaust-up-to-*c* query

Newsome *et al.* [34] also introduced the terminology of an "exhaust-up-to-c query" and this query is very crucial to many hashing-based model counting techniques. This query repeats a satisfiability query up to some number c of solutions, or until there are no satisfying values left. Suppose we have a Boolean formula F and call the exhaust-up-to-c query where F is given and c is equal to 3. If the return value is 0, 1 or 2, then F has exactly 0, 1, or 2 satisfying assignments, respectively. If the return value is 3, then F has 3 or more satisfying assignments. This is a good approach to find a model count if the number of solutions is small. This functionality can be easily implemented if a solver supports the incrementality feature, which can push or pop additional clauses after pre-processing an original formula. This query first asks for a solution with a satisfying assignment and then pushes *blocking clauses*, which is a negation of the found satisfying assignment, to not have the same solution again. It repeats this satisfiability check and adding its blocking clauses c times or until there are no satisfying assignment found.

Algorithm 2.1 shows the pseudocode of the exhaust-up-to-c query. It takes an input formula F and a positive integer c as input and returns the number of solutions which is less than or equal to c. `SAT`(F) returns F's satisfiability and its satisfying assignment if satisfiable. In the loop, it keeps finding a unique solution by adding blocking clauses and breaks out of the loop

Algorithm 2.1: `exhaust-up-to-c`

Input : F, c

Output: $nSat$

```
1 nSat ← 0
2 while nSat < c do
3     res, m ← SAT(F)
4     if res == unsat then
5         break
6     nSat ← nSat + 1
7     F ← F ∧ ¬m                          // Adding blocking clauses
8 return nSat
```

when there is no more solution found.

2.2.4 Particle Filter

A particle filter [42] 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.

2.2.5 ApproxMC

ApproxMC [11] is a well-studied hashing-based model counter and is inspired by MBound [15] which gives probabilistic bounds on the model counts by adding XOR constraints to an input formula as we described in Section 2.2.1. The basic idea of MBound is that adding one XOR constraint reduces the model count by a factor of 2 in expectation, and thus k independent constraints are likely to reduce the model count by 2^k. However, MBound does not automatically compute k, and finding the most appropriate k is quite challenging.

The first version of ApproxMC was proposed to compute k automatically, which runs queries iteratively until it finds the most appropriate k. ApproxMC takes an input formula F, a precision parameter ϵ and a correctness parameter δ as input. ApproxMC was described as an (ϵ, δ) counter such that the true model count is within the interval $[MC(F)/(1+\epsilon), MC(F) \cdot (1+\epsilon)]$ with a probability of at least $1-\delta$. The basic idea of ApproxMC was the combination of MBound and exhaust-up-to-c query. After adding k XOR constraints to an input formula, if an exhaust-up-to-c query of this augmented formula returns n solutions, which is less than c, the model count of this input formula is probably $2^k \cdot n$. Given the parameter ϵ, c was first computed and then ApproxMC ran exhaust-up-to-c queries with adding k constraints iteratively. Once the algorithm found the most appropriate k such that the exhaust-up-to-c query returns n solutions which is less than c, ApproxMC repeatedly queries the exhaust-up-to-c with these computed c and k to guarantee the result with a probability of at least $1 - \delta$.

We describe the core algorithms of ApproxMC from the original paper [11] in Algorithm 2.2 and 2.3. Note that they used the function name `BoundedSAT` and the variable name $thresh$, which is equivalent to the exhaust-up-to-c query and the parameter c, respectively. Algorithm 2.2 is the outer loop of ApproxMC. $thresh$ is a variable which depends on ϵ and determines a maximum number of SAT calls for $nSat$. t is a number of iterations to achieve an estimate to be within the interval $[MC(F)/(1+\epsilon), MC(F) \cdot (1+\epsilon)]$ with a probability of at least $1 - \delta$. The algorithm runs `ApproxMCCore` t times to store a set of estimates and returns a median of C as a final estimate.

Algorithm 2.3 is the inner loop of ApproxMC to find a single estimate. `BoundedSAT` returns a number of satisfying assignments if it is less than $thresh$. If the number of satisfying assignments is greater than or equal to $thresh$, `BoundedSAT` only finds $thresh$ solutions and stops. The algorithm first checks the number of solutions without adding any constraints. If not, it computes $nSat$ until $nSat$ is less than or equal to $thresh$ by increasing k sequentially by 1

Algorithm 2.2: `ApproxMC`

Input : F, ϵ, δ

Output: $finalEstimate$

```
1  thresh ← 1 + 9.84(1 + ε/(1+ε))(1 + 1/ε)^2
2  t ← ⌈17 log2(3/δ)⌉
3  C ← ∅
4  i ← 0
5  while i < t do
6      i ← i + 1
7      nSat ← ApproxMCCore(F, thresh)
8      if nSat ≠ ∅ then
9          AddToList(C, nSat)
10 finalEstimate ← FindMedian(C)
11 return finalEstimate
```

and this means the estimate is $nSat \cdot 2^k$.

In order to achieve the correctness of a probability of at least $1 - \delta$, Lemma 2.2.1 shows the core computation for δ as follows:

Lemma 2.2.1. *Let* $n = |F|, \epsilon \in [0, 1], k \in \mathbb{N}, k \leq \lfloor \log_2(MC(F)) \cdot \epsilon^2/(r \cdot \sqrt[3]{e}) \rfloor$ *and* $nSat =$ `ApproxMCCore`$(F, thresh)$ *where* $h \in H(n, k, r)$ *which is a randomly chosen r-universal hash function. It holds:*

$$Pr\left[(0 < nSat < thresh) \text{ and } (1+\epsilon)^{-1} MC(F) \leq 2^k \cdot nSat \leq (1+\epsilon) MC(F)\right] > 0.6 \tag{2.2}$$

The confidence can be raised to at least $1-\delta$ by invoking `ApproxMCCore` t times to use the median of the returned counts as the probability of at least a number of heads in t independent tosses of a biased coin. We refer the reader to [11] for more detailed analysis.

A family of ApproxMC tools has been improved over and over. ApproxMC2 [40] used a galloping search to reduce linear to logarithmic solver calls and Algorithm 2.4 shows the pseudocode of this galloping search. The first version of ApproxMC increased a number of

Algorithm 2.3: `ApproxMCCore`

Input : $F, thresh$

Output: $nSat$

```
1  nSat ← BoundedSAT(F, thresh + 1)
2  if nSat < thresh + 1 then
3      return nSat
4  else
5      n ← |F|
6      l ← log2(thresh) − 1; k ← l − 1
7      while 1 ≤ nSat ≤ thresh or i = n do
8          k ← k + 1
9          nSat ← BoundedSAT(F_{h,k}, thresh + 1)
10     if nSat > thresh or nSat = 0 then
11         return ∅
12     else
13         return nSat × 2^{k−l}
```

XOR constraints incrementally as Line 8 in Algorithm 2.3. ApproxMC2 replaced this search to a galloping search which multiplies/divides a number of XOR constraints by 2 until a desired number of XOR constraints gets close. When a desired number of XOR constraints gets close (Line 11 and Line 22), it increases/decreases a number of XOR constraints by 1.

ApproxMC3 [43] and ApproxMC4 [23] improved the interaction between the algorithm and its solver calls and optimized CryptoMiniSat [44], a SAT solver which is very suitable for solving a large number of XOR constraints. We refer the reader to [23, 43] for more detailed explanation of the optimizations. Our approach also uses the same SAT solver, and thus the improvements of this solver benefit both ApproxMC-family tools and SearchMC. The main difference between ApproxMC and SearchMC is about the algorithms: how we generate queries based on previous results and compute a lower and an upper bound. Our comparison experiments between ApproxMC and SearchMC are based on the same solver infrastructure to be a fair comparison.

Algorithm 2.4: `GallopingSearch`

Input: $F, thresh, prev$

Output: k

$lo \leftarrow 0$; $hi \leftarrow |F| - 1$; $k \leftarrow prev$
$visited[0] \leftarrow 1$; $visited[|F| - 1] \leftarrow 0$
while *true* **do**
    $nSat \leftarrow$ `BoundedSAT`$(F_{h,i}, thresh + 1)$
    if $nSat \geq thresh$ **then**
        if $visited[k + 1] = 1$ **then**
            return $k + 1$
        $visited[i] \leftarrow 1$ for $i \in \{1, \cdots, k\}$
        $lo \leftarrow k$
        if $|k - prev| < 3$ **then**
            $k \leftarrow k + 1$
        else if $2 \cdot k < |F|$ **then**
            $k \leftarrow k \cdot 2$
        else
            $k \leftarrow (hi + k)/2$
    else
        if $visited[k - 1] = 1$ **then**
            return k
        $visited[i] \leftarrow 0$ for $i \in \{k, \cdots, |F|\}$
        $hi \leftarrow k$
        if $|k - prev| < 3$ **then**
            $k \leftarrow k - 1$
        else
            $k \leftarrow (k + lo)/2$

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 [45]. 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.

2.3.1 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:

$$Pr_{=n}(N,k) = \binom{N}{n} (\frac{1}{2^k})^n (1 - \frac{1}{2^k})^{N-n} \tag{2.3}$$

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:

$$Pr_{\geq n}(N,k) = 1 - \sum_{i=0}^{n-1} Pr_{=i}(N,k) \tag{2.4}$$

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 [11, 35, 40]. Earlier work [15] 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. Also, for use in the very first step of the algorithm we 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.

2.3.2 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 2.5. 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

Algorithm 2.5: `SearchMC`

Input: F, $thres$, cl, α, $initDist$

```
1  cl ← cl + α(1 − cl)                          // Confidence level adjustment
2  width ← getWidth(F)          // The width of an output bit-vector
3  prior ← initDist                              // Initial distribution
4  δ ← width
5  while δ > thres do
6      c, k ← ComputeCandK(prior, width)
7      nSat ← MBoundExhaustUpToC(f, v_p, k, c)
8      post, ub, lb ← UpdateDist(prior, c, k, nSat, cl)      // See Sec. 2.3.1
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
```

more likely that the requested confidence can be met. If $\alpha = 0$, we do not adjust the input confidence level. We have set the value for α to be $\frac{1}{4}$ empirically in our experiments. 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.

2.3.3 Variables

There are several variables: $prior$, $post$, $width$, k, c, $nSat$, ub, lb and δ. $prior$ represents a prior distribution by sampled particles with 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 2.6 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.

Algorithm 2.6: `ComputeCandK`

Input: $prior$, $width$

Output: c, k

$\mu, \sigma \leftarrow$`GetMuSigma`$(prior)$

$c \leftarrow \lceil((2^\sigma + 1)/(2^\sigma - 1))^2\rceil$

$k \leftarrow \lfloor \mu - \frac{1}{2}\log_2 c \rfloor$

if $k \leq 0$ **then**

  $c \leftarrow 2^{width} + 1$ `// In this case, c is effectively infinite`

  $k \leftarrow 0$ `// No constraints`

return c, k

2.3.4 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 Figure 2.1. 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 ruler 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 `UpdateDist` to update the distribution based on the result of the query.

The pseudocode for algorithm `UpdateDist` is described as Algorithm 2.7. A prior distribution $prior$ and a posterior distribution $post$ are represented as a set of sampled particles (influences). We sampled 500 particles for each `UpdateDist` function call. Once we have the updated distribution, we can find out the interval of a given confidence level.

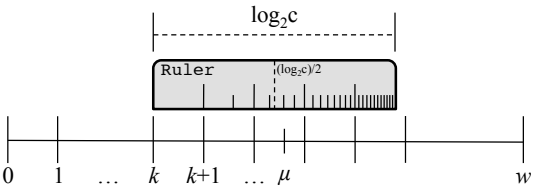


Figure 2.1: SearchMC's ruler intuition

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

Algorithm 2.7: `UpdateDist`

Input: $prior$, c, k, $nSat$, CL

Output: $post, ub, lb$

```
1  for t from 1 to nParticles do
2      x_t ← prior_t                    // A list of sampled particles
       /* Updating each weight of each particle                    */
3      if nSat < c then
4          w_t = Pr_{=nSat}(2^{x_t}, k)
5      else
6          w_t = Pr_{≥nSat}(2^{x_t}, k)
7  w ← normalize()                      // Normalizing the weights
8  post ← sample(x, w, nParticles)      // Resampling based on the
     weights
9  ub, lb ← getBounds(post, cl)
10 return post, ub, lb
```

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 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 2.6.

2.3.5 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 [11], which in turn is a variant on Theorem 5 by Schmidt *et al.* [46]. For convenience we substitute our own terminology as follows.

Lemma 2.3.1. *Let $nSat$ be the return value from* `MBoundExhaustUpToC` *where $0 < nSat < c$ and $k \leq \log_2(MC(F))$. Then,*

$$Pr\left[(1+\epsilon)^{-1}MC(F) \leq 2^k MC(F_h) \leq (1+\epsilon)MC(F)\right] > 0.6$$

Chakraborty *et al.* use *pivot* for what we call c from an exhaust-up-to-c query and *threshold* $= 1 + 9.84(\frac{\epsilon}{1+\epsilon})(1+\frac{1}{\epsilon})^2$. Since $0 < \epsilon < 1$, c (*pivot*) should be always greater than 60 to make the lemma true with a probability of at least 0.6. (The constant 0.6 comes from $(1-e^{-3/2})^2 \approx 0.6035$.)

ApproxMC2, developed concurrently, 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. By comparison SearchMC's approach of maintaining a probabilistic estimate explicitly 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 performance of SAT solvers on CNF-XOR formulas.

2.4 Evaluation

In this section, we present our experimental results. All our experiments were performed on a machine with an Intel Core i7 3.40Ghz CPU and 16GB memory. Our main algorithm is implemented with a Perl script and `UpdateDist` 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 CryptoMiniSat [44] for CNF formulas and Z3 [47] and MathSAT5 [48] 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 [49, 50] 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 [51] and ISCAS89 benchmarks [52] converted to CNF files by TG-Pro [53] and compare the results of the benchmarks with ApproxMC2 [40] and ApproxMC-p [35]. 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.* [54]. 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 hours. The benchmarks that were solved completely by the other tools were also solved completely by SearchMC. Figure 2.2 and 2.5 are based on the benchmarks that were solved by all tools.

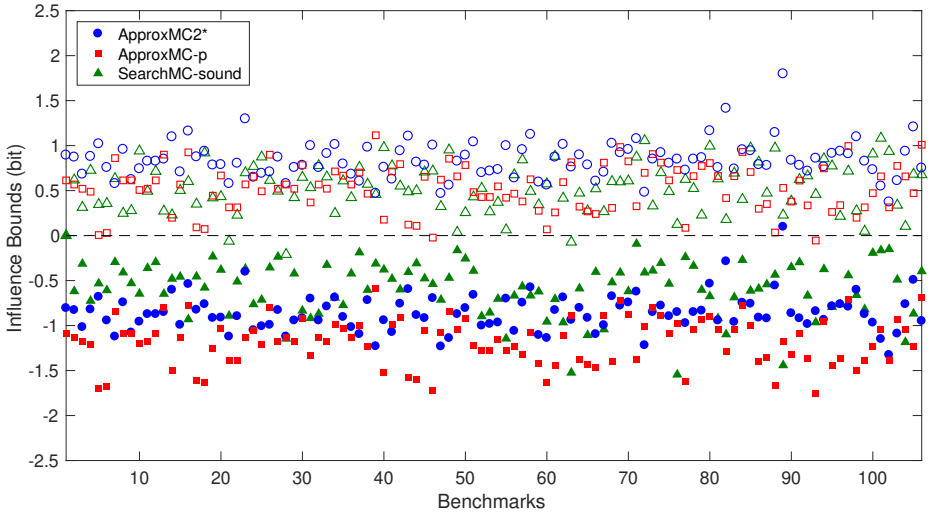


Figure 2.2: Reported lower and upper bounds comparison of SearchMC and ApproxMC2

Figure 2.2 compares the quality of lower bounds and upper bounds computed by SearchMC-sound, ApproxMC-p and ApproxMC2*[2]. Note that the benchmarks are arranged in increasing order of the true influence in Figure 2.2 and 2.5. 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.

Since ApproxMC2 computes the bounds conservatively by using more queries, ApproxMC2's intervals are more closely centered on the true influence, and it out-performs the requested 60%

[2] ApproxMC2* refers to our own re-implementation of the ApproxMC2 algorithm. With the version of ApproxMC2 for our experiments we encountered problems in which the SAT solver would sometimes fail to perform Gaussian elimination, which unfairly hurt the tool's performance.

confidence level. By comparison, our tool and ApproxMC-p compute the bounds more aggressively, and in a few cases the true result is just outside the reported interval (visible as empty markers below the dotted line), though this still occurs somewhat less often than the 40% implied by the confidence level. 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. In brief, SearchMC reported 65 correct bounds out of 106 benchmarks, which is slightly higher than the requested 60% confidence level.

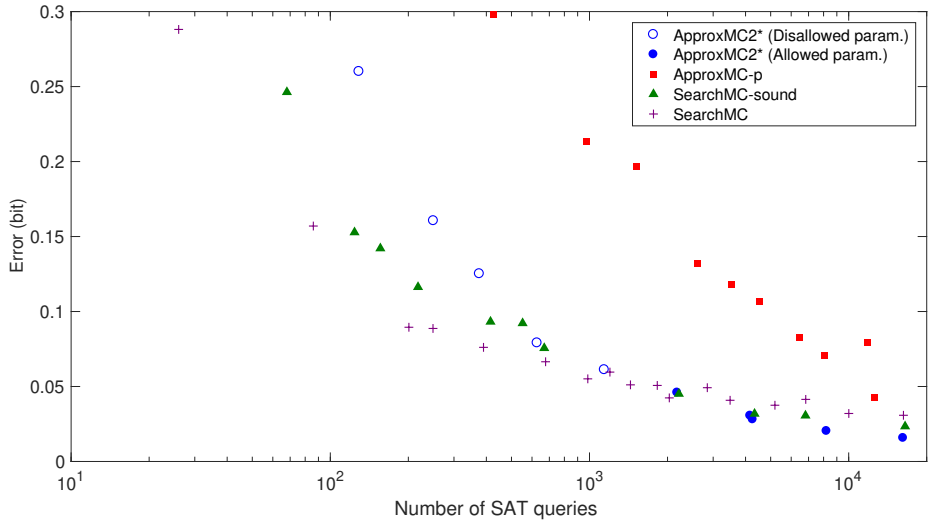


Figure 2.3: Performance vs error trade-off of SearchMC and ApproxMC2

Figure 2.3 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.

