Model-counting approaches for nonlinear numerical constraints

Mateus Borges[1], Quoc-Sang Phan[2], Antonio Filieri[1], and Corina S. Păsăreanu[2,3]
Imperial College London[1], Carnegie Mellon University[2], NASA Ames[3]

Abstract. Model counting is of central importance in quantitative reasoning about systems. Examples include computing the probability that a system successfully accomplishes its task without errors, and measuring the number of bits leaked by a system to an adversary in Shannon entropy. Most previous work in those areas demonstrated their analysis on programs with linear constraints, in which cases model counting is polynomial time. Model counting for nonlinear constraints is notoriously hard, and thus programs with nonlinear constraints are not well-studied. This paper surveys state-of-the-art techniques and tools for model counting with respect to SMT constraints, modulo the bitvector theory, since this theory is decidable, and it can express nonlinear constraints that arise from the analysis of computer programs. We integrate these techniques within the Symbolic Pathfinder platform and evaluate them on difficult nonlinear constraints generated from the analysis of cryptographic functions.

1 Introduction

Model counting is of central importance in quantitative reasoning, with applications in probabilistic inference [1, 2], reliability analysis [3], and quantitative information flow [4–7]. Most previous work in those areas was performed on programs with linear constraints, using model counting tools such as Latte [8]. Model counting for nonlinear constraints is notoriously hard, and thus programs with nonlinear constraints are not well-studied (with only limited support for floating-point values abstracted as real numbers [9]). In this paper we survey state-of-the-art model counting techniques and tools for SMT (satisfiability modulo theories) constraints modulo the bitvector theory, since this theory is decidable and it can express the nonlinear constraints that arise naturally from the analysis of computer programs. Our work is motivated by a security project [10] that aims to develop automated quantitative information flow analysis techniques for complex applications, including cryptographic functions that are very difficult to analyze. The bitvector theory is particularly useful for these functions which typically use operations on bitvector values.

We integrate the surveyed techniques within Symbolic PathFinder (SPF) [11] and evaluate them on difficult nonlinear constraints generated using symbolic execution. Although we restrict our evaluation to cryptographic functions, our study should be relevant to anybody interested in quantitative reasoning over complex, nonlinear systems.

1.1 Symbolic Execution and SPF

SPF performs symbolic execution over Java byte code programs. Symbolic execution [12] is a systematic analysis technique that executes a program on symbolic, rather than concrete, input values and computes the effects of the program as *functions* of these symbolic inputs. The result of symbolic execution is a set of symbolic paths, each with a path condition PC, which is a conjunction of constraints over the symbolic inputs that characterizes all the inputs that follow that path. All the PCs are disjoint by construction.

1.2 Quantification of information leaks

Perfect software security is hard to achieve. Systems often leak information to an adversary who can observe different aspects of program behavior. Research on quantitative information flow aims at quantifying (in number of bits) the expected leakage.

A program can be viewed as a probabilistic function that maps a *high* security input h and a *low* security input l to an *observable* output o. An adversary tries to guess h by providing l and observing the output. The leakage of the program P is defined as the *mutual information* between the secret h and the public output o [13]: $\mathit{Leakage}(P) = \mathcal{H}(o) - \mathcal{H}(o|h)$, where $\mathcal{H}(x)$ denotes the classical Shannon entropy of a random variable x, measuring the "uncertainty" about x. For a deterministic program P, there is no uncertainty about o when h is given. Therefore $\mathcal{H}(o|h) = 0$.The entropy can thus be computed as: $\mathit{Leakage}(P) = \mathcal{H}(o) = -\sum_{i=1,m} p(o_i) \log_2(p(o_i))$.

Intuitively, the leakage gives an estimate on the number of bits in the secret that an adversary can infer by observing the output of the program. If this estimate is small (or zero) then the program can be considered safe. In [4], Backes et al. combined model checking and model counting to compute the leakage when the observable is an output variable. In a similar setting, we used symbolic execution (SPF) combined with Latte to compute an upper bound on the leakage [5].

More recently [6, 7], we used SPF and Latte to compute the leakage when the observables are non-functional characteristics of program executions, i.e. side-channels, such as time consumed, number of memory accessed or packets transmitted over a network. In this model, a symbolic path identified by PC_i leads to a concrete observable o_i. Assuming the secret input has uniform distribution, which means the adversary has no *prior* knowledge about it, the probability of observing o_i can be computed using SPF and model counting as follows: $p(o_i) = \sum_{cost(PC_j)=o_i} \sharp(PC_j)/\sharp D$, where $\sharp(PC_j)$ is the number of solutions (computed with model counting) of constraint PC_j and $\#D$ is the size of the input domain D assumed to be (possibly very large but) finite.

