Maximum Model Counting

Daniel J. Fremont
Markus N. Rabe
Sanjit A. Seshia

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2016-169
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-169.html

November 30, 2016

Maximum Model Counting

Daniel J. Fremont and **Markus N. Rabe** and **Sanjit A. Seshia**
University of California, Berkeley
Email: {dfremont,rabe,sseshia}@berkeley.edu

Abstract

We introduce the problem Max#SAT, an extension of model counting (#SAT). Given a formula over sets of variables X, Y, and Z, the Max#SAT problem is to maximize over the variables X the number of assignments to Y that can be extended to a solution with some assignment to Z. We demonstrate that Max#SAT has applications in many areas, showing how it can be used to solve problems in probabilistic inference (marginal MAP), planning, program synthesis, and quantitative information flow analysis. We also give an algorithm which by making only polynomially many calls to an NP oracle can approximate the maximum count to within any desired multiplicative error. The NP queries needed are relatively simple, arising from recent practical approximate model counting and sampling algorithms, which allows our technique to be effectively implemented with a SAT solver. Through several experiments we show that our approach can be successfully applied to interesting problems.

Introduction

Algorithms for the Boolean satisfiability problem (SAT) have proved tremendously useful in a variety of application areas, including planning, security, and verification. SAT asks for *some* solution (a satisfying assignment) of a Boolean formula, without regard to precisely which solution we get. The maximum satisfiability problem (MaxSAT) asks for an assignment that satisfies the highest number of clauses and thereby enables a search for "good" assignments. A broad range of discrete optimization problems have thus been encoded in MaxSAT, and a number of mature algorithms are available capable of solving large instances (Safarpour et al. 2007; Marques-Silva and Planes 2011; Davies and Bacchus 2013).

A different, quantitative extension of SAT is the Boolean model counting problem (#SAT), which asks for the number of solutions $\#X.\,\psi(X)$ of a given formula ψ. #SAT allows us to encode summations over exponentially many terms *succinctly*, which enables several applications including probabilistic inference (Sang, Beame, and Kautz 2005) and quantitative information flow analysis (Backes, Köpf, and Rybalchenko 2009; Newsome, McCamant, and Song 2009; Klebanov, Manthey, and Muise 2013). Solving #SAT is considered to be a very hard problem (Toda 1991) but effective approximation algorithms have been found recently (Chakraborty, Meel, and Vardi 2013; 2016; Ermon et al. 2013).

In this paper we define the maximum model counting (Max#SAT) problem, which generalizes both MaxSAT and #SAT and thereby combines the ability to optimize with the power of succinct encodings of large summations.

Definition 1. Given a propositional formula $\varphi(X, Y, Z)$ over sets of variables X, Y, and Z, the *Max#SAT* problem is to determine $\max_X \#Y.\,\exists Z.\,\varphi(X, Y, Z)$.

Informally, the Max#SAT problem is to maximize over the variables X the number of assignments to Y such that the formula $\varphi(X, Y, Z)$ is satisfiable.

The main technical contribution of this paper is a randomized approximation algorithm for Max#SAT that estimates the maximum count up to a desired multiplicative error and confidence. The algorithm avoids considering each of the exponentially many assignments over which we maximize and instead encodes the problem in polynomially many queries to an NP oracle. Specifically, given $\varphi(X, Y, Z)$ the algorithm considers a new formula $\psi(X, Y_1, \ldots, Y_k) = \exists Z_1, \ldots, Z_k.\ \bigwedge_i \varphi(X, Y_i, Z_i)$, where each Y_i and Z_i is a fresh copy of Y and Z. If ψ has N solutions, then we prove that for a large enough $k \in O(|X|)$, $\sqrt[k]{N}$ approximates the solution to the Max#SAT problem. Furthermore, uniformly sampling from the solutions of ψ yields a witnessing assignment to X that achieves close to the maximum model count with high probability. The counting and sampling operations need only be approximate, so that we can take advantage of efficient SAT solver-based algorithms (Chakraborty, Meel, and Vardi 2014; 2016; Chakraborty et al. 2015).

In the second part of the paper we present encodings of a variety of problems into Max#SAT and demonstrate the applicability of our approximation algorithm. First we show that Max#SAT generalizes MaxSAT by giving a reduction. This also provides a new approach to the rich set of applications of MaxSAT, complementing existing algorithms that rely on either cardinality constraints (Fu and Malik 2006) or on the joint use of SAT and mixed integer programming (MIP) solvers (Davies and Bacchus 2013).

In *quantitative information flow analysis* (QIF) the goal is to estimate the amount of information a program leaks, typically measured in terms of *channel capacity*. Without an adversary the problem comes down to counting the number of possible outputs of the program, for which various model counting approaches have been considered (Backes, Köpf, and Rybalchenko 2009; Newsome, McCamant, and Song 2009; Klebanov, Manthey, and Muise 2013). We demonstrate that Max#SAT enables effective QIF analysis of programs with adversary-controlled inputs, finding near-optimal choices for the adversary that maximize the leakage.

Next we show that probabilistic inference problems, such as *marginal maximum a posteriori inference* (MAP), have a natural encoding in Max#SAT. This provides the first algorithm to approximate MAP and the functional E-MAJSAT problem (Pipatsrisawat and Darwiche 2009), which is closely related to Max#SAT, with polynomially many calls to an NP oracle.

Finally, in *program synthesis* we seek to find a program, or parts thereof, satisfying a given specification and syntactic restrictions such as a template (Alur et al. 2013). The specification may be a logical formula or a list of examples; either way, it can easily happen that the specification contains contradictory requirements or that a precondition has been omitted. In such cases it can be useful to find a *best-effort program* that is correct on as many inputs as possible. This is naturally formulated as a Max#SAT problem.

In summary, the contributions of this work are as follows:

- we define the Max#SAT problem,
- we give an approximation algorithm for Max#SAT that is polynomial-time relative to an NP oracle,
- we show that this algorithm provides a novel approach to MaxSAT, and
- we apply the algorithm to problems in quantitative information flow analysis, probabilistic inference, and program synthesis.

Overview

In this section we illustrate the Max#SAT problem and our approximation algorithm with an example from quantitative information flow (QIF). Consider Program 1, which models a password-checking program. It has a *public* input that is controlled by an adversarial user, and a *secret* input representing the true password. Normally the program returns either 1 or 0 to indicate whether the user is allowed to log in. However, it also has a backdoor: if the user enters a certain magic value, the program will return the secret password.