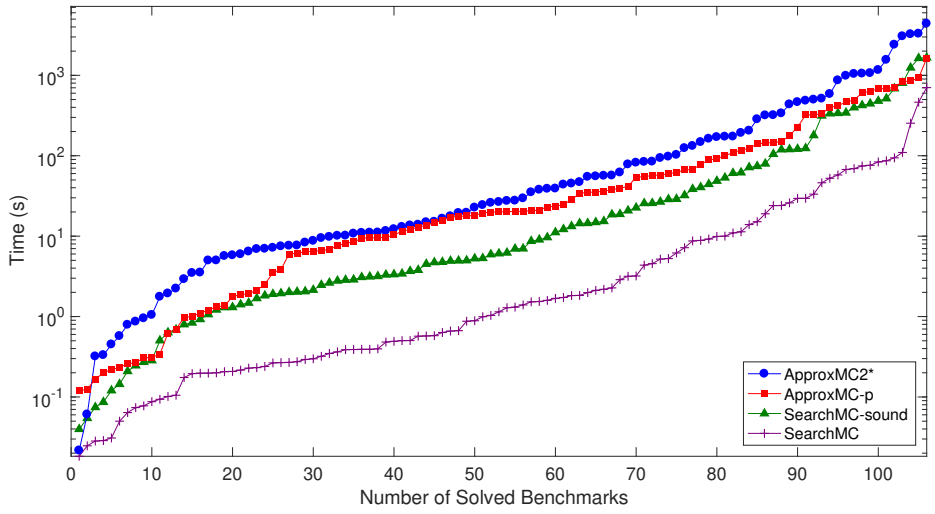


Figure 2.4: Time performance comparison of SearchMC and ApproxMC2

We also compare the running-time performance with ApproxMCs and show the running-time performance comparison on our 106 benchmarks in Figure 2.4. 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 though 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 seconds, compared to an average of 125.48 for ApproxMC-p. ApproxMC2* requires an average of 298.11 seconds 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 Figure 2.5. For this figure the benchmarks are sorted consistently by increasing true model count for all tools. 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.

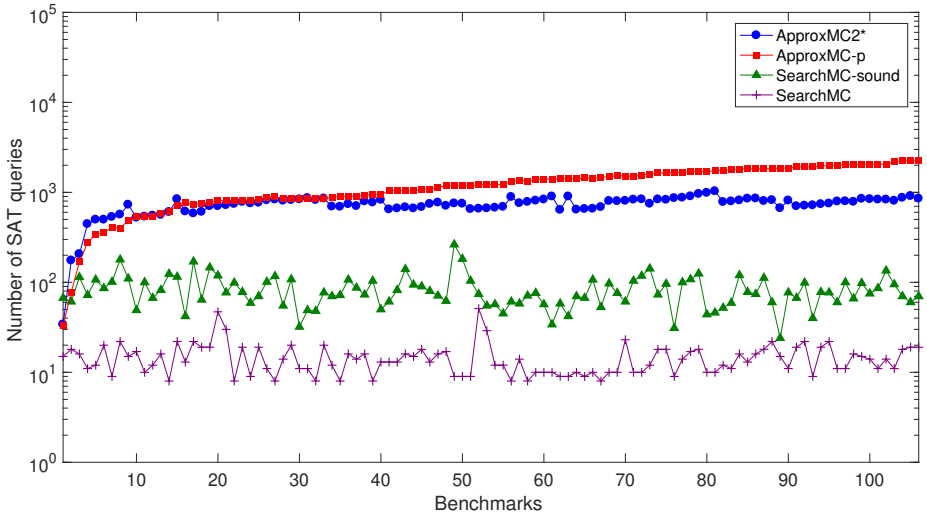


Figure 2.5: Number of SAT queries comparison of SearchMC and ApproxMC2

2.4.1 Case Study: Floating Point / Differential Privacy

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 [55] 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 [56]. 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 $10 + \textit{noise}$ but not as $11 + \textit{noise}$ in our scenario, its release

Problem	All noise			Intersection	
size	Expected	SearchMC	Time (s)	SearchMC	Time (s)
9e7, 2^6	5.977	[5.643, 6.499]	52	3.170	138
10e7 2^6	5.977	[5.872, 6.668]	50	3.459	192
11e7, 2^6	5.977	[6.153, 7.052]	72	2.000	111
12e7, 2^6	5.977	[6.047, 6.792]	72	2.000	127
13e7, 2^7	6.989	[5.757, 6.519]	212	2.585	451
14e7, 2^7	6.989	[5.757, 7.444]	187	2.000	382
15e7, 2^8	7.994	[7.374, 8.069]	164	1.000	312
16e7, 2^9	8.997	[8.566, 9.073]	470	3.322	1585
16e8, 2^{10}	9.999	[10.076, 10.844]	279	4.754	5279
18e8, 2^{10}	9.999	[9.675, 10.099]	583	1.000	1137
19e8, 2^{11}	10.999	[10.825, 11.404]	757	3.585	9848

Table 2.1: Results and performance of model counting ($\log_2$ shown) of naive Laplacian noise in IEEE floating point

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 $10 + \textit{noise}$ as well as the size of the intersection of this set with the $11 + \textit{noise}$ set.

The results and running time are shown in Table 2.1. 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, future solvers may significantly improve the technique's performance.

2.5 Related Work

In 1962, Davis, Putnam, Logemann and Loveland introduced their SAT solving algorithm called DPLL algorithm [1], which is a backtracking search algorithm. Their algorithm is one of the most commonly used algorithms and many efficient SAT solvers have been developed from its basic ideas. GRASP [57] was proposed in 1996, which summarized and integrated previously proposed search-pruning techniques to improve the basic DPLL algorithm flow. The most important contribution of GRASP was the introduction of the Conflict Driven Clause Learning (CDCL) learning process, which is very effective in narrowing the search space with the DPLL algorithm. Subsequent development of the technology can be summarized in the CDCL, including a random restart, heuristic decision-making based on the active value, and effective reasoning to support the data structure. With the introduction of these new technologies, a number of more efficient SAT algorithms such as MiniSat [58], Chaff [59], BerkMin [60], CryptoMiniSat [44], PicoSAT [61], and Lingeling [62] were introduced.

Model counting techniques can be categorized into three areas: exact model counting, randomized approximate model counting and non-randomized approximate model counting. For a given formula, exact model counters give the exact number of solutions and approximate model counters provide a lower bound and/or upper bound. Exact model counters do not perform well as the size/complexity of a problem increases, thus approximate model counters have been proposed to resolve this scalability challenges. It is often sufficient to provide rough estimates instead of the exact model counts, especially when this can be done much faster. The main difference between randomized and non-randomized approximate model counting techniques is probabilistic soundness. Randomized approximate model counting techniques are likely to use random sampling and a SAT solver to produce probabilistic bounds with high probability.

In contrast, non-randomized approximate model counting techniques generally use approximations that are more efficient but do not provide probabilistic bounds.

Some of the earliest Boolean model counters used the DPLL algorithm for counting the exact number of solutions. Birnbaum and Lozinskii [3] formalized this idea and introduced their model counter CDP for counting models of propositional formulas. Based on this idea, Relsat [2], Cachet [4] sharpSAT [5] and DSHARP [6] showed improvements by using several optimizations. Relsat uses component analysis in which the model count of a formula is the product of the model count of each sub-formula (component). Cachet shows optimizations by combining component caching and clause learning. sharpSAT introduces an improved caching technique to reduce the space requirement compared to Cachet. DSHARP is a CNF $\rightarrow$ d-DNNF compiler for efficient reasoning and uses sharpSAT as a back-end. The major contribution of countAntom [7] is techniques for parallelization, but it provides state-of-the-art performance even in single-threaded mode.

Randomized approximate model counting techniques perform well on many kinds of a formula for which finding one solution is efficient. Also, there can sometimes be a smooth trade-off chosen between computational effort and the precision of results. However, this solving is still relatively expensive, so research to get the best precision for a given cost is still important. Wei and Selman [8] 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 [9] improves this sampling idea and gives a lower bound with high probability by using a heuristic sampler. MiniCount [10] is based on a framework to compute an upper bound under statistical assumptions which is that counting the number d of branching decisions (except unit propagations and failed branches) can be used to estimate the total number of solutions by setting a variable to true or false randomly. Specifically, they showed that the expected value of d is not lower than $\log_2$ of the true model count. By estimating the expected value, they can obtain the upper bound of the true model count. Also, they observed that d often has a distribution that is close to a normal distribution. Thus, the expected value can be easily computed under this assumption rather than computing low and high values of d. This only guarantees the upper bound and they used a different method to compute a lower bound.

MBound [15] 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. [11] introduced ApproxMC, an approximate model counter for CNF formulas, which automated the choice of XOR streamlining parameters. An improved algorithm ApproxMC2 [40] 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 [63] and ApproxMC-p [35]. SMTApproxMC proposes word-level constraints based on modular arithmetic instead of bit-level XOR constraints, however these will not likely provide comparable performance until SMT solvers implement modular-arithmetic Gaussian elimination. ApproxMC-p implements projection (counting over only a subset of variables), and also gives more efficient formulas for parameter selection.

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.* [64] compute an upper bound on the number of bits about an input that are revealed by an error report. They measure the entropy loss of an error report by computing the number of bits revealed by subsets of path conditions first and then combining these partial results to get the final result. Meng and Smith [65] 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.* [66] propose a model counting technique over an expressive string constraint language. Their tool computes the bounds on the cardinality of the valid string set and uses generating functions for reasoning about the cardinality of string sets.

2.6 Chapter Summary

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.

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.

Chapter 3

SMC: A Structural Model Counter

3.1 Introduction

Recent research use hashing to reduce approximate model counting to an adaptive sequence of satisfiability queries covering subsets of the solution space as we mentioned in the previous chapter. These algorithms provide probabilistic bounds on the accuracy of their approximation, and they can take advantage of advances in satisfiability solving (both SAT and SMT). However these approaches still have limited scalability in practice: they require a large number of satisfiability queries to achieve tight bounds, and hashing can make individual queries much more expensive, given the complex interactions between hashing constraints and solver optimizations. Some roots of the difficulty of this problem have been investigated by Dudek et al. [67, 68].

Also valuable but less developed are approximation algorithms which achieve guaranteed efficiency and sound bounds, at the expense of not providing an accuracy guarantee. This trade-off can be achieved using algorithms similar to ones used in static program analysis that derive lower and/or upper bounds following the syntactic structure of a formula rather than using semantic decision procedure queries. We refer to these as "structural" approximate model counting algorithms.

Previous structural model counting algorithms have been specialized for narrow domains, or have been built into larger systems in ways that are not easily reusable. In this work, we build a new structural approximate model counting tool for quantifier-free SMT formulas over the theory of bit vectors (SMT-LIB QF_BV), one of the most common theories used to model

bounded arithmetic and software semantics. We also provide a commonly useful generalization known as projected model counting, in which a user can specify a subset of the variables in a formula over which a model should be counted. Our tool uses algorithms which build on the partial description of a previous closed-source tool [69], but we needed to develop new structural rules for cases that were missing or restricted in previous work. We extend the algorithm to cover a more complete set of bit-vector operators, and to use both the signed and unsigned orderings of bit vectors. However our current implementation, like Martin's [69], is limited to conjunctions and not arbitrary Boolean combinations of bit-vector relations. This matches the needs, for instance, of typical single-path symbolic execution, where a path condition is a conjunction of branch conditions, each of which is an equality or inequality over numeric (e.g. bit-vector) terms.

We have built a standalone tool, SMC, that operates on input in the standard SMT-LIB2 format, and which we have open-sourced[1]. We have checked the correctness of the propagation rules for each type of bit-vector expression via bounded exhaustive testing. We empirically compare SMC's performance with representative state-of-the-art exact and approximate model counting tools. The results show that SMC's performance scales much better than these other tools, and that the sound upper and lower model-count bounds that it provides are often usefully accurate. Even if the bounds are very loose, they are still useful for SearchMC [12] to use the bounds as a more refined initial hypothesis. Recall that the initial hypothesis for SearchMC is a uniform distribution over 0 to the maximum bit-width of the output bit-vector. If a more refined prior hypothesis (such as a uniform distribution over a range between a lower and an upper bounds computed by SMC) is given, this reduces search space; thus, SearchMC is able to give a desired answer faster. Our experimental results show the performance benefit up to 1.36x speedup.

3.2 Algorithm

SMC (Structural Model Counter) takes as input an SMT-LIB2 formula which consists of variables and assertions and it outputs lower and upper bounds on the number of satisfying assignments that make a given formula true. First, we describe how this structural model counting technique works with a simple example in Figure 3.1. Since `x` and `y` are 8-bit variables, the

[1] The source code and benchmarks are available at: `https://github.com/seonmokim/smc`

```
(declare-fun x () (_ BitVec 8))
(declare-fun y () (_ BitVec 8))
(assert (= y (bvadd (bvand x 15) 4)))
```

Figure 3.1: A simple SMT-LIB2 formula example for SMC

model counts of two variables are both 256 when they are declared. Then we parse an assertion to analyze the structure of the formula. `(bvand x 15)` has the minimum 0 and the maximum 15. This generates 16 distinct values. Adding 4 makes the minimum 4, the maximum 19 and still 16 distinct values. This is equal to `y` thus `y` has 16 distinct values. The model count of this formula (`x` and `y`) is 256 since `x` has 256 distinct values and `x` determines `y`. This shows a simple process of the structure model counting. In this section, we describe this algorithm in detail, its correctness, some current limitations and differences from the previous work.

The main algorithm is mostly inspired by Martin's FSCB (Fast Solution Count Bounder) algorithm [69]. He proposed this idea to be a fast algorithm that can handle complex expressions such as standard arithmetic or bit shifts. However, FSCB does not consider signed data types and the source code is not available. We extend FSCB to handle both unsigned and signed data types and cover more operators including signed division, signed less-than, signed greater-than and so on. Also, we improve the idea to compute tighter bounds and verify its correctness using small-sized unit tests exhaustively. In this section, we describe two steps of our algorithm. It first computes the bounds for each individual assertion, and then merges them for the model count of a given formula.

3.2.1 Per-Assertion Bounds and Analysis

When a variable is declared or an expression is generated, we create a corresponding node to represent this variable or expression. Each node contains elements as [`ul, uh, sl, sh, lc, hc, hom, vars`]. Since we only consider bit vectors in SMT-LIB2 format, we compute both cases, unsigned and signed representations, in a node. `ul`, `uh`, `sl` and `sh` are the unsigned minimum (low), the unsigned maximum (high), the signed minimum and the signed maximum of the node, respectively. `lc` and `hc` are the low cardinality (lower bound), the high cardinality (upper bound) of the node, respectively. We select the bounds which have a shorter

interval between the unsigned and signed bounds since both the bounds are sound and tighter bounds give a better precision. `hom` is a flag such that a node is homogeneous only if every image has the same number of preimages. For example, `(bvand x 15)` is homogeneous since it generates 16 distinct values and each value (image) has the same number of preimages (16 cases if `x` is 8-bit). This flag is useful when computing bounds more precisely with a constant value. For example, let us assume that we have `(= (bvand x 15) 0)` and `x` is 8-bit. `(bvand x 15)` should be zero to satisfy this assertion and one value of `(bvand x 15)` maps to 16 preimages of `x`. Therefore, this can be computed easily as `x` has exactly 16 distinct values to satisfy this assertion. `vars` is a set of variables which are present in an assertion. As we described above, we represent signed and unsigned values in a single node. One reason that we need both representations is that we want to handle signed operators such as signed division, signed less-than, signed greater-than and so on. Another reason is the ways of computing signed values and unsigned values are different. For example, the minimum value of adding two unsigned values is 0 but we need to consider negative values when we add two signed values. Therefore, we maintain both representations in every operation and a user can select more appropriate bounds as a final answer.

SMC first replaces each constant or variable from an expression with the corresponding node. If it is a constant `c`, the node can be represented as [`c, c, c, c, 1, 1, true,` {}]. Note that if c is a binary or hexadecimal number, we convert the number using the two's complement for `sl` and `sh`. If it is a 8-bit variable `x`, the node can be represented as [`0, 255, -128, 127, 256, 256, true,` {`x`}]. The next step is that SMC breaks down an expression tree and generates a node based on the expression rules starting from sub-expressions. SMC determines how to compute the node values from the expression rules for each supported operation. We can extend SMC to support additional operations by adding new operation rules. We only describe a subset of the operation rules here.

Pseudocode for `bvadd` is shown in Algorithm 3.1. When the tool reaches a `bvadd` operation, it first checks whether the variables are both constant values. This checking applies to other operations as well and we can easily compute all the values if they are both constant values. Next we check that the answer might be an arithmetic positive or negative overflow. If this occurs, we set `ul` and `uh` as the minimum and the maximum of the variable, respectively, based on its bit-width unless the variables are both constants. This applies to `sl` and `sh` to be the negative minimum and positive maximum, respectively. `lc` can be computed as the sum

of two variable's minimum cardinality minus 1. In order to generate the smallest set of adding two sets, one value has to be generated in multiple ways. For example, adding {1, 2, 3} and {2, 3} makes {3, 4, 5, 6} which has 4 elements. If the function has variables in common (for example, `(x&1)+(¬x&1)`), the minimum cardinality could be 1. The maximum cardinality `hc` can be computed as the multiplication of each variable's maximum cardinality. Note that the cardinalities must be less than the distance between its high value and low value. The addition is homogeneous if one variable is homogeneous and another is a constant or they do not have variables in common and both are permutations. We say a variable is a *permutation* if all elements of the variable have not been modified from its declaration such as a node with [`0, 255, -128, 127, 256, 256, true, {x}`].

