Model Counting in the Wild[*]

Arijit Shaw[1,2] , **Kuldeep S. Meel**[3]
[1]Chennai Mathematical Institute, India
[2]IAI, TCG CREST, Kolkata, India
[3]University of Toronto, Canada

Abstract

Model counting is a fundamental problem in automated reasoning with applications in probabilistic inference, network reliability, neural network verification, and more. Although model counting is computationally intractable from a theoretical perspective due to its #P-completeness, the past decade has seen significant progress in developing state-of-the-art model counters to address scalability challenges.

In this work, we conduct a rigorous assessment of the scalability of model counters in the *wild*. To this end, we surveyed 11 application domains and collected an aggregate of 2262 benchmarks from these domains. We then evaluated six state-of-the-art model counters on these instances to assess scalability and runtime performance.

Our empirical evaluation demonstrates that the performance of model counters varies significantly across different application domains, underscoring the need for careful selection by the end user. Additionally, we investigated the behavior of different counters with respect to two parameters suggested by the model counting community, finding only a weak correlation. Our analysis highlights the challenges and opportunities for portfolio-based approaches in model counting.

1 Introduction

Given a Boolean formula F (often presented in conjunctive normal form), the problem of model counting is to compute the number of solutions of the formula. Model counting is a fundamental problem in computer science and has been studied by theoreticians and practitioners alike for the past four decades. From the perspective of theoreticians, model counting is a central problem in computational complexity: the seminal work of Valiant (1979) established that the problem of model counting is #P-complete, where #P is the class of counting problems whose decision versions lie in NP. Toda's celebrated result (Toda 1989) showed that a single call to a #P-oracle suffices to solve a problem in the entire polynomial hierarchy; formally, $\mathrm{PH} \subseteq \mathrm{P}^{\#\mathrm{P}}$. On the other hand, from practitioners' perspectives, model counting emerges as a central problem in a wide variety of domains such as quantitative software verification (Teuber and Weigl 2021; Girol, Farinier, and Bardin 2021), probabilistic inference (Darwiche 2004), network reliability (Kabir and Meel 2023), cryptography (Beck, Zinkus, and Green 2020), synthesis (Golia, Roy, and Meel 2020), product lines (Sundermann et al. 2023; Kuiter et al. 2022), neural network verification (Baluta et al. 2019), and information flow (Bang et al. 2016). Consequently, despite theoretical hardness (in the worst case), there has been a demand for the development of algorithms and tools for model counting.

The earliest algorithmic approaches to model counting were pioneered in the early 2000s, combining advances in conflict-driven clause learning with knowledge compilation (Darwiche 2004; Sang et al. 2004). Subsequently, approaches in the early 2010s based on universal hashing and SAT solving were developed to obtain probably approximate counters (Gomes, Sabharwal, and Selman 2006; Chakraborty, Meel, and Vardi 2013). Since then, there has been a significant surge of interest in the development of model counters. This development has led to substantial improvements in the runtime performance of state-of-the-art counters (Thurley 2006; Lagniez and Marquis 2017; Sharma et al. 2019; Korhonen and Järvisalo 2021; Lai, Meel, and Yap 2021), best evidenced by the launch of the model counting competition in 2020 (Fichte, Hecher, and Hamiti 2020).

The model counting competition also allowed for the standardization of input and output formats, thereby making it easy to use different counters uniformly. The yearly competition also provides a snapshot of the performance of different counters. Given the annual snapshot of performance, one might be tempted to rely on the model counting competition to provide guidance on what counter to use in practice: for example, pick the winner of the latest model counting competition. Such a strategy is generally expected to fare well but may not be optimal since the objective of competitions is often to focus on benchmarks that are *difficult*. While selection focused on *difficult* benchmarks brings forth the weaknesses of state-of-the-art techniques, it may not guide the behavior of counters in the real world.

The primary objective of our investigation is the study of the scalability of model counters in *the wild*. To this end, we focused on the six state-of-the-art model counters, which have consistently performed well in model counting competitions over the past three years and rely on differing underlying techniques. As a next step, we constructed a benchmark suite

[*]Full version of the paper is available at https://arxiv.org/abs/2408.07059. The benchmarks and logfiles are available at https://doi.org/10.5281/zenodo.13284882.

comprising instances from 11 different application categories, including quantitative software verification, probabilistic inference, network reliability, cryptography, synthesis, product lines, neural network verification, and information flow. In total, our benchmark suite consists of 2262 instances. We then performed an extensive analysis of the performance of different counters in these instances. Furthermore, we sought to understand how the performance of counters varies with respect to features of input formulas.

Our experiments with different model counters on this set of benchmarks revealed the following:

1. Among the individual solvers for non-projected instances, SharpSAT-TD performed the best, solving 811 out of 1080 instances. For projected instances, ApproxMC achieved the highest performance, solving 1041 out of 1182 instances.
2. Compilation-based, and hashing-based model counters excelled on different sets of benchmarks, often complementing each other. This complementary nature significantly improved the performance of the virtual best solver, which solved 2106 out of 2262 instances.
3. The performance of compilation-based counters is correlated with treewidth, while the performance of hashing-based counters shows a weak correlation with independent support size.

Organization. The structure of the paper is as follows. First, we introduce notation and preliminaries in Section 2. Next, we discuss various algorithms for model counting in Section 3 and describe the applications of model counting in Section 4. Our experimental results are presented in Section 5. Finally, we conclude in Section 6.

2 Notations and Preliminaries

Let X be the set of Boolean variables, and let F be a Boolean formula in Conjunctive Normal Form (CNF) defined over variables in X. An assignment $\sigma : X \mapsto \{0,1\}$ is called a satisfying assignment or a solution if σ makes F evaluate to True. Given a set of projection variables $P \subseteq X$, a projection of assignment σ to the set P is the subset of assignments only to the variables of P.

