Abstract

Quantitative information flow analyses (QIF) are a class of techniques for measuring the amount of confidential information leaked by a program to its public outputs. Shannon entropy is an important method to quantify the amount of leakage in QIF. This paper focuses on the programs modeled in Boolean clausal constraints and optimises the two stages of the Shannon entropy computation to implement a scalable precise tool PSE. In the first stage, we design a knowledge compilation language called ADD-L that combines Algebraic Decision Diagrams and implied Literals. ADD-L avoids enumerating possible outputs of a program and supports tractable entropy computation. In the second stage, we optimise the model counting queries that are used to compute the probabilities of outputs. We evaluate the performance of PSE, and the experiments demonstrate that PSE significantly enhances the scalability of precise entropy computation and even outperforms the state-of-the-art approximate tool EntropyEstimation.

Introduction

Quantitative information flow (QIF) is an important approach to measuring the amount of information leaked about a secret by observing the running of a program (**??**). In QIF, we often quantify the leakage using entropy-theoretic notions, such as Shannon entropy (**????**) or min-entropy (**????**). Roughly speaking, a program in QIF can be seen as a function from a set of secret inputs X to outputs Y observable to an attacker who may try to infer X based on the output Y. Boolean clausal constraints are a basic representation to model programs (**??**). In this paper, we focus on precisely computing the Shannon entropy of a program expressed in Boolean clausal constraints.

Let $\varphi(X, Y)$ be the Boolean formula modeling the relationship between the input variable set X and the output variable set Y in the given program. We require that for any assignment of X, there is at most one assignment of Y that satisfies the formula $\varphi(X, Y)$. Let p represent a probability distribution defined over the set $\{0,1\}^Y$. For each assignment σ to Y, i.e., $\sigma : Y \mapsto \sigma$, the probability is defined as $p_\sigma = \frac{|Sol(\varphi(Y \mapsto \sigma))|}{|Sol(\varphi)_{\downarrow X}|}$, where $Sol(\varphi(Y \mapsto \sigma))$ denotes the set of solutions of $\varphi(Y \mapsto \sigma)$ and $Sol(\varphi)_{\downarrow X}$ denotes the set of solutions of φ projected to X. The Shannon entropy of φ is $H(\varphi) = \sum_{\sigma \in 2^Y} -p_\sigma \log p_\sigma$. Then we can immediately obtain a measure of leaked information with the computed entropy and the assumption that X follows a uniform distribution [1] (**?**).

The workflow of the current precise methods of computing entropy can often be divided into two stages. In the first stage, we enumerate possible outputs, i.e., the satisfying assignments over Y, while in the second stage, we compute the probability of the current output based on the number of inputs mapped to the output (**?**). The computation in the second stage often invokes model counting (#SAT), which refers to computing the number of solutions $Sol(\varphi)$ for a given set of clausal constraints φ. Due to the exponential possible outputs, the current precise methods are often difficult to scale to programs with a large size of Y. While this paper focuses on improving the precise computation of entropy, we remark that Priyanka et al. (**?**) proposed the first Shannon entropy estimation tool, EntropyEstimation, that guarantees that the estimate lies within $(1 \pm \epsilon)$-factor of $H(\varphi)$ with confidence at least $1 - \delta$. EntropyEstimation uses uniform sampling to avoid generating all outputs, and indeed scales much better than the precise methods.

As we mentioned previously, the current methods for precise Shannon entropy computing are difficult to scale to clausal constraints with a large set of outputs. Theoretically, we sometimes need to perform $2^{|Y|}$ model counting queries. The main contribution of this paper is to improve the scalability of precise computation of Shannon entropy. We enhance the computation process in both stages of precise Shannon entropy computing. For the first stage, we design a knowledge compilation language to guide the search to avoid exhausting the possible outputs. The language combines Algebraic Decision Diagrams (ADD), an influential representation, and implied literals, an important notion in Boolean satisfiability solving. For the second stage, we do not individually perform model counting queries. Instead, we share the component caching with the successive model counting queries. Moreover, we take advantage of literal equivalence to pre-process the clausal constraints corresponding to a given program. Integrating all the techniques mentioned above, we propose a Precise Shannon Entropy tool PSE. We conducted an extensive experimental evaluation over a comprehensive set of benchmarks (361 in total) and compared PSE with the existing precise Shannon entropy computing methods and the current state-of-the-art Shannon entropy estimation tool, EntropyEstimation. Our experiments show that the existing precise Shannon entropy algorithm solves only 17 instances, while PSE can solve 289 instances, representing a significant improvement of 272 instances. EntropyEstimation can solve 264 instances, while PSE can solve 25 more instances, a surprising improvement.

The rest of the paper is structured as follows. We present notation and background in Section . We introduce Algebraic Decision Diagrams (ADD) with implication literals in Section . Section presents the application of ADD-L to QIF and introduces our precise entropy tool, PSE. Section 15 details the specific results and analysis of the experiments. Section 15 discusses related work. Finally, we conclude in Section 15.

Notations and Background

In this paper, we focus on the programs modeled by (Boolean) formulas. In the formulas being discussed, the symbols x and y denote variables, and literal l refers to either the variable x or its negation $\neg x$, where $var(l)$ represents the variable underlying the literal l. $PV = \{x_0, x_1, ..., x_n, ...\} \cup \{y_0, y_1, ..., y_n, ...\}$ denotes a set of Boolean variables. A formula φ is constructed from the constants *true*, *false* and variables using negation operator $\neg$, conjunction operator $\wedge$, disjunction operator $\vee$ and equality operator $\leftrightarrow$, where $Vars(\varphi)$ denotes the set of variables appearing in φ. A

[1] If X does not follow a uniform distribution, techniques exist for reducing the analysis to a uniform case (**?**).

clause C (resp. term T) is a set of literals representing their disjunction (resp. conjunction). A formula in conjunctive normal form (CNF) is a set of clauses representing their conjunction. Given a formula φ, a variable x, and a constant b, the substitution $\varphi[x \mapsto b]$ refers to the transformed formula obtained by substituting the occurrence of x with b throughout φ.