Let us go back to the example in Figure 3.1 and proceed with the SMC algorithm. When `x` and `y` are declared, two corresponding nodes are generated. Initial values of `x` node would be [`0, 255, -128, 127, 256, 256, true, {x}`] and initial values of `y` node would be the same except `vars={y}`. If we have the expression `(= y (bvadd (bvand x 15) 4))`, we parse the expression and generate a node from the expression. First, `(bvand x 15)` generates a node `temp1` with [`0, 15, 0, 15, 16, 16, true, {x}`] and then we replace the sub-expression `(bvand x 15)` with `temp1`. The expression becomes `(bvadd temp1 4)` and this returns a node with [`4, 19, 4, 19, 16, 16, true, {x}`]. When there is an equality or inequality check in an assertion, SMC computes a lower bounds and upper bound of variables in the assertion. We denote `lb` and `hb` which are the lower and upper bounds of a set of variables, respectively. Algorithm 3.2 shows the crucial part to compute the bounds.

This equal rule computes the cardinalities of the left-hand side and the right-hand side and then the cardinality of the intersection between the left-hand side and the right-hand. Depending on the homogeneity of the variable, we compute the low and high density of the variable so we can compute the lower bound and upper bound. We also apply the same procedure to signed values and select the bounds that has a smaller interval. In this example, SMC computes the lower bound as 256 and the upper bound as 256 and we can know 256 is precise because the upper and lower bounds are the same, even if we didn't know the right answer already.

Algorithm 3.1: The operation rule of bvadd (f, g)

Input: f, g

Output: res

if `isConstant(f)` and `isConstant(g)` **then**
  $ul = uh = (f.ul + g.ul)\%2^{f.width}$
  $sl = sh = (f.sl + g.sl)\%2^{f.width}$
  return $ul, uh, sl, sh, 1, 1, true, [f.vars \cup g.vars]$
if $f.uh + g.uh > umax$ **then**
  $ul = umin$
  $uh = umax$
else
  $ul = f.ul + g.ul$
  $uh = f.uh + g.uh$
if $f.sl + g.sl < smin$ or $f.sh + g.sh > smax$ **then**
  $sl = smin$
  $sh = smax$
else
  $sl = f.sl + g.sl$
  $sh = f.sh + g.sh$
if `isCommon(f, g)` **then**
  $lc = 1$
else
  $lc = \texttt{min}(f.lc + g.lc - 1, \texttt{abs}(uh - ul), \texttt{abs}(sh - sl))$
$hc = \texttt{min}(f.hc * g.hc, \texttt{abs}(uh - ul), \texttt{abs}(sh - sl)$
$hom = (f.hom$ and `isConstant(g)`) or (`isConstant(f)` and $g.hom)$
    or (not `isCommon(f, g)` and `isPerm(f)` and `isPerm(g)`)
return $ul, uh, sl, sh, lc, hc, hom, [f.vars \cup g.vars]$

Algorithm 3.2: The operation rule of =

Input: f, g

Output: lb, ub

$leq \leftarrow \texttt{max}(f.ul, g.ul)$

$heq \leftarrow \texttt{min}(f.uh, g.uh)$

$minlhit \leftarrow f.lc - \texttt{max}(f.uh - heq, leq - f.ul)$

$minlhit \leftarrow \texttt{min}(minlhit, heq - leq + 1)$

$minrhit \leftarrow g.lc - \texttt{max}(g.uh - heq, leq - g.ul)$

$minrhit \leftarrow \texttt{min}(minrhit, heq - leq + 1)$

$inter \leftarrow \texttt{max}(1, minlhit + minrhit - (heq - leq + 1))$

$ic \leftarrow 2^{f.width+g.width}$

if $f.hom$ **then**

  $ld_f = 2^{f.width}/f.hc$

  $hd_f = 2^{f.width}/f.lc$

else

  $ld_f = 1$

  $hd_f = 2^{f.width} - f.lc + 1$

if $g.hom$ **then**

  $ld_g = 2^{g.width}/g.hc$

  $hd_g = 2^{g.width}/g.lc$

else

  $ld_g = 1$

  $hd_g = 2^{g.width} - g.lc + 1$

$lb = \texttt{max}(1, \texttt{min}(inter * ld_f * ld_g, ic))$

$ub = \texttt{min}(inter * hd_f * hd_g, ic)$

return lb, ub

3.2.2 Combining Bounds

The second step is to combine per-assertion bounds. For each assertion with comparison operators such as equal, greater-than or less-than, we compute the lower and upper bounds on the number of solutions to variables in the assertion. Recall that `lb` and `hb` are the bounds of a set

of variables. We use `vars` to denote the set of variables in an assertion. Given two such bounds, we can merge per-assertion bounds into bounds that apply to both equations together. We recursively merge the bounds pairwise until the bounds represent all the variables in the system. For example, if two bounds are independent, we can simply multiply each lower bound and upper bound. We present pseudocode for merging bounds in Algorithm 3.3. "Bound" contains a lower bound lb, an upper bound ub and corresponding variables $vars$. We first check whether two bounds are independent or not. If they are independent, we can just simply multiply two bounds. If they have variables in common, then we consider four cases: they are identical, one is subset of another (or vice versa) and they are overlapping sets. We compute the bounds conservatively based on the case. The bounds can be more precise based on the order of merging such as merging similar variables first. This needs an extra running time and we left this for the future work.

3.3 Evaluation

3.3.1 Correctness

We have tested the correctness of the operation rules of bit-vector expression using bounded exhaustive checking. We set each variable to be a bit-vector of width 4 and executed each operator with the power set of each variable. We checked whether `ul`, `uh`, `sl`, `sh`, `lc` and `hc` from our operation rules are sound. We verify unary operators and binary operators. First, we compute `ul`, `uh`, `sl`, `sh`, `lc` and `hc` based on our operation rules. We generate the power set of one variable which has 2^{16} sets and execute an operation which generates 2^{32} sets for binary operators. The same procedure is applied to unary operators. We verify that (1) the unsigned smallest value of each set is greater than or equal to `ul`, (2) the unsigned largest value of each set is less than or equal `uh`, (3) the signed smallest value of each set is greater than or equal to `sl`, (4) the signed largest value of each set is less than or equal to `sh` and (5) the number of elements of each set is less than or equal to `hc` and greater than or equal to `lc`. However, we have not checked the correctness of merging bounds due to the state explosion problem since we need to test the case where bounds contain multiple variables.

Algorithm 3.3: The operation rule of mergeBounds

Input: Bound a, Bound b

Output: Bound res

$vars = a.vars \cup b.vars$
$inter\_vars = a.vars \cap b.vars$
$inter\_max = min(2^{width}, math.gcd(a.ub, b.ub))$
$inter\_min = min(2^{width}, math.gcd(a.lb, b.lb))$
if *a.vars* and *b.vars* are in common **then**
    if *set(a.vars) == set(b.vars)* **then**
        $la = min(a.lb, b.lb)$
        $ha = min(a.ub, b.ub)$
    else if $a.vars \subset b.vars$ **then**
        $lb = b.lb$
        $ub = b.ub$
    else if $b.vars \subset a.vars$ **then**
        $lb = a.lb$
        $ub = a.ub$
else
    $lb = a.lb \times b.lb$
    $ub = a.ub \times b.ub$
return Bound*(lb, ub, vars)*

3.3.2 Experimental Result

In this section, we show our experimental results and all our experiments were performed on a machine with an Intel Core i7 3.40Ghz CPU and 16GB memory. We implement our algorithm with Python and use our own SMT-LIB2 parser. Our algorithm supports SMT-LIB2 format [45] and can be extended to support more formats. We compare our algorithm with state-of-the-art model counters: SearchMC [12] and DSHARP_P [17]. SearchMC is an approximate model counter using XOR hashing constraints to estimate a lower bound and upper bound of the model count. It is a randomized algorithm and gives a desired level of distance between a lower bound and upper bound with a probability of at least 0.6. On each benchmark, we ran SearchMC

		DSHARP_P		SearchMC		SMC	
Benchmark	#Bits	$\log_2(MC)$	Time	Bounds	Time	Bounds	Time
coloring_4	32	30.75	0.82	[30.15, 31.05]	1088.5	[29.61, 32.00]	0.001
FINDpath1	32	21.96	0.09	[20.90, 22.17]	3.99	[4.00, 32.00]	0.001
getopPath1	8	7.92	0.003	[7.50, 8.25]	0.21	[7.92, 7.94]	0.001
queue	16	6.39	0.006	[5.66, 6.97]	0.17	[4.62, 10.78]	0.001
calDate_10	36	16.95	0.002	[16.13, 17.49]	0.30	[16.95, 16.95]	0.001
5_10_1	160	12.02	123.1	[11.34, 12.31]	20.73	[7.24, 24.77]	0.003
5_20_1	160	8.44	351.6	[7.69, 8.88]	29.6	[7.24, 24.77]	0.004

Table 3.1: Comparison results of DSHARP_P, SearchMC and SMC. Runtime is measured in seconds.

until it gave an answer with a desired confidence (60%) and repeated 10 times to compute the average of the results. DSHARP_P is an exact model counter and is implemented on top of DSHARP [6] to support projection. We collected various SMT_BV benchmarks from previous works [13, 70].

Here we show partial results of some representative benchmarks. Table 3.1 shows a comparison on performance and approximation. The second column shows the number of bits we want to count over. Since DSHARP_P is an exact model counter, we take $\log_2$ of the answer which shows in the third column and the running times (in seconds) of DSHARP_P are shown in the fourth column. We also show the bounds (log base 2) computed from SearchMC and SMC following with their running times. DSHARP_P performs well on a small-sized problems and its performance decreases as the size and the complexity of formula increases. In these experiments, SearchMC used the state-of-the-art SMT(BV) solver Z3 [47] and its performance was highly dependent on the performance of the solver.

SMC shows a good precision on some benchmarks if the structure of the formula is well-organized. However, it gave very loose bounds on some benchmarks which SMC was not able to analyze the formula well. For example, 5_10_1 and 5_20_1 are the volume computation problems of convex bodies and consist of a number of inequality constraints. This type of problems show very loose bounds since SMC computes the bounds very conservatively on inequality constraints. The main benefit of SMC is the running times. This shows that our

		SearchMC			SMC+SearchMC		
Benchmark	#Bits	Bounds	Time	#Loops	Bounds	Time	#Loops
coloring_4	32	[30.15, 31.05]	1088.5	6.3	[30.16, 31.53]	830.9	4.9
FINDpath1	32	[20.90, 22.17]	3.99	8.2	[21.24, 22.42]	3.55	7.7
getopPath1	8	[7.50, 8.25]	0.21	4	-	-	-
queue	16	[5.66, 6.97]	0.17	6.1	[5.60, 6.78]	0.15	4.8
calDate_10	36	[16.13, 17.49]	0.30	8.3	-	-	-
5_10_1	160	[11.34, 12.31]	20.73	8.9	[11.20, 12.44]	18.3	6.4
5_20_1	160	[7.691, 8.88]	29.62	8.4	[7.42, 8.85]	21.83	5.3

Table 3.2: Performance of the combination of SMC and SearchMC. Runtime is measured in seconds.

approach is faster than others and it is a trade-off chosen between computational effort and the precision of results in some benchmarks.

Hashing-based model counting techniques like SearchMC rely on prior hypotheses to produce more useful results and start an initial hypothesis from zero knowledge. The initial hypothesis for SearchMC is a uniform distribution over 0 to the maximum bit-width of the output bit-vector. If we gather results from SMC and use them as the initial hypothesis for SearchMC, SearchMC is able to give a desired answer faster. For example, in order to solve coloring_4 the initial hypothesis for SearchMC is a uniform distribution over 0 to 32. If the initial hypothesis for SearchMC is a uniform distribution over 29.61 to 32 which is computed by SMC, SearchMC needs a smaller number of queries to find a desired result.

Table 3.2 shows the performance of the combination of SMC and SearchMC. The results for plain SearchMC are equivalent to the results in Table 3.1. We also measured the average number of iterations (loops) in SearchMC and the results show that the number of iterations was decreased when we used SMC and SearchMC together. Since SMC already computed tight bounds on gettoPath1 and calDate_10, we did not run SearchMC on the benchmarks. This experimental results show that using SMC as a preprocessor of SearchMC gives the performance benefit up to 1.36x speedup.

3.4 Discussion

In this section, we discuss the contributions and the limitations of our current implementation. Our contributions of this work are that we support more operators in SMT-LIB2 format and compute a better precision than FSCB. First, we cover a more complete set of bit-vector operators for both the signed and unsigned orderings of bit vectors. Table 3.3 shows which bit-vector operators are supported by FSCB and SMC. The author of FSCB provided pseudocode of the operation rules in his tech report and we implemented our own SMT-LIB2 version based on the pseudocode. Also, we added more edge cases in operators and fixed some operators to compute a better precision. For example, `bvadd` in SMC computes the low cardinality as `max(f.lc, g.lc)` but we empirically find out that `f.lc+g.lc-1` gives a better result.

FSCB	`bvadd`, `bvsub`, `bvmul`, `bvand`, `bvor`, `bvxor`, `bvshl`, `bvlshr`, `distinct`, =, `ult`, `ugt`
SMC	`bvadd`, `bvsub`, `bvmul`, `bvand`, `bvor`, `bvxor`, `bvshl`, `bvlshr`, `distinct`, =, `ult`, `ugt`, `ule`, `uge`, `slt`, `sgt`, `sle`, `sge`, `bvudiv`, `bvsdiv`, `extract`, `concat`, `sign_extend`

Table 3.3: Bit-vector operators supported by SMC and FSCB

However, some of SMT-LIB2 operators are still not supported in SMC and we are currently working on handling the whole SMT-LIB2 format standard. Also, some operators compute the cardinalities very conservatively due to the limitation of the node representation, hence those operators lead to less precise results (loose bounds). This is for the future work to have more coverage of SMT-LIB2 format standard and design more precise rules.

Lastly, the order of assertions and merging bounds affects precision. For example, let say we have two variable v_1 and v_2 and one constant c. If we have assertions (= v_1 v_2) and (= v_2 c), the results are computed differently depending on which assertion is processed first. For example, (= v_1 v_2) computes the bounds [256, 256] and (= v_2 c) computes the bounds [1, 1] if all the variables are 8-bit. If we merge the two bounds, the final bound would be [256, 256]. But if we flip the order which (= v_2 c) computes the bounds first, then (= v_1 v_2) computes the bounds [1, 1] since we know that v_2 is a constant value this time. This can be resolved by recursively computing the bounds until the bounds do not change but we do not have any proof about its time complexity. This means the bounds can be more precise by merging the

per-assertion bounds in a better order. We believe finding more efficient way to merge bounds will improve the performance and this is for the future work.

3.5 Related Work

Structural approaches to approximate model counting that can provide non-probabilistic bounds (but not guaranteed precision) have seen relatively less tool development. The most direct predecessor of our work in this paper is the FSCB (Fast Solution Count Bounder) algorithm of Martin [69], which as far as we are aware was applied only as part of symbolic execution system to estimate the amount of information revealed by bug reports [64]. Martin's implementation was not available, but we used the description in his technical report as a starting point for the system we developed, as described in more detail in Section 3.2. Other previous structural model-counting systems have been specialized for other domains. Luu et al. [66] build a model counter for string constraints such as arise in symbolic execution of high-level languages like JavaScript; their approach is based on generating functions. Aydin et al.'s MT-ABC [71] uses a combination of structural and automaton-based algorithms for constraints that can include a mix of strings and linear arithmetic. Meng and Smith's two-bit-pattern technique [65] is a hybrid of structural and exact counting approaches to #SAT. It structurally over-approximates the model count by determining, for every pair of bits in a formula, what values they can have in isolation (using many small satisfiability queries). Then the combination of these constraints is a 2CNF formula, which the authors found could be model-counted efficiently in practice (though general 2CNF model counting is still #P-hard).

3.6 Chapter Summary

We propose a structural approximate model counting algorithm, SMC, to compute the lower and upper bound of solutions to a given SMT formula. This is a fast polynomial algorithm compared to other state-of-the-art approximate model counters and it runs in O(n + m) where n is the number of variables and m is the number of assertions. We extend the FSCB algorithm to cover a more complete set of SMT-LIB2 standard operators and to use both the signed and unsigned representations of bit vectors. Our evaluation results illustrate that our technique is most beneficial when time performance is a tight requirement.

Chapter 4

MultiSearchMC: A Scalable Model Counter using a Divide-and-conquer Approach

4.1 Introduction

Hashing-based model counting techniques have been efficient model counting techniques which provide probabilistic guarantees and acceptable precisions. Previous work [11, 14] has shown that they can be more scalable than exact model counting techniques. The hashing-based model counting techniques use SAT or SMT solvers as we add a large number of random XOR constraints to an input formula and count the number of solutions up to a given number. Therefore, a solver for hashing-based model counting techniques needs to be optimized for solving XOR constraints efficiently and enumerating solutions up to a specified number (possibly a huge number). However, solving a formula with a large number of XOR constraints is still a complicated task both theoretically and practically. Also, the length of an XOR constraint becomes larger as the size of an input formula gets larger and this can increase the complexity of solving an XOR constraint.

The main idea of this chapter is to reduce the runtime of solver calls (of model counting) by splitting an input formula into small sub-formulae. We apply *projected model counting*, which determines the number of satisfying assignments of a formula over a subset of variables,

to reduce the number of variables that we count over. The projected model counting problem is #P-complete as the model counting problem [72]. However, projected model counting with a small set of variables becomes less complex practically since a small number of projected variables means a small maximum number of solutions which can be easily enumerated. For example, suppose we have a formula with n variables and we want to compute the number of solutions over n variables. Rather than counting the number of solutions over the total variables, we count the number of solutions over the first half and the second half of n variables separately. Once we have the model counts over two different subsets of variables, we combine the model counts to generate the final bounds conservatively. Since we do not know how satisfying assignments from each subset are connected, we compute the lower and the upper bounds from each model count and the bounds can be very loose. Note that we are not able to compute the exact model count for the final result unless two subsets of variables are totally independent. We apply this idea to hashing-based model counting techniques to reduce the amount of time spent in a solver. We first apply the technique to called *slicing* to find a sub-formula that is totally separable from an input formula. We denote that a sub-formula is *sliceable* if the sub-formula is separable (independent) from an input formula. When we have a sliceable formula, the final bounds become more precise thus we can expect a huge speed-up when we have many sliceable formulae. If a formula is unsiceable, we apply projected model counting to each sub-formulae. As we reduce the size of the input formula, the complexity of solving XOR constraints becomes smaller since the length/number of XOR constraints become smaller. Also, another advantage of this approach is that this can be parallelized since each sub-formula can be computed separately for the final answer.

