Secure Stochastic Multi-party Computation for Combinatorial Problems and A Privacy Concept that Explicitely Factors out Knowledge about the Protocol*

Marius C. Silaghi† and Gerhard Friedrich‡
†Florida Institute of Technology, USA
‡University Klagenfurt, Austria

January 28, 2006

Abstract

High levels of security often imply that the computation time should be independent of the value of involved secrets. When the expected answer of the solver is either a solution or *unsatisfiable*, then the previous assumption leads to algorithms that take always the computation time of the worst case. This is particularly disturbing for NP-hard combinatorial problems.

In this work we start from the observation that sometimes (specially for hard problems) users find it acceptable to receive as answer either a solution, the answer *unsatisfiable* or a failure with meaning *don't know*. More exactly users accept *incomplete* solvers. As argued in [Sil05b], for certain problems privacy reasons lead users to prefer having an answer meaning *don't know* even when the secure multi-party computation could have proven *unsatisfiable* (to avoid revealing that all alternatives are infeasible). While the solution proposed in [Sil05b] is slower than complete algorithms, here we show secure stochastic solutions that are faster than complete solvers, allowing to address larger problem instances. Two new refined concepts of privacy are introduced, namely requested t-privacy that factors out treatment of knowledge of the protocol in *t-privacy*, and a slightly weaker version called *non-uniform requested t-privacy*. In the last section we discuss arithmetic circuits for complete and stochastic solutions to constraint optimization problems.

*The update on Jan 28, 2006 introduced the comments about how the constant round primitive first-in-array can replace certain computations. The last section, on Optimization Problems, was added during a September 2005 update, and contains material from [SPF05]. A subsequent more complete description of that section appears in [SFP06]. The section on the new privacy concept from [Sil05a], is further extended on Dec 31, 2006.

1 Introduction

Typical examples of combinatorial problems are meeting scheduling, resource allocation, time-tabling, auctions with several possible winners. Such a problem is typically defined by a set of variables and constraints on the satisfiable assignments to these variables. The set of all (satisfiable and unsatisfiable) simultaneous assignments of values to all variables defines the *search space* of the problem. An element of the search space is also referred to as an alternative to be considered as a solution to the problem, or simply *alternative*.

A complete solver is one that reports a solution whenever a solution exists. The answer of such a technique is either a solution or *unsatisfiable*. Combinatorial problems can be very hard and therefore we no not have efficient complete secure multi-party computation solvers. Several complete secure solvers were proposed in the past for such problems, and high levels of security always require a computation time that is given by the worst possible case (over all possible values of the secrets).

It was shown that for problems that are solved only once, minimization of privacy loss often requires that the solution be picked randomly, preferably with a uniform distribution among the existing solutions [SR04]. Such a random selection can be achieved if the problem is shuffled prior to solving [Sil03, Sil04]. Two families of techniques were proposed for shuffling a shared description of a combinatorial problem,one based on mix-nets and one based on arithmetic circuits [Sil05c].

Sometimes, the security requirements themselves require an incomplete solver (when the proof of unsatisfiability of the problem leads to unacceptable privacy loss, by revealing that all alternatives are infeasible) [Sil05b]. The answer of such a solver is either a solution or *unsatisfiable*. However, the solution proposed in [Sil05b] is actually slower than complete solutions. It first computes a solution with a complete secure solver and then it hides the solution with some small probability.

In this work we show how the shuffling performed on problem descriptions prior to solving allows to build an incomplete secure stochastic multi-party solver where a high level of privacy is offered. The answers of the solver consists in either a solution or in *don't know*, and nothing is revealed about the set of alternatives that were not explored (except for its size). Notably, these algorithms are strictly faster than the corresponding complete versions and are parametrized with the percentage of the search space to be explored (the search space is the set of all alternatives that may or may not satisfy the combinatorial problem).

By specifying the percentage of the combinatorial problem to be explored, one practically specifies the exact amount of computation (time) that the solver should perform. The proposed techniques are different for shuffling with mix-nets and for shuffling with arithmetic circuits. Arithmetic circuits for optimization problems are discussed in the last sections.

2 Privacy Concept that Explicitely Factors out Knowledge about the Protocol

We refine a concept of privacy that allows to formally identify losses of privacy due to certain multi-party computation protocols. Namely, previous privacy concepts suggest that a protocol is secure if everything *leaked by its execution* can also be inferred from *prior knowledge and obtained result*. However, in this article we stress the need to classify the information leaked due to the knowledge of which protocol is employed as being *leaked by its execution* rather than solely by *prior knowledge and obtained result*. This distinction is non-trivial, as proven by the fact that several recent articles present algorithms designed according to the aforementioned privacy concept and still failing our test of security.