An assignment σ over variable set V is a mapping from V to $\{true, false\}$, and can be seen as a term $T(\sigma) = \bigwedge_{x \mapsto true \in \sigma} x \wedge \bigwedge_{x \mapsto false \in \sigma} \neg x$. The set of all assignments over V is denoted by 2^V. Given a subset $V' \subseteq V$, $\sigma_{\downarrow V'} = \{x \mapsto b \in \sigma \mid x \in V'\}$. Given a formula φ, an assignment over $Vars(\varphi)$ satisfies φ ($\sigma \models \varphi$) if the substitution $\varphi[\sigma]$ is equivalent to $true$. A satisfying assignment is also called solution or model. We use $Sol(\varphi)$ to the set of solutions of φ, and model counting is the problem of computing $|Sol(\varphi)|$. Given two formulas φ and ψ over V, $\varphi \models \psi$ iff $Sol(\varphi) \subseteq Sol(\psi)$. An implied literal l of a formula φ satisfies $\varphi \models l$. Given a formula, we can use its implied literals to simplify it.

Example 0.1. *Given a CNF formula $\varphi = x_1 \wedge (x_1 \vee x_2 \vee x_5) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_4 \vee \neg x_5)$, it has three implied literals $\{x_1, \neg x_2, x_3\}$. Then, φ can be simplified as $x_1 \wedge \neg x_2 \wedge x_3 \wedge (x_4 \vee \neg x_5)$.*

Circuit formula and its Shannon entropy

Given a formula $\varphi(X, Y)$ to represent the relationship between input variables X and output variables Y, if $\sigma_{1\downarrow X} = \sigma_{2\downarrow X} \implies \sigma_1 = \sigma_2$ for each $\sigma_1, \sigma_2 \in Sol(\varphi)$, then we say φ is a circuit formula. It is standard in the security community to employ circuit formulas to model programs in QIF (?).

Example 0.2. *The following CNF formula is a circuit formula with input variables $X = \{x_1, x_2, x_3, x_4\}$ and output variables $Y = \{y_1, y_2, y_3\}$:*

$$\begin{aligned}\varphi(X,Y) =&(x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee y_1) \wedge (x_1 \vee \neg y_1) \wedge (x_3 \vee x_4 \vee y_1) \wedge \\ &(\neg x_3 \vee \neg x_4 \vee y_3) \wedge (x_3 \vee \neg y_3) \wedge (x_4 \vee \neg y_3) \wedge (y_1 \vee \neg y_2) \wedge (\neg x_1 \vee \neg y_2).\end{aligned}$$

Proposition 0.1. *Given a circuit formula φ, the projected model counting of formula φ under X is equal to the number of models of formula φ, i.e. $|Sol(\varphi)_{\downarrow X}| = |Sol(\varphi)|$*

Proof. Let $\sigma_{\downarrow X}$ represent the assignment of variables restricted to X. Since φ is a circuit formula, each input uniquely corresponds to an output. If for each $\sigma_1, \sigma_2 \in Sol(\varphi)$, we have $\sigma_{1_{\downarrow X}} = \sigma_{2_{\downarrow X}} \implies \sigma_{1_{\downarrow X}} = \sigma_{2_{\downarrow X}}$. At the same time, we also know that $X \subseteq Vars(\varphi)$. So, we have $|Sol(\varphi)_{\downarrow X}| = |Sol(\varphi)|$ □

In the computation of Shannon entropy, we focus on the probability of each output. Let p be a probability distribution defined over the set $\{0,1\}^Y$. For each assignment σ to Y, i.e., $\sigma : Y \mapsto \{0,1\}$, the probability is defined as $p_\sigma = \frac{|Sol(\varphi(Y \mapsto \sigma))|}{|Sol(\varphi)_{\downarrow X}|}$, where $Sol(\varphi(Y \mapsto \sigma))$ denotes the set of solutions of $\varphi(Y \mapsto \sigma)$ and $Sol(\varphi)_{\downarrow X}$ denotes the set of solutions of φ projected to X. Then, the entropy of φ is $H(\varphi) = \sum_{\sigma \in 2^Y} -p_\sigma \log p_\sigma$. To facilitate the description of the algorithm, we define the self-information of a subformula $\varphi \wedge T(\sigma)$ with $Vars(T(\sigma)) \subseteq Y$ as $infor(\sigma, \varphi) = \sum_{\sigma' \in 2^Y \wedge \sigma' \models T(\sigma)} -p_{\sigma'} \log p_{\sigma'}$. It is obvious that the self-information of φ is the entropy. Moreover, $\varphi[\sigma]$ is still a circuit formula with the output variables $Y \setminus Vars(T(\sigma))$.

Algebraic Decision Diagrams

The Algebraic Decision Diagram with Propositional Language Containing Implied Literals (ADD-L) proposed in this paper is an extended form based on Algebraic Decision Diagram (ADD (?)). ADD is an extended version of BDD that incorporates multiple terminal nodes, each assigned a real-valued. ADD is a compact representation of a real-valued function as a directed acyclic graph. For functions that have a logical structure, the ADD notation can be exponentially smaller than the explicit notation.

The original design motivation for ADD was to solve matrix multiplication, shortest path algorithms, and direct methods for numerical linear algebra (?). In subsequent research, ADD has also been used for stochastic model checking (?), stochastic programming (?), and weighted model counting (?).

We consider an ADD consisting of a quadruple (X, S, π, G), where X is the set of variables of a Boolean formula, S is an arbitrary set (called the carrier set), and π is the order of occurrence of the variables, G is a rooted directed acyclic graph (?). An ADD is a rooted directed acyclic graph G, where the labels of the non-terminal nodes are elements in the set X and have two outgoing edges labeled 0 and 1 (referred to as lo edge and hi edge in this paper), and all the terminal nodes are labeled with an element in the set S. The order of the appearance of the labels of non-terminal nodes in all paths from the root to the terminal nodes in G must be consistent with the order of the elements in π.