In this chapter, we introduce MultiSearchMC which is a divide-and-conquer algorithm for hashing-based model counting techniques and this can be parallelized to achieve the performance improvement. Our tool focuses on reducing the total computation time of a decision procedure by dividing an input formula into smaller formulae. We use projected model counting techniques to split an input formula into small formulae and apply a divide-and-conquer approach to hashing-based model counting techniques in order to use fewer and shorter XOR constraints. Fewer and shorter XOR constraints reduce the complexity of simplifying XOR constraints (Gaussian elimination) thus the total computation time of a decision procedure can be decreased. We can also expect this approach to be parallelized for solving each sub-formula by hashing-based model counting techniques. Once all the model counts from the sub-formulae

are estimated, we combine the results to compute the final bounds for the original formula. Section 4.2 explains MultiSearchMC's algorithm which is based on a projected model counting approach to divide an input formula into small pieces. We also explain how we combine each result to calculate the final bounds. Section 4.3 shows the experimental results and we also want to show that there is a huge performance improvement in certain problem domains such that an input formula can be split into many sliceable formulae. Section 4.4 summarizes this chapter.

4.2 Algorithm

In this section, we introduce MultiSearchMC's algorithm which is a divide-and-conquer model counting algorithm for hashing-based model counting techniques. The key idea of MultiSearchMC is to apply a projected model counting approach to reduce the complexity of XOR hashing constraints. The expected length of one XOR hashing constraint is a half of the total number of an original variables and adding numerous XOR hashing constraints to an input formula results in poor performance of SAT solving. Our intuition is that if the total number of variables are reduced, then the (expected) length of XOR hashing constraints becomes shorten and the complexity of an augmented formula is decreased. Therefore, we apply a projected model counting approach to divide the total variables into multiple subsets and run a model counter with each subset of projected variables. The main advantage of this approach is *parallelization* which is to execute each model counter in parallel. Once we compute all the model counts of each run, we need to combine the results to compute the bounds for an original formula. Since combining bounds is a problematic task, we need to combine them conservatively as described in Section 4.2.3. If the final bounds are not precise enough, we use this result as an initial hypothesis for a hashing-based model counting technique to generate more precise bounds.

The implementation overview of MultiSearchMC is described in Figure 4.1. Given a SAT/SMT formula, we first apply *projection/slicing* to the formula in order to split into multiple sub-formulae. We keep the original formula and only divide the total variables. Note that a number of projected variables are determined by n threads used in MultiSearchMC. We explain this in more detail in Section 4.2.1. Next, we run the combination of SMC$_n$ and SearchMC$_n$ (as described in Section 3.3.2) for each sub-formula to compute the bounds for the corresponding subset of projected variables. Note that SMC only takes input as an SMT formula and it is

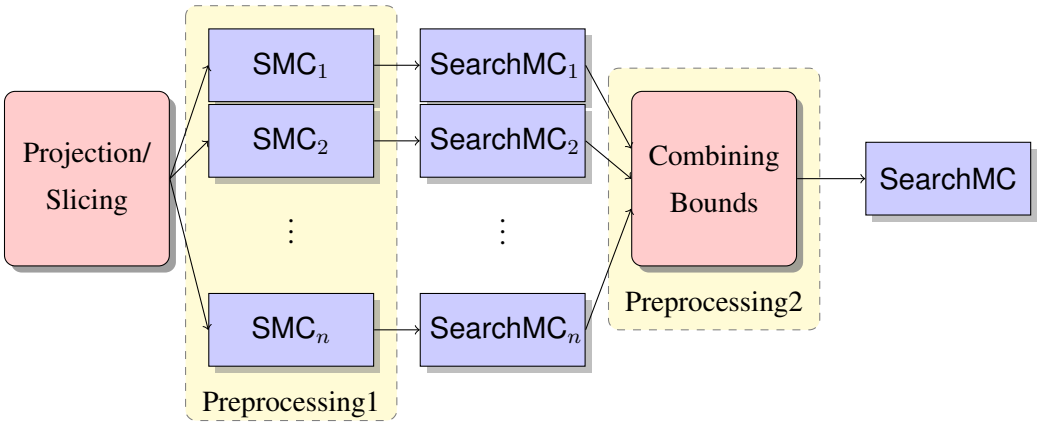


Figure 4.1: The implementation overview of MultiSearchMC

bypassed if an input formula is in CNF. There are two rounds of preprocessing and we denote *preprocessing* as generating a more refined initial distribution for SearchMC. In the *preprocessing1* step, an SMC$_n$'s result may be used as an initial hypothesis of SearchMC$_n$ but we do not need to run SearchMC$_n$ if the SMC$_n$'s result is precise enough. In the *preprocessing2* step, *combining bounds* generates the combined bounds from each result and this result may be used as an initial hypothesis of the final SearchMC. If the *combining bounds*'s result is precise enough, we do not need to run the final SearchMC like *preprocessing1*. We discuss this *preprocessing* in Section 4.2.2. Once we obtain all the results from each SearchMC's run, we combine the results conservatively to generate the first round of lower and upper bounds as described in 4.2.3. The final step is to run SearchMC taking input as an original formula with a refined initial hypothesis.

4.2.1 Projection / Slicing

Projected model counting is to determine the number of satisfying assignments of a formula over a subset of variables. Let F be a Boolean formula, X be a set of variables in F and X_p be a subset of variables in F such that $X_p \subset X$. We denote X_p as *projected variables* and a projected model counter computes the number of unique satisfying assignments for X_p. The projected model counting problem is #P-complete as the model counting problem [72] since the worst case is when $|X_p| = |X|$. However, projected model counting with a small set of variables becomes less complex practically since a small number of projected variables means a

small maximum number of solutions which can be easily enumerated. Projected model counting techniques have been widely used in various areas such as AI, verification, and so on [17, 73].

In this *projection/slicing* procedure, we first simplify an input formula, especially an SMT formula. When a formula contains some variables which are constant, we rather not consider those variables for the total model count since the variables only have 1 solution. If we leave those variables, then this increases the size of XOR constraints and a solver might perform poorly. Therefore, we first check if there is any variable which has a constant value and then we replace the variable with its constant value if there is any.

We denote the *projection* procedure as we divide variables X into n subsets such as X_{p_1}, $X_{p_2}, \cdots, X_{p_n}$ where n is a number of threads. Also, note that any variable should not be overlapped between subsets. Then we generate n sub-formulae differentiated by projected variables and note that the size of each sub-formula is equivalent to the original one since we replicate the original formula. For example, let F be a Boolean formula with $X = \{b_1, b_2, \cdots, b_8\}$ and one way to use 2 threads is to divide X into 2 subsets: $X_{p_1} = \{b_1, \cdots, b_4\}$, $X_{p_2} = \{b_5, \cdots, b_8\}$. The number of projected variables for each sub-formula is determined by the total number of variables divided by a number of threads. If the total number of variables is not divisible by a number of threads, then we can round the number and the last sub-formula takes the rest of variables.

We denote the *slicing* procedure to split a formula when there are some independent variables. This is a special case when some projected variables are independent of other variables and a formula can be actually separated into smaller sub-formulae in this case. From a given variable, we check the *reachability* from this variable to other variables and operators. The reachability procedure starts from one variable and marks any variables and assertions that are associated with the starting variable. If the reachability does not include any new variables or assertions and the marked variables and assertions are not equivalent to an input formula, we detach this independent formula (consists of the marked variables and assertions) from the input formula. The input formula becomes smaller and we repeat this procedure checking until marked variables and assertions are equivalent to this new input formula. Note that this slicing currently supports the SMT-LIB2 format only in our implementation and we believe that this can be applied to the CNF format similar to finding disconnected components in SAT solving.

Figure 4.2 illustrates the basic idea of the slicing with a simple example. Starting from the variable `x`, we search any assertions and variables that affect the satisfiability of `x` and there

```
(declare-fun x () (_ BitVec 8))
(declare-fun y () (_ BitVec 8))
(declare-fun z () (_ BitVec 8))
(assert (bvuge x 4))
(assert (= z (bvadd (bvand y 15) 4)))
```

(a) Original formula

```
(declare-fun x () (_ BitVec 8))
(assert (bvuge x 4))
```

(b) Sub-formula 1

```
(declare-fun y () (_ BitVec 8))
(declare-fun z () (_ BitVec 8))
(assert (= z (bvadd (bvand y 15) 4)))
```

(c) Sub-formula 2

Figure 4.2: Simple SMT-LIB2 formula examples for *slicing*

is only one assertion such that `(= x 4)`. Since there are no other variables or assertions that constrain `x`, we can successfully detach the sub-formula (Figure 4.2b) from the original formula (Figure 4.2a). Next, we search assertions and variables that affect the satisfiability of `y` and one assertion and one variable `z` are found to be related to `y`. Then, since the new variable `z` is included, we search assertions and variables that affect the satisfiability of `z` and repeat this until a new variable is found. Since no new variable is found, we end the slicing procedure and generate the second sub-formula as described in Figure 4.2c.

We say a formula is *sliceable* if the formula contains at least one independent variable determined by the reachability thus the formula can be sliced into multiple sub-formulae as we described above. Note that a formula is *unsliceable* when the formula cannot be sliced since we do not know whether there is any independent variable in the formula. Figure 4.3 illustrates an unsliceable formula where the formula is unsliceable even though `x` and `y` are independent. The assertion $\mathtt{x} + \mathtt{y} \geq 0$ is always true since `x` and `y` are unsigned 8-bit variables based on the

```
(declare-fun x () (_ BitVec 8))
(declare-fun y () (_ BitVec 8))
(assert (bvuge (bvadd x y) 0))
```

Figure 4.3: An unsliceable formula example

unsigned-greater-than-or-equal-to operator `bvuge` ($\texttt{x} \geq 0$ and $\texttt{y} \geq 0$). Thus we can say that `x` and `y` are independent since either one variable does not restrict another variable. However, they become dependent if the assertion were $\texttt{x} + \texttt{y} \geq c$ where $0 < c < 2^8$. The reachability does not check the semantic of the formula therefore we say that the formula is unsliceable. Checking the semantic of a formula efficiently is a complicated task and we leave this for future work.

In the *projection* / *slicing* procedure, we first check whether an input formula is sliceable and we apply the projection to a sub-formula if its size is still large. Also, we can directly apply projection to the input formula based on a given number of threads when the formula is unsliceable. The main goal of this procedure is to generate multiple sub-formulae in order to take the advantage of parallelization.

There is another procedure called *grouping* after this *slicing* procedure which groups *isomorphic* formulae such that two formulae are semantically equivalent except the names of variables and have the same number of solutions. Given two formulae, we first check whether the lengths of two formulae are equal and then replace the names of variables appeared in the formula to temporary variables in order. For example, if we read one formula and found the first variable's declaration, then we replace this variable's name to *temp1* wherever it is used including the declaration. After replacing all the variables, we can check whether two formulae are identical. We sort all the sliced formulae based on their text in alphabetical order and *group* them by checking whether two adjacent formulae are isomorphic formulae or not. The reason why we find isomorphic formulae is that we do not need to compute its model count again if we already have the model count of the same formula.

4.2.2 Preprocessing

The *preprocessing* computes a prior hypothesis to produce more useful results for SearchMC. The initial hypothesis for SearchMC is a uniform distribution over the interval from 0 to the maximum bit-width of the output bitvector which indicates zero knowledge about the model count. In Chapter 3, we discussed that a refined initial hypothesis helps SearchMC to find an estimate of the model count faster. Since SMC computes a lower and upper bounds without any solver calls, we use SMC to generate a refined initial hypothesis in polynomial time.

In the *preprocessing1* procedure, we feed sub-formulae, generated from the projection/slicing procedure, to SMC and the main advantage of this procedure is parallelization. SMC computes the firm bounds, which are 100% correct, in polynomial time and SearchMC uses the bounds as a refined initial hypothesis. The performance can be maximized when an input formula is sliceable and SMC is able to compute the exact model counts for some sub-formulae. SMC performs well on finding the exact model counts for well-structured formulae such as unconstrained formulae or formulae with many constant values. If the results of SMC are precise enough, we do not need to run SearchMC. Also, note that SMC can be bypassed if an input formula is in CNF since SMC only supports an SMT-LIB2. We can also expect one more round of *preprocessing* after *combining results*. If the result from *combining bounds* is not precise as desired, we can feed the result as a refined hypothesis to the final run of SearchMC. In the *preprocessing2* procedure, we generate a refined initial hypothesis from combining all the results from the sub-formulae. Unlike the preprocessing1 procedure, the *preprocessing2* procedure produces the probabilistic bounds. The probability for the bounds might be low but we consider them as a uniform distribution since SearchMC is able to correct the probability distribution over iteration. Another reasonable approach would be using a different distribution other than a uniform distribution and we leave this as future work. If the combined bounds are precise enough, the final SearchMC can be bypassed as well. We discuss how we combine the results from the sub-formulae in the next section.

4.2.3 Combining Bounds

MultiSearchMC is a divide-and-conquer style approach where small pieces of results are gathered and combined into one final result. In order to combine small results, we need to calculate the final bounds conservatively. We can think of two different cases where a formula is either

sliceable or unsliceable when we combine the results. Recall that a formula is sliceable if we can detach a sub-formula from an original formula.

Lemma 4.2.1. *Let F be a sliceable formula which can be sliced into F_1 and F_2 by the slicing procedure. Let a_l and a_u be a lower and upper bounds of the model count of F_1 with a probability of at least p_1, respectively, and let b_l and b_u be a lower and upper bounds of the model count of F_2 with a probability of at least p_2, respectively. Then, the lower bound of F is $a_l \cdot b_l$ and the upper bound of F is $a_u \cdot b_u$ with a probability of at least $p_1 \cdot p_2$.*

Proof. Let us denote that F is a sliceable formula when F can be split into two independent formulae F_1 and F_2 (no assignment in F_1 is determined by any assignment in F_2 and vice versa) by the slicing procedure. For simplicity, we assume that F has N variables such that $X = \{x_1, x_2, \cdots, x_N\}$. Also, we assume that F_1 has m variables such that $X_1 = \{x_1, x_2, \cdots, x_m\}$ and F_2 has n variables such that $X_2 = \{x_{m+1}, x_{m+2}, \cdots, x_N\}$ where $n = N - m$. Since they are independent, one satisfying assignment of F_1 with any satisfying assignment of F_2 is still satisfiable in F. Therefore, the number of cases for satisfying assignments in F is the multiplication of the model count of F_1 and the model count of F_2. If the model counts of F_1 and F_2 are approximate, the multiplication rule applies to bounds since the approximate model counts are non-negative. The lower bound of F is the multiplication of the lower bound of F_1 and the lower bound of F_2 and the upper bound of F is the multiplication of the upper bound of F_1 and the upper bound of F_2. Also, the probability of the combined result is the multiplication of each probability because of the multiplication rule of independent event such that $P(A \cap B) = P(A) \cdot P(B)$. □

Lemma 4.2.2. *Let F be a unsliceable formula, determined by the slicing procedure, with N variables such that $X = \{x_1, x_2, \cdots, x_N\}$. Let $X_1 = \{x_1, x_2, \cdots x_m\}$ and $X_2 = \{x_{m+1}, x_{m+2}, \cdots, x_N\}$ be two non-overlapped subsets of variables from X such that $n + m = N$. Let a_l and a_u be a lower and upper bounds of the model count of F over X_1 with a probability of at least p_1, respectively, and let b_l and b_u be a lower and upper bounds of the model count of F over X_2 with a probability of at least p_2, respectively. Then, the lower bound of F is $\texttt{max}(a_l, b_l)$ and the upper bound of F is $a_u \cdot b_u$ with a probability of at least $p_1 \cdot p_2$.*

Proof. Let us denote that F is a unsliceable formula when F cannot be split into two independent formulae F_1 and F_2 (no assignment in F_1 is determined by any assignment in F_2 and vice

versa) by the slicing procedure. For simplicity, we assume that F has N variables such that $\{x_1, x_2, \cdots x_N\}$. Also, we assume that F_1 has m variables such that $\{x_1, x_2, \cdots x_m\}$ and F_2 has $n(= N - m)$ variables such that $\{x_{m+1}, x_{m+2}, \cdots x_N\}$. Since F is unsliceable, then we cannot determine X_1 and X_2 are independent or not. Thus, one satisfying assignment of F_1 with *some* satisfying assignment of F_2 is satisfiable in F. If we assume they are independent, the upper bound of the model count of F is the multiplication of $a_u \cdot b_u$. If they are dependent, the upper bound of the model count is less than $a_u \cdot b_u$. Therefore, we can assure that the model count of F is less than or equal to $a_u \cdot b_u$. For the lower bound, we consider the case where X_1 and X_2 are dependent. It is straightforward that the model count of F has to be greater than equal to both the model counts of F over X_1 and X_2 since all the satisfying assignment of F_1 has to pair with at least one satisfying assignment of F_2. Also, the probability of the combined result is the multiplication of each probability because it is the minimum probability of two events. □

We can think of this problem as a mapping two sets F_1 and F_2 and computing the minimum and the maximum mappings. Suppose each set has either an exact number of elements or an approximate number of element (lower/upper bound) and *combining results* follows the combining rules to compute a lower/upper bound conservatively as described in Table 4.1. If the model counts of F_1 or F_2 are exact, then we consider the value as $a_l = a_u = a$ or $b_l = b_u = b$, respectively. Note that if the model count of a sub-formula is exact, then the probability of the model count is 1. The result generated from *combining results* might be sufficient enough when an input formula can be divided into independent formulae. If they are not independent, the precision could be extremely poor so we can run another round of SearchMC using the result as an initial hypothesis.

