Formally Certified Approximate Model Counting

Citation for the original published paper (version of record):
Tan, Y., Yang, J., Soos, M. et al (2024). Formally Certified Approximate Model Counting. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 14681 LNCS: 153-177. http://dx.doi.org/10.1007/978-3-031-65627-9_8

Formally Certified Approximate Model Counting

Yong Kiam Tan[1(✉)], Jiong Yang[2], Mate Soos[2], Magnus O. Myreen[3], and Kuldeep S. Meel[4]

[1] Institute for Infocomm Research (I2R), A*STAR, Singapore, Singapore
tanyk1@i2r.a-star.edu.sg
[2] National University of Singapore, Singapore, Singapore
jiong@comp.nus.edu.sg, soos.mate@gmail.com
[3] Chalmers University of Technology, Gothenburg, Sweden
myreen@chalmers.se
[4] University of Toronto, Toronto, Canada
meel@cs.toronto.edu

Abstract. Approximate model counting is the task of approximating the number of solutions to an input Boolean formula. The state-of-the-art approximate model counter for formulas in conjunctive normal form (CNF), ApproxMC, provides a scalable means of obtaining model counts with *probably approximately correct* (PAC)-style guarantees. Nevertheless, the validity of ApproxMC's approximation relies on a careful theoretical analysis of its randomized algorithm and the correctness of its highly optimized implementation, especially the latter's stateful interactions with an incremental CNF satisfiability solver capable of natively handling parity (XOR) constraints.

We present the first certification framework for approximate model counting with formally verified guarantees on the quality of its output approximation. Our approach combines: (i) a *static*, once-off, formal proof of the algorithm's PAC guarantee in the Isabelle/HOL proof assistant; and (ii) *dynamic*, per-run, verification of ApproxMC's calls to an external CNF-XOR solver using proof certificates. We detail our general approach to establish a rigorous connection between these two parts of the verification, including our blueprint for turning the formalized, randomized algorithm into a verified proof checker, and our design of proof certificates for both ApproxMC and its internal CNF-XOR solving steps. Experimentally, we show that certificate generation adds little overhead to an approximate counter implementation, and that our certificate checker is able to fully certify 84.7% of instances with generated certificates when given the same time and memory limits as the counter.

Keywords: approximate model counting · randomized algorithms · formal verification · proof certification

Y. K. Tan and J. Yang—The first two authors contributed equally.

A. Gurfinkel and V. Ganesh (Eds.): CAV 2024, LNCS 14681, pp. 153–177, 2024.
https://doi.org/10.1007/978-3-031-65627-9_8

1 Introduction

State-of-the-art automated reasoning solvers are critical software systems used throughout formal methods. However, even skilled and trusted developers of such tools can inadvertently introduce errors. Two approaches have evolved to provide assurances that automated reasoning tools behave as intended. The first involves the use of theorem provers to formally verify the correctness of solver implementations [20,30]. This approach guarantees correct outputs for all inputs, but struggles to scale to complex systems such as SAT solvers. The second approach is based on *certifying algorithms* [38], where a solver is required to produce a certificate alongside its output [6,10,24,35,37,53,58]. A *certificate* checker (also called *proof* checker)—which is often formally verified—then checks the correctness of this certificate, ensuring that the system's output adheres to the desired specifications. This latter method has gained significant traction in the SAT solving community, wherein a SAT solver either returns a satisfying assignment that is easy to check through evaluation or a proof of unsatisfiability as a certificate [58]. However, neither of these approaches have been applied to probabilistic systems that rely on *randomized* algorithms. In fact, McConnell et al. [38] argue that randomized algorithms resist deterministic certification.

In this paper, we propose a hybrid approach that harnesses the power of both theorem-proving and certificate-based approaches to certify probabilistic systems. We present our approach on ApproxMC, a probabilistic automated reasoning system which computes approximate model counts for Boolean formulas. Model counting is a fundamental problem in computer science that serves as a key component in a wide range of applications including control improvisation [22], network reliability [14,56], neural network verification [5], probabilistic reasoning [11,18,45,46], and so on. Therefore, it is crucial that the results computed by an approximate model counter, such as ApproxMC, can be trusted.

Two key questions must be tackled by our approach. First, what does it mean to trust a *random* run of ApproxMC? Here, we propose a *verification modulo randomness* approach, i.e., our certification results are modulo a trusted random bit generator. Second, how do we handle the huge volume of (incremental) CNF-XOR satisfiability solver calls which are tightly integrated in ApproxMC [49,50]? Here, we design the certificate format to require only the results of solver calls that are crucial for ApproxMC's correctness. In particular, ApproxMC makes $\mathcal{O}(\varepsilon^{-2} \cdot \log n \cdot \log \delta^{-1})$ many calls to its solver, where n is the number of (projected) variables of the formula, ε is the tolerance parameter, and δ is the confidence parameter (see Sect. 3 for definitions); our crucial insight is that to certify ApproxMC, we only need to check the correctness of $\mathcal{O}(\log \delta^{-1})$ UNSAT calls, which is independent of n. We then observe that existing CNF-XOR UNSAT checkers fail to scale to formulas that are handled by ApproxMC. To this end, we adapt existing solving and verified proof checking pipelines to natively support proof certificates for CNF-XOR unsatisfiability. With this design, our framework is able to independently check certificates generated by a state-of-the-art (but untrusted) implementation of ApproxMC, with *all* of the latter's optimizations enabled. Overall, the key idea is to combine a *static*, once-off, formal proof of

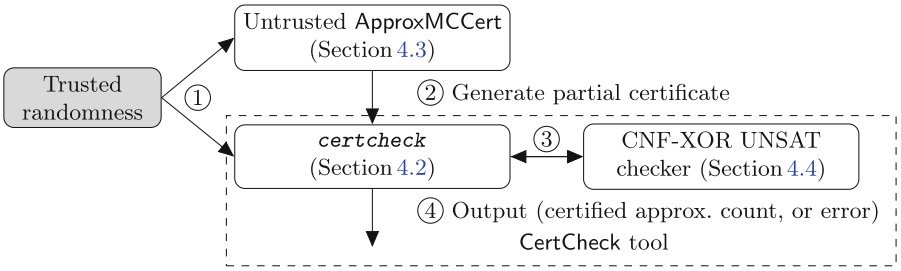

Fig. 1. The certified approximate model counting workflow.

the algorithm's correctness guarantee in Isabelle/HOL [42,43] with *dynamic*, per-run, certification of ApproxMC's calls to an external CNF-XOR solver.

In summary, our contributions are as follows:

1. An abstract specification of ApproxMC and a formal proof of its probably approximately correct (PAC) guarantee in Isabelle/HOL (Sect. 4.1).
2. A refinement of the abstract specification to a concrete certificate format and checker implementation for ApproxMC (Sects. 4.2 and 4.3).
3. Updates to various tools to realize a formally verified proof checking pipeline with native support for CNF-XOR unsatisfiability (Sect. 4.4).
4. Empirical evaluation of the framework on an extensive suite of model counting benchmarks to demonstrate its practical utility (Sect. 5).

Our workflow for certified approximate model counting is shown in Fig. 1. In step ①, it uses a trusted external tool to generate uniform random bits which are handed to an *untrusted* certificate generator ApproxMCCert and to the verified certificate checker CertCheck (extracted from Isabelle/HOL); the random bits are used identically by ApproxMCCert and CertCheck to generate random XOR constraints as part of the counting algorithm. For step ②, ApproxMCCert generates a *partial* certificate which is subsequently checked in step ③ by CertCheck; the certificate is partial because it does not contain CNF-XOR unsatisfiability proofs. Instead, CertCheck calls an external CNF-XOR unsatisfiability checking pipeline (with verified proof checking in CakeML [36,53]). In the final step ④, an approximate model count is returned upon successful certification.

As part of our commitment to reproducibility, all code and proofs have been made available with a permissive open-source license [2,21,54].

Impact. Although our main objective was to enhance end-user trust in answers to their counting queries, undertaking this project led to unexpected benefits that are worth highlighting. While modifying ApproxMC's underlying solver, `CryptoMiniSat` [52], to emit certificates (Sect. 4.4), a bug in `CryptoMiniSat`'s XOR manipulation system was discovered. The bug was introduced during the development of part of the BIRD system [50] that keeps *all* XOR constraints' clausal versions (as well as their compact XOR versions) in-memory at all times.