ADD-L: A New Probabilistic Representation

In order to compute the Shannon entropy of a circuit formula $\varphi(X, Y)$, we need to use the probability distribution over the outputs. Algebraic Decision Diagrams (ADDs) are an influential compact probability representation that can be exponentially smaller than the explicit representation. Macii and Poncino (?) showed that ADD supports efficient exact computation of entropy. However, we observed in the experiments that the sizes of ADDs often exponentially explode with large circuit formulas. We take inspiration from a Boolean representation called Ordered Binary Decision Diagram with implied Literals (OBDD-L) (?), which reduces its size by recursively extracting implied literals and can be exponentially smaller than the original OBDD. Accordingly, we propose a probabilistic representation called Algebraic Decision Diagrams with implied Literals (ADD-L) and show it supports tractable entropy computation. ADD-L is a general form of ADD and is defined as follows:

Definition 0.1. *An ADD-L is a rooted DAG, where each node u is either terminal or non-terminal. Each terminal node u is labeled with a real weight $\omega(u)$ and a set of implied literals $L(u)$. Each non-terminal node u is associated* *with a variable $var(u)$, two children, and a set of implied literals $L(u)$, which satisfies that $var(u)$ does not appear in $L(u)$ and that each variable appearing in $L(u)$ does not appear in the descendants of u. The children of non-terminal node u are referred to as the low child $lo(u)$ and the high child $hi(u)$, and connected by dashed lines and solid lines, respectively, corresponding to the cases where $var(u)$ is assigned the value of false and true. An ADD-L is imposed with a linear ordering $\prec$ of variables such that given a node u and its non-terminal child v, $var(u) \prec var(v)$.*

Hereafter, we denote to the set of variables that appear in v (i.e., the variables in $var(v)$ and $L(v)$ as well as all the variables that appear in its descendant nodes) as $Vars(v)$. It is obvious that when each set of implied literals is empty, an ADD-L is equivalent to an ADD. In our experiments, we observed that given a large circuit formula, assigning a decision variable often results in a large number of implied literals, and therefore, the ADD-L often has a much smaller size than the corresponding ADD. We now turn to show that how an ADD-L defines a probability distribution:

Definition 0.2. *Let u be an ADD-L node over a set of variables Y and let σ be an assignment over Y. The weight of σ is defined as follows:*

$$\omega(\sigma, u) = \begin{cases} 0 & \sigma \not\models L(u) \\ \omega(u) & u \text{ is terminal and } \sigma \models L(u) \\ \omega(\sigma, lo(u)) & u \text{ is non-terminal and } \sigma \models L(u) \cup \{\neg var(u)\} \\ \omega(\sigma, hi(u)) & u \text{ is non-terminal and } \sigma \models L(u) \cup \{var(u)\} \end{cases}$$

The weight of an non-constant ADD-L rooted at u is denoted by $\omega(u)$ and defined as $\sum_{\sigma \in 2^{Vars(u)}} \omega(\sigma, u)$. The probability of σ over u is defined as $p(\sigma, u) = \frac{\omega(\sigma,u)}{\omega(u)}$.

Figure 1 depicts an ADD-L to represent the probability distribution of the circuit formula in Example 0.2 over its outputs. Let u be its root and let $\sigma = \{y_1 \mapsto false, y_2 \mapsto false, y_3 \mapsto false\}$. By Definition 0.2, $\omega(\sigma, u) = 2$, $\omega(u) = 7$, and $p(\sigma, u) = \frac{2}{7}$.

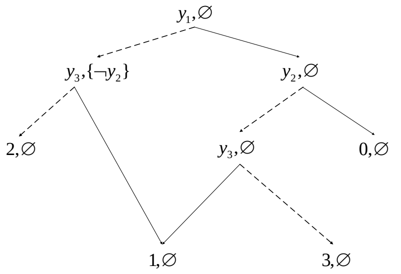

Figure 1: An ADD-L example corresponding to Example 0.2, where the weight of a terminal node is the model count of $\varphi[\sigma]$ (σ is a partial assignment corresponding to the decisions on a path from the root to it)

Tractable Computation of Weight and Entropy

The computation of Shannon entropy of ADD-L depends on the computation of its weight. We first show that for an ADD-L node u, we can compute $\omega(u)$ in polynomial time.

Proposition 0.2. *Given a non-terminal node u in ADD-L, its weight $\omega(u)$ can be recursively computed as follows in polynomial time:*

$$\omega(u) = 2^{n_0} \cdot \omega(lo(u)) + 2^{n_1} \cdot \omega(hi(u))$$

where $n_0 = |Vars(u)| - |Vars(lo(u))| - 1 - |L(u)|$ and $n_1 = |Vars(u)| - |Vars(hi(u))| - 1 - |L(u)|$.

Proof. The time complexity is immediate by using dynamic programming. We prove the equation can compute the weight correctly by induction on the number of variables of the ADD-L rooted at u. It is obvious that the weight of a terminal node is the real value labeled. Assume that when $|Vars(u)| \leq n$, this proposition holds. For the case where $|Vars(u)| = n+1$, we use Y_0 and Y_1 to denote $Vars(lo(u))$ and $Vars(hi(u))$, and we have $|Y_0| \leq n$ and $|Y_1| \leq n$. Thus, $\omega(lo(u))$ and $\omega(hi(u))$ can be computed correctly. According to Definition 0.2, $w(u) = \sum_{\sigma \in 2^{Vars(u)}} \omega(\sigma, u)$. The assignments over $Vars(u)$ can be divided into three categories:

- The assignment $\sigma \not\models L(u)$: $\omega(\sigma, u) = 0$.
- The assignment $\sigma \models L(u) \cup \{\neg var(u)\}$: It is obvious that $\omega(\sigma, u) = \omega(\sigma_{\downarrow Y_0}, lo(u))$. Each assignment over Y_0 can be extended to exactly 2^{n_0} different assignments over $Vars(u)$ in this category. Thus, we have the following equation:
$$\sum_{\sigma \in 2^{Vars(u)} \wedge \sigma \models L(u) \cup \{\neg var(u)\}} \omega(\sigma, u) = 2^{n_0} \cdot \omega(lo(u)).$$
- The assignment $\sigma \models L(u) \cup \{var(u)\}$: This case is similar to the above case.

To sum up, we can obtain that $\omega(u) = 2^{n_0} \cdot \omega(lo(u)) + 2^{n_1} \cdot \omega(hi(u))$. □

Now we explain how ADD-L computes its Shannon entropy in polynomial time.