One of main challenges for *combining results* is the decrement of combined probability. We want to address the problem where the probability of all the independent events is the multiplication of each event's probability. For example, if we slice an input formula into 10 independent formulae and the probability of one computed (approximate) model count is at least 0.6, then the combined probability would be true at least $0.6^{10} = 0.006$. There is a huge gap between the proven probability and practical probability because some over-estimations and under-estimations would be cancelled out. Therefore, we need to provide better ways of computing probabilities. One possible approach is finding many conservative results efficiently. If the exact model counts of some formulae can be computed very quickly, we can increase the

	$MC(F_1)$	$MC(F_2)$	$MC(F)$
sliceable	a	b	$a \cdot b$
	$[a_l \quad a_u]$	b	$[a_l \cdot b \quad a_u \cdot b]$
	a	$[b_l \quad b_u]$	$[a \cdot b_l \quad a \cdot b_u]$
	$[a_l \quad a_u]$	$[b_l \quad b_u]$	$[a_l \cdot b_l \quad a_u \cdot b_u]$
unsliceable	a	b	$[\texttt{max}(a, b) \quad a \cdot b]$
	$[a_l \quad a_u]$	b	$[\texttt{max}(a_l, b) \quad a_u \cdot b]$
	a	$[b_l \quad b_u]$	$[\texttt{max}(a, b_l) \quad a \cdot b_u]$
	$[a_l \quad a_u]$	$[b_l \quad b_u]$	$[\texttt{max}(a_l, b_l) \quad a_u \cdot b_u]$

Table 4.1: Rules for combining two bounds. A model count can be either an exact value or an interval. A square bracket and a parenthesis represent a closed interval and an open interval, respectively.

refined probability. For example, if the model count is a small number or the formula is unconstrained, it is likely that we can compute the model count exactly by SMC with probability of 100%. This does not lower the final probability after the multiplication. Practically, if we apply our techniques to word-level problems such as image processing or document benchmarks, many pixel or character data would be easy to analyze; each data representation have same or similar constraints. Computing more applicable probability of *combining results* is for future work.

4.3 Evaluation

In this section, we present our experimental results of MultiSearchMC. All our experiments were performed on a machine with an Intel Core i7 3.40Ghz CPU and 16GB memory. Our main algorithm is implemented with a Perl script and SMC and SearchMC are called by the main script. Our algorithm can be applied to both SMT formulas and CNF formulas.

We first want to compare the performances among SearchMC, MultiSearchMC and ApproxMC4 [23] and use CryptoMiniSat5 as the back-end solver with all the tools for fair comparison. For the parameters for the tools, we measured the running time to reach a 60% confidence level which corresponds to the first round of ApproxMC4 and generates provable sound

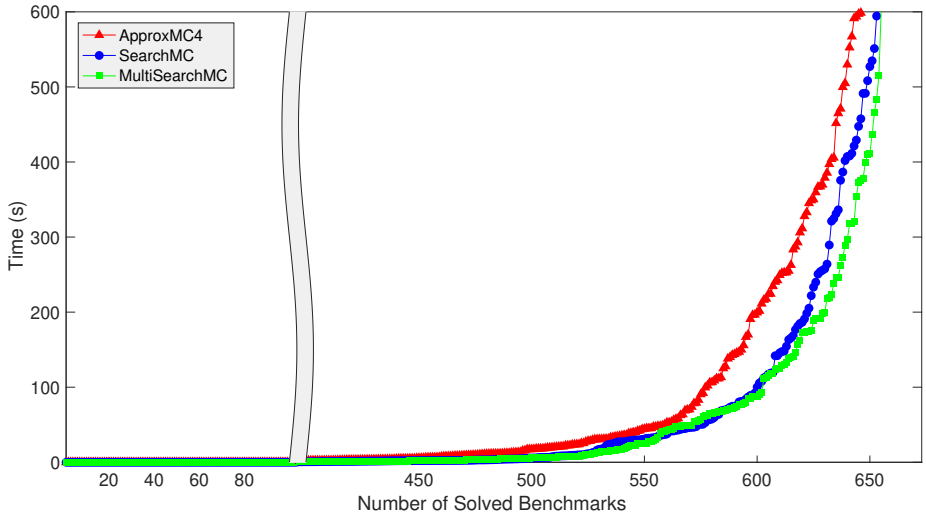


Figure 4.4: Performance comparison of SearchMC, MultiSearchMC and ApproxMC4

bounds for SearchMC and MultiSearchMC. We also set a desired interval length of 1.7 which is equivalent to $\epsilon = 0.8$ for ApproxMC4 and $thres = 1.7$ for SearchMC and MultiSearchMC. We ran our algorithm with the whole benchmarks from [23] representing a wide range of application areas including probabilistic reasoning, plan recognition, Bayesian networks, combinatorial circuits, quantified information flow, program synthesis, functional synthesis, logistics, and so on. For MultiSearchMC, we use 4 threads to run SearchMC with 2 iterations in parallel to generate a refined hypothesis and then run a single-threaded SearchMC to compute the final bounds.

Figure 4.4 shows the performance comparison over a subset of benchmarks. The experimental results show the number of benchmarks was solved in a given amount of time. MultiSearchMC, SearchMC and ApproxMC4 were able to solve 654, 653, and 646 benchmarks within 600 seconds, respectively. 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. Although, some benchmarks were solved faster with a single-threaded ApproxMC4 or SearchMC than MultiSearchMC, the figure demonstrates the speed up achieved through MultiSearchMC mode.

Table 4.2 shows more drastic performance results. We built a program, called pwdstr, to

			$\log_2$	ApproxMC4		MultiSearchMC	
Benchmark	#Vars	#Cls	(MC)	Bounds	Time (s)	Bounds	Time (s)
pwdstr_8	9577	22319	39.10	[38.33, 39.93]	0.57	39.10	0.15
pwdstr_16	19153	44639	76.52	[75.81, 77.41]	70.01	76.52	0.49
pwdstr_24	28601	66599	114.4	[113.6, 115.2]	483.95	114.4	0.82
pwdstr_28	33341	77624	130.8	[130.1, 131.7]	2615.7	130.8	0.87
pwdstr_32	38081	88649	151.3	n/a	-	151.3	1.38
pwdstr_64	75953	176714	294.9	n/a	-	294.9	5.79
pwdstr_128	152001	353699	595.4	n/a	-	595.4	23.19
pwdstr_256	303985	707354	1191.4	n/a	-	1191.4	94.05

Table 4.2: Performance comparison of ApproxMC4 and MultiSearchMC for a text processing program. '-' indicates timeout after 7200 seconds.

check strength of password in C code, which was inspired by [74]. Given a password, the program checks the following conditions: it contains at least one lowercase, one uppercase, one digit, and one special character and its length is at least 8. This program checks each character from an input password with the conditions and turns on a corresponding flag if a condition is satisfied. If the password satisfies all the five conditions, then the program says the password is strong. If not, it says the password is weak. This is related to quantitative information-flow analysis such as we want to measure how many inputs can be considered as *a strong password* in a given length. We generated random strings from the length of 8 to 256 (which is shown next to pwdstr in the first column) and fed these strings to pwdstr to measure the number of passwords which satisfied the identical sequence of conditions as the input string. We used FuzzBall [75], a symbolic execution tool, to generate the path conditions (an SMT formula) following the same execution for a given string and treating an input string symbolic. Each character (an 8-bit variable) is independent of others and has 26 solutions for one lowercase or uppercase, 10 solutions for one digit and 32 solutions for one special character. We used STP [50] to convert the SMT formula into the corresponding CNF formula with its projected variables. The second and the third column of the table represents the total number of variables and clauses, respectively, from this conversion. We ran each benchmark with ApproxMC for a 60% confidence level but ApproxMC was not able to compute the estimate from a password with 32 characters in length.

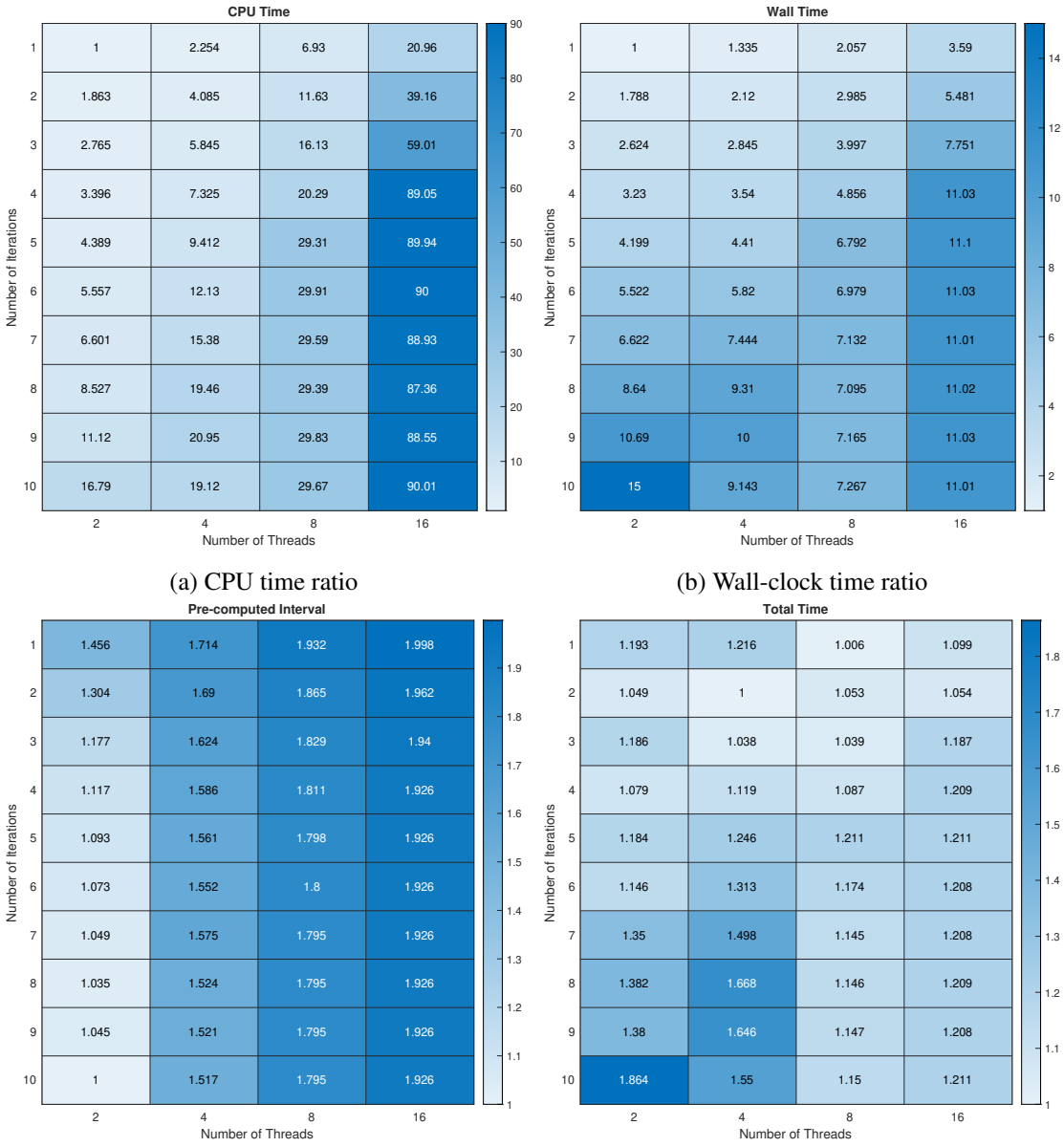
	ApproxMC4	SearchMC		MultiSearchMC	
	CMS5	CMS5	CMS2	CMS5	CMS2
Benchmark	Time (s)	Time (s)	Time (s)	Time (s)	Time (s)
or-70-5-1-UC-20	**1.05**	6.98	1.57	5.81	1.66
or-100-5-6-UC-20	967.51	-	300.14	-	**243.47**
tire-3	**56.49**	-	748.74	-	138.409
blasted_case107	64.71	-	**5.12**	-	23.825
min-8	15.16	4.14	-	**2.56**	-
min-16s	10.05	**3.27**	-	3.19	-
prod-4	2.52	14.28	**1.13**	7.99	86.46
s15850a_15_7	130.80	**17.78**	-	31.93	-
sort.sk_8_52	5.99	75.72	**17.39**	132.23	30.86
parity.sk_11_11	917.70	**470.69**	657.27	759.00	1802.08
04B-2	**265.35**	-	399.10	-	407.81
karatsuba.sk_7_41	-	**1755.57**	-	-	-
signedAvg.sk_8_1020	**481.58**	775.84	-	753.45	-
prod-28	**631.12**	756.65	-	633.77	-
leader_sync4_11	17.11	14.16	-	**9.20**	-
leader_sync6_64	28.99	**12.65**	-	24.04	-

Table 4.3: Performance comparison of hashing-based model counters using different versions of CryptoMiniSat. '-' indicates timeout after 2000 seconds.

For MultiSearchMC, *slicing* was applied to each character since they were all independent. For example, pwdstr_8 was sliced into 8 SMT sub-formulae successfully. Then, we converted each SMT formula into the CNF formula for MultiSearchMC experiments. SearchMC was able to compute the exact model count for each sliced sub-formula with SMC and 5 iterations of SearchMC. Since we were able to compute the exact model counts of independent sub-formulae, we did not need to run the final round of SearchMC. In this experiment, we wanted to show the performance of MultiSearchMC in more extreme cases. Therefore, we computed the exact model count for each sliced sub-formula and MultiSearchMC handled the large-sized problems well when a given formula was sliceable.

We experienced that the optimization of a decision procedure highly affected the running time in the same algorithm. Table 4.3 shows that the running-time comparison based on the different version of CryptoMiniSat. Note that some benchmarks in Table 4.3 are not presented in Figure 4.4 due to the timeout. We set up the parameters and a number of threads as same as previous experiment. We only differentiate the back-end solver. We highlighted the best performances for each benchmark and it is clear that there is no algorithm which is always dominant to others. This is caused by different optimization techniques from CryptoMiniSat2 and CryptoMiniSat5. In order to achieve the best performance result, we need to first select the most suitable decision procedure for a given benchmark. We can resolve this issue by running a portfolio-style parallel approach so we can select the result whichever computed the first. We leave this as future work and we refer a detailed discussion in Section 6.4.

For MultiSearchMC, there are some factors that affect the performance other than a decision procedure. In the preprocessing procedure, we have two parameters: a number of threads and a number of iterations. The number of threads represents some facts such as how many threads you use and how small a sliced formula is. If you increase the number of threads, the sliced formulae become less expensive but the combined result gives less precision. Also, you can control the number of iterations for each SearchMC's run. As you iterate more, you can get more precise answers but you do not get fruitful results if you iterate too many times. Figure 4.5 shows the heatmaps of ratio measurements based on the different settings of a number of threads and a number of SearchMC iterations when you have 8 threads at maximum. We selected 10 random benchmarks and measure the average-times spent for preprocessing. For (a), (b), and (d), the shortest running time is 1 and we compute ratio compared to the shortest running time. For (a) and (b), we measure the preprocessing time and it is clear that you increase the running time as you increase the number of threads and the number of iterations. In (c), it shows the computed bound interval (upper bound minus lower bound) for preprocessing and the shortest interval is represented as 1 and we also compute ratio compared to others. As you can see, using 2 threads and 10 iterations gives the most tight bounds but we know this takes a lot of time from other experiments. (d) shows the comprehensive results such that we measure the total running time to achieve a 60% confidence level based on different number of threads and iterations. This graph shows using 4 threads with 2 iterations gives the best result. However, we believe that the best parameter setting can differ from various factors such as the type of benchmarks, the maximum number of cores, etc.



(a) CPU time ratio

(b) Wall-clock time ratio

(c) Interval ratio

(d) Total running time ratio

Figure 4.5: MultiSearchMC's ratio measurements parameterized by a number of threads and iterations for each SearchMC execution. Smaller ratios represent faster results for the time measurements and more precise results for the interval measurements.

4.4 Chapter Summary

We introduced MultiSearchMC to improve the scalability of hashing-based model counting techniques and compared the performances with ApproxMC4 in this chapter. The performance of hashing-based model counting techniques is highly dependent on the performance of a decision procedure (SAT or SMT solver) and adding numerous hashing constraints to a formula might cause a solver to perform poorly. The key idea of improving the scalability for MultiSearchMC is to apply a divide-and-conquer approach that can be parallelized. This is achieved by dividing an input formula into smaller pieces, feeding them into SearchMC algorithm in parallel and combining their results. Once we obtain all the results from each SearchMC's run, we combine the results conservatively to generate a final lower and upper bounds. We showed the scalability for MultiSearchMC compared to other existing approaches and our experimental result showed MultiSearchMC performed well in large-sized programs.

Chapter 5

Applications of Model Counting to Software

This chapter shows the applications of model counting techniques to software for program analysis and testing. We first explain quantitative information flow analysis using model counting techniques and show how we generate path conditions with a symbolic execution tool to measure potential information leakage. From the path conditions, we compute the model count of input/output data to quantify the information leakage. The experimental results show that our approach scales to analyze large/complex programs.

We also show uniform sampling using model counting techniques. Uniform sampling are fundamental problems in computer science with a wide range of applications ranging from constrained random simulation, probabilistic inference to network reliability and beyond. Recently, random hashing-based techniques have been proposed and UniGen [22] is a near-uniform sampler which provides the scalability and strong theoretical guarantees of the uniformity as well. UniGen uses ApproxMC to estimate the model count and compute the most appropriate number of XOR constraints. Since MultiSearchMC outperforms ApproxMC, we replace the model counting algorithm of UniGen to MultiSearchMC. The technical and mathematical details are based on Unigen3 [23], a state-of-the-art hashing-based sampler, and we compare our approach with Unigen3 and show that our tool outperforms Unigen3.

5.1 Quantitative Information Flow using Model Counting

Quantitative information flow (QIF) analysis is a powerful approach to measure the amount of secret information revealed to the public outputs by a program or a function. QIF can be applied to measure the information flow over variables of the program. An unexpected large (or small) flow of information can be interpreted as a potential leakage. Practically, the maximum flow which can be revealed from a program (known as *channel capacity*) is $\log_2$ of number of possible output values. Newsome *et al.* [34] proposed the terminology of "influence" for a specific application of model counting in quantitative information-flow measurement. Based on this channel capacity, influence 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.