In all the previous work mentioned above, Latte was used to perform model counting; it implements the polynomial time Barvinok algorithm to count models for a system of linear integer inequalities. However Latte cannot handle nonlinear constraints. In this paper we study approaches for the fixed-width bitvector theory, which can represent such constraints. In the following, we use the term "bitvector" and "word" interchangeably.

2 Model counting techniques and tools

In this section we evaluate several tool-supported approaches for counting the models of bitvector constraints. These approaches can be classified according to two orthogonal dimensions: exact vs approximate and bit-level vs word-level.

Exact techniques count the exact number of models for a given constraint. Approximate techniques only explore a portion of the solution space, carefully selected to provide probabilistic guarantees on the accuracy ($0 < \epsilon < 1$) and confidence ($0 < \delta < 1$) of the result. In particular, they guarantee that $\Pr\big((1-\epsilon)c \leq c^* \leq (1+\epsilon)c\big) \geq 1-\delta$, where c^* is the approximate result and c is the exact (unknown) count. Other randomized approaches not providing formal guarantees (e.g., [14, 15]) are not considered in this study.

Bit-level approaches address the model counting problem for propositional (SAT) formulas, i.e., #SAT. Model counting for bit vector formulas can be performed as follows. A bitvector formula is first converted to a propositional formula using bit blasting to

generate an equivalent Boolean circuit based on bit-level behavior of bitvector operations. This Boolean circuit is interpreted as a propositional logic problem and converted in conjunctive normal form (CNF); at this point #SAT approaches can be used to count the number of models. While the procedure is general, the conversion of Boolean circuits into CNF is usually based on the Tseitin transformation [16], which introduces additional Boolean variables in the process. While this transformation guarantees a model for the CNF form is also a model for the initial problem, the introduction of additional variables may lead to different model counts. For this reason, in this paper we use only #SAT tools supporting projection, i.e., able to project the solution space only on the variables appearing in the Boolean circuit, ignoring the ones introduced by Tseitin transformation.

We found five tools for #SAT that support projection and can thus be used in our setting for bitvector counting: SharpCDCL, All-SAT, SharpSAT and Dsharp, which compute exact solutions, and ApproxMC-p, which produces approximate solutions.

- SharpCDCL [17] is an enumeration-based approach; it iteratively invokes the SAT solver to produce at each iteration a new model, keeping trace of the set of models and their number.
- All-SAT [18] and SharpSAT [19] extend the DPLL algorithm to count the number of solutions of a SAT problem. They both use caching mechanisms and use constraint propagation for pruning the DPLL, which avoid the exhaustive exploration of subtrees containing no solutions.
- Dsharp [20] reuses the algorithmic core of SharpSAT, adapting it to work with a deterministic Decomposable Negation Normal Form (d-DNNF) representation of the SAT problem. d-DNNF provides a more compact representation of the constraints in memory that, according to [20], may better support model counting.
- ApproxMC-p [21] takes as input accuracy and confidence targets and produce an approximate count which deviates from the exact count by at most a factor $1 \pm \epsilon$ with probability at least $1 - \delta$. The approach uses universal hash functions to perform a uniform sampling within the domain. The ratio between the number of models for this sample and the sample size is used as an estimate of the ratio of models over the entire problem domain. The samples is automatically decided to achieve ϵ and δ.

Word-level approaches aim to avoid the cost of bit blasting by defining counting procedures that operate directly on SMT variables and operations. We investigate a recent tool that provides an approximate counting procedure for bitvectors: SMTApproxMC [2]. SMTApproxMC uses word-level hashing functions to sample a finite number of candidate models and then an SMT solver to check how many of these candidate models satisfy the constraint. The number of models found within the sample are used to build a robust statistical estimator achieving the desired probabilistic guarantees. SMTApproxMC can avoid bit blasting whenever the SMT solver can check a constraint without it (e.g., for linear constraints); however, for nonlinear constraints (all the subjects of this study), SMTApproxMC requires bit blasting.

Chistikov et al. [1] also extend the hashing-based approach used for #SAT (e.g., in [21]) to counting for SMT problems. Hashing functions allow to uniformly sample candidate solutions. Statistics on the sample are used to estimate the total number of models. However, no tool is available and, according to [2], SMTApproxMC is faster.