Proposition 0.3. *Given an ADD-L rooted at u, we use $infor(v)$ to denote the self-information of the subgraph rooted at v. We can recursively compute $infor(v)$ in polynomial time as follows:*

$$infor(v) = \begin{cases} -\frac{\omega(v)}{\omega(u)} \cdot \log \frac{\omega(v)}{\omega(u)} & v \text{ is terminal} \\ 2^{n_0} \cdot infor(lo(u)) + 2^{n_1} \cdot infor(hi(u)) & otherwise \end{cases}$$

where $n_0 = |Vars(v)| - |Vars(lo(v))| - 1 - |L(v)|$ and $n_1 = |Vars(v)| - |Vars(hi(v))| - 1 - |L(v)|$. The entropy of the probability distribution represented by u satisfies $H(u) = infor(u)$.

Proof. According to Proposition 0.2, $\omega(u)$ can be computed in polynomial time, and therefore the time complexity in this proposition is obvious. According to Definition 0.2, $H(u) = -\sum_{\sigma \in 2^{Vars(u)}} \frac{\omega(\sigma,u)}{\omega(u)} \log \frac{\omega(\sigma,u)}{\omega(u)}$. We modify the weight of each terminal node v in the ADD-L rooted at u as $-\frac{\omega(v)}{\omega(u)} \cdot \log \frac{\omega(v)}{\omega(u)}$ and assume that the root of the new ADD-L is u'. It is obvious that $\omega(u') = infor(u)$. According to

Proposition 0.2, we know the equation in this proposition can compute $\omega(u')$ correctly. Then this proposition holds due to the fact $H(u) = \omega(u')$. $\square$

We close this section by explaining why we use the orderedness in the design of ADD-L. Actually, Propositions 0.2–0.3 still hold when we only use the more general read-once property: each variable appears at most once in each path from the root of an ADD-L to a terminal node. First, we observed in our experiments that the linear ordering given by minfill in our tool PSE works better than the dynamic orderings in the state-of-the-art model counters, where the former imposes the orderedness and the latter imposes the read-once property. Second, ADD-L can provide tractable equivalence checking between probability distributions beyond this study.

PSE: Scalable Precise Entropy Computation

In this section, we present our tool, PSE, which computes the Shannon entropy of a given circuit CNF formula over its outputs. Similar to other Shannon entropy tools, the main process of PSE is divided into two stages: the Y-stage (corresponding to outputs) and the X-stage (corresponding to inputs). During the Y-stage, we conduct a search pertaining to ADD-L in order to compute the Shannon entropy precisely. For the X-stage, we perform multiple optimized model counting. PSE, depicted in Algorithm 1, takes in a CNF formula φ, an input set X, and an output set Y, and returns the self-information $infor(\sigma, \varphi)$ of the formula, where σ is the assignment made in the ancestor calls. As mentioned in Proposition 0.3, the entropy of the original formula is the output of the root call.

Algorithm 1: PSE(φ,X,Y)

```
Input: A circuit CNF formula $\varphi$ with inputs $X$ and outputs $Y$
Output: the self-information of $\varphi$
1  if $\varphi = false$ then return $0$
2  $n \leftarrow |Vars(\varphi)|$
3  $L \leftarrow$ GetImpliedLiterals($\varphi$)
4  $\varphi, Y \leftarrow$Simplify($\varphi, Y, L$)
5  if $Cache(\varphi) \neq nil$ then return $Cache(\varphi)$
6  if $Y = \emptyset$ then
7      $m \leftarrow$ CountModels($\Phi$)
8      $p \leftarrow \frac{m}{TotalCount}$
9      return $Cache(\varphi) \leftarrow -p \cdot \log p$
10 $y \leftarrow$ PickGoodVar($Y$)
11 $\varphi_0 \leftarrow \varphi[y \mapsto false]$
12 $\varphi_1 \leftarrow \varphi[y \mapsto true]$
13 $infor_0 \leftarrow$ PSE($\varphi_0, X, Y \backslash y$)
14 $infor_1 \leftarrow$ PSE($\varphi_1, X, Y \backslash y$)
15 return $Cache(\varphi) \leftarrow 2^{n-|Vars(\varphi_0)|-|L|-1} \cdot infor_0 +$
       $2^{n-|Vars(\varphi_1)|-|L|-1} \cdot infor_1$
```

We first deal with the cases in which φ is *false* in line 1. When the formula φ is *false*, its self-information with respect to the assignment made in the ancestor calls is 0. In line 2, we record the number of variables in φ. Then, in line 3, we utilize the `GetImpliedLiterals` function to extract the implied literals of φ. The implementation of `GetImpliedLiterals` relies on implicit Boolean Constraint Propagation (i-BCP) (**?**). In line 4, we simplify φ and Y based on the extracted implied literals L. Detect whether the simplified formula φ has been stored in the cache in line 5, and return its $infor$ if φ hits the cache. If the current Y is an empty set (in line 6), it means that a satisfiable assignment under the restriction of the output set Y has been obtained. We did not directly handle the case where φ is $true$, because in such a scenario, Y is inevitably an empty set, which is a prominent characteristic of the circuit formula. Therefore, the case where Y is an empty set already encompasses the scenario where φ is $true$. Lines 7–9 query model counting for the residual formula and calculate its $infor$, corresponding to the terminal case of proposition 0.3. $TotalCount$ represents the total number of models for the original formula, i.e., for the original formula φ, $TotalCount = |Sol(\varphi)|$. This calculation is obtained directly through the invocation of any model counter before PSE. In line 9, the $infor$ is stored in the cache and returned as the result. If any remaining variables in Y have not been assigned values, we make decisions about them in line 10. `PickGoodVar` is a heuristic algorithm that heuristically selects a variable from Y (the selection method depends on the heuristic used). Next, in lines 11–12, we separately obtain the residual formulas φ_0 and φ_1 for the variable y being *false* and $true$, respectively, and recursively calculate their $infor$ in lines 13–14. Since φ is a circuit formula, all residual formulas in the recursive process after the variables in decision Y are also circuit formulas. Finally, we compute the information for φ (corresponding to the non-terminal case of proposition 0.3) and store it in the cache, returning it as the result in line 15.