Program 1 CHECKPASSWORD(*public*, *secret*)

```
if public = 0x42CB88FF then
    return secret
else
    return public = secret
end if
```

Even without the backdoor, it is clear that a password-checking program must leak some information about the secret. If an adversary guesses a password at random, they will almost certainly guess wrong, but in so doing eliminate one possible value for the true password. This leak is unavoidable, but it is not a security problem, since it is negligible compared to the size of the space of passwords. On the other hand, the backdoor in Program 1 allows the program to leak a large amount of information.

In QIF leaks are measured with information-theoretic metrics. One of the simplest and most popular is *channel capacity*. For deterministic programs with public and secret inputs as above, the channel capacity C is the logarithm of the number of possible output values, maximized over the public inputs. More formally, if $\varphi(P, O, S)$ holds if and only if the program returns O on inputs P and S, then

$$C = \log_2 \left(\max_P \#O.\, \exists S.\, \varphi(P, O, S) \right). \tag{1}$$

For example, if the inputs to Program 1 are 32-bit words, then its channel capacity is 32 bits, as when *public* has the magic backdoor value the program has 2^{32} possible outputs.

There are standard techniques that can construct a propositional formula φ as needed above for any loop-free program. For Program 1, we could define $\varphi(P, O, S)$ to be

$$\begin{aligned} &(P = \texttt{0x42CB88FF} \to O = S) \\ \wedge\, &(P \neq \texttt{0x42CB88FF} \to (P = S \to O = 1) \\ &\qquad\qquad \wedge\, (P \neq S \to O = 0)) \,. \end{aligned}$$

Then computing the inner quantity in (1) is exactly an instance of Max#SAT.

How might we solve such a problem? A first observation is that for a particular choice of public input p, finding $\#O.\, \exists S.\, \varphi(p, O, S)$ is an instance of (projected) *model counting*, for which there are efficient approximation algorithms using SAT solvers (Chakraborty, Meel, and Vardi 2013; 2016). So one simple approach would be to choose public inputs at random, computing the output count for each and taking the maximum: if a significant fraction of the inputs cause large leaks, this could find them quickly. However, this method clearly breaks down if large leaks are caused by relatively few inputs, and Program 1 is an extreme example: there is exactly 1 public input with a leak of 32 bits, and the remaining $2^{32} - 1$ inputs leak only 1 bit.

To fix this problem, we need a way to sample public inputs that puts more weight on those that have large output counts. Consider the solutions (p, o) to the formula $\psi(P, O) = \exists S.\, \varphi(P, O, S)$. In our example there are 2^{32} solutions with p being the backdoor value, while there are only two solutions for every other value of p. Taking a random sample from this formula, for which there are again efficient approximation algorithms (Chakraborty, Meel, and Vardi 2014; Chakraborty et al. 2015), likely gives us a solution with the backdoor value for p. To be precise ψ has $2^{32} + (2^{32} - 1) \cdot 2 = 3 \cdot 2^{32} - 2$ solutions, of which 2^{32} have the backdoor value for p. So if we sample from them uniformly, we will obtain the backdoor value with probability $2^{32}/(3 \cdot 2^{32} - 2) \geq 1/3$. Unlike our first attempt where we picked values of p unintelligently, we will now discover the maximum leak after taking just a few samples.

Notice that we are taking advantage of the fact that there is a large gap between the maximum leak and the "typical"

leak. If the maximum leak were smaller, say only 16 bits, then the probability of sampling the single public input p leading to it would be $2^{16}/(2^{16}+(2^{32}-1)\cdot 2) < 2^{-16}$. Here even though p has many more solutions leading to it than any other input, they are not enough to make up a significant fraction of the whole solution space. We can further bias the sampling towards p by giving it more solutions, and a natural way to do this is by duplicating the formula:

$$\psi(p, o_1, o_2) = \exists s_1.\, \varphi(p, o_1, s_1) \wedge \exists s_2.\, \varphi(p, o_2, s_2).$$

Now the solutions leading to p are those of the form (p, o_1, o_2) for any possible outputs o_1 and o_2 on public input p. The number of solutions is squared, which amplifies the gap between the leaks: a 16-bit leak will now have 2^{32} corresponding solutions, while a 1-bit leak will only have 2^2. So the probability of finding the 16-bit leak will be $2^{32}/(2^{32} + (2^{32}-1)\cdot 2^2) \geq 1/5$, which is once again large enough for sampling to succeed. In general, making k copies of the formula will raise the number of solutions corresponding to a leak to the k-th power. As k increases, the solution space is dominated more and more by solutions corresponding to public inputs with large counts, and the probability of finding such an input increases accordingly. We will show that finding an input whose count is within any desired multiplicative factor of the maximum only requires a polynomially large k, leading to an algorithm that is polynomial-time relative to the underlying sampling and counting primitives.

Maximum Model Counting

We call an assignment x to X a *witness* for the Max#SAT instance, and say its *count* C_x is $\#Y.\, \exists Z.\, \varphi(x, Y, Z)$. The solution to the Max#SAT problem is $M = \max_x C_x$, so for any witness x we have $C_x \leq M$. The *quality* of x is the ratio C_x/M, which indicates how close x comes to achieving the maximum possible count in terms of multiplicative error. A quality-1 witness has the maximum count, a quality-$1/2$ witness has count within a factor of 2 of the maximum, and so forth. Often when solving a Max#SAT problem we want not only to bound the maximum count but also to find a high-quality witness.

Approximate Solution Technique

By work of Toda (1991), the decision version of Max#SAT is in $\mathsf{NP}^{\mathsf{PP}}$. The $\mathsf{NP}^{\mathsf{PP}}$-complete problem E-MAJSAT (Littman, Goldsmith, and Mundhenk 1998) can be trivially reduced to it, so it is also $\mathsf{NP}^{\mathsf{PP}}$-complete. However, this does not determine how difficult Max#SAT is to approximate. In this section we present an algorithm, MAXCOUNT, that approximately solves the Max#SAT problem using only polynomially-many queries to an NP oracle.

The algorithm uses the notion of the *k-self-composition* φ^k of a given formula $\varphi(X, Y, Z)$ for some $k \in \mathbb{N}$:

$$\varphi^k(X, \overline{Y}, \overline{Z}) \equiv \varphi(X, Y_1, Z_1) \wedge \cdots \wedge \varphi(X, Y_k, Z_k)\,,$$

where $\overline{Y} = (Y_1, \ldots, Y_k)$ and $\overline{Z} = (Z_1, \ldots, Z_k)$.