A related approach is implemented in the MathSAT solver [22], which provides a functionality, called All-SMT, that given a set of Boolean variables V_I, it can enumerate all the models of the problem projected on V_I. The source code of the tool is not available,

nor a technical description of the All-SMT feature, thus we do not know the details of the counting algorithm it implements but can only report its execution time.

Other approaches. We have also investigated other techniques for model counting: blocking-clause enumeration, BDD-based enumerations, counting with Gröbner bases and a brute-force enumeration that we use as baseline.

Blocking clause enumeration make the solver find all the models for a problem by iteratively adding the negation of already found models to the initial problem. The iteration terminates when no more solutions can be found. Intuitively, this method can work only for complex problems with few models. We implemented it on top of Z3 SMT solver [23] to practically confirm this intuition.

BDD-based enumeration represents a propositional formula as a binary decision diagram and then counts the paths from its root to the leaf representing the Boolean constant "true". We implemented a prototype based on the BDD library CUDD [24], which builds a BDD corresponding to a constraint bitblasted with Z3. Unfortunately, for all the subjects in this study the execution time exceeded the timeout of 1hr.

Gröbner bases are used in computational algebra to reason about polynomials over finite fields. Boolean variables and and operators from propositional logic can be mapped into corresponding variables and functions over polynomials. Each zero of such polynomials corresponds to exactly one model of the initial propositional formula [25, 26]. Algebraic solvers can be used to find those zeroes. We implemented this technique using PolyBoRi [27], but its execution timed out for all the subjects.

Finally, we also implemented as a reference a brute force approach which encodes the constraints as bitwise operations on unsigned integers in C. The mapping is straightforward from the smtlib representation. The program iterates over the entire domain and count the number of models for a constraint. We compiled the C sources using level 1 optimization in GCC.

3 Evaluation

Subjects. We study modular exponentiation ($modPow(b, e, m) = b^e \bmod m$) and modular multiplication ($modMul(x, y) = x * y \bmod m$) implementations. These are core routines for most public-key cryptographic systems, most notably RSA. In the past, some implementations have been found vulnerable to side channel attacks [28, 29], mostly as effect of optimizations. Our goal is to localize side channels by quantifying information leaks with symbolic execution and model counting (see Section 1).

For our experiments, we analyzed a set of randomly selected path conditions from two different implementations of the modular operations (the source code is given in the appendix). The first implementation (subjects `a-`$\star$ in the following), taken from [6], optimizes $modPow$ with a reduction step at each iteration, but uses a naive implementation of $modMul$. We analyze the program with the same configurations from [6]: the modulus m can be either 1717, 834443, or 1964903306; both the base b and exponent e are symbolic, with $b \leq m$ and $e \leq 31$.

The second implementation (benchmarks `b-`$\star$ in the following) is more realistic as it uses Java's `BigInteger` class to encode large messages and secrets (this example was provided to us by DARPA at a recent engagement) and uses fast multiplication. Here modulus m is fixed with a 1536-bit value; the base b is also a concrete 1532-bit value; the exponent e is symbolic BigInteger with 40 bits. We analyze both $modPow$ and $modMul$, where both x and y are symbolic 24-bit BigInteger.

Subject	a-1	a-2	a-3	a-4	a-5	a-6	a-7	b-1	b-2	b-3	b-4
N. Ops	11	26	15	37	121	57	117	250	243	1428	1428
Domain Size	10K	10K	10K	25M	25M	59B	59B	4T	4T	32B	32B
N. Solutions	1.7K	7	1.7K	208K	109K	80M	77M	2B	66B	1	1
N. CNF clauses	40K	78K	58K	67K	114K	58K	78K	2K	2K	2K	2K
Execution time											
BitBlasting	15s	30s	24s	25s	44s	23s	30s	1s	1s	1s	2s
SharpCDCL	1s	1s	1s	43m	-	-	-	-	-	1s	1s
All-SAT	1s	8s	2s	31m*	59m*	15m*	19m*	-	-	1s	1s
SharpSAT	5s	2s	11s	29m	53m	-	-	1s	1s	1s	1s
Dsharp	12m	32s	22m	-	-	-	-	1s	1s	1s	1s
ApproxMC (f)	4s	2s	5s	16s	32s	1m	1m	4s	5s	1s	1s
ApproxMC (p)	4s	2s	6s	2m	5m	21m	24m	16s	25s	1s	1s
SMTapproxMC (f)	6m	15m	8m	-	-	-	-	-	-	2m	2m
SMTapproxMC (p)	-	15m	-	-	-	-	-	-	-	2m	2m
MathSAT	2s	2s	5s	38m	54m	-	-	-	-	1s	1s
Z3-BC	12s	3s	18s	-	-	-	-	-	-	1s	1s
Brute Force	1s	1s	1s	1s	1s	8m	8m	-	-	2m	2m

Fig. 1. Execution time comparison.