Implementation

We now discuss the implementation details that are crucial for the runtime efficiency of PSE. In particular, we adopt many techniques in the state-of-the-art model counters due to the close relationship between entropy computation and model counting.

First, we can use different methods for the model counting query `CountModels` in line 7. The first considered method is to individually employ state-of-the-art model counters, such as SharpSAT-TD (**?**), Ganak (**?**), ExactMC (**?**), and D4 (**?**). The second method, called **ConditionedCounting**, requires the prior construction of a representation of the original formula φ to support tractable model counting. Eligible languages include d-DNNF (**?**), OBDD[$\wedge$] (**?**), and SDD (**?**). When we reach line 7, we perform conditioned model counting with the compiled representation and the partial assignment made in the ancestor calls. The last method, called **SharedCounting**, also solves model counting query by invoking the exact model counter. Unlike the first method, this method does not just query the model counting individually. Instead, we share the component cache in the counter for all model counting queries, and we refer to this strategy as **XCache**. To distinguish it from the cache method in the X-stage, we refer to the cache method in the Y-stage as **YCache** in the following. We observed that the last method works best in PSE.

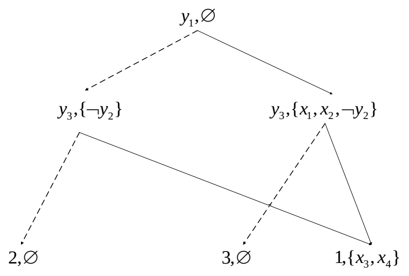

Figure 2: An ADD-L example corresponding to Example 0.2, where the implied literals include the variables in X.

Implied Literal From the definition of entropy, the assignment σ is restricted to Y, so the implied literals theoretically only contain variables from Y. However, we observed that the structure of ADD-L can be further simplified and optimized if the implied literals are not limited to variables in Y but can also include variables in X. Based on this observation, we made some improvements to the computation of implied literals, removing restrictions on implied literals to allow them to include variables in X. Additionally, we found that this technique can enhance the efficiency of the search without compromising the accuracy of entropy computation. We use an example to illustrate the correctness and effectiveness of this idea, as shown in Figure 2. From the comparison between Figure 1 and 2, it is easy to see that the approach optimizes the structure of ADD-L without affecting the entropy result.

Variable Decision Heuristic We apply the current state-of-the-art model counting heuristics to the Shannon entropy computing problem and perform experimental comparisons, including **VSADS** (**?**), **minfill** (**?**), **SharpSAT-TD heuristic** (**?**), and **DLCP** (**?**). We experimented with these heuristics, and our experiments show that the **minfill** heuristic is the one that performs best in the entropy computing problem. We use the **minfill** heuristic by default in our subsequent experiments.

Preprocessing Since model counting is the core of entropy computing, we extend the preprocessing technique based on literal equivalence in model counting and integrate it into our entropy tool, PSE. This idea is inspired by the work of Lai et al. (**?**) on capturing literal equivalence in knowledge compilation. However, due to the special nature of the entropy computing process, we cannot arbitrarily replace the literals corresponding to variables in the output set with equivalent ones. This is because the new formula φ' obtained after the equivalent substitution of literals is equivalent to φ in model counting, but not necessarily equivalent in entropy (it is not equivalent when the variables of substituted literals belong to Y). In the beginning, we had the simple idea of restoring all equivalent literals after the calculation so as not to affect the calculation of the entropy and, at the same time, reduce the size of the formula. We improve this idea by restoring only the literals corresponding to the variables in Y, since the assignments that affect the entropy are only in the part of Y. The new preprocessing method is called **Pre** in the following. Experiments have demonstrated that this preprocessing method provides some performance enhancement to the tool (See Section 15). This idea of preprocessing is motivated by the fact that, on the one hand, after preprocessing using literal equivalence, it can simplify the formula and thereby improve the efficiency of the subsequent model counting. On the other hand (and more importantly), it reduces the treewidth of the tree decomposition, which leads to a more optimal order of variable selection obtained by the tree decomposition heuristic, and can be very effective in improving the efficiency of PSE calculations.

Experiments

To evaluate the performance of PSE, we implemented a prototype in C++ that employs ExactMC as the model counter. We evaluate PSE in 361 benchmarks, including the QIF benchmarks (**?**), combinatorial circuits, plan recognition (**?**), bit-blasted versions of SMTLIB benchmarks (**?**), and QBFEval competitions (**?**). All experiments were run on a computer with Intel(R) Core(TM) i9-10920X CPU @ 3.50GHz and 32GB RAM. RAM. Each instance was run on a single core with a timeout of 3000 seconds and 8GB memory.

Through our evaluations and analysis, we sought to answer the following research questions:

- **RQ1:** How does the runtime performance of PSE compare to the state-of-the-art precise Shannon entropy tools?
- **RQ2:** How does the runtime performance of PSE compare to the state-of-the-art Shannon entropy estimation tool?
- **RQ2:** How do the utilized methods impact the runtime performance of PSE?

Comparison with precise Shannon entropy tools

instance	$\lvert X\rvert$	$\lvert Y\rvert$	baseline-SharpSAT-TD		baseline-ExactMC		PSE	
			Entropy	Time(s)	Entropy	Time(s)	Entropy	Time(s)
blasted_case102.cnf	11	23	8	81.29	8	0.39	8	0.16
s27_15_7.cnf	7	25	3.3	142.77	3.3	0.15	3.3	0.14
small-bug1-fixpoint-5.cnf	66	21	12.81	79.51	12.81	2.99	12.81	0.18
small-bug1-fixpoint-6.cnf	79	25	15.31	1406.47	15.31	51.9	15.31	0.21
blasted_case144.cnf	138	627	-	-	-	-	77.62	35.42
s1423a_15_7.cnf	91	773	-	-	-	-	88.17	232.65
s382_15_7.cnf	24	326	-	-	-	-	23.58	0.22
CVE-2007-2875.cnf	752	32	-	-	-	-	32	0.72
10.sk_1_46.cnf	47	1447	-	-	-	-	13.58	0.18

Table 1: Entropy computing performance of baseline and PSE. ”-” represents that the entropy cannot be solved within the specified time limit.