Recently, QIF approaches are based on program analysis techniques such as symbolic execution or model checking to generate a SAT/SMT formula representing the program's behavior and apply model counting to measure the information flow. While program analysis scales to large and complex programs, the formulae generated by a symbolic execution or model checking tool are too complex for exact model counting. Therefore, approximate model counting techniques are often used for analyzing a large/complex program or handling a large size of inputs. In this chapter, we show that MultiSearchMC is able to provide a large performance increase for a very small loss of precision, allowing the analysis of SAT/SMT formulas produced from complex programs. Our approach is based on the combination of model counting and symbolic execution such that we obtain a execution path which might leak some private data from symbolic execution tools and measure the leakage using model counting techniques.

Here we are interested in two basic scenarios of QIF analysis: input and output measurements. The first scenario is to measure the amount of inputs which cause a specific behaviour of a program. For example, when a program crashes, you might want to send an error report to developers to fix bugs. This raises privacy concerns such that error reports may contain sensitive user data. One solution to not reveal any private input data is that you can send an execution path that leads to a crash. However, this is still in danger to reveal some of input data if the number of possible inputs that cause a crash is very small. The worst case is that you can reveal the exact original input by observing a reported execution path if only one input reaches a crash. By measuring the amount of information in the input, we can tell how much information can be

leaked in an error report and you can decide whether to submit this report or not.

Secondly, if there is a program that receives some private input and produces some public output, we want to measure the information leakage by counting the number of outputs. The basic idea is that an adversary might be able to guess something about the input by observing this output. We would like to quantify the amount of information in the output to measure the information leaked as the number of possible outputs. For example, you want to apply an image processing technique such as pixelation, blurring or swirling to hide your identity from a picture. Some image processing functions could hide your identity from an image by looking at it with your eyes but the output image still preserved a quite amount of the original image data so the original image was retrievable as much as you could be identified. We can measure image processing functions how much information it preserves after conversion and tell which function is more secure to protect your identity.

5.1.1 A Symbolic Execution Tool with MultiSearchMC

Symbolic execution is a program analysis technique to explore all the paths generated by a program and observe interesting behaviours of the program. We can find out which input values cause a program to execute a certain path or reach a undesirable state such as NULL pointer dereference, division by zero, etc. We can reach a undesirable state by testing a program on a concrete input value (without symbolic value) and repeat this until a violation is found. However, testing only explores a single path and exposes a single input value which leads to the undesirable state. The main concept of symbolic execution is to execute a program on symbolic input values rather than concrete values, which means that single execution path is explored with multiple inputs. Therefore, we can find out all the input values that cause a specific execution path.

Symbolic execution maintains each explored control flow path which contains a path condition (satisfied by the branches taken along that path) and a symbolic memory store that maps variables to symbolic expressions or values. Branch execution keeps adding conjuncts to the formula, while assignments update the symbolic store. An SMT solver is used to verify whether there are any violations of the property along each explored path and if the path itself is realizable, i.e., if its formula can be satisfied by some assignment of concrete values to the symbolic arguments. Suppose we have a simple example shown as Figure 5.1. Here we want to determine whether `x` is equal to `y` and this program fails if `x` is equal to `y`. We take `a` and `b` as symbolic

```
1: void foo (int a , int b) {
2:     int x = 4, y = 0;
3:     if (a > 0) {
4:         y = 2 + x;
5:         if (b > 0)
6:             x = 2*(a + b);
7:     }
8:     assert (x != y);
9: }
```

Figure 5.1: Simple symbolic execution example

inputs which can take 2^{32} distinct integer values for a single variable. There exists 3 possible condition paths: $\neg$`(a > 0)`, `(a > 0)` $\wedge \neg$`(b > 0)` and `(a > 0)` $\wedge$ `(b > 0)`. First, if $\neg$`(a > 0)` is true, then `x = 4` and `y = 0`. Therefore, the program does not fail. Second, if `(a > 0)` $\wedge \neg$`(b > 0)` is true, then the program executes `y = 2 + x`. This makes `x = 4` and `y = 6` and the program also does not fail. Lastly, if `(a > 0)` $\wedge$ `(b > 0)` is true, then the program executes `y = 2 + x` and `x = 2*(a + b)`. Since `y = 6`, the program would fail if `(b > 0)` $\wedge$ `(a > 0)` $\wedge$ `(a + b = 3)`. An SMT solver determines whether there is a satisfying assignment for `a` and `b` and gives a satisfying assignment such as `a = 2, b = 1` or `a = 1, b = 2`.

FuzzBALL [76] is an open-source symbolic execution tool [75] which determines what inputs cause certain behaviors of a program. FuzzBALL executes the program (binary executable) and generates path conditions by replacing concrete values by symbolic variables. The STP[50] decision procedure finds satisfying assignments to those symbolic variables whenever any branch is found and explores feasible execution paths. We use FuzzBALL to generate the SMT file of the execution path in a certain program and we run MultiSearchMC with the SMT file over the input/output variables to measure how much information is leaked.

5.2 Uniform Sampling using Model Counting

As the model counting problem has been widely studied, uniform sampling are fundamental problems in computer science with a wide range of applications ranging from constrained random simulation, probabilistic inference to network reliability and beyond. The past few years have witnessed the rise of hashing-based approaches that use XOR-based hashing and employ SAT solvers to solve the resulting CNF formulas conjuncted with XOR constraints. Since most of the runtime of hashing-based techniques is spent inside the SAT queries, improving CNF-XOR solvers has emerged as a key challenge. As we discussed in Chapter 4, improving the scalability of hashing-based model counting techniques lead to solving large-sized and realistic problems.

5.2.1 A Uniform Sampler with MultiSearchMC

Constraint random sampling has been emerging in industrial problems such as functional verification of digital systems. Recently, random hashing-based techniques have been proposed and UniGen [22] is a near-uniform sampler which provides the scalability and strong theoretical guarantees of the uniformity as well. The idea of UniGen is to divide the solution space of an input formula into small pieces using random hashing functions and select one piece randomly. If this piece has the acceptable number of solutions based on input parameters, then select one random solution from this piece and return this solution as a sample. Unigen2 [77] was introduced to show the performance improvement compared to UniGen. The key difference is that Unigen2 generates multiple samples in a single piece and preserves strong guarantees of the uniformity with generated samples. This feature allows Unigen2 to outperform UniGen by approximately 10 times faster. Unigen3 [23] improved the performance of the solver and reduced the running time by reusing the results of SAT queries. We refer the reader to [77] for the proofs of the guarantees of the uniformity.

Here we explain the two stages of Unigen3. The first stage (Algorithm 5.1) performs a preprocessing step for a given F and ϵ to compute parameters before the sampling process. These parameters determine the scalability and the theoretical guarantees of the uniformity. 'loThresh' and 'hiThresh' represent the acceptable number of solutions in a single piece that is divided by the XOR-based hashing function. κ is the tolerance parameter, computed from ϵ, to determine a low and high thresholds for the size of each piece. 'pivot' is the ideal size of each

Algorithm 5.1: `EstimateParameters`

Input : F, ϵ

Output: k, loThresh, hiThresh

Compute κ such that $\epsilon = (1+\kappa)(7.44 + \frac{0.392}{(1-\kappa)^2}) - 1$

pivot $\leftarrow 4.03(1+\frac{1}{\kappa})^2$

hiThresh $\leftarrow 1 + \sqrt{2}(1+\kappa)$pivot

loThresh $\leftarrow \frac{1}{\sqrt{2}(1+\kappa)}$pivot

count $\leftarrow$ `ApproxMC` $(F, 0.8, 0.8)$

if count == $\perp$ **then**

  return $\emptyset$

return $\log \frac{1.8\text{count}}{\text{pivot}}$, loThresh, hiThresh

Algorithm 5.2: `GenerateSamples`

Input : F, k, loThresh, hiThresh

Output: loThresh number of samples

`/* k, loThresh, hiThresh are returned values from EstimateParameters */`

foreach $i \in \{k-2, k-1, k\}$ **do**

  Sols $\leftarrow$ `MBoundExhaustUptoC`(F, i, hiThresh)

  if loThresh $\leq$ |Sols| $\leq$ hiThresh **then**

    return Pick loThresh distinct elements of Sols randomly

return $\emptyset$

piece and loThresh and hiThresh are computed by κ and pivot. k is the number of XOR hashing constraints to represents the most appropriate value to generate samples and is computed by a hashing-based model counter such as ApproxMC.

Algorithm 5.2 illustrates the second stage of Unigen3 where the sample generation is performed. Given k computed from Algorithm 5.1, `MBoundExhaustUptoC` is executed with $k-2$, $k-1$ and k XOR hashing constraints in order to find an applicable piece where its solutions are greater than or equal to loThresh and less than or equal to hiThresh. If the chosen

piece meets the condition, it returns loThresh number of samples by selecting loThresh elements randomly from the piece. Unigen3 provides a theoretical analysis of the uniformity and defer all proofs to the original work [77]. Unigen3 has shown that the failure probabilities of `EstimateParameters` and `GenerateSamples`.

Theorem 5.2.1. *`EstimateParameters` and `GenerateSamples` return $\emptyset$ with probabilities at most 0.009 and 0.38 respectively.*

This shows that ApproxMC returns the estimate with a probability at least 0.991 and one invocation of `GenerateSamples` generates loThresh number of samples with a probability at least 0.62. Also, Unigen3 has shown that a single invocation of `GenerateSamples` provides guarantees nearly as strong as those of an almost-uniform generator.

Theorem 5.2.2. *For given F, and ϵ, let L be the set of samples generated using Unigen3 with a single call to `GenerateSamples`. Then for each sample y in F, we have*

$$\frac{\text{loThresh}}{(1+\epsilon)MC(F)} \leq Pr[y \in L] \leq 1.02 \cdot (1+\epsilon)\frac{\text{loThresh}}{MC(F)} \tag{5.1}$$

For example, when ϵ is 16 and loThresh is 11, the probability of generated sample is greater than or equal to $(11/17)p \approx 0.647p$ and less than or equal to $1.02 \cdot 17 \cdot 11 \cdot p = 190.74p$ where $p = 1/MC(F)$, which is the ideal probability. Decreasing ϵ gains more uniformity but loses scalability since loThresh also decreases and `GenerateSamples` generates less number of samples per iteration. As we discussed in Chapter 4, we showed that MultiSearchMC is more scalable than ApproxMC, thus we replace ApproxMC with MultiSearchMC for our experiments. We also optimize this sampling process where if an input formula is independently sliceable, we generate samples from each slice and combine them. This optimization shows the performance improvement significantly in some problem domains.

5.3 Evaluation

Here we present our experimental results. We provide our experiments on a machine with an Intel Core i7 3.40Ghz CPU and 16GB memory. We show QIF case studies using MultiSearchMC where it can scale up to large-sized input/output data. We also show uniform random sampling using MultiSearchMC to speed up the model counting process and generate samples based on the model count. In some problem domains, our approach generates random samples significantly faster than Unigen3.

5.3.1 QIF Case Study: Error Report System

This experiment is inspired by [64] where generating a bug report with path conditions might leak some private information. If the amount of inputs (which caused the bug) is too small, then this is still in danger to reveal some of input data. We assume that a normal execution path has the same complexity as a buggy one so that we explore some normal execution paths and measure how many inputs cause these execution paths. We used FuzzBall [75], a symbolic execution tool, to generate the path conditions (an SMT formula) following the same execution for a given string and treating an input string symbolic.

In Section 4.3, we used pwdstr to show the scalability of MultiSearchMC. Given a password, pwdstr is a password strength checker program that checks the following four conditions: it contains at least one lowercase, one uppercase, one digit, and one special character. Table 5.1 shows that MultiSearchMC performs differently based on the setting of SearchMC iterations for preprocessing. Note that our tool was able to slice the input formula and group sub-formulae into the equivalent formula in the *slicing* procedure. We differentiated the number of SearchMC iterations in the *preprocessing* procedure and measured the time performances and the computed bounds. 'Prob' represents the proven probability based on Lemma 2.2.1. Note that we ran MultiSearchMC with 5 SearchMC iterations and SearchMC gave the exact model count of each sub-formula shown as Table 4.2 in Section 4.3. Therefore, it computed the exact model count of the input formula which guarantees a 100% correctness. When we ran MultiSearchMC with 4 SearchMC iterations, SearchMC was able to compute the exact model counts for a character containing one digit, one lowercase or one uppercase. Since the model count for a character containing a special character was estimated, the proven correctness of the final bounds is at least 60%. SearchMC with 3 pre-iterations gave none of the exact model counts. Since pwdstr_8 contained three types of character, the correctness of the final bounds is at least $0.6^3 \approx 0.22$. From pwdstr_16, the input string contained three types of characters, the correctness of the final bounds is at least $0.6^4 \approx 0.13$. The computed bounds become loosen as the number of SearchMC iterations gets decreased and it shows there is a huge gap between the proven probability and the actual probability of the bounds . The experiments showed that using more *preprocessing* time might help the precision and the correctness.

Table 5.2 shows more realistic text programs for QIF analysis. The input file was a sample text file generated by a test data generator[78] which is 930 Bytes with 17 lines. head is a Linux command-line program which mainly prints out a given number of lines from the beginning of

	MultiSearchMC with 4 SearchMC iterations			MultiSearchMC with 3 SearchMC iterations		
Benchmark	Bounds	Time (s)	Prob	Bounds	Time (s)	Prob
pwdstr_8	[24.1, 54.1]	0.14	0.6	[16.8, 63.2]	0.13	0.22
pwdstr_16	[49.5, 103.5]	0.37	0.6	[34.4, 121.5]	0.36	0.13
pwdstr_24	[84.4, 144.4]	0.79	0.6	[48.2, 183.0]	0.83	0.13
pwdstr_32	[115.3, 187.3]	1.32	0.6	[64.7, 240.9]	1.27	0.13
pwdstr_64	[243.9, 345.9]	5.31	0.6	[132.0, 462.6]	4.81	0.13
pwdstr_128	[478.4, 712.4]	22.01	0.6	[262.7, 940.2]	19.66	0.13
pwdstr_256	[951.4, 1431.4]	88.23	0.6	[528.1, 1882.5]	78.18	0.13

Table 5.1: Performance comparison of MultiSearchMC based on a number of SearchMC iterations

a file. We ran the head command to print out 1, 5 and 10 lines from the beginning of this input file. FuzzBall generated the path conditions such that the printed characters were concrete and others were symbolic. For example, head_1 prints out the first line of the input file which is 101 Bytes and the rest of file is symbolic. This makes 829 bytes (6632 bits) unconstrained therefore log based 2 of the model count is 6632. Our tool was able to slice the input formula and group unconstrained sub-formulae to reduce the running time. When the head benchmarks were sliced, there are one type of formulae where an input variable is constant and another type of formulae where an input variable is unconstrained. SMC computed the exact model counts for the unconstrained formulae and MultiSearchMC with 5 SearchMC's pre-iterations gave the exact model counts for the formulae that has only one solution. We also ran the tail command to print out 1, 5 and 10 lines from the end of the same input file. In the tail experiments, our tool was able to slice the input formula into three types of formulae: formulae with one solution, unconstrained formulae and one unsliceable formula. Due to the behaviour of tail, some input bytes were concatenated and unsliceable. Therefore, we ran SearchMC to compute the estimate of this unsliceable formula. We ran the grep command to print out 2, 10 and 16 lines from the beginning of this input file. From the investigation of the path conditions for grep, the generated path conditions were not sliceable since the path conditions constrained some bytes to not contain the target string. Since the path conditions for grep were not sliceable

				MultiSearchMC	
Benchmark	#Vars	#Cls	$\log_2(MC)$	Time (s)	Bounds
head_1	51158	108854	6632	101.53	6632
head_5	73236	175160	3904	93.68	3904
head_10	88661	221555	1976	92.37	1976
tail_1	50129	102671	7160	99.87	[7159.31, 7160.47]
tail_5	70518	152510	6048	99.13	[6046.94, 6048.60]
tail_10	94764	211736	4640	97.66	[4639.65, 4640.28]
grep_2	512089	1349483	n/a	30.04*	[837, 5970]
grep_10	772622	2070233	n/a	49.41*	[654.91, 4592.87]
grep_16	568215	1452206	n/a	150.20*	[798.44, 799.33]

Table 5.2: Performance of MultiSearchMC for Linux text processing programs. '*' indicates the running time of the first round of MultiSearchMC and did not complete the second round.

at all, we applied the *projection* procedure using 8 threads. Once we computed the combined bounds and we ran SearchMC for the final bounds. SearchMC was able to compute the final bounds for grep_16 so the running time represents the combination of the *preprocessing* procedure and the final SearchMC run. However, we were not able to compute the final bounds for grep_2 and grep_10 within 2 hours and the running times only represents the *preprocessing* procedure. The exact model counts for head and tail were computed by hands and DSHARP [6] was not able to compute the exact model counts of grep benchmarks within 2 hours. Note that ApproxMC was not able to compute the estimates of head, tail and grep benchmarks within 2 hours.