We now present the algorithm MAXCOUNT, which given a Max#SAT problem and two parameters ϵ and δ computes a tuple (c, x) consisting of an estimate c of the maximum model count and a witness x. MAXCOUNT is based on techniques for projected model counting and projected sampling. The number of solutions of a formula $\psi(X, Y)$ *projected onto* X is the number of solutions distinct in X. Accordingly, *projected sampling* returns an assignment to X uniformly at random from those that satisfy $\exists Y.\, \psi(X, Y)$.

For our purposes it is sufficient to estimate the projected solution count with any desired multiplicative error. Likewise we only need *almost-uniform* sampling, where each element is returned with a probability within a multiplicative factor of the uniform probability. Both can be done with an NP oracle using the algorithms of Chakraborty, Meel, and Vardi (2016) and Chakraborty et al. (2015).

The first step of MAXCOUNT is to sample almost-uniformly from the solutions of φ^k, projected onto X and $\overline{Y}$. Observe that for any assignment x to X, we have

$$\#\overline{Y}.\exists \overline{Z}.\varphi^k(x, \overline{Y}, \overline{Z}) = [\#Y.\exists Z.\varphi(x, Y, Z)]^k = C_x^k. \quad (2)$$

Therefore the probability of obtaining x from our sample is proportional to C_x^k. This means that increasing k concentrates the probability on assignments with large counts, and for appropriate k our sample x will be such an assignment with high probability.

The second phase of MAXCOUNT is simply to estimate C_x, which is a projected model counting problem. The resulting value is our estimate for M.

The procedure is given in more detail as Algorithm 2. The parameters ϵ and δ specify the desired tolerance and failure probability. The function $\textsc{Sample}(\psi, V, n, \rho)$ generates n independent samples from ψ projected onto the variables V, with a distribution that is uniform to within a multiplicative error of ρ. The function $\textsc{Count}(\psi, V, \rho, \delta)$ estimates the number of models of ψ projected onto the variables V, returning a value that is accurate to within a multiplicative error of ρ with probability at least $1 - \delta$.

Algorithm 2 $\textsc{MaxCount}(\varphi(X, Y, Z), \epsilon, \delta)$

$k \leftarrow \lceil 2|X|/\log_2(1+\epsilon) \rceil$
$n \leftarrow \lceil 34 \ln(2/\delta) \rceil$
$s_1, \ldots, s_n \leftarrow \textsc{Sample}(\varphi^k(X, \overline{Y}, \overline{Z}), X \cup \overline{Y}, n, 17)$
for all $i \in \{1, \ldots, n\}$ **do**
  $(x_i, \overline{y}_i) \leftarrow s_i$
  $c_i \leftarrow \textsc{Count}(\varphi(x_i, Y, Z), Y, \sqrt{1+\epsilon}, \delta/2n)$
end for
$i \leftarrow \arg\max_i c_i$
return (c_i, x_i)

Theorem 1. *For any φ and $\epsilon, \delta > 0$, suppose that $\textsc{MaxCount}(\varphi, \epsilon, \delta)$ returns $(\tilde{c}, \tilde{x})$. Then with probability at least $1-\delta$, $M/(1+\epsilon) \leq \tilde{c} \leq (1+\epsilon)M$ and $C_{\tilde{x}} \geq M/(1+\epsilon)$.*

Proof. Let A be the set of all assignments to X. Let $B \subseteq A$ consist of all assignments such that $C_x \geq M/\sqrt{1+\epsilon}$, i.e. all x which have "close" to the largest possible count.

Each sample s_i is obtained by almost-uniformly sampling from solutions of $\varphi^k(X, \overline{Y}, \overline{Z})$ projected onto X and $\overline{Y}$. By Equation (2), for any x the number of such projected solutions with x as the assignment to X is C_x^k. So the fraction

f of projected solutions where the assignment to X is in B (and thus the x_i obtained from that sample is in B) is

$$\frac{\sum_{x\in B} C_x^k}{\sum_{x\in B} C_x^k + \sum_{x\in A\setminus B} C_x^k} \geq \frac{\sum_{x\in B} C_x^k}{\sum_{x\in B} C_x^k + |A|\left(\frac{M}{\sqrt{1+\epsilon}}\right)^k}.$$

Now since B contains at least one witness achieving the maximum count, $\sum_{x\in B} C_x^k \geq M^k$ and so

$$f \geq \left(1 + \frac{|A|}{(1+\epsilon)^{k/2}}\right)^{-1} \geq \left(1 + \frac{|A|}{2^{|X|}}\right)^{-1} = \frac{1}{2},$$

where we have used $k = \lceil 2|X|/\log_2(1+\epsilon)\rceil = \left\lceil 2\log_{1+\epsilon}(2^{|X|})\right\rceil$. Therefore, the probability that at least one x_i is in B is at least

$$\begin{aligned} 1 - (1 - (f/17))^n &\geq 1 - \exp(-nf/17) \\ &\geq 1 - \exp(-n/34) \geq 1 - \delta/2. \end{aligned}$$

Also, with probability at least $(1-\delta/2n)^n$, we have $C_{x_i}/\sqrt{1+\epsilon} \leq c_i \leq C_{x_i}\sqrt{1+\epsilon}$ for every x_i, i.e. the estimates c_i are accurate for *every* sample. So with probability at least $(1-\delta/2)\cdot(1-\delta/2n)^n \geq 1-\delta$ we have:

- $x_i \in B$ for some i;
- $M/(1+\epsilon) \leq C_{x_i}/\sqrt{1+\epsilon} \leq c_i$ (since $x_i \in B$ and c_i is accurate);
- $c_i \leq \tilde{c}$ (since $\tilde{c}$ is chosen as the max);
- $\tilde{c} \leq C_{\tilde{x}}\sqrt{1+\epsilon} \leq M\sqrt{1+\epsilon} \leq M(1+\epsilon)$ (since $\tilde{c}$ is accurate).