The existing precise Shannon entropy tools do not use the techniques in the state-of-the-art model counters. Just like (**?**), we implemented the precise Shannon entropy baseline with state-of-the-art model counting techniques. In the baseline, we enumerate each assignment $\sigma \in Sol(\varphi)_{\downarrow Y}$ and compute $p_\sigma = \frac{|Sol(\varphi(Y \mapsto \sigma))|}{|Sol(\varphi)_{\downarrow X}|}$, where $Sol(\varphi(Y \mapsto \sigma))$ denotes the set of solutions of $\varphi(Y \mapsto \sigma)$ and $Sol(\varphi)_{\downarrow X}$ denotes the set of solutions of φ projected to X. As can be seen from the previous proposition, $|Sol(\varphi)_{\downarrow X}|$ can be replaced by $|Sol(\varphi)|$. Finally, entropy is computed as $H(\varphi) = \sum_{\sigma \in 2^Y} -p_\sigma \log p_\sigma$. For a formula with an output set size of m, 2^m model counting queries are required, and we chose the state-of-the-art model counters SharpSAT-TD and ExactMC to the model counting queries, respectively.

In our experiment, the baseline with SharpSAT-TD (resp. ExactMC) can only solve 17 instances within the specified time of 3000 seconds, while PSE can solve 289 instances. Table 1 shows the comparison between baseline and PSE on some instances. The results show a significant improvement in the efficiency of PSE for computing the precise Shannon entropy, which answers **RQ1** as well.

Comparison with Shannon entropy estimation tool

We compare PSE with EntropyEstimation, the current state-of-the-art Shannon entropy estimation tool on 361 instances. Table 2 shows the performance of PSE and EntropyEstimation (**?**) in entropy computing.

tool	Instances Solved			Average PAR-2 score
	Unique	Fastest	Total	
EntropyEstimation	10	0	264	1574.59
PSE	**35**	**254**	**289**	**1193.37**

Table 2: Detailed performance comparison of PSE and EntropyEstimation. Unique represents the number of instances that can only be solved by a specific tool. Fastest represents the number of instances that a tool solves with the shortest time.

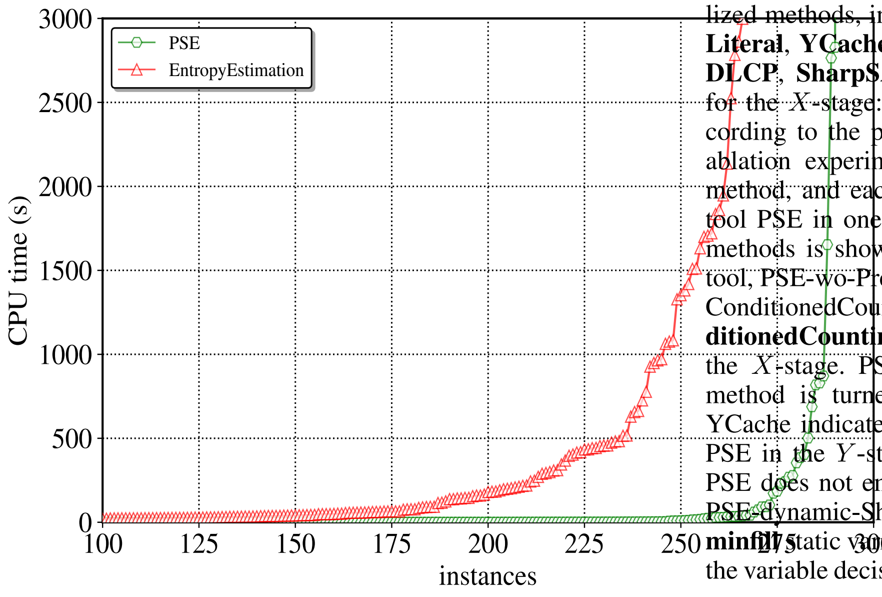

Figure 3: Cactus plot comparing the computing time of PSE and EntropyEstimation.

From Table 2, it can be seen that out of a total of 361 instances, under the time limit of 3000 seconds, our tool PSE is able to solve 25 more instances than EntropyEstimation, and the overall performance is better. For all instances where both PSE and EntropyEstimation can be solved, PSE has a shorter computing time, as well as a much smaller average PAR-2 score [2] than EntropyEstimation. Figure 3 shows the cactus plot of runtime for PSE and EntropyEstimation, where the x-axis represents the number of benchmarks and the y-axis represents the runtime.

Meanwhile, we also observed that there were some instances that PSE could not solve while EntropyEstimation could. We observed in our experiments that in the instances where PSE could not be solved, the number of model counting queries was so high that it had reached an order of magnitude of ten million and still did not solve for entropy. We believe this may have reached the bottleneck of precise Shannon entropy computing. Therefore, we believe that more powerful techniques are needed to accurately solve the entropy of these instances. We analyze that better preprocessing methods and more suitable variable heuristics for entropy computing may become a breakthrough. However, we need to emphasize that EntropyEstimation is only a Shannon entropy estimation tool, and the entropy it obtains is subject to a certain degree of error. Our precise Shannon entropy tool PSE performs better than entropy estimators in most instances, which demonstrates that our techniques significantly enhance the scalability of precise Shannon entropy. Based on the results, it is evident that PSE outperforms EntropyEstimation in the majority of instances. This provides a positive answer to **RQ2**.