Model Counting. Let $\mathsf{Sol}(F)$ denote the number of solutions of a given formula F. The model counting problem is determining $|\mathsf{Sol}(F)|$. *An exact model counter* takes in formula F, and returns $|\mathsf{Sol}(F)|$. *An approximate model counter* takes in a formula F, tolerance parameter ε, confidence parameter δ and returns c such that $\Pr\left[\frac{|\mathsf{Sol}(F)|}{1+\varepsilon} \leq c \leq (1+\varepsilon)|\mathsf{Sol}(F)|\right] \geq 1-\delta$.

Projected Model Counting. Let $\mathsf{Sol}(F)_{\downarrow S}$ denote the set of projected assignments satisfying the given formula F and a projection set S. The problem of *projected model counting* is to compute $|\mathsf{Sol}(F)_{\downarrow S}|$. *An exact projected model counter* takes in formula F, and returns $|\mathsf{Sol}(F)_{\downarrow S}|$. An *approximate projected model counter* takes in a formula F, projection set S, parameters ε, and δ, and returns c such that $\Pr\left[\frac{|\mathsf{Sol}(F)_{\downarrow S}|}{1+\varepsilon} \leq c \leq (1+\varepsilon)|\mathsf{Sol}(F)_{\downarrow S}|\right] \geq 1-\delta$.

To differentiate between model counting and projected model counting, we use the term *non-projected model counting* to refer to model counting without projection.

Independent Support. For a given assignment σ over X and a subset of variables $S \subseteq X$, let $\sigma_{\downarrow S}$ represent the assignment of variables restricted to S. Given a Boolean formula F over the set of variables X and a projection set $S \subseteq X$, a subset of variables $\mathcal{I}$ such that $\mathcal{I} \subseteq S$ is called independent support (or simply *support*) of S if $\forall \sigma_1, \sigma_2 \in \mathsf{Sol}(F), \sigma_{1\downarrow\mathcal{I}} = \sigma_{2\downarrow\mathcal{I}} \implies \sigma_{1\downarrow S} = \sigma_{2\downarrow S}$. Several preprocessing techniques for model counting have been proposed, which compute a small independent support for the input formula and simplify the formula based on that support (Lagniez, Lonca, and Marquis 2016; Soos and Meel 2019).

Treewidth. Treewidth is a measure of how *tree-like* a graph is. A tree decomposition of a graph $G = (V, E)$ is a pair $(T, \{B_i\}_{i\in I})$ where T is a tree with nodes indexed by I, and $\{B_i\}_{i\in I}$ are subsets of V (bags) such that:

1. Every vertex $v \in V$ is in at least one bag B_i.
2. For every edge (u, v), there is a bag B_i with $u, v \in B_i$.
3. For each vertex v, the bags containing v form a connected subtree of T.

The width of a tree decomposition $(T, \{B_i\}_{i\in I})$ is $\max_{i\in I}(|B_i| - 1)$. The treewidth of G, denoted $\text{tw}(G)$, is the minimum width among all tree decompositions of G: $\text{tw}(G) = \min_{(T,\{B_i\})} \max_{i\in I}(|B_i| - 1)$.

3 The Landscape of Model Counting

Significant progress has been made in developing efficient algorithms for model counting. In this section, we provide a brief overview of the different approaches.

1. **Compilation-based Exact Model Counters.** The remarkable success of SAT solvers has motivated researchers to develop model counters that leverage the search techniques employed by these solvers. Darwiche (2004) introduced the use of deterministic decomposable negation normal form (d-DNNF) to efficiently obtain the model count in a model counter named c2d. During the search procedure of DPLL, the solver may encounter sub-formulas that have already been seen in a prior branch of the tree. To avoid redundant computations, it is essential to recognize such sub-formulas and reuse their model counts efficiently. To address this challenge, researchers have introduced the concept of component caching, which has led to the development of highly efficient model counters like Cachet (Sang et al. 2004) and SharpSAT (Thurley 2006). Lagniez and Marquis (2017) further improved this approach by utilizing dynamic decomposition techniques to enhance the efficiency of d-DNNF-based techniques and designed the D4 model counter. Subsequently,

heuristics were integrated into component caching in the probabilistic model counter Ganak (Sharma et al. 2019). Korhonen and Järvisalo (2021) combined the concept of tree decomposition was combined with component caching in SharpSAT-TD. GPMC uses a similar strategy as SharpSAT, but optimizes it for projected model counting. Lai, Meel, and Yap (2021) introduced a generalization of d-DNNF known as Constrained Conjunction and Decision Diagram (CCDD), which was implemented in the model counter ExactMC. While all the aforementioned techniques employ a search-based top-down compilation approach, Dudek, Phan, and Vardi (2020) utilized algebraic decision diagram (ADD) based bottom-up compilation methods to design the model counter ADDMC, which also performs effectively due to its early-projection techniques.

2. **Hashing-based Approximate counter.** Over the past decade, there has been the development of scalable approximate model counters that rely on XOR-based pairwise independent functions to divide the solution space into smaller parts and then invoke state-of-the-art SAT solvers to enumerate models in a randomly chosen cell to accurately estimate the model count (Chakraborty, Meel, and Vardi 2013; Chakraborty, Meel, and Vardi 2016; Soos and Meel 2019; Soos, Gocht, and Meel 2020). The state-of-the-art hashing-based counter, ApproxMC, has shown to work well in practice. It also supports preprocessing techniques based on independent set detection and scales better when Arjun (Soos and Meel 2022) provides a small independent set.

In the context of this survey, we focus on the six state-of-the-art model counters that have performed well in model counting competitions over the past year. The first five are top-down compilation techniques, with different technical improvements on top of the algorithm.