So with probability at least $1-\delta$ we have $M/(1+\epsilon) \leq \tilde{c} \leq M(1+\epsilon)$ and $C_{\tilde{x}} \geq M/(1+\epsilon)$. □

Thus MAXCOUNT allows us to solve Max#SAT problems to any desired accuracy with high probability, and in fact to find arbitrarily high-quality witnesses. It can also be implemented efficiently using SAT solvers:

Theorem 2. *Relative to an NP oracle,* MAXCOUNT *runs in* $O(|\varphi||X|\log_2(|X||Y|)\log_2^2(1/\delta)/\epsilon^2)$ *time.*

Proof. We have $k \in O(|X|/\log_2(1+\epsilon)) \subseteq O(|X|/\epsilon)$, so φ^k has size $O(|\varphi||X|/\epsilon)$. Implementing SAMPLE requires $n \in O(\log_2(1/\delta))$ calls to the algorithm of Chakraborty et al. (2015). Replacing its internal counter with the improved algorithm of Chakraborty, Meel, and Vardi (2016), its runtime will be only logarithmic in the number of projection variables. There are $|X| + k|Y| = O(|X||Y|/\epsilon)$ of these, and we use a constant sampling tolerance, so each sampling call takes $O(|\varphi^k| \cdot \log_2(|X||Y|/\epsilon)) = O(|\varphi||X|\log_2(|X||Y|/\epsilon)/\epsilon)$ time relative to an NP oracle. We can implement COUNT with the algorithm of Chakraborty, Meel, and Vardi (2016). We project onto $|Y|$ variables, so each of the n calls takes $O(|\varphi|\log_2|Y|\log_2(2n/\delta)/\tilde{\epsilon}^2)$ time relative to an NP oracle, where $1+\tilde{\epsilon} = \sqrt{1+\epsilon}$ and so $1/\tilde{\epsilon} \in O(1/\epsilon)$.

In total, the sampling queries require $O(|\varphi||X|\log_2(|X||Y|/\epsilon)\log_2(1/\delta)/\epsilon)$ time and the counting queries require $O(|\varphi|\log_2|Y|\log_2^2(1/\delta)/\epsilon^2)$ time. Adding these and simplifying, MAXCOUNT runs in $O(|\varphi||X|\log_2(|X||Y|)\log_2^2(1/\delta)/\epsilon^2)$ time relative to an NP oracle. □

Discussion

We designed MAXCOUNT with two goals in mind: (1) provide guarantees for both lower and upper bounds and (2) quickly find witnesses with large counts. If we are primarily interested in finding witnesses to obtain lower bounds, it makes sense to start with $k = 0$ (sampling assignments to X uniformly at random with no constraints), incrementally increasing k until we get a large enough lower bound for our purposes. If we reach $k \sim |X|/\log_2(1+\epsilon)$, then our bounds are likely tight. This method has two advantages. First, runtime generally increases with k, and directly setting k to $2|X|/\log_2(1+\epsilon)$ would yield formulas too large for current sampling and counting techniques. Second, in many of our experiments, there was a large gap between the "typical" counts achieved by most witnesses and the maximum count. This allowed the incremental approach to find good lower bounds with values of k that were much smaller than those required in the worst case.

This ability to give good results with small k is also why we use a combination of sampling and counting in MAXCOUNT, when counting would be sufficient. Observe that counting instead of sampling on line 3 would return an estimate of $S = \sum_{x\in A} C_x^k$, which satisfies $M^k \leq S \leq |A|M^k$. So $S^{1/k}/2^{|X|/k} \leq M \leq S^{1/k}$, and taking $k \sim |X|/\log_2(1+\epsilon)$ suffices for $S^{1/k}$ to be a good estimate of the maximum count. For this counting approach, however, such a large value of k is also *necessary*, since otherwise the lower bound $S^{1/k}/2^{|X|/k}$ can be extremely weak. On the other hand, the sampling approach used in MAXCOUNT can potentially find strong lower bounds with low values of k, since it returns the (estimated) count of a single witness. With a favorable distribution of counts the witness is likely near the maximum, even for small k.

Implementation Notes

As mentioned above, for projected model counting and sampling we use the algorithms of Chakraborty, Meel, and Vardi (2016) and Chakraborty et al. (2015) respectively. In our tool we use an improved implementation of these algorithms by Mate Soos and Kuldeep Meel, which is pending publication.

It can be helpful in practice to switch between several counting techniques for the second phase of the algorithm. In particular, if the density of solutions is high then simple Monte Carlo sampling, i.e. choosing assignments y uniformly at random and checking $\exists Z.\varphi(x,y,Z)$ with a SAT solver, can be much faster than the hashing-based counting algorithm. Our implementation has a simple heuristic to decide which methods to use.

Finally, there is a considerable amount of flexibility in choosing the internal parameters — counting and sampling tolerances, number of samples, size of k, etc. — so that the desired overall tolerance ϵ and confidence δ are achieved. For example, we can relax the needed sampling tolerance by taking more samples, or the counting tolerance by using a larger value of k. Exposing these parameters as arguments to MAXCOUNT would obscure the actual tolerance and confidence achieved and complicate the statements of Theorems 1 and 2. To keep the presentation above simple,

we have expressed everything in terms of only ϵ and δ and fixed arbitrary convenient values for the internal parameters (e.g. 17 for the sampling tolerance). In practice, trading off the different sources of error (approximate sampling, approximate counting, small k) by adjusting these parameters can improve efficiency. Our tool allows such adjustments to be made easily, and computes the resulting overall tolerance and confidence.

Experimental Setup

In the following sections we discuss a number of experiments that were all performed on identical machines with Intel Core i7 3.1GHz processors. Memory requirements were always moderate, so we only report execution times.

From Maximum Satisfiability to Max#SAT

MaxSAT is a well studied problem with several algorithms competing in an annual evaluation (Argelich et al. 2008). In this section we discuss a simple reduction from weighted partial MaxSAT to Max#SAT. This shows that Max#SAT generalizes MaxSAT and enables a novel algorithmic approach to it. Since there are highly optimized MaxSAT solvers and MaxSAT resides in a lower complexity class than Max#SAT, this will hardly give us a competitive approach to the problems already encoded in MaxSAT. However, in Max#SAT we can use counting variables to succinctly encode exponentially large sums, while in MaxSAT they have to be listed explicitly as clauses. Applications of MaxSAT that allow for such a succinct encoding may thereby profit from our approximation algorithm.

A *weighted partial MaxSAT* (MaxSAT for short) problem consists of a set of clauses $\mathcal{F}$ over variables X, partitioned into soft clauses $soft(\mathcal{F})$ and hard clauses $hard(\mathcal{F})$, and a weight function wt mapping soft clauses to positive integers. We assume that $hard(\mathcal{F})$ is satisfiable and define the *cost* of an assignment to X to be the total weight of the soft clauses that are not satisfied. The MaxSAT problem is to determine the minimal cost over all assignments satisfying $hard(\mathcal{F})$.