5.3.2 QIF Case Study: Privacy Measurement

This experiment is inspired by [79] which measured the amount of leakage from image processing functions. Applying an image processing technique such as pixelation or blurring can hide your identity from a picture. However, some image processing functions could hide your identity from an image visually but the output image still preserves a quite amount of the original image data so the original image was retrievable as much as you could be identified. The basic idea is to measure how much information an image processing function preserves after

Benchmark	$\log_2$ (MC)	ApproxMC4 Bounds	ApproxMC4 Time (s)	MultiSearchMC Bounds	MultiSearchMC Time (s)
gray_4x4	128	[127.2, 128.8]	3.44	128	0.93
gray_8x8	512	[511.2, 512.8]	92.65	512	9.93
gray_16x16	2048	[2047.2, 2048.8]	4018.1	2048	55.61
gray_32x32	8192	n/a	-	8192	482.12
pixelate_4x4	96	[95.2, 96.8]	1.88	96	0.91
pixelate_8x8	384	[383.2, 384.8]	44.63	384	4.06
pixelate_16x16	1536	[1534.2, 1536.8]	1187.1	1536	66.02
pixelate_32x32	6144	n/a	-	6144	1134.4
blur_4x4	n/a	n/a	-	[43.4, 363]	59.00*
blur_8x8	n/a	n/a	-	[44.5, 1463.7]	673.95*
blur_16x16	n/a	n/a	-	[44.5, 5877.9]	3856.4*
blur_32x32	n/a	n/a	-	n/a	-

Table 5.3: Performance of MultiSearchMC for image processing functions. '-' indicates time-out after 7200 seconds. '*' indicates the running time of the first round of MultiSearchMC and did not complete the second round.

conversion and tell which function is more secure to protect your identity. We implemented our own image processing functions (in an SMT-LIB 2 formula) based on the OpenCV library [80] which takes input as an nxn input data and used STP [50] to convert the SMT formula into the corresponding CNF formula with its projected variables. Each pixel is a 24-bit variable which contains red, green and blue for 8-bit each, and thus a 4x4 input image file contains 384 bits. We implemented three different image processing functions: bw, pixelate and blur. bw is the function which converts a color image to a black-and-white image using the average method. pixelate is the function that fills every 2x2 areas with the average of pixels in one area. blur is the box blurring function that takes a 3x3 area of pixels surrounding a central pixel, averages all these pixels together, and replaces the central pixel with the average. Table 5.3 shows applying various image processing functions with different input sizes and measuring the possible outputs to see how much each function preserves input data. Each output pixel of bw is independent of other output pixels since each output pixel is associated with the input pixel which has the same

position as the output pixel thus each pixel has 256 solutions. Also, each 2x2 area of pixelate is independent of other output areas since each area is computed by the same area of the input. MultiSearchMC was able to slice bw and pixelate benchmarks and was requested to compute the exact model count for each sliced formula. ApproxMC4 was requested to compute with a 80% confidence level and $\epsilon = 0.8$. MultiSearchMC was only able to slice blur benchmarks by a color data since each color data is computed separately. However, each pixel was not sliceable because all input pixels are connected by a 3x3 filter. We applied the *projection* procedure to unsliceable formulae using 8 threads. However, MultiSearchMC was not able to compute for the final bounds with the combined bounds. The results for blur benchmarks in the table are before the final SearchMC runs.

Note that the exact model counts for bw and pixelate were computed by hands and DSHARP was not able to compute the exact model counts of blur benchmarks within 2 hours. The experiments shows that MultiSearchMC scales better than ApproxMC4 and performs well with some image processing functions.

5.3.3 Uniform Sampling Experiments

Our main algorithm MultiSearchMC is implemented in Unigen3 replacing its model counting algorithm, ApproxMC. This implies we simply followed all the technical and theoretical details of Unigen3's sampling techniques. As we described in Section 5.2, ApproxMC first returns k (the most appropriate number of XOR constraints) in Unigen3. Based on k, Unigen3 generates a number of samples based on input parameters. We replaced ApproxMC to MultiSearchMC, which computed k faster.

Figure 5.2 shows the performance comparison between Unigen3 with ApproxMC and MultiSearchMC with a timeout of 3600 seconds. We explored over 1800 benchmarks. Unigen3 with ApproxMC was able to solve only 387 benchmarks while Unigen3 with MultiSearchMC was able to solve 395 benchmarks. Since the majority of running time in uniform sampling is the sampling process rather than the model counting process, the performance benefit may not be significant.

We also evaluated the quality of sampling as we compared a uniform sampler with our approach. We implemented a simple ideal uniform sampler using a random number generator to pick a number uniformly at random. We refer the reader to [23] for detailed discussion of this experiment. As described in [23], we also chose the CNF instance blasted_case110 where it

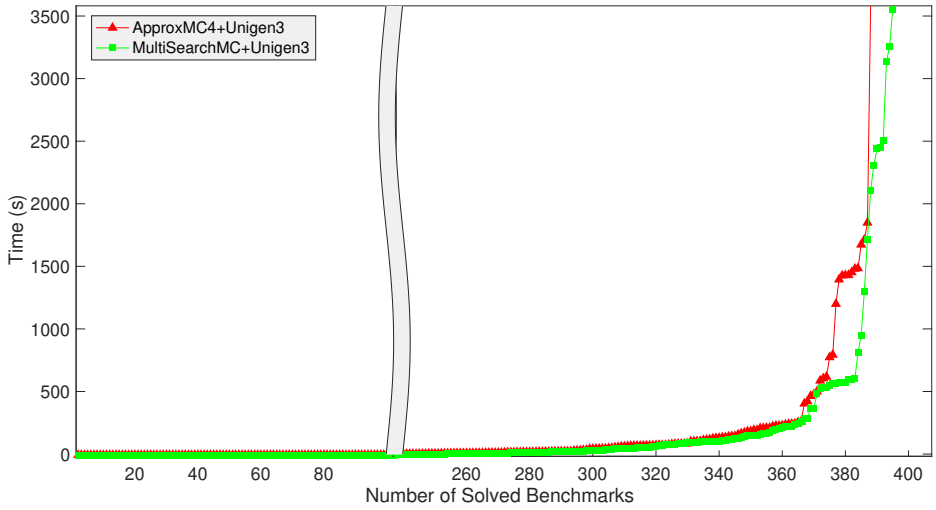


Figure 5.2: Unigen3 performance comparison of ApproxMC4 and MultiSearchMC as a back-end model counter

has 287 variables and 16384 solutions. We then generate 4,234,928 samples ($\approx 2^{22}$) from both approaches. In each case, we recorded the number of times various solutions were generated and represented as a distribution of the counts. Figure 5.3 shows the distributions of frequencies with the number of each solution generated. The x-axis represents the number of each generated solutions and the y-axis represents the number of samples appearing the specified number of times. For example, the point (240,194) represents that each of 194 distinct solutions were generated 240 times among the 4,234,928 samples. The figure compares the distributions of the uniform sampler using a random number generator and Unigen3 using MultiSearchMC. The yellow shaded area represents the binomial distribution of this experiment, which is theoretically ideal. While UniGen3 provides guarantees of almost-uniformity only, the two distributions are statistically indistinguishable. In particular, the KL divergence of the distribution by UniGen3 from that of the uniform sampler is 0.0074.

Table 5.4 shows more drastic results similar to Section 5.3.1. We can also take the advantage of slicing independent formulae to boost the performance and apply the sampling approach independently to each slice. Since `grep` was not able to take the advantage of the slicing procedure, it was not able to generate 500 samples within 2 hours. This experiment shows that Unigen3 with MultiSearchMC is able to scale to large-sized input data.

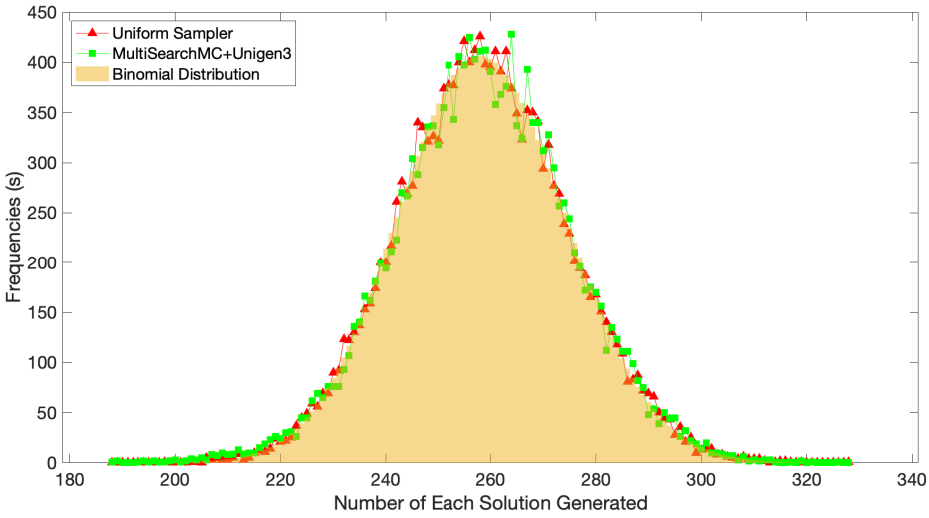


Figure 5.3: Uniformity comparison of a uniform sampler and UniGen3 with MultiSearchMC

5.4 Related Work

5.4.1 Quantitative Information Flow

Prior work on QIF has largely followed the paradigm of characterizing the set of a program's outputs. Phan *et al.* [81] 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. Specifically, it recursively calls the solver by adding a bit constraint for finding a single satisfying assignments as DPLL-based search. This search tree approach is useful when the program verification system does not expose the underlying logical representation or when the used solver cannot generate models. Klebanov *et al.*. [73] 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. They explore both exhaustive enumeration and the existing DSHARP and sharpSAT tools, but only counting distinct values of the output variables. Val *et al.* [36] 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. However, precisely counting solutions is a #P-complete problem and these exact

Benchmark	#Vars	#Cls	$\log_2(MC)$	Unigen3 +ApproxMC4 Time (s)	Unigen3 +MultiSearchMC Time (s)
pwdstr_8	9577	22319	39.10	10.31	8.12
pwdstr_16	19153	44639	76.52	321.81	15.24
pwdstr_24	28601	66599	114.42	5321.75	25.39
pwdstr_28	33341	77624	130.77	-	30.11
pwdstr_32	38081	88649	151.25	-	33.56
pwdstr_64	75953	176714	294.89	-	67.22
pwdstr_128	152001	353699	595.42	-	132.54
pwdstr_256	303985	707354	1191.44	-	278.13
head_1	51158	108854	6632	-	879.33
head_5	73236	175160	3904	-	912.10
head_10	88661	221555	1976	-	921.87
tail_1	50129	102671	7168	-	891.17
tail_5	70518	152510	6048	-	905.56
tail_10	94764	211736	4640	-	911.23

Table 5.4: Sampling experiments with text programs in Section 5.3.1 for generating 500 samples within 2 hours

model counters typically only work well with small-sized problems or ones with only simple constraints. For many practical problems, it is infeasible to count the exact number of solutions in a reasonable amount of time. Castro *et al.* [64] 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.* [34] show how an untrusted input affect a program and present a compound approach to obtain precise channel capacity measurements for a set of small, synthetic benchmark programs, and very coarse approximations to large, real-world programs including x86 binaries. Meng and Smith [65] present a method to obtain empirically good upper bounds on the channel capacity of various small synthetic example programs, also contributing to standardizing a set of benchmark programs to the field of QIF.

Klebanov et al. [73] show how to obtain precise measurements on channel capacity (in addition to conditional Shannon entropy) for a number of small synthetic programs, in addition to two examples of real C code on the order of magnitude of 100 lines. McCamant and Ernst [79] use a coarse upper-bounding approach for channel capacity based on network flows in order to show how to scale to hundreds of thousands of lines of real code, in addition to contributing smaller case studies as benchmarks. Phan and Malacaria [82] present a method that is able to analyze and compute upper bounds on the channel capacity for C implementations of several well-known protocols and three few-hundred-line case studies including parts of the Linux kernel. Biondi et al. [54] present ApproxFlow, using ApproxMC2, to measure the approximate channel capacity of deterministic C programs as well as a new case study based on the OpenSSL Heartbleed bug. While some of the above work has demonstrated that generating SAT formulas is possible even for large programs, complex program structures such as pointers often result in SAT formulas that are too difficult for model counting.

Weigl [83] presents a tool sharpPI, which implements different search heuristics for model counting applied to measurement of Shannon entropy, presenting results for a small, scalable synthetic C program. Biondi et al. [84] propose a technique to measure Shannon entropy for a number of scalable case studies expressed in a simple imperative language. Backes et al. [85] present a technique to analyze small, synthetic programs with respect to various information-theoretic measures.

5.4.2 Uniform Sampling

Jerrum, Valiant, and Vazirani [86] studied uniform generation of SAT solutions. They showed that the problem can be solved in probabilistic polynomial time and near-uniform generation is inter-reducible in a polynomial time with approximate model counting. Bellare, Goldreich, and Petrank [19] improved this idea but unfortunately, their algorithm fails to scale beyond few tens of variables in practice [20]. Yuan et al. [87] introduced a different approach which used a random walk over a weighted binary decision diagram (WBDD) in order to generate samples. Still, this approach had the limitations of high space requirement and its applicability in practice.

The interest of constraint random sampling has been emerged in some industrial domains but earlier approaches worked via heuristic methods which provide very weak or no guarantees of uniformity. In general, a good trade-off between the scalability and the uniformity has been

always challenging for uniform sampling. Strengthening the uniformity eventually loses the scalability and increasing the scalability provides weak theoretical guarantees. Earlier research focused on increasing the scalability rather than the uniformity [59, 88, 89, 90].

Recently, several random hashing-based techniques have been proposed to bridge the wide gap between scalable algorithms and those that give strong guarantees of uniformity when sampling witnesses of propositional constraints [20, 22, 91]. Sipser [92] first introduced the basic idea of hashing-based sampling techniques and Jerrum et al [86] and Bellare et al [19] improved the idea further. The key idea in hashing-based techniques is to divide the solution space into small pieces of roughly equal size using an independent hashing function and then select a random solution from a randomly chosen piece. Chakraborty, Meel, and Vardi [20] showed UniWit with XOR streamlining [15] to show a scalable near-uniform generator. Further algorithmic improvements [91] and a new algorithm named UniGen [11] were proposed upon the ideas of UniWit. UniGen provided stronger guarantees of uniformity and showed to scale to formulae with hundreds of thousands of variables. Several improvements to UniGen [23] have been applied such as scaling by parallelization to increase the performance.

5.5 Chapter Summary

In this chapter, we first described the combination of symbolic execution and model counting techniques in quantitative information flow. We used FuzzBall, a symbolic execution tool, to explore a program path and generate a formula which represents path conditions. In some program paths, there might be private information leakage so we apply model counting techniques to quantify potential information leakage. Our experimental results showed that MultiSearchMC was able to provide a precise information leakage from realistic programs in a reasonable amount of time.

As the uniform sampling based on hashing-based model counting techniques has emerged since the last few years, previous research showed that this approach generates samples in a near-uniform distribution. We showed how uniform sampling based on hashing-based model counting techniques performed. The core of this technique is to apply a hashing-based model counting technique to find the most appropriate number of XOR constraints first and use this value to generate a given number of samples. UniGen3 has been a state-of-the-art almost-uniform sampler using ApproxMC, a hashing-based model counting technique. Since we

showed the scalability of MultiSearchMC compared to ApproxMC, we switched Unigen3's model counting approach to ours to increase its scalability and performance while we followed the same infrastructure for generating samples. Our experimental results showed that our approach was able to generate samples faster than Unigen3 for large input data.

Chapter 6

Future Work

We introduced MultiSearchMC, a scalable model counting technique, which is a divide-and-conquer algorithm for SearchMC, a hashing-based model counting technique. The gap between theory and practice in this model counting technique is still more to be explored. Here we list several open future work that would be critical to improve MultiSearchMC's performance and precision.

6.1 A Parallel Solver with Gaussian Elimination

The performance of a decision procedure is a crucial factor of MultiSearchMC's performance. As we discussed in Chapter 4, hashing-based model counters tend to use a SAT solver which optimizes for handling XOR constraints efficiently by Gaussian Elimination. Also, the solver requires to support the incrementality feature. CryptoMiniSat [44] is a state-of-the-art SAT solver which supports Gaussian Elimination and the incrementality feature. Therefore, many hashing-based model counters use CryptoMiniSat as the back-end SAT solver.

Recently, many parallel SAT solvers have been introduced and can be categorized in in two approaches mainly: portfolio and divide-and-conquer approaches. The portfolio approaches are based on running different optimizations and search strategies in parallel and taking the fastest result [93, 94, 95]. The divide-and-conquer approaches divide an input formula and distribute the total computation to each thread [96, 97, 98, 99]. However, to the best of our knowledge, there is no state-of-the-art SAT solver which supports Gaussian Elimination and parallelization both. Previous research [100] has shown that Gaussian elimination is suitable to solve solutions

of linear equations in parallel but there has not been any on-going research about this. We expect that it is worth exploring to build a parallel SAT solver with Gaussian elimination and beneficial to improve the performance of hashing-based model counters.

6.2 XOR Streamlining Probabilistic Distribution

Most of hashing-based model counters are based on XOR streamlining, which is a 3-universal hash function. Since it is difficult to formulate the probabilistic distribution of XOR streamlining precisely, SearchMC uses a binomial-distribution model to update the probability distribution of an expected model count. This probability distribution provides empirical bounds of the model count and SearchMC requires more queries to achieve the probabilistic soundness. Also, there is a certain gap between the actual probabilistic soundness and the provable probability. Section 2.4 showed that SearchMC-sound and ApproxMC2 requested the answers with a 60% confidence level and they both outperformed the requested level, which was more than 90% correctness. Therefore, SearchMC would be able to generate probabilistic sound bounds and achieve a higher confidence level using less number of SAT queries than the current implementation if a more accurate probabilistic distribution of XOR streamlining is provided.

6.3 Precision of SMC

Currently, some of SMT-LIB2 operators are not supported in SMC and increasing the coverage of the whole SMT-LIB2 format standard will be very useful. Also, some operators compute the cardinalities very conservatively due to the limitation of the node representation, hence those operators lead to less precise results (loose bounds). The precision of SMC can be increased by having more coverage of SMT-LIB2 format standard and designing more precise rules. Lastly, the order of assertions and merging bounds can be improved since this affects precision. SMC recursively computes the bounds until the bounds do not change but we do not have any proof about its time complexity. This means the bounds can be more precise by merging the per-assertion bounds in a better order. We expect finding a more efficient way to merge bounds will improve the performance.