This allows a substantial level of interaction between XOR and clausal constraints. However, it also led to large overhead in terms of the often hundreds of thousands of clauses needed to encode the XORs in their clausal form. The compromise made by the developers was to detach the clausal representation of XORs from the watchlists. However, that seemed to have led to a level of complexity that both allowed the bug to occur, and more importantly, made it impossible to discover via `CryptoMiniSat`'s standard fuzzing pipeline. Our version of `CryptoMiniSat` fixes this by not keeping around a clausal encoding of all XORs, instead introducing (and deleting) them whenever needed for the proof.

Furthermore, we have also found minor flaws in the theoretical analysis of ApproxMC (see discussion of `events_prob`) and in the implementation, e.g., the sampling of random bits was slightly biased, and an infinite loop could be triggered on certain random seeds. None of these bugs were known to the authors of ApproxMC or were previously reported by users of the tool. All of these issues have been fixed and upstreamed to their tools' respective codebases.

2 Related Work

This discussion is focused on formally verified algorithms and proof checkers. Readers are referred to Chakraborty et al. [13] and references therein for related literature on approximate model counting.

Certified Model Counting. Prior research on certificate-based approaches focuses on deterministic methods in model counting. Prior work on certified *exact* model counting focuses either on the development of proofs, such as MICE [19] and CPOG [10], along with their respective toolchains, or on analyzing the complexity of the proof system [7]. Some efforts have been directed toward certifying deterministic approximate counting algorithms which, however, require access to a Σ_2^{P} oracle and did not yield practical implementations [40]. Our work develops the first certification framework for randomized approximate model counting.

Formalization of Randomized Algorithms. Various randomized algorithms have been formally analyzed in Isabelle/HOL, including randomized quicksort, random binary tree data structures [15], and approximation of frequency moments in data streams [31–34]. These prior efforts as well as ours, all build upon the foundations for measure and probability theory in Isabelle/HOL [16,28]. Properties of approximate membership query structures (including Bloom filters) have been verified in Coq [26]. Pioneering work on formal verification of randomized algorithms, including the Miller-Rabin primality test, was carried out by Joe Hurd in HOL4 [29]. A common objective of these prior efforts, and that of ours, is to put the guarantees of randomized algorithms on formal foundations.

Verified Proof Checking. Formally verified proof checkers have been developed for several (deterministic) algorithms and theories, such as the CNF unsatisfiability checkers used by the SAT community [27,37,53]. Within Isabelle/HOL,

the Pastèque tool [35] checks proofs in the practical algebraic calculus, which can be used to validate algebraic reasoning; the CeTA tool [55] is based on an extensive library of results for certifying properties of rewriting systems; and the LEDA project developed specialized proof checkers for graph algorithms [1]. CoqQFBV [47] is similar in design to our approach in that a higher-level Coq-generated tool for verified bit-blasting is used in concert with a lower-level verified proof checker for CNF formulas.

CNF-XOR Unsatisfiability Checking. Given ApproxMC's reliance on CNF-XOR formulas, certification of CNF-XOR unsatisfiability emerged as a key challenge in our work. To this end, we provide a brief overview of three prior state-of-the-art approaches for certified CNF-XOR reasoning.

1. The first approach uses proof generation and certification of XOR reasoning based on Binary Decision Diagrams (BDDs) [48]. It uses `CryptoMiniSat` [52], a SAT solver specifically made to work on CNF-XOR instances and `TBUDDY` [9] to produce FRAT proof certificates [3] for `CryptoMiniSat`'s XOR reasoning; `FRAT-rs` [3] is used as the elaboration backend and a verified LRAT proof checker [27,53] can be used to check the elaborated proofs.
2. The second approach, due to Gocht and Nordström [24], relies on pseudo-Boolean reasoning and its associated proof system to justify both CNF and parity reasoning. This approach was demonstrated on MiniSat equipped with an XOR reasoning engine, with VeriPB as a proof checker; pseudo-Boolean proofs are also supported by a verified proof checker [23].
3. The third approach is to rely on the standard SAT solvers accompanied with standard CNF proof formats and (verified) checkers [27,37,53].

3 Background

This section gives a brief introduction to ApproxMC (Sect. 3.1) and to theorem-proving in Isabelle/HOL (Sect. 3.2).

3.1 Approximate Model Counting

Given a Boolean formula F, the *model counting* problem is to calculate the number of models (also called *solutions* or *satisfying assignments*) of F. Model counting is known to be #P-complete, and therefore has been a target of sustained interest for randomized approximation techniques over the past four decades. The current state-of-the-art approximate approach, ApproxMC [12], is a hashing-based framework that relies on reducing the model counting problem to SAT queries, which are handled by an underlying solver. Importantly, ApproxMC is a *probably approximately correct* (PAC) projected model counter, i.e., it takes in a formula F, a projection set $S \subseteq \mathsf{Vars}(F)$, a tolerance parameter $\varepsilon > 0$, and a confidence parameter $\delta \in (0, 1]$, and returns a count c satisfying the PAC guarantee: $\Pr\left[\frac{|\mathsf{sol}(F)_{\downarrow S}|}{1+\varepsilon} \leq c \leq (1+\varepsilon)|\mathsf{sol}(F)_{\downarrow S}|\right] \geq 1 - \delta$, where $|\mathsf{sol}(F)_{\downarrow S}|$ denotes the number of the solutions of F projected on S.

Algorithm 1. ApproxMC $(F, S, \varepsilon, \delta)$

1: thresh $\leftarrow 9.84\left(1+\frac{\varepsilon}{1+\varepsilon}\right)\left(1+\frac{1}{\varepsilon}\right)^2$
2: $Y \leftarrow$ BoundedSAT$(F, S,$ thresh$)$
3: **if** $(|Y| <$ thresh$)$ **then return** $|Y|$
4: $t \leftarrow$ computeIter(δ) ▷ probability amplification using the median method
5: $C \leftarrow$ emptyList, iter $\leftarrow 0$
6: **repeat**
7: iter $\leftarrow$ iter $+ 1$
8: nSols $\leftarrow$ ApproxMCCore$(F, S,$ thresh$)$
9: AddToList$(C,$ nSols$)$
10: **until** (iter $\geq t$)
11: **return** FindMedian(C)

Algorithm 2. ApproxMCCore $(F, S,$ thresh$)$

1: Choose $|S| - 1$ random XOR constraints $X = (X_1, \ldots, X_{|S|-1})$ over S
2: $m \leftarrow$ FindM$(F, S, X,$ thresh$)$ ▷ search for $m \in \{1, \ldots, |S|\}$ using BoundedSAT
3: **if** $(m \geq |S|)$ **then return** $(2^m \times 1)$ ▷ dummy value for failed round
4: $c \leftarrow$ BoundedSAT $(F \wedge X_1 \wedge \cdots \wedge X_m, S,$ thresh$)$;
5: **return** $(2^m \times c)$

An outline of ApproxMC is shown in Algorithms 1 and 2. At a high level, the key idea of ApproxMC is to partition the set of solutions into small cells of roughly equal size by relying on the power of XOR-based hash families [12,25], then randomly picking one of the cells and enumerating all the solutions in the chosen small cell up to a threshold thresh via calls to BoundedSAT$(F, S,$ thresh$)$. The estimated count is obtained by scaling the number of solutions in the randomly chosen cell by the number of cells, and the success probability of this estimation is amplified to the desired level by taking the median result from several trials.

Syntactically, the solution space partition and random cell selection is accomplished by introducing randomly generated XOR constraints of the form $(\bigoplus_{y \in Y} y) = b$ for a random subset $Y \subseteq S$ and random bit b. A crucial fact about random XOR constraints exploited by ApproxMC is their 2-universality when viewed as a hash family on assignments—briefly, given any two distinct Boolean assignments over the variable set S, the probability of each one satisfying a randomly chosen XOR constraint is independent and equal to $\frac{1}{2}$.

Accordingly, the BoundedSAT queries made in Algorithms 1 and 2 are conjunctions of the input formula and random XOR constraints, i.e., CNF-XOR formulas. The current implementation of ApproxMC relies on `CryptoMiniSat` for its ability to handle CNF-XOR formulas efficiently and incrementally [49,50]. Furthermore, the real-world implementation also relies on three key optimizations. **(1)** The search for the correct value of m in Algorithm 2 (FindM) combines a linear neighborhood search, a galloping search, and a binary search [12]. **(2)** The underlying SAT solver is used as a library, allowing to solve under a set of assumptions, a technique introduced as part of MiniSat [17]. This allows the

solver to keep learned lemmas between subsequent calls to `solve()`, significantly improving solving speed, which is especially helpful for proving unsatisfiability. **(3)** To improve the speed of finding satisfying assignments, a solution cache of past solutions is retained [49] which is especially helpful when the optimal number of XORs to add is N, but N+1 have been added and were found to be too much. In these cases, all solutions that are valid for N+1 XORs are also solutions to N XORs and can be reused.