Let $n > 0$ be the number of soft clauses and let m be the maximal weight. Introducing two bitvectors y and y' with lengths $\lceil \log(n) \rceil$ and $\lceil \log(m) \rceil$, we can encode the MaxSAT problem as the following Max#SAT problem: maximize over X the number of solutions in y and y' of the formula

$$hard(\mathcal{F}) \ \wedge \ y{<}n \ \wedge \bigwedge_{0 \le i < n} [(y{=}i \ \wedge \ y'{<}wt(C_i)) \Rightarrow C_i] \,,$$

where C_i is the i-th soft clause, $=$ encodes bitwise equality, and $<$ encodes the *less than* relation on unsigned bitvectors. Then we have the following:

Theorem 3. *The minimal cost of a MaxSAT problem is equal to $\sum_{C \in soft(\mathcal{F})} wt(C)$ minus the maximum model count of the Max#SAT encoding above.*

Proof. Let $\mathbf{x}$ be an assigment to X satisfying $hard(\mathcal{F})$ with cost c. The constraint $y < n$ permits n values for y. For each value, all but one of the conjuncts $(y{=}i \ \wedge \ y'{<}wt(C_i)) \Rightarrow C_i$ are true by falsifying the antecedent. If for the remaining conjunct the soft clause C_i is satisfied, the conjunct admits $wt(C_i)$ assignments for y'; otherwise it admits none. So the number of solutions (projected onto y and y') is the total weight of the satisfied soft clauses, which is $wt(soft(\mathcal{F}))$ minus the weight of the unsatisfied clauses. □

We tested this encoding with benchmarks from the 2016 MaxSAT competition. In some cases MAXCOUNT provided better upper bounds on the minimum cost than MAXHS (Davies and Bacchus 2013), the winner of the competition. For example, with $k = 5$ we sampled a solution for the benchmark `keller4.clq` with cost 1552 in 174 seconds, while the best upper bound MAXHS could find before the 3600s timeout was 1576. However, on the vast majority of the problems from the competition the Max#SAT approach could not compete with MAXHS.

Application 1: Quantitative Information Flow

We consider the problem of finding the largest information leak, measured by channel capacity, in a program with an adversary-controlled input. Modeling such a public input is crucial when looking for backdoors or other large leaks that occur only for particular values of the input. However, existing QIF approaches that deal with public inputs take time exponential in the size of the input or the size of the leak (Köpf and Basin 2007; Heusser and Malacaria 2010). On the other hand, as we saw in the Overview section, this task is naturally represented as a Max#SAT problem, so we can solve it in polynomial time relative to an NP oracle using MAXCOUNT. The multiplicative error in our estimate gives an additive error in the channel capacity (e.g. a factor of 16 gives 4 bit accuracy). In this section we demonstrate that MAXCOUNT is effective in practice on interesting examples.

We are not aware of a standard set of QIF benchmarks with public inputs. Several actual vulnerabilities in the Linux kernel were studied by Heusser and Malacaria (2010). We experimented with two, CVE-2007-2875 and CVE-2009-3002, but they turn out to be uninteresting from our perspective: almost all values of the public input trigger the leak, so they are easily solved with $k = 0$ (i.e. just testing random public inputs). Therefore we created several more challenging benchmarks where the public input must be carefully chosen (see the Appendix in the full version of the paper for benchmark details). The next table shows the least k needed to lower bound the maximum leak to within 4 bits (with at least 80% confidence), the actual leak, the lower bound obtained, and the time taken.

Name	k	leak (bits)	bound	time (s)
CVE-2007-2875	0	32	32	0
CVE-2009-3002	0	64	60	30
pwd-backdoor	1	64	64	70
bin-search-16	1	16	16	72
reverse	2	32	29	4360
reverse2	2	32	28	1153
backdoor-2x16-8	3	16	15	59
backdoor-32-24	4	32	32	150

For completeness we implemented the approach of Heusser and Malacaria (2010) and applied it to the benchmarks above. Unsurprisingly, since the method is exponen-

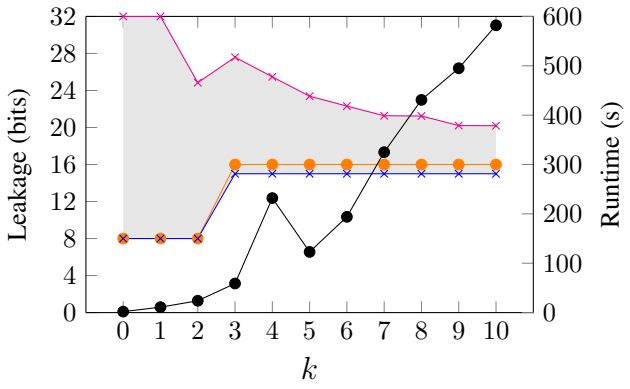

Figure 1: Leak estimate (orange), upper and lower bounds (purple, blue), and time to compute them (black), as a function of k for benchmark "backdoor-2x16-8". Lower and upper bound confidences are 0.98 and 0.6 respectively. Experiments with $n = 20$ samples and counting tolerance $\epsilon = 1$.

tial in the size of the leak, in every case it ran out of memory (with a 10 GB limit) before being able to lower bound the leak to within 4 bits.

Our technique is particularly suited to finding lower bounds, as this does not require a large k in many cases. Sometimes it is also possible to prove nontrivial upper bounds. An example is shown in Figure 1. At $k = 3$, when the maximum leak of 16 bits was first found, there is already an upper bound of about 27.6 bits. As k increases the upper bound gets tighter, although the time needed to obtain it grows. However, the absolute value of the upper bound may increase, as we see in Figure 1 between $k = 2$ and $k = 3$. Here the estimate of the leak increases more than the improvement in the accuracy of the upper bound.

Application 2: Probabilistic Inference

We consider challenging problems in probabilistic inference, such as finding the *marginal maximum a posteriori hypothesis* (MAP) or the *maximum expected utility* (MEU). For sets X and Y of variables over finite domains and suitable functions $f_1, \ldots, f_n$ these problems can be represented as an optimization over a *sum of products* (Dechter 1999):

$$\max_X \sum_Y \prod_{i \le n} f_i(X, Y)$$

In this section we show that Max#SAT naturally captures sum-of-products problems by giving a simple reduction from MAP to Max#SAT. We also discuss how the representation of these problems critically affects the performance of Max#SAT.

Markov networks consist of variables over finite domains and a list of *potentials* f_i mapping variable assignments to $[0, 1]$, given as tables. Given a partition of the variables $X \dot\cup Y$ and *evidence* in the form of a partial assignment σ to the variables, the MAP problem for Markov nets is to determine $\max_X \sum_{Y' \supseteq \sigma} \prod_{i \le n} f_i(X, Y')$, where Y' are assignments extending σ. With suitable quantization and scaling we can approximate the problem by a sum of products with functions mapping to the natural numbers. We also encode the original variables by Boolean variables in a standard way, and write τ for the set of unit clauses asserting the assignment σ. Finally, introducing bitvectors z_i of lengths $\log_2(\max_{X,Y} f_i(X, Y))$, we encode the MAP instance as the following Max#SAT problem:

$$\max_X \#Y, z_1, \ldots, z_n . \tau \wedge \bigwedge_{i \le n} (z_i < f_i(X, Y))$$

where $<$ represents the *less than* relation on bitvectors.