Impact of algorithmic configurations

To better verify the effectiveness of the PSE methods and answer **RQ3**, we conducted a comparative study on all the utilized methods, including methods for the Y-stage: **Implied Literal**, **YCache**, **Pre**, variable decision heuristics(**minfill**, **DLCP**, **SharpSAT-TD heuristic**, **VSADS**), and methods for the X-stage: **XCache** and **ConditionedCounting**. According to the principle of control variables, we designed ablation experiments to verify the effectiveness of each method, and each experiment was only different from the tool PSE in one method. The cactus plot for the different methods is shown in Figure 4, where PSE represents our tool, PSE-wo-Pre means that **Pre** is turned off in PSE. PSE-ConditionedCounting indicates that PSE employed the **ConditionedCounting** method rather than **SharedCounting** in the X-stage. PSE-wo-XCache indicates that the caching method is turned off in PSE in the X-stage. PSE-wo-YCache indicates that the caching method is turned off in PSE in the Y-stage. PSE-wo-ImpliedLiteral indicates that PSE does not employ the **Implied Literal** in the Y-stage. PSE-dynamic-SharpSAT-TD means that PSE replaces the **minfill** static variable order with the dynamic variable order: the variable decision-making heuristic method of SharpSAT-

[2]The average PAR-2 scoring scheme gives a penalized average runtime, assigning a runtime of two times the time limit for each benchmark that the tool fails to solve

TD (all other configurations are the same as in PSE, only the variable decision-making heuristic is different). Similarly, PSE-dynamic-DLCP and PSE-dynamic-VSADS respectively indicate the selection of dynamic heuristic **DLCP** and **VSADS**.

As can be seen from the experimental results, the power of implied literal is very strong. The effect of caching is also significant, as has been demonstrated in previous studies of knowledge compilation. Although the effect of preprocessing is not very significant, it can still increase the number of instances solved. The heuristic strategies minfill and SharpSAT-TD heuristic both performed well, and the results indicate that the static heuristic minfill is more suitable for entropy computing. In the technique of the X-stage, the **ConditionedCounting** method performs better than **SharedCounting** without **XCache**, but not as well as the **SharedCounting** method. This comparative experiment indicates that the shared component caching is quite effective. The biggest advantage of the **ConditionedCounting** method is that its time complexity is linear (**?**), although it also has a major disadvantage in that we need to construct an ADD-L based on the static variable order first, which is a process with a large time overhead for more complex problems. Although the **ConditionedCounting** method is not the most effective, we believe it is still a promising and scalable method. For instances where an ADD-L can be constructed quickly based on the static variable order, the **Conditioned-Counting** method may work better than the SharedCounting method when it is difficult to model counting in X-stage. The **ConditionedCounting** method is not the best method, but we believe it is still a promising and scalable method. Finally, PSE selects the **SharedCounting** strategy in the X-stage, and chooses **YCache**, **Implied Literal**, **Pre**, and the **minfill** heuristic method in the Y-stage.

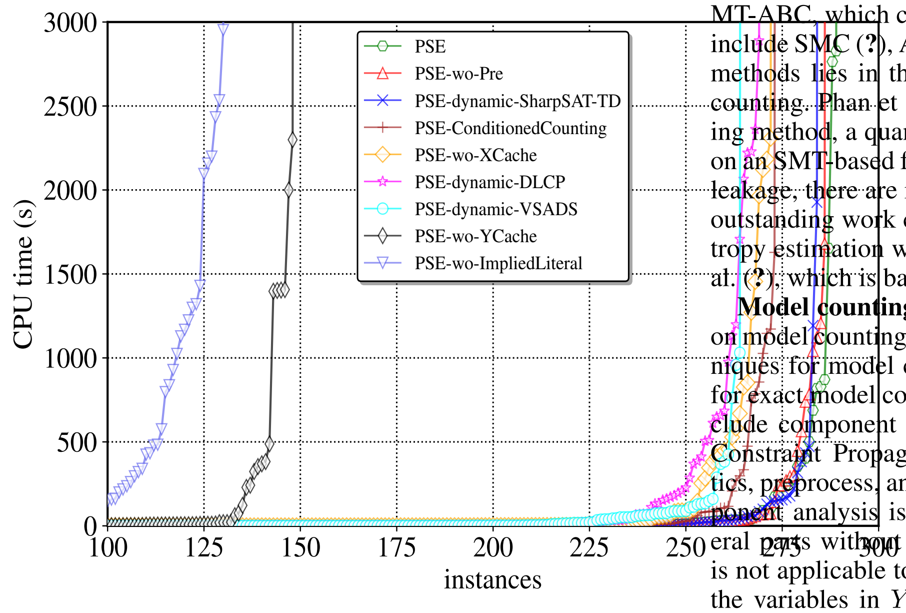

Figure 4: Cactus plot comparing different methods.

Related work

Our work is based on the close relationship between QIF, model counting, and knowledge compilation. We introduce relevant work from three perspectives: (1) quantitative information flow analysis, (2) model counting, and (3) knowledge compilation.

Quantified information flow analysis Currently, the QIF method based on model counting faces two major challenges. The first one is to construct the logical postcondition Π_{proc} for a program $proc$ (**?**). This process can be achieved through symbolic execution, but the existing symbolic execution tools have certain limitations and are difficult to be extended to some complex programs, such as those involving symbolic pointers. The second challenge is model counting. Our work focuses on the challenges of model counting. For programs modeled by Boolean clause constraints, Shannon entropy can be solved through model counting queries to quantify the amount of information leakage. Golia et al. have made outstanding contributions to this work. They proposed the first efficient Shannon entropy estimation(**?**) with PAC guarantees based on sampling and model counting. Their core idea is to reduce the number of model counting queries through sampling. However, their idea can only provide an approximate estimation of entropy. Our work is inspired by the research conducted by Golia et al., but our approach and optimization direction differ. We optimize the existing framework of model counting for precise Shannon entropy to reduce the number of model counting queries, while simultaneously enhancing the efficiency of model counting solutions. Based on this idea, we propose an efficient and precise method for computing the Shannon entropy. Besides Boolean clause constraints, converting programs into strings and SMT constraints has also attracted extensive research. For string constraints, Aydin et al. (**?**) proposed an automata-based model counting solver, MT-ABC, which can be applied to QIF. Similar solvers also include SMC (**?**), ABC (**?**), etc. The difference between their methods lies in the improvement of the automaton model counting. Phan et al. (**?**) proposed an abstract model counting method, a quantified information leakage method based on an SMT-based framework. For approximate estimation of leakage, there are many sampling-based methods. The most outstanding work currently is the first efficient Shannon entropy estimation with PAC guarantees proposed by Golia et al. (**?**), which is based on sampling and model counting.