3.2 Formalization in Isabelle/HOL

Notation. All Isabelle/HOL syntax is typeset in *typewriter* font with boldface Isar **keywords**; $\bigwedge$ and $\Longrightarrow$ are the universal quantifier and implication of Isabelle's metalogic, respectively. Type variables are written as *'a, 'b*. The type of (total) functions from *'a* to *'b* is written as *'a ⇒ 'b*, and the type of partial functions, which are only defined on some elements of type *'a*, is *'a ⇀ 'b*. For clarity, we often annotate terms with their type using the notation *term :: type*. For types such as reals, integers, or natural numbers, the interval from *i* to *j* (inclusive) is written as *{i..j}*; the same interval except endpoint *j* is *{i..<j}*. More comprehensive introductions can be found in standard references [4,42].

Locales and Probability. Isabelle/HOL is equipped with *locales* [4], a system of user-declared modules consisting of syntactic parameters, assumptions on those parameters, and module-specific theorems. These modules can be instantiated and inherited, giving users a powerful means of managing mathematical relationships. The following snippet, taken from the Isabelle/HOL standard library, shows an example **locale** declaration for probability spaces followed by an **interpretation** command claiming that the measure space associated with any probability mass function (PMF) *p* is a probability space [28].

```
locale prob_space = finite_measure +
assumes emeasure_space_1: "emeasure M (space M) = 1"
...
interpretation measure_pmf: prob_space "measure_pmf p"
```

Thanks to the locale interpretation, all definitions and theorems associated with probability spaces can be used with PMFs. For example, the probability of an event *A :: 'a set* occurring under *p* is *measure_pmf.prob p A*. The support of PMF *p* is *set_pmf p*, which is *finite* for all PMFs considered in this work.

4 Approximate Model Counting in Isabelle/HOL

This section outlines our formalization of ApproxMC in Isabelle/HOL and its verified certificate checker implementation. The proof follows a refinement-based approach, starting with an abstract mathematical specification of ApproxMC, where its probabilistic approximation guarantees can be formalized without low-level implementation details getting in the way (Sect. 4.1). Then, the abstract

specification is progressively concretized to a verified certificate checker which we call CertCheck (Sect. 4.2) and we extend ApproxMC to ApproxMCCert, a certificate-generating counter (Sect. 4.3). As part of CertCheck, we also built a native CNF-XOR unsatisfiability checker, which is external to Isabelle/HOL, but is also based on formally verified proof checking (Sect. 4.4).

4.1 Abstract Specification and Probabilistic Analysis

Throughout this section, the type *'a* abstracts the syntactic representation of variables. For example, in the DIMACS CNF format, variables are represented with positive numbers, while in other settings, it may be more convenient to use strings as variable names. A solution (or model) *w :: 'a ⇒ bool* is a Boolean-valued function on variables and a projection set *S :: 'a set* is a (finite) set of variables. The main result of this section is formalized in a locale with two parameters *sols*, *enc_xor*, and an assumption relating the two:

```
locale ApproxMC =
fixes sols :: "'fml ⇒ ('a ⇒ bool) set"
fixes enc_xor :: "'a set × bool ⇒ 'fml ⇒ 'fml"
assumes "⋀F xor.
  sols (enc_xor xor F) = sols F ∩ {ω. satisfies_xor xor {x. ω x}}"
```

Here, type *'fml* abstracts the syntactic representation of formulas, *sols F* is the set of all solutions of a formula *F*, and *enc_xor xor F* is a formula whose set of solutions satisfies both *F* and the XOR constraint *xor*. An instantiation of the *ApproxMC* locale would need to provide implementations of *sols*, *enc_xor* and prove that they satisfy the latter assumed property.

The PAC theorem for ApproxMC is formalized as follows:

```
theorem approxmc_prob:
assumes "δ > 0" "δ < 1" "ε > 0" "ε ≤ 1" "finite S"
shows "let sz = real (card (proj S (sols F))) in
  measure_pmf.prob (approxmc F S ε δ n)
    {c. c ∈ {sz / (1 + ε) .. (1 + ε) * sz}} ≥ 1 - δ"
```

Here, *sz* is the true count of projected solutions, i.e., the cardinality of the set *proj S (sols F)*, interpreted as a real number. The conclusion says that *approxmc* returns an ε-approximate count *c* with probability at least *1 - δ*. The argument *n* is a user-specifiable minimum number of iterations of ApproxMCCore calls inside ApproxMC; in practice, a sufficient number of rounds is automatically determined using the median method. Since the *ApproxMC* locale can be instantiated for *any* Boolean theory in which XOR constraints can be syntactically encoded, this theorem shows that the approximate model counting algorithm of Chakrabory et al. [12] works for any such theory.

The rest of this section gives an overview of our proof of *approxmc_prob*. Technical differences compared to the original proofs are discussed in remarks.

Formalized Analysis of ApproxMCCore. For simplicity, we write `S ⇒ bool` for the type of solutions projected onto set `S` and `[n] ⇒ bool` for n-dimensional bit-vectors, i.e., the type of Boolean-valued functions on domain $0, 1, \ldots, n-1$. A hash function `h :: (S ⇒ bool) ⇒ ([n] ⇒ bool)` maps projected solutions into n-dimensional bit-vectors. Let `W :: ('a ⇒ bool) set` be any set of solutions, such as `sols F`. Abstractly, ApproxMCCore is a way of approximating the cardinality of the projected set `proj S W`, given an *oracle* that can count up to a specified threshold `thresh` number of solutions. Without loss of generality, assume `thresh` $\leq$ `proj S W` (otherwise, the oracle returns the exact count).

Remark 1. The simple type theory of Isabelle/HOL does not support dependent function types like `S ⇒ bool` and `[n] ⇒ bool`. Our formalization represents functions with type `S ⇒ bool` as partial functions `'a ⇀ bool` along with an assumption that their function domain is equal to `S`.

For any fixed bit-vector α `:: [card S - 1] ⇒ bool`, the sets of hash functions `T`, `L`, and `U` used in the analysis are defined as follows, where `card_slice h i` counts the number of entries of `w` $\in$ `proj S W` such that the hash value `h w` agrees with α on their first `i` entries (also called the i-th slices).

```
definition μ where "μ i = card (proj S W) / 2 ^ i"
definition T where "T i = {h. card_slice h i < thresh}"
definition L where "L i = {h. card_slice h i < μ i / (1+ε)}"
definition U where "U i = {h. card_slice h i ≥ μ i * (1 +
ε/(1+ε))}"
```

For any input hash function `h`, the following `approxcore` function (cf. Algorithm 2 Lines 2–5) finds the first index `m`, if one exists in `[1..<card S]`, where `h` $\in$ `T m`. It returns the approximate model count as a multiplier (`2 ^ m`) and cell size (`card_slice h m`). The *failure event* `approxcore_fail` is the set of hash functions `h` such that `approxcore` returns a non-`(1+ε)`-factor-approximate count.

```
definition approxcore where "
  approxcore h = (
  case List.find (λi. h ∈ T i) [1..<card S] of
    None ⇒ (2 ^ card S, 1)
  | Some m ⇒ (2 ^ m, card_slice h m))"

definition approxcore_fail where "
  approxcore_fail  =
  {h. let (cells,sols) = approxcore h ; sz = card (proj S W) in
    cells * sols ∉ {sz / (1 + ε) .. (1 + ε) * sz}}"
```

The key lemma for `approxcore` (shown with proof sketch below) is that, for hash functions `h`, which are randomly sampled from an appropriate hash family `H`, the probability of the aforementioned failure event is bounded above by 0.36 [12]. The lemma uses Isabelle/HOL's formalization of hash families which is *seeded* [31], i.e., `p` is a PMF on seeds and `H` is a `2`-universal hash family for seeds drawn from `p`;

`map_pmf (λs w. H w s) p` is a PMF which samples a random seed `s` and then returns the hash function associated with that seed according to the family `H`.

```
lemma approxcore_fail_prob:
assumes "(1 + ε / (1 + ε)) * (9.84 * (1 + 1 / ε)^2) ≤ thresh"
assumes "ε ≤ 1" "finite (set_pmf p)"
assumes "prob_space.k_universal (measure_pmf p) 2 H
         {α. dom α = S} {α. dom α = {0..<card S - 1}}"
shows "
  measure_pmf.prob (map_pmf (λs w. H w s) p) approxcore_fail ≤ 0.36"
```

Proof. The proof of `approxcore_fail_prob` proceeds via several sub-lemmas [12], which we discuss inline below. We first show that an index `mstar` exists with the following properties (**obtains** is the Isar keyword for existential claims):

```
lemma mstar_exists:
obtains mstar where
  "μ (mstar - 1) * (1 + ε / (1 + ε)) > thresh"
  "μ mstar * (1 + ε / (1 + ε)) ≤ thresh"
  "mstar ≤ card S - 1"
```

This is proved by noting that there exists `m` satisfying the first two properties separately in the finite interval `1,2,...,card S - 1`, so there must be an `mstar` satisfying all three properties in that interval.