For given assignments to X and Y each conjunct has a number of solutions equal to the value of the corresponding potential (after scaling), and these solution counts multiply since each variable z_i occurs in exactly one of the conjuncts. The total model count for an assignment to X adds these products over all assignments to Y consistent with τ, equivalently σ, and so the Max#SAT problem has the same solution as the MAP problem up to any quantization error. Applying MaxCount, this yields the first algorithm to give lower and upper bounds for MAP to arbitrary precision with only polynomially many calls to an NP oracle.

From a practical perspective, however, the reduction was not competitive with existing approaches (Marinescu, Dechter, and Ihler 2014). The problem instances in the UAI competition (Zhang and Tian 2014) tend to use *many small function tables*, leading to a large number of variables z_i. Already for $k = 1$ this led to challenging queries for the underlying counting and sampling algorithms. Our algorithm could, however, show better performance on problems consisting of *a few large function tables*, if we can represent them succinctly as formulas. It would be interesting to see how MaxCount performs on succinct MAP problems, e.g. those studied by Mauá, De Campos, and Cozman (2015).

Application 3: Program Synthesis

Program synthesis asks to find a program satisfying a given specification, which constrains both the syntax and the behavior of the desired program (Alur et al. 2013). Here we consider *sketching*, a class of synthesis problems that fix most of the structure of the program but ask for a number of integer constants to fill in 'holes', indicated by double question marks (??) (Solar-Lezama et al. 2006). For example, consider this sketch of a program computing the sign of an integer:

```
short sign(short x) {
    if (x > ??)
        return 1;
    else
        return -1;
}
```

This sketch is, of course, unrealizable for the specification of the sign function as it omits an option to provide the output 0 for $x = 0$. However, we can still attempt to synthesize a program P that is consistent with the sketch and satisfies the specification φ on the maximal possible number of inputs I, a *best-effort* program: $\max_P \#I . \varphi(P, I)$.

Finding best-effort programs may also be of use for programming by example, where the correctness specification

comes in the form of a list of input-output examples. It is straightforward to adapt the formulation above of best-effort program synthesis to the search for a program that matches the examples as closely as possible given some error model.

In Table 1 we summarize a set of program synthesis tasks from the Sketch performance benchmarks which are substantially larger than the example above (208–1808 LOC). We considered the problem of finding a best-effort program and we ran the experiments until we found with 80% confidence a "good" program, i.e. one that is correct on at least 95% of the inputs. The experiment shows that MAXCOUNT is able to sample almost correct programs and that the value of k for which this succeeds is much lower than theoretically required. For some of the sketches we were even able to sample fully correct programs, e.g. within 146 seconds using $k = 12$ on the sketch 'GuidanceService'.

Name	$k_{\max}$	k	$Q_{80\%}$	time
ActivityService	952	1	.965	28
ActivityService2	952	1	.965	28
ConcreteActivityService	965	1	.970	33
ConcreteRoleAffectation	10036	-	-	TO
GuidanceService	938	1	.981	24
GuidanceService2	938	1	.981	24
IssueServiceImpl	1046	3	.955	45
IterationService	952	1	.965	29
LoginService	1248	4	.959	89
LoginService2	1370	5	.951	210
NotificationServiceImpl2	1182	13	.957	512
PhaseService	953	1	.965	28
ProcessBean	2248	4	.951	274
ProjectService3	1816	1	.974	81
UserServiceImpl	1181	4	.953	69

Table 1: Experiments on the Sketch performance benchmark set. We indicate $k_{\max}$, the value of k for which we are guaranteed to sample a good program with 80% chance, the first k for which a good program was actually found, the fraction $Q_{80\%}$ of the inputs on which the program is correct with 80% confidence, and the computation time in seconds.

We also tried sampling programs uniformly at random, corresponding to $k = 0$, but this did not result in a good program in any of the cases above. This suggests that while sampling uniformly at random produces programs that fail on most inputs, sampling from programs that are correct on a few randomly selected inputs is sufficient to get programs that work on most inputs. Besides providing partial solutions when no wholly correct solution exists, such best-effort programs may prove useful as a starting point for other program synthesis tools.

Related Work

Functional E-MAJSAT (Pipatsrisawat and Darwiche 2009) is similar to Max#SAT. Given a formula $\varphi(X, Y)$ and a function θ mapping literals of Y to $[0, 1]$, the functional E-MAJSAT problem is to compute $\max_x \sum_{y \models \varphi(x,\cdot)} \prod_{l \in y} \theta(l)$ where $l \in y$ are the literals true in y. If we choose $\theta(l) = 1$ for all literals, we obtain Max#SAT without projection variables Z. We note that although they do not change the theoretical complexity, projection variables are crucial for natural and efficient encoding of applications like QIF. In the other direction, we can reduce functional E-MAJSAT to Max#SAT with the weighted-to-unweighted translation used above.

Recently, Xue et al. (2016) independently proposed an algorithm, XOR_MMAP, for marginal MAP based on universal hashing. Like MAXCOUNT, XOR_MMAP creates a number of copies of the original problem, but they are used in a different way. Also, after adding hash functions MAXCOUNT only requires satisfiability queries, while XOR_MMAP still has to solve *optimization* problems (via MIP solvers).

Conclusion

In this paper we define the Max#SAT problem, which generalizes MaxSAT and #SAT and thereby allows us to optimize over succinctly represented problems. We give an approximation algorithm MAXCOUNT that can estimate, with any desired multiplicative error, the maximum model count of a Boolean formula using only polynomially many queries to an NP oracle. The algorithm can also return witnesses that have a model count close to the maximum.

We demonstrate with examples in quantitative information flow analysis, probabilistic inference, and program synthesis that MAXCOUNT enables interesting new applications. There are many more applications in areas such as probabilistic programming and probabilistic model checking that MAXCOUNT could potentially be applied to, which we leave for future work.

Acknowledgments

We thank Armando Solar-Lezama for providing an extension to Sketch to generate the program synthesis examples used in this paper. We also thank several anonymous reviewers for their helpful comments. This work is supported in part by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1106400, by NSF grants CCF-1139138, CNS-1528108, and CNS-1646208, and by TerraSwarm, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

Alur, R.; Bodik, R.; Juniwal, G.; Martin, M. M. K.; Raghothaman, M.; Seshia, S. A.; Singh, R.; Solar-Lezama, A.; Torlak, E.; and Udupa, A. 2013. Syntax-guided synthesis. In *IEEE International Conference on Formal Methods in Computer-Aided Design*, 1–17.