Model counting Since the computation of entropy relies on model counting, we conducted a survey on powerful techniques for model counting. Among the existing techniques for exact model counting, the most effective ones mainly include component decomposition, caching, Implicit Binary Constraint Propagation (i-BCP), variable decision heuristics, preprocess, and so on. The key idea of the disjoint component analysis is to divide the constraint graph into several parts without variable intersection. However, this idea is not applicable to the computation of Shannon entropy, as the variables in Y do not support component decomposition. Caching technique has been used in various fields for a long time, and it also has excellent effects in model counting. Bacchus et al. analysis three distinct caching schemes: simple caching, component caching,and linear-space caching (?). Their research showed that component caching has the most strong prospect. Combining component caching with conflict driven clause learning is an idea pioneered by Sang et al (?) in 2004 with their model counter Cachet which is based on the well-known SAT solver zCHaff (?). We also utilized caching technique in the process of computing entropy, and our experiments once again demonstrated the power of caching technique. Thurley (?) proposed improved component encoding schemes with a solver called sharpSAT to make cache components more concise. sharpSAT first employed i-BCP in model counting, which is an improvement of the well-known "look-ahead" technique based on Boolean constraint propagation. The computation of implied literals relies on IBCP, and our experiments have shown that implied literals are also beneficial to the process of computing entropy. Sharma et al. (?) proposed a probabilistic caching strategy in their model counter Ganak, demonstrating its efficiency. Our tool PSE does not currently integrate a probabilistic caching method, which is a direction for future exploration. There have been many studies on variable decision heuristics for model counting, which can be divided into static heuristics and dynamic heuristics. In static heuristics, the **minfill** (?) heuristic is very effective, while in dynamic heuristics, **VSADS** (?), **DLCP** (?), and **SharpSAT-TD heuristic** (?) have been the most significant in recent years. We have already discussed the impact of these heuristics on entropy computation in Section 15. Preprocessing methods in model counting have been discussed in detail by Lagniez et al (?). Not all preprocessing methods in model counting are suitable for entropy computation. Our research found that the most effective method is literal equivalence, and we have made some improvements to this preprocessing method to make it suitable for entropy computation (in Section).

Knowledge compilation The motivation for utilizing knowledge compilation for model counting lies in the fact that it is difficult to compute the number of models for the original Boolean formula, whereas the new target language can solve the model counting problem quickly. Darwiche et al. first proposed a compiler called c2d (?) to convert the given CNF formula into Decision-DNNF. The core technique of c2d compile CNF to Decision-DNNF is decomposition. Based on the exact solver sharpsat (?), Muise and McIlraith et al. developed a new compiler, Dsharp (?), which converts CNF to Decision-DNNF. Unlike c2d, Dsharp utilizes two significant features of sharpSAT: dynamic decomposition and implicit binary constraint propagation. Lagniez and Marquis (?) proposed an improved Decision-DNNF compiler, called D4, is a top-down compiler that compiles CNF formulas into d-DNNF. Similar to Dsharp, D4 also uses dynamic decomposition, although the decomposition strategy is different. In order to better improve the efficiency of the algorithm, D4 makes full use of heuristic strategies such as disjoint component analysis, component caching, conflict analysis and non-technical backtracking, which perform well in c2d and Dsharp. Exploiting literal equivalence, Lai et al. (?) proposed a generalization of Decision-DNNF, called CCDD, to capture literal equivalence. They demonstrate that CCDD supports model counting in linear time and design a model counter called ExactMC based on CCDD.

In order to compute the Shannon entropy, the focus of this paper is to design a compiled language that supports the representation of probability distributions. Numerous target representations have been used to concisely model probability distributions, providing more concise factorizations than Bayesian networks in the presence of local structure(?). d-DNNF can be used to compile relational Bayesian networks for exact inference (?). Probabilistic Decision Graph (PDG) is a representation language for probability distributions based on BDD (?). Probabilistic Sentential Decision Diagram (PSDD) is a complete and canonical representation of probability distributions defined over the models of a given propositional theory (?). Each parameter of a PSDD can be viewed as the (conditional) probability of making a decision in a corresponding Sentential Decision Diagram (SDD). And/Or Multi-Valued Decision Diagrams (AOMDDs) (?) take advantage of the AND/OR structure to represent probability distributions. Affine ADDs(AADDs) (?), an extended form of ADDs, achieve more compression through affine transformations, which is beneficial for probabilistic inference algorithms. Macii and Poncino (?) demonstrated that ADD enables efficient and precise computation of entropy. Nevertheless, our experiments revealed that the size of ADD frequently grows exponentially when handling large circuit formulas, suggesting that ADD is not suitable for computing entropy of large-scale circuit formulas. Based on the core idea of simplifying the size of ADD, we propose an extended form of ADD, ADD-L, which utilizes the power of implied literals to make the graph structure more concise and easier to hit the cache during the construction process. Apart from ADD, there have been studies utilizing BDDs for approximating entropy. Stanković et al. (?) observed that the process of computing entropy estimates could be implemented on BDDs, thus avoiding redundant calculations and ensuring the efficiency of the process. They used BDDs to estimate the entropy of a given vector in further studies (?). The complexity of their proposed algorithm is proportional to the number of nodes in the decision graph, which is similar to the complexity of our entropy calculation. However, it is worth noting that they only estimated the entropy, while we solve it precisely. In addition, the number of ADD-L nodes proposed by us is significantly less than that of BDDs, theoretically making our efficiency much higher.

Conclusion

In this paper, we propose a new compilation language, ADD-L, which combines ADD and implied literals to optimize the search process in the first stage of accurate Shannon entropy computation. In the second stage of accurate Shannon entropy computation, we optimize model counting queries using shared component cache. We integrated preprocessing, heuristic, and other methods into the precise Shannon computation tool PSE, with its trace corresponding to ADD-L. Experimental results demonstrate that PSE significantly enhances the scalability of precise Shannon entropy computation, even outperforming the state-of-the-art entropy estimator EntropyEstimation in overall performance. We believe that PSE has opened up new research directions for entropy computing in Boolean formula modeling, such as caching schemes, variable heuristics, preprocessing, etc. We look forward to designing more suitable techniques for Shannon entropy computation in the future, in order to further enhance the scalability of precise Shannon entropy.

References