Next, the failure event (which is a set of hash functions) is proved to be contained in the union of four separate events involving `mstar` using the properties from `mstar_exists` and unfolding the respective definitions of `T`, `L`, and `U`:

```
lemma failure_subset:
shows "approxcore_fail ⊆
  T (mstar-3) ∪ L (mstar-2) ∪ L (mstar-1) ∪ (L mstar ∪ U mstar)"
```

Finally, we bound the probability for each of the four events separately.

```
lemma events_prob:
assumes "(1 + ε / (1 + ε)) * (9.84 * (1 + 1 / ε)^2) ≤ thresh"
assumes "finite (set_pmf p)"
assumes "prob_space.k_universal (measure_pmf p) 2 H
         {α. dom α = S} {α. dom α = {0..<card S - 1}}"
shows "let Hp = map_pmf (λs w. H w s) p in
  (ε ≤ 1 ⟶ measure_pmf.prob Hp (T (mstar-3)) ≤ 1 / 62.5) ∧
  measure_pmf.prob Hp (L (mstar-2)) ≤ 1 / 20.68 ∧
  measure_pmf.prob Hp (L (mstar-1)) ≤ 1 / 10.84 ∧
  measure_pmf.prob Hp (L mstar ∪ U mstar) ≤ 1 / 4.92"
```

Lemma `approxcore_fail_prob` follows from `failure_subset`, `events_prob`, and the union bound on probabilities. □

Remark 2. Our implicit construction of `mstar` in `mstar_exists` avoids an explicit calculation from `F`, `S` and ε [12], which is more intricate to analyze. Additionally, in `events_prob`, the first bound for `T (mstar-3)` works only when $\varepsilon \leq$ `1`, an omitted condition from the pen-and-paper proof [12, Lemma 2]; we also verified a looser bound of `1 / 10.84` without this condition, but this leads to a weaker overall guarantee for ApproxMCCore (which we do not use subsequently).

Formalized Analysis of ApproxMC. Random XORs and XOR-based hash families are defined as follows:

```
definition random_xor where "
  random_xor V = pair_pmf (pmf_of_set (Pow V)) (bernoulli_pmf (1/2))"
definition random_xors where "
  random_xors V n = prod_pmf {..<n} (λ_. map_pmf Some (random_xor
V))"
definition xor_hash where "
  xor_hash w xors =
  (map_option (λxor. satisfies_xor xor {x. w x = Some True}) ∘
xors)"
```

Here, `random_xor V` is the PMF which samples a pair of a uniformly randomly chosen subset of the (projection) variables `V` and the outcome of a fair coin flip; `random_xors V n` is the PMF that samples `n` independent XORs according to `random_xor V`. Given `card S - 1` randomly chosen seed `xors`, the associated `xor_hash` hash function takes a projected solution `w` to the bit-vector whose bit `i` indicates whether the `i`-th XOR is satisfied by `w`.

The following definition of `approxmccore` (cf. Algorithm 2) randomly samples `card S - 1` XOR constraints over the variables `S` and runs `approxcore_xors` (`approxcore` instantiated with XOR-based hash families using `xor_hash`). The top-level function `approxmc` (cf. Algorithm 1) selects appropriate values for `thresh` and the number of rounds `t` for amplification using the median method.

```
definition approxmccore :: "'fml ⇒ 'a set ⇒ nat ⇒ nat pmf"
where "approxmccore F S thresh =
  map_pmf (approxcore_xors F S thresh) (random_xors S (card S - 1))"

definition approxmc::"'fml ⇒ 'a set ⇒ real ⇒ real ⇒ nat ⇒ nat
pmf"
where "approxmc F S ε δ n = (
  let thresh = compute_thresh ε in
  if card (proj S (sols F)) < thresh
  then return_pmf (card (proj S (sols F)))
  else
    let t = compute_t δ n in
    map_pmf (median t)
      (prod_pmf {0..<t::nat} (λi. approxmccore F S thresh)))"
```

The main result *approxmc_prob* follows from 2-universality of XOR-based hash families and the facts that *compute_thresh* returns a correct value of *thresh* and *compute_t* chooses a sufficient number of rounds for the median method.

Library Contributions. We added reusable results to Isabelle/HOL's probability libraries, such as the Paley-Zigmund inequality (a concentration inequality used in the analysis of ApproxMCCore) and a slightly modified (tighter) analysis of the median method based on the prior formalization by Karayel [31,33]; the latter modification does not change the asymptotic analysis of the method but it is needed as ApproxMC implementations use the tighter calculation to reduce the number of rounds for success probability amplification.

We also formalized the 3-universality of XOR-based hash families [25], which implies its 2-universality, as needed by ApproxMC. The proof is sketched in the online extended version of this paper. Our (new) proof is of independent interest as it is purely combinatorial, using a highly symmetric case analysis which helps to reduce formalization effort because many cases can be proved using without-loss-of-generality-style reasoning in Isabelle/HOL.

4.2 Concretization to a Certificate Checker

The specification from Sect. 4.1 leaves several details abstract. For example, *card_slice* refers to set cardinalities and *approxmc* uses a bounded solution counter as an oracle, neither of which are *a priori* computable terms. This section gives a concrete implementation strategy where the abstract details are obtained from certificates generated by an untrusted external implementation, and checked using verified code. The main result is formalized in a locale *CertCheck* with two key extensions compared to *ApproxMC* from Sect. 4.1: (i) the *ApproxMCL* locale, switching from set-based to computable list-based representations for the projection set and XORs; (ii) the additional locale parameters *check_sol* determining whether a formula is satisfied by a specified assignment, and *ban_sol* that syntactically blocks a solution from further consideration.

```
locale CertCheck = ApproxMCL sols enc_xor
for sols :: "'fml ⇒ ('a ⇒ bool) set"
and enc_xor :: "'a list × bool ⇒ 'fml ⇒ 'fml" +
fixes check_sol :: "'fml ⇒ ('a ⇒ bool) ⇒ bool"
fixes ban_sol :: "'a sol ⇒ 'fml ⇒ 'fml"
assumes "⋀F w. check_sol F w ⟷ w ∈ sols F"
assumes "⋀F vs. sols (ban_sol vs F) =
  sols F ∩ {ω. map ω (map fst vs) ≠ map snd vs}"
```

The correctness of the *certcheck* checker (shown below) has two conjuncts in its conclusion. In both conjuncts, *f* models an external (untrusted) implementation returning a certificate and *r* is a random seed passed to both *f* and *certcheck*. The checker either returns an error string (*isl*) or a certified count. The *soundness* guarantee (left conjunct) says that the probability of the checker

returning an incorrect count (without error) is bounded above by δ. Note that for a buggy counter `f` that always returns an invalid certificate, `certcheck` returns an error for all random seeds, i.e., it returns a count (whether correct or not) with probability 0. Thus, the *promise-completeness* guarantee (right conjunct) says that *if* the function `f` is promised to return valid certificates for all seeds `r`, then the checker returns a correct count with probability `1 - δ`.

```
theorem certcheck_prob:
assumes "(⋀F. check_unsat F ⟹ sols F = {})"
assumes "δ > 0" "δ < 1" "ε > 0" "distinct S"
shows "
  let sz = real (card (proj (set S) (sols F))) in
  let seeds = random_seed_xors (find_t δ) (length S) in
  let pr = measure_pmf.prob
    (map_pmf (λr. certcheck check_unsat F S ε δ (f r) r) seeds) in
  pr {c. ¬isl c ∧ projr c ∉ {sz / (1 + ε) .. (1 + ε) * sz}} ≤ δ ∧
  ( (∀r ∈ set_pmf seeds.
      ¬isl (certcheck check_unsat F S ε δ (f r) r)) ⟶
    pr {c. projr c ∈ {sz / (1 + ε) .. (1 + ε) * sz}} ≥ 1 - δ)"
```

Additional differences in `certcheck_prob` compared to `approxmc_prob` are: (iii) the oracle function `check_unsat`, which is assumed to be an interface to an external unsatisfiability checker; (iv) the additional certificate arguments `m0` and `ms`; and (v) the eager sampling of XORs using random bits (`random_seed_xors`), compared to `approxmc` which samples lazily.