1. SharpSAT-TD: Developed on top of SharpSAT, combined with a tree-decomposition-based heuristics.
2. Ganak: Developed on top of SharpSAT, enhanced with probabilistic component caching.
3. D4: A Decision-DNNF compilation based on dynamic decomposition.
4. GPMC: Another top-down compilation-based counter, which uses optimizations for projected counting.
5. ExactMC: Another top-down compilation-based model counter using CCDD for compilation.
6. ApproxMC: A hashing-based approximate model counter.

Among these counters, ExactMC and SharpSAT-TD solve the problem of only non-projected model counting. The remaining solvers can solve the problems of both projected and non-projected model counting. We compare the performance of all the solvers in the categories in which they can solve the problem.

4 Benchmarks

We selected a large set of benchmarks from various practical domains to evaluate the model counters. Below is a list of these domains:

1. **Software Verification.** In software verification, some quantitative problems are solved by reducing the problems to model counting. Here are two such problems:
   (a) *Reliability Estimation.* When the functional correctness of a program cannot be established, a potential approach to assess the software's reliability is to quantify it as the ratio of failing program runs to all terminating runs. Teuber and Weigl (2021) reduced this approach to model counting, where the model count corresponds to the number of inputs that trigger or bypass assertions or assumptions.
   (b) *Robust Reachability.* Determining the extent to which a bug can be replicated is frequently relevant. Girol, Farinier, and Bardin (2021) addressed this issue by employing the formalism of robust reachability. They also introduced the concept of quantitative robust reachability, which seeks to identify a controlled input that maximizes the number of uncontrolled inputs capable of reaching the intended target. Model counting can be used to lower bound the runtime cost by the cost of determining the number of uncontrolled inputs satisfying a path constraint for a given controlled input.
2. **Probabilistic Inference.** Model counting is used to solve the problem of probabilistic inference. Sang, Beame, and Kautz (2005) encoded the inference problem on Boolean Bayesian networks as a model counting problem.
3. **Network Reliability.** For critical infrastructure like power transmission grids, it is important to know the reliability of the infrastructure. Kabir and Meel (2023) encoded the problem of network reliability as a weighted model counting problem. They then used chain formulas (Chakraborty et al. 2015) to encode the problem as unweighted model counting problems. These benchmarks encode power transmission grids from different cities and states. The number of solutions to these formulas relates to the network reliability.
4. **Cryptography.** Certain problems in cryptography can also be tackled with model counting. Beck, Zinkus, and Green (2020) used model counting to automate the development of chosen-ciphertext attacks.
5. **Synthesis.** Given the specification of a function or program, the task of synthesis is to generate the function or program.
   (a) *Program and Function Synthesis.* Some algorithms for synthesis (Golia, Roy, and Meel 2020) use model counts in certain parts. These benchmarks consist of instances where the specifications of the functions like arithmetic, disjunctive instances etc.
   (b) *Synthesis for Control Improvisation.* The control improvisation (CI) framework helps synthesize randomized systems with strict and flexible constraints. Gittis, Vin, and Fremont (2022) includes quantitative constraints on

benchmark	Treewidth	Support Size
Robust Reachability	9.13	1565.38
Bayes Net	15.52	2410.69
Industrial Config	18.29	749.45
Linux Config	21.50	799.32
Network Reliability	34.33	941.58
Control Improvisation	35.00	58.27
Cryptographic	41.49	292.97
Software Reliability	55.49	1111.65
Information Flow	66.84	10268.38
Functional Synthesis	83.57	681.82
Neural Net Verification	N.A.	71.26

Table 1: Average Treewidth and Independent Support Size.

expected costs and randomness constraints for diversity based on labels and encodes the problem as a model counting problem.

6. **Feature Counting.** Product lines efficiently manage groups of products sharing a core set of features. Determining the number of valid configurations is often a crucial task.
   (a) *Industrial Product Lines.* Product lines are commonly employed to handle families of products sharing a core set of features, with feature models serving as a standard to define valid feature combinations. However, not all feature configurations are permissible. These models facilitate standardized analyses of the system's variability, and many of these analyses require calculating the number of valid configurations. Sundermann et al. (2023) surveyed these problems as a model counting problem.
   (b) *Configuration Spaces of Software Systems.* Kuiter et al. (2022) studied the problem of feature modeling, which helps systematically model features and dependencies in software systems. The authors encode feature models into propositional formulas, where the number of solutions of the formula corresponds to the number of possible features in a software system.
7. **Quantitative Verification of Neural Networks.** An intriguing aspect of neural network verification is assessing the frequency with which a specific property is valid. The NPAQ (Neural Property Approximate Quantifier) framework, introduced by Baluta et al. (2019), facilitates the evaluation of various properties in binarized neural networks. The benchmarks test the following properties on the MNIST and UCI datasets: *fairness*, which encodes bias towards marital status, race, or sex; *robustness*, which measures the impact of 2-3-bit adversarial input perturbations; and *Trojan attacks*, which account for the number of inputs with a trojan pattern. These benchmarks were initially encoded as pseudo-Boolean constraints and later converted to CNFs, thereby encoding many arithmetic circuits.
8. **Quantitative Information Flow.** Information leaks in modern software systems are an important problem. Bang et al. (2016) introduced an analysis method that estimates both minimum and maximum leak amounts, even when some paths aren't fully explored. This method was added to KLEE to analyze information leaks in C programs.

Among these benchmark sets, functional synthesis, reliability estimation, control improvisation, and neural network verification benchmarks consist of projected counting instances, while the remaining are non-projected model counting problems.

5 Experimental Evaluation

To evaluate the performance and effectiveness of the various tools discussed in Section 3, we conducted the following experiments.

Experimental Setup. The experiments were carried out on a supercomputing cluster equipped with AMD EPYC 7713 CPUs. Each experiment involved running a tool on a specific benchmark using a single core with a memory limit of 16 GB. For approximate counters, we set $\varepsilon = 0.8$ and $\delta = 0.2$. We adhered to the competition standard timeout of 3600 seconds. Initial experiments with a higher timeout showed a minimal increase in the number of instances solved. For the experiments, we used the versions of the counters submitted to the Model Counting Competition 2023.