Argelich, J.; Li, C.-M.; Manya, F.; and Planes, J. 2008. The first and second Max-SAT evaluations. *Journal on Satisfiability, Boolean Modeling and Computation* 4:251–278.

Backes, M.; Köpf, B.; and Rybalchenko, A. 2009. Automatic discovery and quantification of information leaks. In *30th IEEE Symposium on Security and Privacy*, 141–153. IEEE.

Chakraborty, S.; Fremont, D. J.; Meel, K. S.; Seshia, S. A.; and Vardi, M. Y. 2015. On parallel scalable uniform SAT witness generation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 304–319. Springer.

Chakraborty, S.; Meel, K. S.; and Vardi, M. Y. 2013. A scalable approximate model counter. In *Principles and Practice of Constraint Programming*, 200–216. Springer.

Chakraborty, S.; Meel, K. S.; and Vardi, M. Y. 2014. Balancing scalability and uniformity in SAT witness generator. In *51st Design Automation Conference*, 1–6. ACM.

Chakraborty, S.; Meel, K. S.; and Vardi, M. Y. 2016. Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic SAT calls. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI-16)*, 3569–3576.

Clarke, E.; Kroening, D.; and Lerda, F. 2004. A tool for checking ANSI-C programs. In Jensen, K., and Podelski, A., eds., *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, 168–176. Springer.

Davies, J., and Bacchus, F. 2013. Exploiting the power of MIP solvers in MAXSAT. In *Proceedings of Theory and Applications of Satisfiability Testing*, 166–181. Springer.

Dechter, R. 1999. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence* 113(1):41–85.

Ermon, S.; Gomes, C. P.; Sabharwal, A.; and Selman, B. 2013. Taming the curse of dimensionality: Discrete integration by hashing and optimization. In *Proceedings of the 30th International Conference on Machine Learning. JMLR: W&CP volume 28.*

Fu, Z., and Malik, S. 2006. On solving the partial MAX-SAT problem. In *International Conference on Theory and Applications of Satisfiability Testing*, 252–265. Springer.

Heusser, J., and Malacaria, P. 2010. Quantifying information leaks in software. In *26th Annual Computer Security Applications Conference*. ACM.

Klebanov, V.; Manthey, N.; and Muise, C. J. 2013. SAT-based analysis and quantification of information flow in programs. In *Quantitative Evaluation of Systems*.

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

Littman, M. L.; Goldsmith, J.; and Mundhenk, M. 1998. The computational complexity of probabilistic planning. *Journal of Artificial Intelligence Research* 9(1):1–36.

Marinescu, R.; Dechter, R.; and Ihler, A. 2014. AND/OR search for marginal MAP. In *Uncertainty in Artificial Intelligence (UAI)*, 563–572. Citeseer.

Marques-Silva, J., and Planes, J. 2011. Algorithms for maximum satisfiability using unsatisfiable cores. In *Advanced Techniques in Logic Synthesis, Optimizations and Applications*. Springer New York. 171–182.

Mauá, D. D.; De Campos, C. P.; and Cozman, F. G. 2015. The complexity of map inference in bayesian networks specified through logical languages. *Cancer* 1:Y2.

Meng, Z., and Smith, G. 2011. Calculating bounds on information leakage using two-bit patterns. In *Proceedings of PLAS*. ACM.

Newsome, J.; McCamant, S.; and Song, D. 2009. Measuring channel capacity to distinguish undue influence. In *ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*. ACM.

Pipatsrisawat, K., and Darwiche, A. 2009. A new d-DNNF-based bound computation algorithm for functional E-MAJSAT. In *21st international jont conference on Artifical intelligence (IJCAI)*, 590–595.

Safarpour, S.; Mangassarian, H.; Veneris, A.; Liffiton, M. H.; and Sakallah, K. A. 2007. Improved design debugging using maximum satisfiability. In *Proceedigns of the International Conference on Formal Methods in Computer Aided Design (FMCAD)*, 13–19. IEEE.

Sang, T.; Beame, P.; and Kautz, H. 2005. Solving bayesian networks by weighted model counting. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*, volume 1, 475–482.

Solar-Lezama, A.; Tancau, L.; Bodík, R.; Seshia, S. A.; and Saraswat, V. A. 2006. Combinatorial sketching for finite programs. In *12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 404–415. ACM Press.

Toda, S. 1991. PP is as hard as the polynomial-time hierarchy. *SIAM J. Comput.* 20(5):865–877.

Xue, Y.; Li, Z.; Ermon, S.; Gomes, C. P.; and Selman, B. 2016. Solving marginal map problems with np oracles and parity constraints. In Lee, D. D.; Luxburg, U. V.; Guyon, I.; and Garnett, R., eds., *Advances In Neural Information Processing Systems 29*. Curran Associates, Inc. 1127–1135.

Zhang, N. L., and Tian, J., eds. 2014. *Proceedings of the Thirtieth Conference on Uncertainty in Artificial Intelligence, UAI 2014, Quebec City, Quebec, Canada, July 23-27, 2014*. AUAI Press.

QIF Benchmarks

Our QIF benchmarks were written as functions in C taking inputs *public* and *secret*. These were converted into propositional formulas using CBMC (Clarke, Kroening, and Lerda 2004). We now give complete listings of the benchmarks, as well as brief descriptions.

pwd-backdoor

This benchmark models a password checker for 8-byte passwords that has a backdoor. If the backdoor value is given as the public input, the secret is leaked (64 bits); otherwise only whether or not the secret password has been guessed correctly is leaked (1 bit).

```
unsigned long long pwd_backdoor(
      unsigned long long public,
      unsigned long long secret)
{
```

```
    if (public == 4414850668108406108 \
               9415163175489421777ULL)
        return secret;
    if (public == secret)
        return 1;
    else
        return 0;
}
```

bin-search-16

This is a variant of a benchmark from Meng and Smith (2011), where all the bits of the secret are eventually leaked through successive rounds of binary search. We have introduced a public input in place of the constant 1, which allows a wide variety of leaks: if *public* = 0 the leak is 0 bits, if *public* = 1 the leak is 16 bits, and many intermediate values are possible.

```
unsigned short bin_search_16(
        unsigned short public,
        unsigned short secret)
{
    unsigned short out = 0;
    for (int i = 0; i < 16; i++) {
        unsigned short m =
          public << (15 - i);
        if (out + m <= secret)
            out += m;
    }
    return out;
}
```