Remark 3. Note that `ban_sol` and `check_sol` are locale parameters with assumptions that must be *proven* when `CertCheck` is instantiated to a Boolean theory; in contrast, `check_unsat` appears as an *assumption*. The pragmatic reason for this difference is that `ban_sol` and `check_sol` can be readily implemented in Isabelle/HOL with decent performance. In contrast, developing efficient verified *unsatisfiability* proof checkers and formats, e.g., for CNFs, is still an active area of research [3,27,37,53]. Leaving `check_unsat` outside the scope of Isabelle/HOL allows us to rely on these orthogonal verification efforts (as we do in Sect. 4.4).

From `approxmc` to `certcheck`. We briefly list the steps in transporting the PAC guarantee from `approxmc` to `certcheck`, with reference to the differences labeled (i)–(v) above. The proof follows a sequence of small refinement steps which are individually straightforward as they do not involve significant probabilistic reasoning. First, cf. (v), a variant of `approxmc` is formalized where all XORs are eagerly sampled upfront, as opposed to lazily at each call to `approxmccore`. Without loss of generality, it suffices to sample `t × (card S - 1)` XORs. Next, cf. (i), the representations are swapped to executable ones, e.g., the projection set is represented as a list `S` of distinct elements. Accordingly, the left-hand side of each XOR is represented as a list of `length S` bits, where the `i`-th bit indicates whether the `i`-th entry of `S` is included in the XOR. Note that it suffices to sample `t × (card S - 1) × (card S + 1)` bits for ApproxMC. Finally, cf. (iv),

Input formula

```
p cnf 10 7
1 2 3 4 5 0
6 7 8 9 10 0
-1 -6 0
-2 -7 0
-3 -8 0
-4 -9 0
-5 -10 0
```

Approximate count

```
...
s mc 184
```

certcheck partial certificate file

```
0   // initial m0
73  // number and list of solutions
-1 2 -3 -4 -5 6 -7 -8 -9 -10 0
...
1 -2 -3 4 5 -6 7 -8 -9 -10 0
2   // round 1 value of m
73  // number and list of solutions
... //   after adding m - 1 XORs
51  // number and list of solutions
... //   after adding m XORs
    //   *UNSAT after excluding 51 solutions
    //   checked by external pipeline
2   // round 2 value of m
    // ... repeat for t rounds ...
```

Fig. 2. An example pigeon-hole formula (2 pigeons, 5 holes, 180 solutions) in DIMACS format and a valid certificate for the checker at $\varepsilon = 0.8$ and $\delta = 0.2$ ($\texttt{thresh} = 73, \texttt{t} = 9$). The certificate is shown with colored comments and with redundant spaces added for clarity. In clauses, the negative (resp. positive) integers are negated (resp. positive) literals, with a 0 terminator; solutions are lists of literals assigned to true. Part of the certificate (marked with $*$) is checked with an external UNSAT proof checking pipeline.

partial certificates are introduced. The key observation is that the final value of `m` in `approxcore` from Sect. 4.1 can be readily *certified* because it is the first entry where adding `m` XORs causes the solution count to fall below `thresh`—the solution count is monotonically decreasing as more XORs are added. Thus, for a claimed value of `m` it suffices to check, cf. (ii) and (iii) that the following three conditions hold. **(1)** Firstly, `1 ≤ m ≤ card S - 1`. **(2)** Secondly, the solution count after adding `m - 1` XORs reaches or exceeds `thresh`, which can be certified (`check_sol`) by a list of solutions of length at least `thresh`, which are distinct after projection on `S`. **(3)** Thirdly, if `m < card S - 1`, then the solution count after adding `m` XORs is below `thresh`, which can be certified (`check_sol`) by a list of solutions of length below `thresh`, which are distinct after projection, and where the formula after excluding all those projected solutions (`ban_sol`) is unsatisfiable (`check_unsat`).

An example partial certificate is shown in Fig. 2. Note that we call these *partial* certificates because of the reliance on an external pipeline for checking unsatisfiability, as illustrated in the example.

Code Extraction for CertCheck. To obtain an executable implementation of `certcheck`, we instantiated the Isabelle/HOL formalization with a concrete syntax and semantics for CNF-XOR formulas, and extracted source code using Isabelle/HOL's Standard ML extraction mechanism. The extracted implementation is compiled together with user interface code, e.g., file I/O, parsing, and

interfacing with a trusted random bit generator and CNF-XOR unsatisfiability checking, as shown in Fig. 1. The resulting tool is called CertCheck.

4.3 Extending ApproxMC to ApproxMCCert

To demonstrate the feasibility of building a (partial) certificate generation tool, we modified the mainline implementation of ApproxMC to accept and use an externally generated source of random bits. We also modified it to write its internally calculated values of `m` and a log of the respective models reported by its internal solver to a file. The resulting tool is called ApproxMCCert. An implementation of ApproxMC (and thus ApproxMCCert) requires logarithmically many solver calls to *find* the correct value of `m` and it can employ many search strategies [12]. The partial certificate format is agnostic to how `m` is found, requiring certification only for the final value of `m` in each round.

Remark 4. It is worth remarking that CertCheck requires checking the validity of $\mathcal{O}(\varepsilon^{-2} \cdot \log \delta^{-1})$ solutions (each of size n, the number of variables), and unsatisfiability for $\mathcal{O}(\log \delta^{-1})$ formulas, while ApproxMC requires $\mathcal{O}(\varepsilon^{-2} \cdot \log n \cdot \log \delta^{-1})$ calls to its underlying solver. In the next section, we instantiate `check_unsat` with a CNF-XOR unsatisfiability checking pipeline that generates proofs which are checkable by a verified checker in polynomial time (in the size of the proofs).

4.4 CNF-XOR Unsatisfiability Checking

A crucial aspect of CertCheck is its reliance on an external checker for unsatisfiability of CNF-XOR formulas. As mentioned in Sect. 2, there are several prior approaches for certified CNF-XOR reasoning that can be plugged into CertCheck.

We opted to build our own *native* extension of FRAT [3] because none of the previous options scaled to the level of efficient XOR proof checking needed for certifying ApproxMC (as evidenced later in Sect. 5). For brevity, the new input and proof format(s) are illustrated with inline comments in Fig. 3. We defer a format specification to the tool repository.

In a nutshell, when given an input CNF-XOR formula, `CryptoMiniSat` has been improved to emit an unsatisfiability proof in our extended FRAT-XOR format. Then, our `FRAT-xor` tool elaborates the proof into XLRUP, our extension of Reverse Unit Propagation (RUP) proofs [27] with XOR reasoning. The latter format can be checked using `cake_xlrup`, our formally verified proof checker. Such an extension to FRAT was suggested as a possibility by Baek et al. [3] and we bear their claim out in practice.

Extending `FRAT-rs` to `FRAT-xor`. Our `FRAT-xor` tool adds XOR support to `FRAT-rs` [3], an existing tool for checking and elaborating FRAT proofs. This extension is designed to be *lightweight*—`FRAT-xor` does not track XORs nor check the correctness of any XOR-related steps; instead, it defers the job to an underlying verified proof checker. Our main changes were: (i) adding parsing support for XORs; (ii) ensuring that clauses implied from XORs can be properly

Input CNF-XOR formula

```
p cnf 3 4
1 2 0
-1 -2 0
x 1 2 -3 0
-3 0
```

FRAT-XOR proof file

```
...
o x 1 1 2 -3 0
i x 2 1 2 0 1 1 2 0
a x 3 3 0 1 1 2 0
i 4 3 0 1 3 0
a 5 0
...
```

XLRUP proof file

```
// Add at XOR ID 1, XOR x 1 2 -3 0,
// from the input formula
o x 1 1 2 -3 0
// Add at XOR ID 2, XOR x 1 2 0,
// implied by clauses 1 and 2
i x 2 1 2 0 1 2 0
// Add at XOR ID 3, XOR x 3 0,
// implied by XORs 1 and 2
x 3 3 0 1 2 0
// Add at clause ID 4, Clause 3 0,
// implied by XOR 3
i 4 3 0 3 0
// Derive empty clause by RUP,
// hints generated by FRAT-xor
5 0 3 4 0
```

Fig. 3. (top left) A sample input CNF-XOR formula where XOR lines start with `x` and indicate the literals that XOR to 1, e.g., the line `x 1 2 -3` represents the XOR constraint $x_1 \oplus x_2 \oplus \bar{x}_3 = 1$; (bottom left) a FRAT-XOR proof; (right) an XLRUP proof. The steps in **bold** indicate newly added XOR reasoning. Note that the XOR steps are (mostly) syntactically and semantically unchanged going from FRAT-XOR to XLRUP, so we focus on the latter here. The meaning of each XLRUP step (analogously for FRAT-XOR) is annotated in color-coded comments above the respective line.

used for further clausal steps, including automatic elaboration of RUP [3]; and (iii) ensuring the clauses used to imply XORs are trimmed from the proof at proper points, i.e., after the last usage by either a clausal or XOR step.