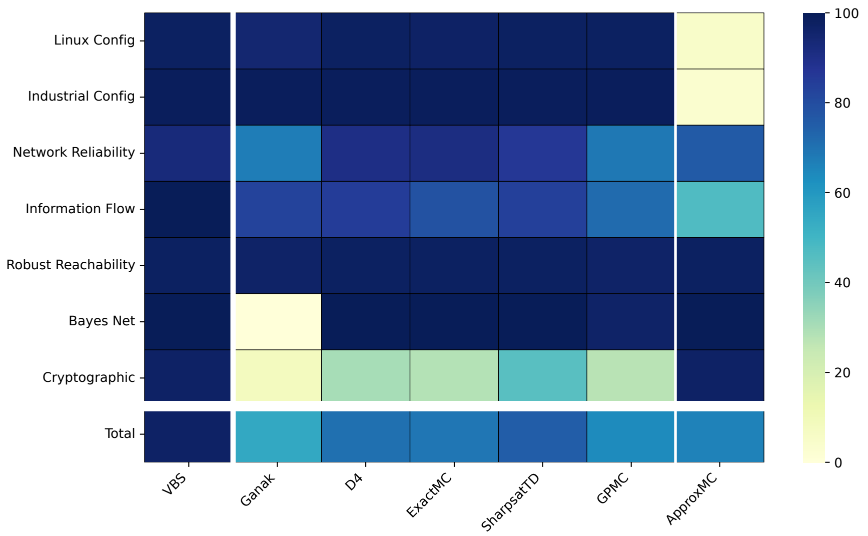
Virtual Best Solver. We included the results of a Virtual Best Solver (VBS) in our comparisons. A VBS is a hypothetical solver that performs well and is the best method for each benchmark. If solvers $s_1, \dots s_n$ solve a problem in time $t_1, \dots t_n$ seconds, then VBS solves it in $\min(t_1, \dots, t_n)$ sec.

Correctness. The correctness of the model counters is well-established; in all model counting competitions, the counters consistently produce correct counts. Additionally, approximate model counters provide counts with deficient error. Therefore, we assume the counters are correct and do not focus on this aspect.
Since not all model counters can handle projected model counting, we analyzed projected and non-projected instances separately.
In this work, we sought to answer the following research questions:

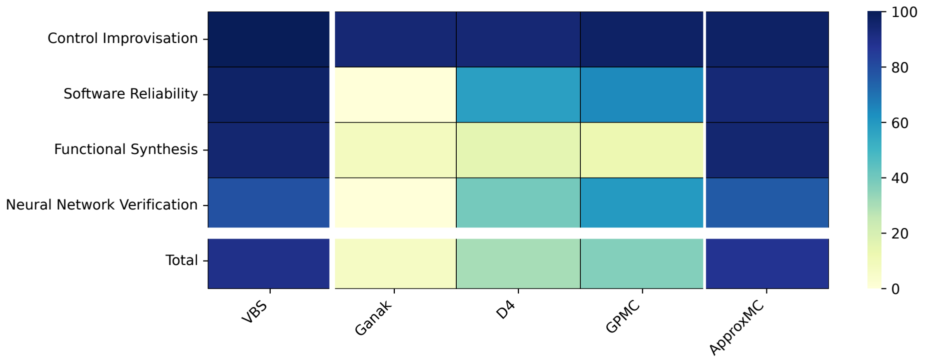
RQ1. How do the overall performances of different model counters compare, and how do they vary across different sets of benchmarks?

RQ2. How do benchmark parameters relate to the performance of various solvers?

Summary of Results. The highest number of non-projected instances solved by a single solver was achieved by SharpSAT-TD, which successfully solved 811 out of 1080 instances. For projected instances, ApproxMC demonstrated the best performance, solving 1041 out of 1182 instances. Compilation-based and hashing-based model counters excelled in solving different sets of benchmarks, and their performances were often complementary. This complementary performance resulted in a much better performance of the VBS, which solved 2106 out of 2262 instances. Treewidth



(a) Non-projected instances



(b) Projected instances

Figure 1: Heatmap of the percentage of instances solved by model counters for

correlates with the performance of compilation-based counters, while independent support size correlates weakly with the hashing-based counter.

5.1 RQ1: Evaluation on Different Benchmark Sets

We evaluate the performance of the solvers on different benchmark sets by analyzing the total number of instances solved, examining each benchmark set individually, and considering the performance of the virtual best solver.

Total Benchmarks Solved First, we analyze the total number of instances solved for both non-projected and projected instances.

Non-projected Instances. Table 3 presents the number of problems solved by various model counters across different benchmark sets. When aggregating all benchmarks, SharpSAT-TD exhibits the best performance, solving 75% of the benchmarks (811 out of 1080). D4 and ExactMC also perform well, solving 762 and 745 instances, respectively. In contrast, ApproxMC solves 715 instances, performing relatively less effectively.

Projected Instances. Table 2 presents the number of instances solved by various model counters on projected model counting problems. Among the 1182 instances, ApproxMC performs the best, solving 1041 instances. The other projected model counters do not perform as well, with GPMC and D4 solving 434 and 361 instances, respectively, placing them second and third.

Runtime Variation Table 2 and Table 3 present the number of problems solved by different model counters across various benchmark sets, highlighting significant performance variations among them. The key observations are as follows:

1. Hashing-based counters perform exceptionally well on specific benchmark sets, particularly in cryptographic benchmarks, functional synthesis benchmarks, and neural network verification benchmarks.