6.4 Portfolio-style Parallelization

As we discussed in Section 4.3, the majority of the performance depends on a decision procedure and every decision procedure has different optimization techniques which can affect the performance. As we described above, CryptoMiniSat is a state-of-the-art SAT solver which works well on hashing-based model counters. CryptoMiniSat2 and CryptoMiniSat5 have different optimization techniques and there might be the setting of parameters that makes identical optimizations between two versions. However, it is clear that some SAT instances are more suitable for CryptoMiniSat2 even though CryptoMiniSat5 is the most recent version. Also, it is difficult to determine which optimization technique works best for a given formula in advance. The analysis of the solvers and SAT instances needs to be further explored and we believe this is an interesting topic to the SAT community. One possible solution to eliminate this uncertainty of solver selection is to execute MultiSearchMC in a portfolio-style approach. For example, we run MultiSearchMC using CryptoMiniSat2 with a half of total cores and CryptoMiniSat5 with another half. The result whichever generated first would be the final result. There might be more optimization techniques to select a more appropriate solver and use cores more efficiently. Therefore, an interesting direction of future research would be to study how to determine a better solver for hashing-based model counters using parallelization.

Chapter 7

Conclusion

Recently, hashing-based model counting techniques have been an emerging area where they provide precise bounds and reasonable theoretic guarantees. Since the techniques computes the approximation of model counts, they have shown the scalability compared to exact model counting techniques. Due to strong theoretical and practical interest in the hashing-based model counting techniques, many researchers have worked on reducing a huge gap between theory and practice. Since there is a trade-off between theoretical guarantees on the quality of approximation and practical scalability at the cost of offering weaker or no guarantees. With the effort of finding an appropriate spot between guarantees and scalability, we present 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.

We proposed another approximate model counting algorithm, SMC, to compute the lower and upper bound of solutions to a given SMT formula. We categorize SMC as "structural" approximate model counting algorithms which analyze the syntactic structure of a formula. Previous structural model counting algorithms have been specialized for narrow domains, or have been built into larger systems in ways that are not easily reusable. Our tool uses algorithms which build on the partial description of a previous closed-source tool [69], but we needed to develop new structural rules for cases that were missing or restricted in previous work. We extend the algorithm to cover a more complete set of bit-vector operators, and to use both the signed and unsigned orderings of bit vectors. The experimental results showed that SMC's

performance scales much better than these other tools, and that the sound upper and lower model-count bounds that it provides are often usefully accurate. Also, the combination of SMC and SearchMC was useful since SMC provides a more refined hypothesis to reduce a search space for SearchMC.

However, solving a formula with XOR constraints is still a complicated task as the size/complexity of a formula increases. We proposed MultiSearchMC, a divide-and-conquer algorithm to increase the scalability of the hashing-based techniques and this approach can be parallelized. We used projected model counting techniques to split an input formula into small formulae and ran SearchMC with each formula in parallel. We combined all the results to provide the final result and our experimental results showed that the performance of MultiSearchMC scaled better than a single-threaded SearchMC's run.

We focused on using this scalable model counting technique to analyze programs in two areas: quantitative information-flow analysis and uniform sampling for testing. Quantitative information flow (QIF) analysis is a powerful approach to measure the amount of sensitive information leakage. We used symbolic execution tools to analyze realistic binary files and apply quantitative information-flow analysis. Our experimental result showed that MultiSearchMC was able to provide a precise information leakage from realistic programs in a reasonable amount of time. Another application of approximate model counting techniques is uniform sampling for testing. Uniform sample generation for SAT/SMT formulas is widely used in various areas such as probabilistic reasoning in AI systems, functional verification and so on. Generating independent uniformly distributed samples over a set of satisfying assignments is a challenging problem both theoretically and practically. Our work was inspired by Unigen3 [23], which uses hashing-based techniques to compute an estimate model count and generate near-uniform samples. Basically, we switched Unigen3's model counting approach to ours to increase its scalability and performance and we built our approach using the same infrastructure for generating samples. Our experiments showed that our version achieves a more speed-up than Unigen3. Overall, we were able to show the scalability of our hashing-based model counting techniques.

References

[1] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[2] Roberto J. Bayardo, Jr. and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 203–208, 1997.

[3] Elazar Birnbaum and Eliezer L. Lozinskii. The good old Davis-Putnam procedure helps counting models. *Journal of Artificial Intelligence Research (JAIR)*, 10(1):457–477, 1999.

[4] Tian Sang, Fahiem Bacchus, Paul Beame, Henry A. Kautz, and Toniann Pitassi. Combining component caching and clause learning for effective model counting. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 2004.

[5] Marc Thurley. SharpSAT: Counting models with advanced component caching and implicit BCP. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT)*, page 424–429, 2006.

[6] Christian Muise, Sheila A. McIlraith, J. Christopher Beck, and Eric I. Hsu. Dsharp: Fast d-DNNF compilation with sharpSAT. In *Advances in Artificial Intelligence*, pages 356–361, 2012.

[7] Jan Burchard, Tobias Schubert, and Bernd Becker. Laissez-Faire caching for parallel #SAT solving. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 46–61, 2015.

[8] Wei Wei and Bart Selman. A new approach to model counting. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT)*, page 324–339, 2005.

[9] Carla P. Gomes, Joerg Hoffmann, Ashish Sabharwal, and Bart Selman. From sampling to model counting. In *Proceedings of the International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 2293–2299, 2007.

[10] Lukas Kroc, Ashish Sabharwal, and Bart Selman. Leveraging belief propagation, backtrack search, and statistics for model counting. In *Proceedings of the International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR)*, pages 127–141, 2008.

[11] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. A scalable approximate model counter. In *Proceedings of International Conference on Principles and Practice of Constraint Programming (CP)*, pages 200–216, 2013.

[12] Seonmo Kim and Stephen McCamant. Bit-vector model counting using statistical estimation. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 133–151, 2018.

[13] Cunjing Ge, Feifei Ma, Tian Liu, Jian Zhang, and Xutong Ma. A new probabilistic algorithm for approximate model counting. In *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR)*, 2018.

[14] Jonathan Kuck, Tri Dao, Shengjia Zhao, Burak Bartan, Ashish Sabharwal, and Stefano Ermon. Adaptive hashing for model counting. In *Proceedings of the Uncertainty in Artificial Intelligence Conference (UAI)*, pages 271–280, 2020.

[15] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Model counting: A new strategy for obtaining good bounds. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 54–61, 2006.

[16] Seonmo Kim and Stephen McCamant. Structural bit-vector model counting. In *Proceedings of the International Workshop on Satisfiability Modulo Theories (SMT)*, pages 26–36, 2020.

[17] Rehan Abdul Aziz, Geoffrey Chu, Christian Muise, and Peter Stuckey. #∃SAT: Projected model counting. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 121–137, 2015.

[18] Shujun Deng, Zhiqiu Kong, Jinian Bian, and Yanni Zhao. Self-adjusting constrained random stimulus generation using splitting evenness evaluation and XOR constraints. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 769–774, 2009.

[19] Mihir Bellare, Oded Goldreich, and Erez Petrank. Uniform generation of NP-witnesses using an NP-oracle. *Information and Computation*, 163(2):510–526, 2000.

[20] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. A scalable and nearly uniform generator of sat witnesses. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, pages 608–623, 2013.

[21] Rina Dechter, Kalev Kask, Eyal Bin, and Roy Emek. Generating random solutions for constraint satisfaction problems. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, page 15–21, 2002.

[22] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. Balancing scalability and uniformity in SAT witness generator. In *Proceedings of the Design Automation Conference (DAC)*, page 1–6, 2014.

[23] Mate Soos, Stephan Gocht, and Kuldeep S. Meel. Tinted, detached, and lazy cnf-xor solving and its applications to counting and sampling. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, pages 463–484, 2020.

[24] Roberto Bruttomesso. *RTL Verification: From SAT to SMT (BV)*. PhD thesis, University of Trento, 2008.

[25] David Cyrluk, Oliver Möller, and Harald Rueß. An efficient decision procedure for the theory of fixed-sized bit-vectors. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, pages 60–71, 1997.

[26] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, page 151–158, 1971.

[27] Dorothy Elizabeth Robling Denning. *Cryptography and Data Security*. Addison-Wesley Longman Publishing Co., Inc., 1982.

[28] J.W. Gray. Toward a mathematical foundation for information flow security. In *Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy (S&P)*, pages 21–34, 1991.

[29] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[30] Konstantinos Chatzikokolakis, Catuscia Palamidessi, and Prakash Panangaden. Anonymity protocols as noisy channels. *Information and Computation*, 206(2–4):378–401, 2008.

[31] Boris Köpf and David Basin. An information-theoretic model for adaptive side-channel attacks. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, page 286–296, 2007.

[32] Richard McPherson, Reza Shokri, and Vitaly Shmatikov. Defeating image obfuscation with deep learning. *CoRR*, abs/1609.00408, 2016.

[33] Yehuda Naveh, Michal Rimon, Itai Jaeger, Yoav Katz, Michael Vinov, Eitan Marcus, and Gil Shurek. Constraint-based random stimuli generation for hardware verification. In *Proceedings of the Conference on Innovative Applications of Artificial Intelligence (IAAI)*, page 1720–1727, 2006.

[34] James Newsome, Stephen McCamant, and Dawn Song. Measuring channel capacity to distinguish undue influence. In *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 73–85, 2009.

[35] Vladimir Klebanov, Alexander Weigl, and Jörg Weisbarth. Sound probabilistic #SAT with projection. In *International Workshop on Quantitative Aspects of Programming Languages and Systems (QAPL)*, 2016.

[36] Celina G. Val, Michael A. Enescu, Sam Bayless, William Aiello, and Alan J. Hu. Precisely measuring quantitative information flow: 10K lines of code and beyond. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 31–46, 2016.

[37] Alexander Ivrii, Sharad Malik, Kuldeep S. Meel, and Moshe Y. Vardi. On computing minimal independent support and its applications to sampling and counting. *Constraints*, 21(1):41–58, 2016.

[38] Mateus Borges, Antonio Filieri, Marcelo d'Amorim, Corina S. Pasareanu, and Willem Visser. Compositional solution space quantification for probabilistic software analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 123–132, 2014.

[39] Dmitry Chistikov, Rayna Dimitrova, and Rupak Majumdar. Approximate counting in SMT and value estimation for probabilistic programs. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 320–334, 2015.

[40] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic SAT calls. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, page 3569–3576, 2016.

[41] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Near-uniform sampling of combinatorial spaces using XOR constraints. In *Proceedings of the International Conference on Neural Information Processing Systems (NIPS)*, page 481–488, 2006.

[42] P. Del Moral. Nonlinear filtering: Interacting particle solution. *Markov Processes and Related Fields*, 2(4):555–580, 1996.

[43] Mate Soos and Kuldeep S. Meel. BIRD: Engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 1592–1599, 2019.

[44] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 244–257, 2009.

[45] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB standard: Version 2.0. In *Proceedings of the International Workshop on Satisfiability Modulo Theories (SMT)*, 2010.

[46] Jeanette P. Schmidt, Alan Siegel, and Aravind Srinivasan. Chernoff-hoeffding bounds for applications with limited independence. *SIAM J. Discret. Math.*, 8(2):223–250, May 1995.

[47] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

[48] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, page 93–107, 2013.

[49] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, pages 519–531, 2007.

[50] STP. `http://stp.github.io/`.

[51] Bayesian-inference as model-counting benchmarks. `http://www.cs.rochester.edu/users/faculty/kautz/Cachet/Model_Counting_Benchmarks/index.htm`.

[52] F. Brglez, D. Bryan, and K. Kozminski. Combinational profiles of sequential benchmark circuits. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1929–1934 vol.3, 1989.

[53] Huan Chen and Joao Marques-Silva. TG-Pro: A SAT-based ATPG system. *Journal of Satisfiability, Boolean Modeling and Computation*, 8(1-2):83–88, 2012.

[54] Fabrizio Biondi, Michael A. Enescu, Annelie Heuser, Axel Legay, Kuldeep S. Meel, and Jean Quilbeuf. Scalable approximation of quantitative information flow in programs. In *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 71–93, 2018.

[55] Cynthia Dwork. Differential privacy. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, pages 1–12, 2006.

[56] Ilya Mironov. On significance of the least significant bits for differential privacy. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 650–661, 2012.

[57] João P. Marques Silva and Karem A. Sakallah. GRASP-a new search algorithm for satisfiability. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 220–227, 1996.

[58] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 502–518, 2004.

[59] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference (DAC)*, pages 530–535, 2001.

[60] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 142–149, 2002.

[61] Armin Biere. PicoSAT essentials. *JSAT*, 4:75–97, 2008.

[62] Armin Biere. Lingeling , Plingeling , PicoSAT and PrecoSAT at SAT Race 2010. Technical report, Johannes Kepler University, 2010.

[63] Supratik Chakraborty, Kuldeep S. Meel, Rakesh Mistry, and Moshe Y. Vardi. Approximate probabilistic inference via word-level counting. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 2016.

[64] Miguel Castro, Manuel Costa, and Jean-Philippe Martin. Better bug reporting with better privacy. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 319–328, 2008.

[65] Ziyuan Meng and Geoffrey Smith. Calculating bounds on information leakage using two-bit patterns. In *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 1:1–1:12, 2011.

[66] Loi Luu, Shweta Shinde, Prateek Saxena, and Brian Demsky. A model counter for constraints over unbounded strings. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 565–576, 2014.

[67] Jeffrey Dudek, Kuldeep S. Meel, and Moshe Y. Vardi. Combining the k-CNF and XOR phase-transitions. In *Proceedings of International Joint Conference on Artificial Intelligence*, 2016.

[68] Jeffrey Dudek, Kuldeep S. Meel, and Moshe Y. Vardi. The hard problems are almost everywhere for random CNF-XOR formulas. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 2017.

[69] Jean-Philippe Martin. Upper and lower bounds on the number of solutions. Technical Report MSR-TR-2007-164, Microsoft Research, 2007.

[70] Wei Gao, Hengyi Lv, Qiang Zhang, and Dunbo Cai. Estimating the volume of the solution space of SMT(LIA) constraints by a flat histogram method. *Algorithms*, 11:142, 2018.

[71] Abdulbaki Aydin, William Eiers, Lucas Bang, Tegan Brennan, Miroslav Gavrilov, Tevfik Bultan, and Fang Yu. Parameterized model counting for string and numeric constraints. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, (ESEC/SIGSOFT FSE)*, pages 400–410, 2018.

[72] L. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.

[73] Vladimir Klebanov, Norbert Manthey, and Christian Muise. SAT-based analysis and quantification of information flow in programs. In *Proceedings of the International Conference on Quantitative Evaluation of Systems (QEST)*, pages 156–171, 2013.

[74] Program to Check Strength of Password. `https://www.geeksforgeeks.org/program-check-strength-password/`.

[75] FuzzBALL. `http://bitblaze.cs.berkeley.edu/fuzzball.html`.

[76] Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. Path-exploration lifting: Hi-fi tests for lo-fi emulators. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 337–348, 2012.

[77] Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia, and Moshe Y. Vardi. On parallel scalable uniform SAT witness generator. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 304–319, 2015.

[78] FHIR Test Data Generator. `https://github.com/smart-on-fhir/sample-patients/blob/master/data/familyhistory.txt`.

[79] Stephen McCamant and Michael D. Ernst. Quantitative information flow as network flow capacity. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 193–205, 2008.

[80] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[81] QuocSang Phan, Pasquale Malacaria, Oksana Tkachuk, and Corina S. Păsăreanu. Symbolic quantitative information flow. *SIGSOFT Softw. Eng. Notes*, 37(6):1–5, 2012.

[82] Quoc-Sang Phan and Pasquale Malacaria. Abstract model counting: A novel approach for quantification of information leaks. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 283–292, 2014.

[83] Alexander Weigl. Efficient SAT-based pre-image enumeration for quantitative information flow in programs. In *Data Privacy Management and Security Assurance*, pages 51–58, 2016.

[84] Fabrizio Biondi, Axel Legay, Pasquale Malacaria, and Andrzej Wasowski. Quantifying information leakage of randomized protocols. In *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 68–87, 2013.

[85] Michael Backes, Boris Kopf, and Andrey Rybalchenko. Automatic discovery and quantification of information leaks. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 141–153, 2009.

[86] Mark R. Jerrum, Leslie G. Valiant, and Vijay V. Vazirani. Random generation of combinatorial structures from a uniform distribution. *Theoretical Computer Science*, 43:169–188, 1986.

[87] Jun Yuan, K. Albin, A. Aziz, and C. Pixley. Simplifying Boolean constraint solving for random simulation-vector generation. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 123–127, 2002.

[88] Nathan Kitchen and Andreas Kuehlmann. Stimulus generation for constrained random simulation. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 258–265, 2007.

[89] Alexander Nadel. Generating diverse solutions in SAT. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 287–301, 2011.

[90] M.A. Iyer. Race: A word-level ATPG-based constraints solver system for smart random simulation. In *Proceedings of the International Test Conference (ITC)*, pages 299–308, 2003.

[91] Stefano Ermon, Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Embed and project: Discrete sampling with universal hashing. In *Proceedings of the International Conference on Neural Information Processing Systems (NIPS)*, page 2085–2093, 2013.

[92] Michael Sipser. A complexity theoretic approach to randomness. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, page 330–335, 1983.

[93] Gilles Audemard and Laurent Simon. Lazy clause exchange policy for parallel SAT solvers. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 197–205, 2014.

[94] Armin Biere et al. Lingeling, plingeling and treengeling entering the SAT competition 2013. *Proceedings of SAT competition*, 2013:1, 2013.

[95] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. ManySAT: A parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6(4):245–262, 2010.

[96] Antti E. J. Hyvärinen, Tommi Junttila, and Ilkka Niemelä. Partitioning SAT instances for distributed solving. In *Proceedings of the International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, pages 372–386, 2010.

[97] Marijn J. H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *Hardware and Software: Verification and Testing*, pages 50–65, 2012.

[98] S Plaza, I Markov, and Valeria Bertacco. Low-latency SAT solving on multicore processors with priority scheduling and XOR partitioning. In *International Workshop on Logic and Synthesis (IWLS)*, 2008.

[99] Carsten Sinz, Wolfgang Blochinger, and Wolfgang Küchlin. PaSAT-parallel SAT-checking with lemma exchange: Implementation and applications. *Electronic Notes in Discrete Mathematics*, 9:205–216, 2001.

[100] K.N.Balasubramanya Murthy and C.Siva Ram Murthy. A new gaussian elimination-based algorithm for parallel solution of linear equations. *Computers & Mathematics with Applications*, 29(7):39–54, 1995.