Experimental Results. Fig. 1 summarizes the performance of the different tools. The results indicate that enumeration-based techniques perform well for complex problem with few solutions (SharpCDCL, Z3-BC). Exact techniques based on DPLL (All-SAT and SharpSAT) scale better than enumeration, but fail for the subjects involving complex constraints over large domains, like `a-6` and `a-7` which have approximately 58k and 78k CNF clauses over a domain of 59B points. Notably, All-SAT produced the correct count only for the first three subjects. For all the others (marked with *), it significantly under-approximated the count. However, the most recent release dates back to 2004 and the tool is not maintained, making difficult to get the tool fixed.

The performance of approximate methods (ApproxMC and SMTApproxMC) depends on the required accuracy ϵ and confidence δ. The correct counts and the approximate ones are shown in a table in the appendix. We run the tools with two different settings: (f) ϵ=0.5, δ=0.05 and (p) ϵ=0.1, δ=0.05. SMTApproxMC provides a bad performance on our subjects; this is however expected since its internal solver is required to bit blast our nonlinear constraints for each query. From our experience, low-accuracy approximate methods can be used for a preliminary assessment of the number of solutions: if the coarse approximate count is small, exact methods may then be used for an exact solution. Similarly, if the count is close to the domain size, it is possible to count exactly the models of the negation of the problem (which should be only a few). If the count is far from its extreme values (0 and domain size) or if the problem is particularly complex ($> 50k$ CNF clauses on our subjects), exact counters will probably fail if the domain is large and a more precise approximate solution can be pursued.

Not surprisingly, the brute force approach is faster than model counting tools when the domain size is small enough ($< 10^9$), but it is not a viable solution for larger problems.

4 Conclusion

We surveyed model counting techniques that are applicable to complex nonlinear constraints. We restricted our study to techniques and tools that are capable of providing formal guarantees on the results. Our survey suggests that that the most promising techniques use approximate model counting and bit-level hashing, however the performance of the tools can degrade when increased precision is required. SMT-based model counting is still a very young research area, but its relevance for quantitative analysis

can be an effective driver for its development, as program verification has effectively driven the development in SMT solving.

References

[1] Chistikov, D., Dimitrova, R., Majumdar, R.: Approximate Counting in SMT and Value Estimation for Probabilistic Programs. TACAS'15 (2015) 320–334

[2] Chakraborty, S., Meel, K.S., Mistry, R., Vardi, M.Y.: Approximate Probabilistic Inference via Word-level Counting. AAAI'16 (2016) 3218–3224

[3] Filieri, A., Păsăreanu, C.S., Visser, W.: Reliability Analysis in Symbolic Pathfinder. ICSE, IEEE Press (2013) 622–631

[4] Backes, M., Kopf, B., Rybalchenko, A.: Automatic Discovery and Quantification of Information Leaks. SP '09 (2009) 141–153

[5] Phan, Q.S., Malacaria, P., Păsăreanu, C.S., d'Amorim, M.: Quantifying Information Leaks Using Reliability Analysis. SPIN 2014, ACM (2014) 105–108

[6] Păsăreanu, C.S., Phan, Q.S., Malacaria, P.: Multi-run Side-Channel Analysis Using Symbolic Execution and Max-SMT. CSF '16 (June 2016) 387–400

[7] Bang, L., Aydin, A., Phan, Q.S., Păsăreanu, C.S., Bultan, T.: String Analysis for Side Channels with Segmented Oracles. FSE 2016, ACM (2016) 193–204

[8] Loera, J.A.D., Hemmecke, R., Tauzer, J., Yoshida, R.: Effective lattice point counting in rational convex polytopes. Journal of Symbolic Computation **38**(4) (2004) 1273 – 1302

[9] Borges, M., Filieri, A., d'Amorim, M., Păsăreanu, C.S., Visser, W.: Compositional Solution Space Quantification for Probabilistic Software Analysis. PLDI, ACM (2014) 123–132