	Total	VBS	Ganak	D4	GPMC	ApproxMC
Control Improvisation	33	33	31	31	**32**	**32**
Software Reliability	123	119	0	71	79	**115**
Control Improvisation	33	33	31	31	**32**	**32**
Functional Synthesis	609	577	50	94	74	**577**
Neural Network Verification	417	329	0	165	249	**317**
Total	1182	1058	81	361	434	**1041**

Table 2: Number of instances solved by different model counters on projected instances.

	Total	VBS	Ganak	D4	ExactMC	SharpsatTD	GPMC	ApproxMC
Linux Config	135	130	126	**130**	129	**130**	**130**	7
Industrial Config	128	126	**126**	**126**	**126**	**126**	**126**	4
Network Reliability	256	177	129	173	**174**	165	131	145
Information Flow	117	106	88	**90**	83	89	76	50
Robust Reachability	93	91	90	**91**	**91**	**91**	90	**91**
Bayes Net	29	29	0	**29**	**29**	**29**	28	**29**
Cryptographic	411	389	34	123	113	181	110	**389**
Total	1169	1048	593	762	745	**811**	691	715

Table 3: Number of instances solved by different model counters on non-projected instances.

2. Compilation-based counters excel in certain benchmark sets, such as Linux configuration, industrial configuration feature counting benchmarks, and quantitative information flow benchmarks. The performance differences among the search-based counters are minimal.
3. For the remaining benchmark sets, nearly all counters perform very well.

In Table 2 and Table 3, the dashed lines separate the benchmark sets based on which type of counter performs best.

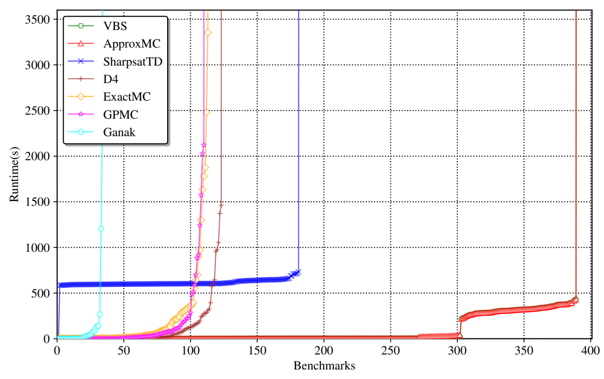
Figure 1 provide a heatmap representation of the percentage of problems each model counter has successfully solved. Each cell in the heatmap indicates the percentage of benchmarks solved by a solver in a specific class of benchmarks. The color scale is shown to the right, with darker colors corresponding to a higher percentage of instances solved.

The cactus plots in Figure 2 and Figure 3 provide additional insight into performance variations. In these plots, the x-axis represents the number of instances, while the y-axis indicates the time taken. A point (i, j) on the plot signifies that a solver completed j benchmarks out of the total set in less than or equal to i seconds. The key insights from the figures are as follows:

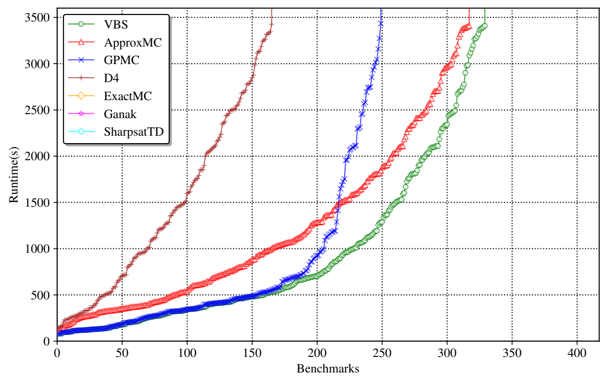
1. In Figure 2 (a), cryptographic instances are either solved within seconds or not at all. The VBS closely follows ApproxMC, indicating that ApproxMC typically has minimal runtime in most cases.
2. A different pattern emerges for neural network verification in Figure 2 (b). Here, the counters take a considerable amount of time to solve the instances. The VBS closely follows the curve of GPMC for approximately 700 seconds, suggesting that GPMC can solve around 200 instances more quickly within this timeout. Between 1000 and 3600 seconds, ApproxMC gradually solves around 150 additional instances, a trend not observed in any other benchmark set.
3. In Figure 2 (d), the information flow benchmarks exhibit another interesting behavior, with counters taking varying times between 0 and 2000 seconds to solve instances, managing to solve a maximum of 90 out of 106 instances. However, the VBS solves all the instances within 200 seconds.

The other cactus plots also show similar patterns. We have not included the cactus plots for robust reachability, control improvisation, and Bayes net benchmark sets because, in these sets, all the counters solve the instances within seconds. Similarly, we skipped the industrial and Linux configuration benchmarks since, in these cases, the compilation-based counters solve all the instances within seconds. In contrast, the hashing-based counter solves a negligible number of instances.

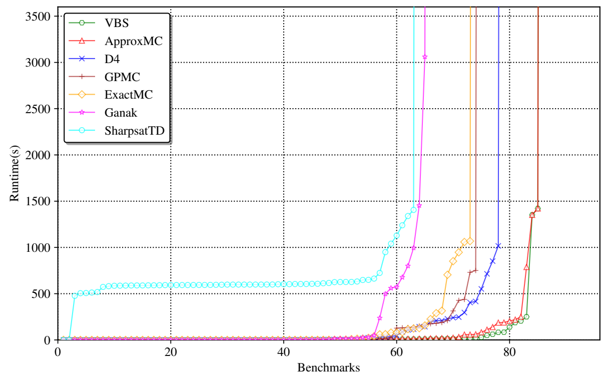
Virtual Best Solver The VBS can solve significantly more instances than any individual solver. For the non-projected instances, out of 1080 instances, the VBS can solve 1048 instances. This number is much higher than any individual counter; the best counter for non-projected benchmarks is SharpSAT-TD, which solves 811 instances,



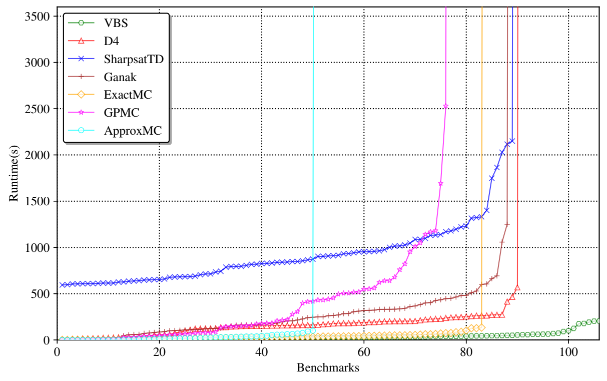
(a) Cryptographic Benchmarks



(b) Neural Net Verification Benchmarks



(c) Software Reliability Benchmarks



(d) Information Flow Benchmarks

Figure 2: Cactus plot of the numbers of benchmarks solved by model counters on different benchmark sets.