Extending `cake_lpr` to `cake_xlrup`. We also modified `cake_lpr` [53], a verified proof checker for CNF unsatisfiability, to support reasoning over XOR constraints. The new tool supports: (i) clause-to-clause reasoning via RUP steps; (ii) deriving new XORs by adding together XORs; (iii) XOR-to-clause and clause-to-XOR implications. The main challenge here was to represent XORs efficiently using byte-level representations to take advantage of native machine instructions in XOR addition steps. The final verified correctness theorem for `cake_xlrup` is similar to that of `cake_lpr` [53] (omitted here).

Modifications to CryptoMiniSat. A refactoring of `CryptoMiniSat` was performed in response to the bug described in Sect. 1 and in order to add FRAT-XOR proof logging. As part of this rewrite, a new XOR constraint propagation engine has been added that had been removed as part of BIRD [50]—that system did not need it, as it kept all XOR constraints also in a blasted form. Furthermore, XOR constraints have been given IDs instead of a pointer to a `TBUDDY` BDD previously used, and all XOR manipulations such as XOR-ing

together XOR constraints, constant folding [57], satisfied XOR constraint deletion, etc., had to be documented in the emitted FRAT-XOR proof log. Further, `CryptoMiniSat` had to be modified to track which clause IDs were responsible for recovered XOR constraints. To make sure our changes were correct, we modified `CryptoMiniSat`'s fuzzing pipeline to include XOR constraint-generating problems and to check the generated proofs using our certification tools.

5 Experimental Evaluation

To evaluate the practicality of partial certificate generation (ApproxMCCert) and certificate checking (CertCheck), we conducted an extensive evaluation over a publicly available benchmark set [41] of 1896 problem instances that were used in previous evaluations of ApproxMC [49,51]. The benchmark set consists of (projected) model counting problems arising from applications such as probabilistic reasoning, plan recognition, DQMR networks, ISCAS89 combinatorial circuits, quantified information flow, program synthesis, functional synthesis, and logistics. Most instances are satisfiable with large model counts and only approximately 6% are unsatisfiable for testing corner cases.

To demonstrate the effectiveness of our new CNF-XOR unsatisfiability checking pipeline, we also compared it to the three prior state-of-the-art approaches discussed in Sect. 2. The approaches are labeled as follows:

CMS+frat-xor. Our new (default) pipeline based on FRAT-XOR (Sect. 4.4); here, `CMS` is short for `CryptoMiniSat`.

CMS+tbuddy. The pipeline consisting of `CryptoMiniSat` with `TBUDDY`, `FRAT-rs`, and a verified CNF proof checker (Sect. 2, item 1).

MiniSatXOR+pbp. The pipeline consisting of `MiniSat` with XOR engine, VeriPB, and its verified proof checker (Sect. 2, item 2)

CaDiCaL+lrat. A state-of-the-art SAT solver CaDiCaL [8,44] which generates proofs checkable by a verified CNF proof checker (Sect. 2, item 3).

We experimented with each of these approaches as the CNF-XOR unsatisfiability checking pipeline for CertCheck, checking the same suite of certificates produced by ApproxMCCert.

The empirical evaluation was conducted on a high-performance computer cluster where every node consists of an AMD EPYC-Milan processor featuring 2×64 real cores and 512 GB of RAM. For each instance and tool (ApproxMC, ApproxMCCert, or CertCheck), we set a timeout of 5000 s, memory limit of 16GB, and we used the default values of $\delta = 0.2$ and $\varepsilon = 0.8$ for all tools following previous experimental conventions [49]. For each given tool, we report the PAR-2 score which is commonly used in the SAT competition. It is calculated as the average of all runtimes for solved/certified instances out of the relevant instances for that tool, with unsolved/uncertified instances counting for double the time limit (i.e., 10000 s).

Our empirical evaluation sought to answer the following questions:

RQ1 How does the performance of ApproxMCCert and CertCheck compare to that of ApproxMC?
RQ2 How does the performance of CMS+frat-xor compare to prior state-of-the-art approaches for CNF-XOR UNSAT checking for use in CertCheck?

RQ1 Feasibility of Certificate Generation and Checking. We present the results for ApproxMC, ApproxMCCert, and CertCheck in Table 1. For certificate generation, our main observation is that ApproxMCCert is able to solve and generate certificates for 99.3% (i.e., 1202 out of 1211) instances that ApproxMC can solve alone, and their PAR-2 scores (out of 1896 instances) are similar. Indeed, in the per-instance scatter plot of ApproxMC and ApproxMCCert runtimes in Fig. 4, we see that for almost all instances, the overhead of certificate generation in ApproxMCCert is fairly small. This is compelling evidence for the practicality of adopting *certificate generation* for approximate counters with our approach.

Table 1. Performance comparison of ApproxMC, ApproxMCCert, and CertCheck. The PAR-2 score is calculated out of 1896 instances for ApproxMC and ApproxMCCert, and out of the 1202 instances with certificates for CertCheck.

	ApproxMC	ApproxMCCert	CertCheck
Counted Instances	1211	1202	1018
PAR-2 Score	3769	3815	1743

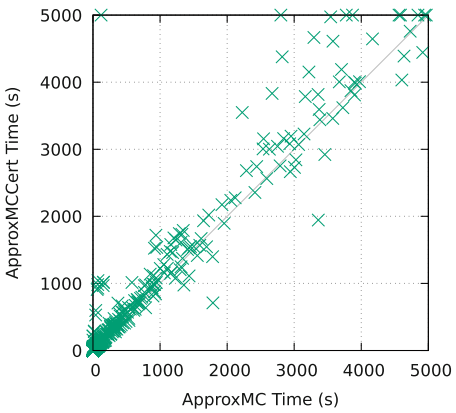

Fig. 4. Per instance runtime (s) comparison for ApproxMCCert and ApproxMC.

Turning to the feasibility of certificate checking, we observe in Table 1 that CertCheck is able to fully certify 84.7% of the instances (i.e., 1018 out of 1202) with certificates. Of the remaining instances, CertCheck timed out for 46 and ran out of memory for 138 instances (no certificate errors were reported in our latest

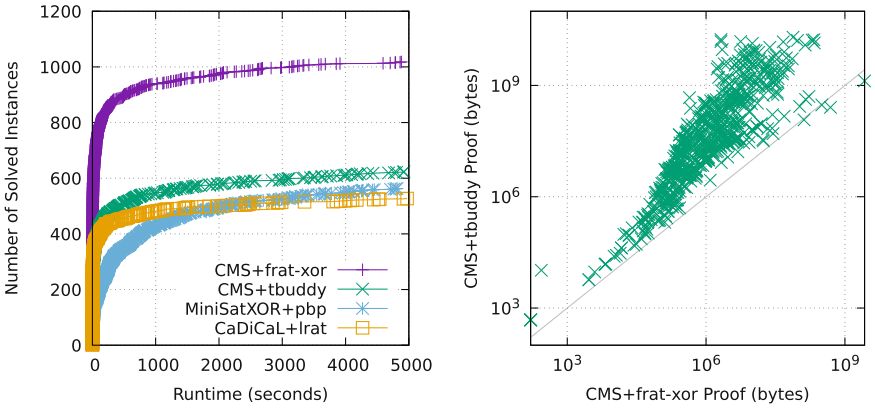

Fig. 5. (left) Runtime performance comparison between CNF-XOR unsatisfiability checkers. (right) Per instance CNF-XOR unsatisfiability proof size (bytes) comparison for CMS+frat-xor and CMS+tbuddy.

versions of the tools). On average, CertCheck requires 4.6 times the runtime of ApproxMCCert across all certified instances. Note that each instance of CertCheck requires nine separate calls to the CNF-XOR unsatisfiability checking pipeline (because $\delta = 0.2$). It is worth emphasizing that in other certificate checking setups, such as the SAT competitions, one would typically provide an order of magnitude more time and memory to the checkers compared to solvers. Thus, CertCheck performs well even though our time and memory limits are stringent. Furthermore, we believe that CertCheck's ability to achieve a fairly low PAR-2 score (computed out of 1202 instances) is compelling evidence for the practicality of certificate checking in approximate counting. Future work could explore *parallelized* certificate checking since each round used in CertCheck can be checked independently of each other.