[10] ISSTAC project. `http://www.cmu.edu/silicon-valley/research/isstac`

[11] Păsăreanu, C.S., Visser, W., Bushnell, D., Geldenhuys, J., Mehlitz, P., Rungta, N.: Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. Automated Software Engineering (2013) 1–35

[12] King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7) (July 1976) 385–394

[13] Malacaria, P.: Algebraic foundations for quantitative information flow. Mathematical Structures in Computer Science **25** (2 2015) 404–428

[14] Wei, W., Selman, B. In: A New Approach to Model Counting. Springer (2005) 324–339

[15] Rubinstein, R.: Stochastic enumeration method for counting np-hard problems. Methodology and Computing in Applied Probability **15**(2) (2013) 249–291

[16] Tseitin, G.S.: On the Complexity of Derivation in Propositional Calculus. In: Automation of Reasoning: 2: Classical Papers on Computational Logic. Springer (1983) 466–483

[17] Klebanov, V., Manthey, N., Muise, C.: SAT-based analysis and quantification of information flow in programs. QEST'16 (2013) 177–192

[18] Grumberg, O., Schuster, A., Yadgar, A.: Memory efficient all-solutions sat solver and its application for reachability analysis. FMCAD'04, Springer (2004) 275–289

[19] Thurley, M.: sharpSAT–counting models with advanced component caching and implicit BCP. SAT'06, Springer (2006) 424–429

[20] Muise, C., McIlraith, S.A., Beck, J.C., Hsu, E.I. In: Dsharp: Fast d-DNNF Compilation with sharpSAT. Springer (2012) 356–361

[21] Klebanov, V., Weigl, A., Weisbarth, J.: Sound Probabilistic #SAT with Projection. QAPL'16 (2016) 15–29

[22] Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The mathsat5 smt solver. TACAS'13 (2013) 93–107

[23] De Moura, L., Bjørner, N.: Z3: an efficient SMT solver. TACAS'08 (2008) 337–340

[24] Somenzi, F.: Cudd: Cu decision diagram package release 3.0. 0. (2015)

[25] Gao, S.: Counting zeros over finite fields using gröbner bases. Master's thesis, Carnegie Mellon University (2009)

[26] Tran, Q., Vardi, M.Y.: Groebner bases computation in boolean rings for symbolic model checking. MOAS, ACTA Press (2007) 440–445

[27] Brickenstein, M., Dreyer, A.: Polybori: A framework for grbner-basis computations with boolean polynomials. Journal of Symbolic Computation **44**(9) (2009) 1326 – 1345

[28] Kocher, P.C.: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. CRYPTO (1996) 104–113

[29] Brumley, D., Boneh, D.: Remote Timing Attacks Are Practical. SSYM'03, USENIX Association (2003) 1–1

Appendix

1 Source code of case studies used in the experiments

1.1 Modular exponentiation with reduction steps

The method `modPow1` is taken from [6]. Modular exponentiation is optimized with a reduction step at each iteration.

```
int modPow1(int num, int e, int m){
  int s = 1, y = num, res = 0;
  while (e > 0) {
    if (e % 2 == 1) {
      //reduction:
      int tmp = s * y;
      if (tmp > m){
        tmp = tmp - m;
      }
      res = tmp % m;
    } else {
      res = s;
    }
    s = (res * res) % m;
    e /= 2;
  }
  return res;
}
```