Counter	Contribution
ApproxMC	1114
ExactMC	372
GPMC	368
Ganak	186
D4	67
SharpSAT-TD	0
Total	2106

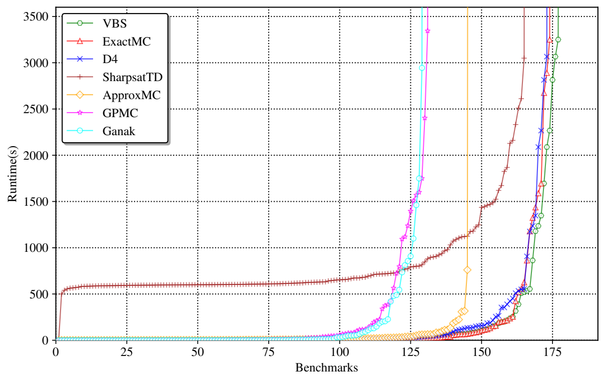
Table 4: Contribution of the solvers to the VBS.

representing 77% of the instances solved by the VBS. In the case of projected instances, the trend continues, with the VBS demonstrating superior performance.

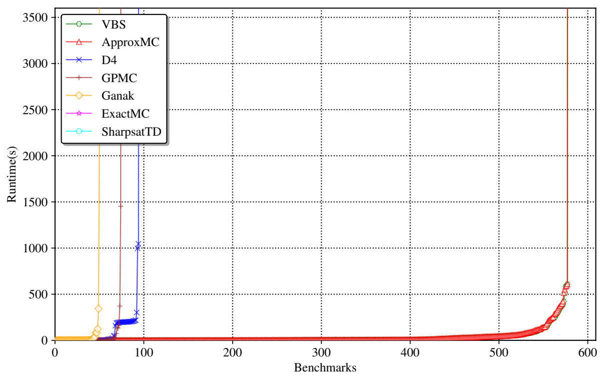
Contribution of Counters to the VBS. In Table 4, we show the number of instances contributed to the VBS by each model counter. ApproxMC makes the most substantial contribution to the VBS, accounting for 1114 out of 2106 instances. The following most significant contributors are ExactMC and GPMC, contributing 372 and 368 instances, respectively. Interestingly, ExactMC only solves non-projected instances. SharpSAT-TD does not contribute any instances to the VBS, likely because it requires a constant 600 seconds to run the tree-decomposition component before starting the actual counting, resulting in longer execution times compared to other counters.

5.2 RQ2. Correlation with Benchmark Parameters

While the primary observation with model counters on different types of benchmarks was that the performance varies significantly across benchmarks, we sought to identify the underlying parameters from a formula that influences the difficulty of model counting. We consider *treewidth* and size of *independent support set* of a formula as parameters to predict which count would be efficient to count the formula. Computing the values of each parameter is a computationally hard problem; therefore, we heuristically determine these values. We use FlowCutter (Strasser 2017) for computing the treewidth and Arjun for calculating the independent support size. In Table 1, we list the average treewidth and independent support size for each benchmark set. For neural net verification instances, FlowCutter timed out, which we denote by N.A. in the table.

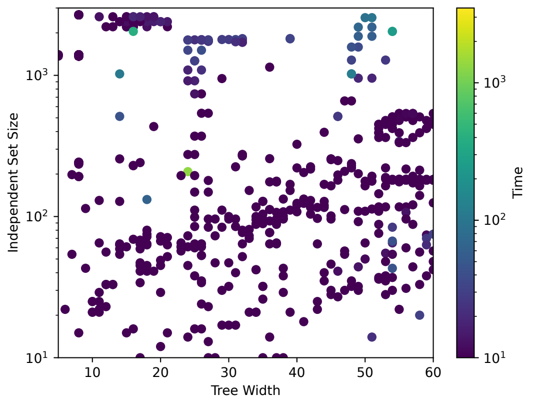


(a) Network Reachability Benchmarks

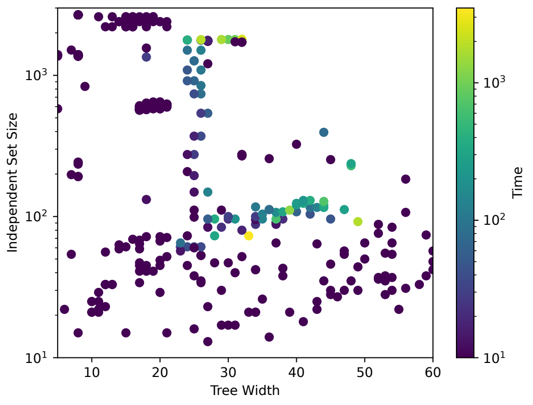


(b) Functional Synthesis Benchmarks

Figure 3: Cactus plot of the numbers of benchmarks solved by model counters on different benchmark sets.



(a) ApproxMC



(b) ExactMC

Figure 4: Correlation of runtime, independent support size and treewidth (best seen in color).