RQ2 Comparison of CNF-XOR Unsatisfiability Checkers. We present results using various alternative unsatisfiability checking pipelines as part of CertCheck in Table 2. Here, we observe that the use of CMS+frat-xor allows CertCheck to fully certify significantly more instances than can be certified by prior approaches, and with a much lower PAR-2 score.

Table 2. Performance comparison of CNF-XOR unsatisfiability checkers in CertCheck. The PAR-2 score is calculated out of the 1202 instances with certificates for all checkers.

Total	CaDiCaL+lrat	MiniSatXOR+pbp	CMS+tbuddy	CMS+frat-xor
Counted Instances	527	563	623	**1018**
PAR-2 Score	5742	5659	5027	**1743**

Figure 5 (left) visualizes the performance gap between CMS+frat-xor and the prior methods using a CDF (cumulative distribution function) plot; a point (x, y) indicates that the corresponding tool certifies y number of instances when given a timeout of x seconds for each instance. This plot provides strong justification for our claim of the need to develop CMS+frat-xor for native CNF-XOR unsatisfiability proof checking in Sect. 4.4. The ability to log XOR proof steps compactly in our new CNF-XOR unsatisfiability proof format is also significant. This is illustrated in Fig. 5 (right) which gives a scatter plot comparing FRAT (resp. FRAT-XOR) proof sizes generated by CMS+tbuddy (resp. CMS+frat-xor) within 600 s on instances that were successfully certified by CMS+tbuddy. Recall that the solver in CMS+tbuddy supports XOR reasoning and uses `TBUDDY` to emit its proof log in terms of a clausal proof system, i.e., without native XOR proof steps. Overall, our new proof format achieves an average 30-fold reduction in proof size, with the maximum reduction reaching up to 8,251 times.

6 Conclusion and Future Work

This work shows that it is feasible to use proof assistants to formalize practical randomized automated reasoning algorithms. Such formalizations are valuable—our end-to-end certification approach for ApproxMCCert has led to bug-fixes for both ApproxMC and its underlying `CryptoMiniSat` solver.

An interesting line of future work would be to support recently proposed techniques such as *sparse hashing* [39] or *rounding* [60] in the context of ApproxMC. Furthermore, this work leaves preprocessing techniques, such as independent support identification, out of scope. It is worth noting that efficient identification of the independent support set, in conjunction with a new rounding-based algorithm [60], significantly boosts the counting performance of ApproxMC; in the experimental setting of Table 1, this combination solves 1787 instances with a PAR-2 score of 625. Thus, certifying these extensions is a tantalizing avenue for future research. Another potential line of future work involves developing extensions for theories other than CNF-XOR model counting [59].

Acknowledgement. This work has been financially supported by the Swedish Research Council grant 2021-05165, National Research Foundation Singapore under its NRF Fellowship Programme [NRF-NRFFAI1-2019-0004], Ministry of Education Singapore Tier 2 Grant [MOE-T2EP20121-0011], Ministry of Education Singapore Tier 1 Grant [R-252-000-B59-114], and by A*STAR, Singapore. The computational experiments were performed on resources of the National Supercomputing Centre, Singapore https://www.nscc.sg. Part of this work was carried out while some of the authors participated in the Spring 2023 *Extended Reunion: Satisfiability* program at the Simons Institute for the Theory of Computing and at Dagstuhl workshop 22411 *Theory and Practice of SAT and Combinatorial Solving.*

References

1. Abdulaziz, M., Mehlhorn, K., Nipkow, T.: Trustworthy graph algorithms (invited talk). In: Rossmanith, P., Heggernes, P., Katoen, J. (eds.) MFCS. LIPIcs, vol. 138, pp. 1:1–1:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPICS.MFCS.2019.1
2. ApproxMCCert and CertCheck tool repository. https://github.com/meelgroup/approxmc-cert
3. Baek, S., Carneiro, M., Heule, M.J.H.: A flexible proof format for SAT solver-elaborator communication. Log. Methods Comput. Sci. **18**(2) (2022). https://doi.org/10.46298/LMCS-18(2:3)2022
4. Ballarin, C.: Locales: a module system for mathematical theories. J. Autom. Reason. **52**(2), 123–153 (2014). https://doi.org/10.1007/s10817-013-9284-7
5. Baluta, T., Shen, S., Shinde, S., Meel, K.S., Saxena, P.: Quantitative verification of neural networks and its security applications. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) CCS, pp. 1249–1264. ACM (2019). https://doi.org/10.1145/3319535.3354245
6. Barbosa, H., Blanchette, J.C., Fleury, M., Fontaine, P.: Scalable fine-grained proofs for formula processing. J. Autom. Reason. **64**(3), 485–510 (2020). https://doi.org/10.1007/s10817-018-09502-y
7. Beyersdorff, O., Hoffmann, T., Spachmann, L.N.: Proof complexity of propositional model counting. In: Mahajan, M., Slivovsky, F. (eds.) SAT. LIPIcs, vol. 271, pp. 2:1–2:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). https://doi.org/10.4230/LIPICS.SAT.2023.2
8. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT competition 2020. In: Balyo, T., Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.) Proceedings of SAT Competition 2020 – Solver and Benchmark Descriptions. Department of Computer Science Report Series B, vol. B-2020-1, pp. 51–53. University of Helsinki (2020)
9. Bryant, R.E.: TBUDDY: a proof-generating BDD package. In: Griggio, A., Rungta, N. (eds.) FMCAD, pp. 49–58. TU Wien Academic Press (2022).https://doi.org/10.34727/2022/ISBN.978-3-85448-053-2_10
10. Bryant, R.E., Nawrocki, W., Avigad, J., Heule, M.J.H.: Certified knowledge compilation with application to verified model counting. In: Mahajan, M., Slivovsky, F. (eds.) SAT. LIPIcs, vol. 271, pp. 6:1–6:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). https://doi.org/10.4230/LIPIcs.SAT.2023.6
11. Chakraborty, S., Fremont, D.J., Meel, K.S., Seshia, S.A., Vardi, M.Y.: Distribution-aware sampling and weighted model counting for SAT. In: Brodley, C.E., Stone, P. (eds.) AAAI, pp. 1722–1730. AAAI Press (2014). https://doi.org/10.1609/AAAI.V28I1.8990
12. Chakraborty, S., Meel, K.S., Vardi, M.Y.: Algorithmic improvements in approximate counting for probabilistic inference: from linear to logarithmic SAT calls. In: Kambhampati, S. (ed.) IJCAI, pp. 3569–3576. IJCAI/AAAI Press (2016). http://www.ijcai.org/Abstract/16/503
13. Chakraborty, S., Meel, K.S., Vardi, M.Y.: Approximate model counting. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability - Second Edition, Frontiers in Artificial Intelligence and Applications, vol. 336, pp. 1015–1045. IOS Press (2021). https://doi.org/10.3233/FAIA201010
14. Dueñas-Osorio, L., Meel, K.S., Paredes, R., Vardi, M.Y.: Counting-based reliability estimation for power-transmission grids. In: Singh, S., Markovitch, S. (eds.) AAAI, pp. 4488–4494. AAAI Press (2017). https://doi.org/10.1609/AAAI.V31I1.11178