1.2 Modular exponentiation with fast multiplication

The method `modPow2` is taken from SnapBuddy, a web application for image processing and sharing. `modPow2` is implemented with BigInteger, and it does not have reduction steps. The modulus has 1536 bits; the base `clientPublic` (value not shown here) has 1532 bits.

```
public KeyExchangeServer(String secretKey) {
  String modp1536 = "
    ↪ FFFFFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD"
    + "129024E088A67CC74020BBEA63B139B22514A08798E3404"
    + "DDEF9519B3CD3A431B302B0A6DF25F14374FE1356D6D51C"
    + "245E485B576625E7EC6F44C42E9A637ED6B0BFF5CB6F406"
    + "B7EDEE386BFB5A899FA5AE9F24117C4B1FE649286651ECE"
    + "45B3DC2007CB8A163BF0598DA48361C55D39A69163FA8FD"
    + "24CF5F83655D23DCA3AD961C62F356208552BB9ED529077"
    + "096966D670C354E4ABC9804F1746C08CA237327FFFFFFFF"
    + "FFFFFFFF";
  this.modulus = new BigInteger(modp1536, 16);
```

```
  this.secretKey = secretKey.startsWith("0x") ?
    new BigInteger(secretKey.substring(2), 16)
    : new BigInteger(secretKey);
  // ...
}

public BigInteger generateMasterSecret(BigInteger clientPublic)
    ↪ {
  return ModPow.modPow2(clientPublic, this.secretKey, this.
    ↪ modulus);
}
```

```
public static BigInteger modPow2(final BigInteger base, final
    ↪ BigInteger exponent, final BigInteger modulus) {
    BigInteger s = BigInteger.valueOf(1L);
    for (int width = exponent.bitLength(), i = 0; i < width; ++i
    ↪ ) {
      s = s.multiply(s).mod(modulus);
      if (exponent.testBit(width - i - 1)) {
        s = fastMultiply(s, base).mod(modulus);
      }
    }
    return s;
  }
}
```

1.3 Modular multiplication

For $x * y \mod m$, we use the method `fastMultiply` from SnapBuddy and m be the 1536-bit modulus defined in the constructor of `KeyExchangeServer`.

```
public static BigInteger fastMultiply(final BigInteger x, final
    ↪ BigInteger y) {
  final int xLen = x.bitLength();
  final int yLen = y.bitLength();
  if (x.equals((Object)BigInteger.ONE)) {
    return y;
  }
  if (y.equals((Object)BigInteger.ONE)) {
    return x;
  }
  BigInteger ret = BigInteger.ZERO;
  int N = Math.max(xLen, yLen);
  if (N <= 800) {
    ret = x.multiply(y);
  }
  else if (Math.abs(xLen - yLen) >= 32) {
    ret = standardMultiply(x, y);
```

```
17    }
      else {
19      N = N / 2 + N % 2;
        final BigInteger b = x.shiftRight(N);
21      final BigInteger a = x.subtract(b.shiftLeft(N));
        final BigInteger d = y.shiftRight(N);
23      final BigInteger c = y.subtract(d.shiftLeft(N));
        final BigInteger ac = fastMultiply(a, c);
25      final BigInteger bd = fastMultiply(b, d);
        final BigInteger crossterms = fastMultiply(a.add(b), c.add(d
        ↪ ));
27      ret = ac.add(crossterms.subtract(ac).subtract(bd).shiftLeft(
        ↪ N)).add(bd.shiftLeft(2 * N));
      }
29    return ret;
  }
31
  public static BigInteger standardMultiply(final BigInteger x,
      ↪ final BigInteger y) {
33    BigInteger ret = BigInteger.ZERO;
      for (int i = 0; i < y.bitLength(); ++i) {
35      if (y.testBit(i)) {
          ret = ret.add(x.shiftLeft(i));
37      }
      }
39    return ret;
  }
```

2 Detailed counts reported for each tool

Benchmark	a-1	a-2	a-3	a-4	a-5	a-6	a-7	b-1	b-2	b-3	b-4
N. Ops	11	26	15	37	121	57	117	250	243	1428	1428
Domain Size	10K	10K	10K	25M	25M	59B	59B	4T	4T	32B	32B
N. CNF clauses	40K	78K	58K	67K	114K	58K	78K	2K	2K	2K	2K
Reported Counts											
SharpCDCL	1701	7	1696	208096	-	-	-	-	-	1	1
All-SAT	1701	7	1696	51666	21298	40478	30810	-	-	1	1
SharpSAT	1701	7	1696	208096	109495	-	-	2081157128	66597028096	1	1
Dsharp	1701	7	1696	-	-	-	-	2081157128	66597028096	1	1
ApproxMC (f)	1664	7	1664	172032	126976	83886080	77594624	2147483648	66571993088	1	1
ApproxMC (p)	1700	7	1696	209152	108544	78643200	76021760	2097152000	66571993088	1	1
MathSAT	1701	7	1696	208096	109495	-	-	-	-	1	1
SMTapproxMC (f)	1799	2	1799	-	-	-	-	-	-	0	0
SMTapproxMC (p)	-	2	-	-	-	-	-	-	-	0	0
Z3-BC	1701	7	1696	-	-	-	-	-	-	1	1
Brute Force	1701	7	1696	208096	109495	79963411	76589491	-	-	1	1

Fig. 2. Counts reported by the tools under analysis