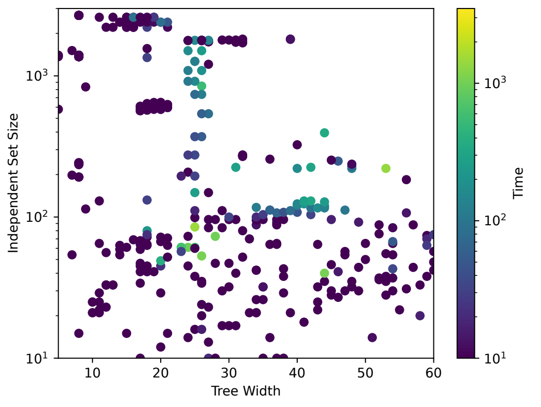
	Treewidth	Support Size
Ganak	0.38	-0.22
D4	0.41	-0.20
ExactMC	0.42	-0.18
SharpSAT-TD	0.48	-0.12
GPMC	0.47	-0.09
ApproxMC	-0.03	0.27

Table 5: Correlation between solver runtime and formula features. (Value ranges from -1 to 1, 0 is no linear correlation.)

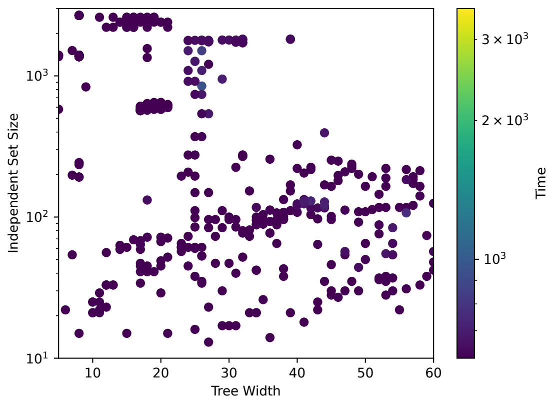
In Table 5, we present the Pearson correlation between the different benchmark parameters and the solvers' runtimes. The value ranges between -1 and 1, where a greater absolute value indicates a higher correlation. The key observations are as follows:

1. The performance of knowledge-compilation-based model counters has a positive correlation with treewidth, while hashing-based counters do not exhibit such a correlation. Among all counters, SharpSAT-TD shows the highest correlation of 0.48 between treewidth and runtime.
2. The runtime of the hashing-based counter ApproxMC shows a weak positive correlation of 0.27 with independent support size, whereas the compilation-based counters show no correlation.

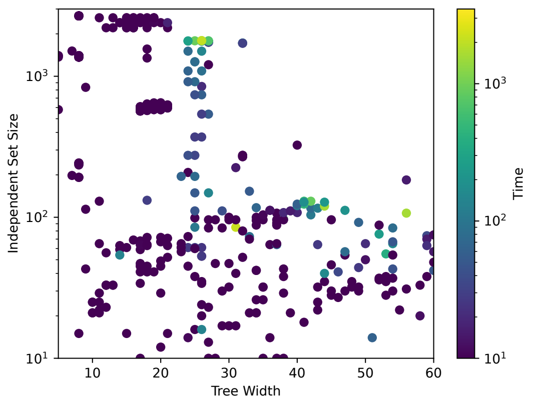
In Figure 4 and 5, we represent the relationship among treewidth, independent support size, and the runtime of a specific solver in a heatmap. The x-axis represents the treewidth, and the y-axis represents the independent support size. Thus,



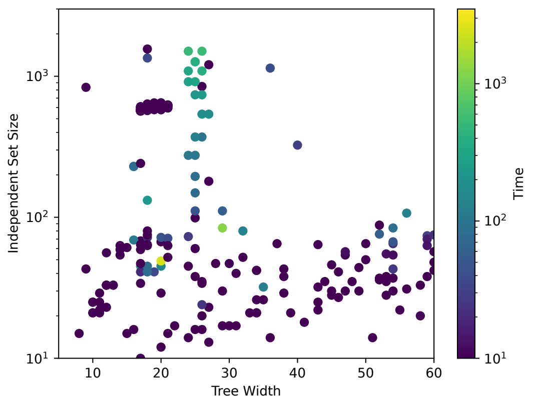
(a) D4



(b) SharpSAT-TD



(c) GPMC



(d) Ganak

Figure 5: Correlation of runtime, independent support size and treewidth (best seen in color).

a point with coordinates (i, j) represents an instance with treewidth i and independent support size j. The color of the point indicates the runtime for the solver depicted in the graph. The color scale is shown to the right, with darker colors generally indicating shorter runtimes. These heatmaps shed more light on the lack of correlation between the performance of the counters and the benchmark parameters. The heatmaps reveal the following insights:

1. In Figure 4 (a), ApproxMC takes approximately 10^3 seconds to solve instances of treewidth 10, while most of the instances with treewidth between 50 and 60 are solved in less than 100 seconds. The relationship between independent support size and solving time is also not very clear. There are many instances with independent support sizes above 1000 that are solved in less than 10 seconds.
2. The performance of ExactMC in Figure 4 (b) is very different. For instances with treewidth higher than 20, it gradually takes an increasing amount of time. Instances with higher treewidth and higher independent support sizes seem to be particularly challenging for ExactMC. If the treewidth is greater than 50 and the independent support size is greater than 100, the time taken to solve is generally more than 200 seconds.
3. The results for D4 in Figure 5 (a) and GPMCin Figure 5 (c) are not very different from those of ExactMC.
4. The behavior of SharpSAT-TD, however, appears a little different in Figure 5 (b). It solves instances with higher treewidth relatively faster, while instances with treewidth greater than 30 and independent support size greater than 100 take more time to solve.

6 Conclusion

We conducted a comprehensive study on a diverse set of benchmarks, revealing that different solvers excel on different subsets. Our findings indicate that the virtual best solver can solve nearly all instances, a feat unattainable by any individual solver alone. This performance of VBS demonstrates that the complementary strategies employed by various solvers enable them to address distinct sets of instances effectively. Consequently, our study underscores the significance of integrating these approaches or developing a portfolio-based solver model.