15. Eberl, M., Haslbeck, M.W., Nipkow, T.: Verified analysis of random binary tree structures. J. Autom. Reason. **64**(5), 879–910 (2020). https://doi.org/10.1007/s10817-020-09545-0
16. Eberl, M., Hölzl, J., Nipkow, T.: A verified compiler for probability density functions. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 80–104. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46669-8_4
17. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24605-3_37
18. Ermon, S., Gomes, C.P., Sabharwal, A., Selman, B.: Taming the curse of dimensionality: discrete integration by hashing and optimization. In: ICML. PMLR, vol. 28, pp. 334–342. PMLR (2013). http://proceedings.mlr.press/v28/ermon13.html
19. Fichte, J.K., Hecher, M., Roland, V.: Proofs for propositional model counting. In: Meel, K.S., Strichman, O. (eds.) SAT. LIPIcs, vol. 236, pp. 30:1–30:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPICS.SAT.2022.30
20. Fleury, M.: Optimizing a verified SAT solver. In: Badger, J.M., Rozier, K.Y. (eds.) NFM 2019. LNCS, vol. 11460, pp. 148–165. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-20652-9_10
21. FRATxor and cakexlrup tool repository. https://github.com/meelgroup/frat-xor
22. Gittis, A., Vin, E., Fremont, D.J.: Randomized synthesis for diversity and cost constraints with control improvisation. In: Shoham, S., Vizel, Y. (eds.) CAV. LNCS, vol. 13372, pp. 526–546. Springer, Heidelberg (2022). https://doi.org/10.1007/978-3-031-13188-2_26
23. Gocht, S., McCreesh, C., Myreen, M.O., Nordström, J., Oertel, A., Tan, Y.K.: End-to-end verification for subgraph solving. In: Wooldridge, M.J., Dy, J.G., Natarajan, S. (eds.) AAAI, pp. 8038–8047. AAAI Press (2024). https://doi.org/10.1609/AAAI.V38I8.28642
24. Gocht, S., Nordström, J.: Certifying parity reasoning efficiently using pseudo-Boolean proofs. In: AAAI, pp. 3768–3777. AAAI Press (2021). https://doi.org/10.1609/AAAI.V35I5.16494
25. Gomes, C.P., Sabharwal, A., Selman, B.: Near-uniform sampling of combinatorial spaces using XOR constraints. In: Schölkopf, B., Platt, J.C., Hofmann, T. (eds.) NIPS, pp. 481–488. MIT Press (2006)
26. Gopinathan, K., Sergey, I.: Certifying certainty and uncertainty in approximate membership query structures. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12225, pp. 279–303. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_16
27. Heule, M., Hunt, W., Kaufmann, M., Wetzler, N.: Efficient, verified checking of propositional proofs. In: Ayala-Rincón, M., Muñoz, C.A. (eds.) ITP 2017. LNCS, vol. 10499, pp. 269–284. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66107-0_18
28. Hölzl, J., Lochbihler, A., Traytel, D.: A formalized hierarchy of probabilistic system types. In: Urban, C., Zhang, X. (eds.) ITP 2015. LNCS, vol. 9236, pp. 203–220. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22102-1_13
29. Hurd, J.: Formal verification of probabilistic algorithms. Technical Report. UCAM-CL-TR-566, University of Cambridge, Computer Laboratory (2003). https://doi.org/10.48456/tr-566
30. Kan, S., Lin, A.W., Rümmer, P., Schrader, M.: CertiStr: a certified string solver. In: Popescu, A., Zdancewic, S. (eds.) CPP, pp. 210–224. ACM (2022) https://doi.org/10.1145/3497775.3503691

31. Karayel, E.: Formalization of randomized approximation algorithms for frequency moments. In: Andronick, J., de Moura, L. (eds.) ITP. LIPIcs, vol. 237, pp. 21:1–21:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.ITP.2022.21
32. Karayel, E.: Formalization of randomized approximation algorithms for frequency moments. Archive of Formal Proofs (2022). https://isa-afp.org/entries/Frequency_Moments.html, Formal proof development
33. Karayel, E.: Median method. Archive of Formal Proofs (2022). https://isa-afp.org/entries/Median_Method.html, Formal proof development
34. Karayel, E.: Universal hash families. Archive of Formal Proofs (2022). https://isa-afp.org/entries/Universal_Hash_Families.html, Formal proof development
35. Kaufmann, D., Fleury, M., Biere, A.: The proof checkers Pacheck and Pastèque for the practical algebraic calculus. In: FMCAD, pp. 264–269. TU Wien Academic Press (2020).https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_34
36. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: a verified implementation of ML. In: Jagannathan, S., Sewell, P. (eds.) POPL, pp. 179–192. ACM (2014). https://doi.org/10.1145/2535838.2535841
37. Lammich, P.: Efficient verified (UN)SAT certificate checking. J. Autom. Reason. **64**(3), 513–532 (2020). https://doi.org/10.1007/s10817-019-09525-z
38. McConnell, R.M., Mehlhorn, K., Näher, S., Schweitzer, P.: Certifying algorithms. Comput. Sci. Rev. **5**(2), 119–161 (2011). https://doi.org/10.1016/J.COSREV.2010.09.009
39. Meel, K.S., Akshay, S.: Sparse hashing for scalable approximate model counting: theory and practice. In: Hermanns, H., Zhang, L., Kobayashi, N., Miller, D. (eds.) LICS, pp. 728–741. ACM (2020). https://doi.org/10.1145/3373718.3394809
40. Meel, K.S., Chakraborty, S., Akshay, S.: Auditable algorithms for approximate model counting. In: Wooldridge, M.J., Dy, J.G., Natarajan, S. (eds.) AAAI, pp. 10654–10661. AAAI Press (2024). https://doi.org/10.1609/AAAI.V38I9.28936
41. Meel, K.S., Soos, M.: Model counting and uniform sampling instances (2020). https://doi.org/10.5281/zenodo.3793090
42. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
43. Paulson, L.C.: The foundation of a generic theorem prover. J. Autom. Reasoning **5**(3), 363–397 (1989). https://doi.org/10.1007/BF00248324
44. Pollitt, F., Fleury, M., Biere, A.: Faster LRAT checking than solving with CaDiCaL. In: Mahajan, M., Slivovsky, F. (eds.) SAT. LIPIcs, vol. 271, pp. 21:1–21:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). https://doi.org/10.4230/LIPICS.SAT.2023.21
45. Roth, D.: On the hardness of approximate reasoning. Artif. Intell. **82**(1–2), 273–302 (1996). https://doi.org/10.1016/0004-3702(94)00092-1
46. Sang, T., Beame, P., Kautz, H.A.: Performing Bayesian inference by weighted model counting. In: Veloso, M.M., Kambhampati, S. (eds.) AAAI, pp. 475–482. AAAI Press/The MIT Press (2005). http://www.aaai.org/Library/AAAI/2005/aaai05-075.php
47. Shi, X., Fu, Y.-F., Liu, J., Tsai, M.-H., Wang, B.-Y., Yang, B.-Y.: CoqQFBV: a scalable certified SMT quantifier-free bit-vector solver. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12760, pp. 149–171. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81688-9_7
48. Soos, M., Bryant, R.E.: Proof generation for CDCL solvers using Gauss-Jordan elimination. CoRR arxiv:2304.04292 (2023). https://doi.org/10.48550/ARXIV.2304.04292

49. Soos, M., Gocht, S., Meel, K.S.: Tinted, detached, and lazy CNF-XOR solving and its applications to counting and sampling. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12224, pp. 463–484. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53288-8_22
50. Soos, M., Meel, K.S.: BIRD: engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting. In: AAAI, pp. 1592–1599. AAAI Press (2019). https://doi.org/10.1609/AAAI.V33I01.33011592
51. Soos, M., Meel, K.S.: Arjun: An efficient independent support computation technique and its applications to counting and sampling. In: Mitra, T., Young, E.F.Y., Xiong, J. (eds.) ICCAD, pp. 71:1–71:9. ACM (2022). https://doi.org/10.1145/3508352.3549406
52. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 244–257. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02777-2_24
53. Tan, Y.K., Heule, M.J.H., Myreen, M.O.: Verified propagation redundancy and compositional UNSAT checking in CakeML. Int. J. Softw. Tools Technol. Transf. **25**(2), 167–184 (2023). https://doi.org/10.1007/s10009-022-00690-y
54. Tan, Y.K., Yang, J.: Approximate model counting. Archive of Formal Proofs (2024). https://isa-afp.org/entries/Approximate_Model_Counting.html, Formal proof development
55. Thiemann, R., Sternagel, C.: Certification of termination proofs using CeTA. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 452–468. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_31
56. Valiant, L.G.: The complexity of enumeration and reliability problems. SIAM J. Comput. **8**(3), 410–421 (1979). https://doi.org/10.1137/0208032
57. Wegman, M.N., Zadeck, F.K.: Constant propagation with conditional branches. ACM Trans. Program. Lang. Syst. **13**(2), 181–210 (1991). https://doi.org/10.1145/103135.103136
58. Wetzler, N., Heule, M.J.H., Hunt, W.A.: DRAT-trim: efficient checking and trimming using expressive clausal proofs. In: Sinz, C., Egly, U. (eds.) SAT 2014. LNCS, vol. 8561, pp. 422–429. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09284-3_31
59. Yang, J., Meel, K.S.: Engineering an efficient PB-XOR solver. In: Michel, L.D. (ed.) CP. LIPIcs, vol. 210, pp. 58:1–58:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021https://doi.org/10.4230/LIPIcs.CP.2021.58
60. Yang, J., Meel, K.S.: Rounding meets approximate model counting. In: Enea, C., Lal, A. (eds.) CAV. LNCS, vol. 13965, pp. 132–162. Springer, Heidelberg (2023). https://doi.org/10.1007/978-3-031-37703-7_7