reverse

This benchmark is based on a classical technique for reversing the bits in a 32-bit word. We have replaced the constant `0x55555555` by the public input: when the input is chosen to be that value, the function reverses the secret and so leaks all 32 bits of it. Many smaller leaks of different sizes exist.

```
unsigned int reverse(
        unsigned int public,
        unsigned int secret)
{
    unsigned int res = secret;
    res = ((res >> 1) & public)
        | ((res & public) << 1);
    res = ((res >> 2) & 0x33333333)
        | ((res & 0x33333333) << 2);
    res = ((res >> 4) & 0x0F0F0F0F)
        | ((res & 0x0F0F0F0F) << 4);
    res = ((res >> 8) & 0x00FF00FF
        | ((res & 0x00FF00FF) << 8));
    res = (res >> 16) | (res << 16);
    return res;
}
```

reverse2

This benchmark is an elaboration of the previous. Two different methods of reversing the bits in a byte are used to reverse the secret, which is then reversed again using the previous method (assuming the public input is set correctly).

```
typedef unsigned char byte;
unsigned int reverse2(
        unsigned int public,
        unsigned int secret)
{
    unsigned long mask = 0x22110UL;

    byte b1 = secret & 0xFF;
    byte b2 = (secret >> 8) & 0xFF;
    byte b3 = (secret >> 16) & 0xFF;
    byte b4 = (secret >> 24) & 0xFF;

    byte t;
    t = ((b1 * 0x0802UL & mask)
        | (b1 * 0x8020UL & 0x88440UL))
        * 0x10101UL >> 16;
    b1 = ((b4 * 0x0802UL & mask)
        | (b4 * 0x8020UL & 0x88440UL))
        * 0x10101UL >> 16;
    b4 = t;

    t = ((b2 * 0x80200802ULL)
       & 0x0884422110ULL)
        * 0x0101010101ULL >> 32;
    b2 = ((b3 * 0x80200802ULL)
       & 0x0884422110ULL)
        * 0x0101010101ULL >> 32;
    b3 = t;

    unsigned int res = b4;
    res = (res << 8) | b3;
    res = (res << 8) | b2;
    res = (res << 8) | b1;

    res = ((res >> 1) & public)
        | ((res & public) << 1);
    res = ((res >> 2) & 0x33333333)
        | ((res & 0x33333333) << 2);
    res = ((res >> 4) & 0x0F0F0F0F)
        | ((res & 0x0F0F0F0F) << 4);
    res = ((res >> 8) & 0x00FF00FF
        | ((res & 0x00FF00FF) << 8));
    res = (res >> 16) | (res << 16);

    return res;
}
```

backdoor-2x16-8

This benchmark models a program which normally leaks 8 bits of the 32 bit secret. However, there are two separate backdoors, each of which causes all 32 bits to be leaked. The backdoors leak different sets of bits to ensure that CBMC will not statically determine some bits to be fixed.

```
unsigned int backdoor_2x16_8(
        unsigned int public,
        unsigned int secret)
{
    if (public == 0x42CB88FF)
        return secret & 0x0000FFFF;
    else if (public == 0xC141F975)
        return secret & 0xFFFF0000;
    else
        return secret & 0x000000FF;
}
```

backdoor-32-24

This benchmark models a program which normally leaks 24 bits of the 32 bit secret, but has a backdoor causing it to leak all 32 bits.

```
unsigned int backdoor_32_24(
        unsigned int public,
        unsigned int secret)
{
    if (public == 0x42CB88FF)
        return secret;
    else
        return secret & 0x00FFFFFF;
}
```

CVE-2007-2875

This benchmark is taken directly from Heusser and Malacaria (2010). If the public input is too large, an integer underflow occurs and secret data is unintentionally returned.

```
int CVE_2007_2875(
        int secret,
        long long public) {
  int bufsz;
  unsigned int nbytes;
  bufsz=1024;
  nbytes=20;
  if (public + nbytes > bufsz)
       nbytes = bufsz - public;
  if(public + nbytes > bufsz) {
     return secret;
  } else {
     return 0;
  }
}
```

CVE-2009-3002

This is another benchmark from Heusser and Malacaria (2010). The structure `sat` is stored in kernel memory but not completely initialized, so that (potentially sensitive) parts of kernel memory are leaked. The code was reconstructed from the original Linux kernel source, simplified to omit some irrelevant error-checking code and structure fields. For convenience we have also consolidated all of the public inputs into a single structure `getname_query`.

```
struct sock { int foo; };
struct socket { struct sock *sk; };
struct atalk_sock {
        struct sock sk;
        unsigned short dest_net;
        unsigned short src_net;
        unsigned char dest_node;
        unsigned char src_node;
        unsigned char dest_port;
        unsigned char src_port;
};
struct atalk_addr {
        unsigned short s_net;
        char s_node;
};
struct sockaddr_at {
        unsigned short sat_family;
        char sat_port;
        struct atalk_addr sat_addr;
        char sat_zero[8];
};
struct getname_query {
        struct atalk_sock at;
        struct sockaddr_at uaddr;
        int uaddr_len;
        int peer;
};

struct getname_query CVE_2009_3002(
        struct sockaddr_at sat
        struct getname_query query)
{
  struct atalk_sock *at = &query.at;
  struct sockaddr_at *uaddr =
       &query.uaddr;
  int *uaddr_len = &query.uaddr_len;
  int peer = query.peer;

  *uaddr_len =
        sizeof(struct sockaddr_at);

  if(peer) {
    sat.sat_addr.s_net = at->dest_net;
    sat.sat_addr.s_node =
        at->dest_node;
    sat.sat_port = at->dest_port;
  } else {
    sat.sat_addr.s_net = at->src_net;
    sat.sat_addr.s_node = at->src_node;
    sat.sat_port = at->src_port;
  }

  sat.sat_family = 42;

  uaddr->sat_family = sat.sat_family;
  uaddr->sat_port = sat.sat_port;
  uaddr->sat_addr.s_net =
        sat.sat_addr.s_net;
  uaddr->sat_addr.s_node =
```

```
        sat.sat_addr.s_node;
    uaddr->sat_zero[0] = sat.sat_zero[0];
    uaddr->sat_zero[1] = sat.sat_zero[1];
    uaddr->sat_zero[2] = sat.sat_zero[2];
    uaddr->sat_zero[3] = sat.sat_zero[3];
    uaddr->sat_zero[4] = sat.sat_zero[4];
    uaddr->sat_zero[5] = sat.sat_zero[5];
    uaddr->sat_zero[6] = sat.sat_zero[6];
    uaddr->sat_zero[7] = sat.sat_zero[7];

    return query;
}
```