Acknowledgments

We are thankful to Jiong Yang and Yash Pote for the many useful discussions and grateful to the anonymous reviewers for their constructive comments on improving this paper. We thank Gabrielle Beck, Guillaume Girol, Samuel Teuber, and Maximilian Zinkus for providing us with different sets of benchmarks. This work was supported in part by the National Research Foundation Singapore under its NRF Fellowship Programme [NRF-NRFFAI1-2019-0004] and Natural Sciences and Engineering Research Council of Canada (NSERC) [RGPIN-2024-05956]. Part of the work was done when Arijit Shaw was a visiting graduate student at the University of Toronto. The computational work for this article was performed on resources of the National Supercomputing Centre, Singapore (https://www.nscc.sg).

References

Baluta, T.; Shen, S.; Shinde, S.; Meel, K. S.; and Saxena, P. 2019. Quantitative verification of neural networks and its security applications. In *Proc. of CCS*.

Bang, L.; Aydin, A.; Phan, Q.-S.; Păsăreanu, C. S.; and Bultan, T. 2016. String analysis for side channels with segmented oracles. In *Proc. of ICSE*.

Beck, G.; Zinkus, M.; and Green, M. 2020. Automating the development of chosen ciphertext attacks. In *Proc. of USENIX Security*.

Chakraborty, S.; Fried, D.; Meel, K. S.; and Vardi, M. Y. 2015. From weighted to unweighted model counting. In *Proc. of IJCAI*.

Chakraborty, S.; Meel, K. S.; and Vardi, M. Y. 2013. A scalable approximate model counter. In *Proc. of CP*.

Chakraborty, S.; Meel, K. S.; and Vardi, M. Y. 2016. Algorithmic Improvements in Approximate Counting for Probabilistic Inference: From Linear to Logarithmic SAT Calls. In *Proc. of IJCAI*.

Darwiche, A. 2004. New advances in compiling cnf to decomposable negation normal form. In *Proc. of ECAI*.

Dudek, J. M.; Phan, V. H.; and Vardi, M. Y. 2020. ADDMC: Weighted model counting with algebraic decision diagrams. *Proc. of AAAI*.

Fichte, J. K.; Hecher, M.; and Hamiti, F. 2020. The model counting competition 2020. *arXiv preprint arXiv:2012.01323*.

Girol, G.; Farinier, B.; and Bardin, S. 2021. Not all bugs are created equal, but robust reachability can tell the difference. In *Proc. of CAV*.

Gittis, A.; Vin, E.; and Fremont, D. J. 2022. Randomized synthesis for diversity and cost constraints with control improvisation. In *Proc. of CAV*.

Golia, P.; Roy, S.; and Meel, K. S. 2020. Manthan: A data-driven approach for boolean function synthesis. In *Proc. of CAV*.

Gomes, C. P.; Sabharwal, A.; and Selman, B. 2006. Model counting: A new strategy for obtaining good bounds. In *Proc. of AAAI*.

Kabir, M., and Meel, K. S. 2023. A fast and accurate asp counting based network reliability estimator. In *Proc. of LPAR*.

Korhonen, T., and Järvisalo, M. 2021. Integrating tree decompositions into decision heuristics of propositional model counters. In *Proc. of CP*.

Kuiter, E.; Krieter, S.; Sundermann, C.; Thüm, T.; and Saake, G. 2022. Tseitin or not tseitin? the impact of cnf transformations on feature-model analyses. In *Proc. of CASE*.

Lagniez, J.-M., and Marquis, P. 2017. An improved decision-dnnf compiler. In *Proc. of IJCAI*.

Lagniez, J.-M.; Lonca, E.; and Marquis, P. 2016. Improving model counting by leveraging definability. In *Proc. of IJCAI*.

Lai, Y.; Meel, K. S.; and Yap, R. H. 2021. The power of literal equivalence in model counting. In *Proc. of AAAI*.

Sang, T.; Beame, P.; and Kautz, H. 2005. Solving bayesian networks by weighted model counting. In *Proc. of AAAI*.

Sang, T.; Bacchus, F.; Beame, P.; Kautz, H. A.; and Pitassi, T. 2004. Combining component caching and clause learning for effective model counting. *Journal of SAT*.

Sharma, S.; Roy, S.; Soos, M.; and Meel, K. S. 2019. Ganak: A scalable probabilistic exact model counter. In *Proc. of IJCAI*.

Soos, M., and Meel, K. S. 2019. BIRD: engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting. In *Proc. of AAAI*.

Soos, M., and Meel, K. S. 2022. Arjun: An efficient independent support computation technique and its applications to counting and sampling. In *Proc. of ICCAD*.

Soos, M.; Gocht, S.; and Meel, K. S. 2020. Tinted, detached, and lazy CNF-XOR solving and its applications to counting and sampling. In *Proc. of CAV*.

Strasser, B. 2017. Computing tree decompositions with flowcutter: PACE 2017 submission. *arXiv preprint arXiv:1709.08949*.

Sundermann, C.; Heß, T.; Nieke, M.; Bittner, P. M.; Young, J. M.; Thüm, T.; and Schaefer, I. 2023. Evaluating state-of-the-art# sat solvers on industrial configuration spaces. *Empirical Software Engineering*.

Teuber, S., and Weigl, A. 2021. Quantifying software reliability via model-counting. In *Proc. of QEST*.

Thurley, M. 2006. sharpSAT–counting models with advanced component caching and implicit BCP. In *Proc. of SAT*.

Toda, S. 1989. On the computational power of pp and (+) p. In *Proc. of FOCS*.

Valiant, L. G. 1979. The complexity of enumeration and reliability problems. *SIAM Journal on Computing* 8(3).