The implications of the two ways of accounting privacy loss becomes easily clear for problems where several solutions are possible (optimal). For such problems, the knowledge of the algorithm used to select the solution may leak information about where the other (optimal) solutions may or may not be located. While these implications were underlined in our previous work, the new concept of privacy shown here offers a formal way to evaluate solutions proposed in the past for this problem. This allows to motivate protocols leaking less information for applications such as auctions.

Different privacy concepts and levels of privacy were identified for multi-party computations [BOGW88, FMW01], and they are relevant for different computational models and assumptions. The t-privacy concept introduced in [BOGW88] is very well known and is applicable to many types of assumptions:

Definition 1 *A multi-party computation is t-private if an attacker controlling any at most t participants cannot learn anything from the computation, except for what can be inferred from its outputs and prior knowledge.*

Security of a t-private scheme can be computational (if breaking it is computationally intractable for the attacker) or information theoretical (if even infinite computation power cannot infer anything about secrets since no information about it is contained in the data obtained by the attacker).

Given secret inputs σ, the prior knowledge Γ of the t colluders and a multi-party computation process Π with answer α, the technique is t-private if the probability distribution of the secrets is conditionally independent on Π given answer α and knowledge Γ.

$$P(\sigma|\alpha, \Gamma, \Pi) = P(\sigma|\alpha, \Gamma) \tag{1}$$

However, many algorithms provide answers α that contain more information than what is actually needed. We typically decompose α in a desired data α^* and an algorithmic dependent unrequested data $\overline{\alpha}$. For example[1] the desired data can be an assignment of some variables satisfying secret constraints, and the unrequested data is what can be obtained from peculiarities of the used algorithm $\mathcal{A}$ (e.g., the solution is the first/last in some known order on alternatives).

[1] With DisCSPs[Sil04].

Definition 2 *We say that an algorithm $\mathcal{A}$ achieves* requested t-privacy *if the probability distribution of the secrets is conditionally independent on $\Pi, \mathcal{A}$ and $\overline{\alpha}$ given requested data α^* and prior knowledge Γ.*

$$P(\sigma|\alpha, \Gamma, \Pi, \mathcal{A}) = P(\sigma|\alpha^*, \Gamma) \quad (2)$$

Examples of algorithms designed to be secure, according to Equation 1 [BOGW88], but insecure according to the definition introduced in Equation 2 are proposed in [YSH02a, Sil03, YS04, NZ05].

For problems with several solutions, *requested t-privacy* typically implies the return of uniformly random selected solutions whenever the problem may have more than one solution. Examples of techniques secure according to the introduced concept are [Sil04, Sil05a]. Sometimes uniform randomness in selecting the solution requires very expensive computations. Non-uniform randomness in selecting the solution, while shown to be often clearly better then deterministic approaches and often achievable with significantly reduced computation effort, is less secure [Sil04]. The level of privacy achieved in that case is nevertheless interesting and worth its own definition. We propose to call it *non-uniform requested t-privacy*, and can be describes by the property:

Definition 3 *We say that an algorithm $\mathcal{A}$ achieves* non-uniform requested t-privacy *if for any secret $\overline{\sigma} \in \sigma$ that is not deterministically revealed given requested data α^* and prior knowledge Γ, it is also not deterministically relealed given $\Pi, \mathcal{A}$ and $\overline{\alpha}$.*

$$\forall \overline{\sigma} \in \sigma P(\overline{\sigma}|\alpha^*, \Gamma) < 1 \Rightarrow P(\overline{\sigma}|\alpha, \Gamma, \Pi, \mathcal{A}) < 1 \quad (3)$$

3 Background

Combinatorial problems have been often discussed in Computer Science and many examples are known to be very hard. For example SAT was the first proven NP-complete problem and Constraint Satisfaction Problems are largely addressed with stochastic and incomplete solvers.

A Constraint Satisfaction Problem (X,D,C) is defined by a set of variables $X = \{x_1, ..., x_m\}$, a set of domains $D = \{D_1, ..., D_m\}$ where D_i is the domain for x_i, and a set of constraints $C = \{\phi_1, ..., \phi_c\}$. Each constraint ϕ_j specifies the acceptable combinations of assignments of values to a subset X_j of the variables. A tuple is a vector of assignments of values to distinct variables. A solution of the CSP is a tuple of assignments of values to all the variables and that satisfies all the constraints. The search space of the CSP is defined by the Cartesian product $D_1 \times ... \times D_m$. An element of the search space is called an *alternative*. The i^{th} alternative is denoted by ϵ_i.

A distributed CSP is a CSP (X, D, C) where a set of participants $A = \{A_1, ..., A_n\}$ have secret shares of C, none of them knowing the whole set C.

3.1 Shuffling an array of shared secrets

Secure multi-party computations can simulate any arithmetic circuit [BOGW88] or boolean circuit [Kil88, Gol04] evaluation. An *arithmetic circuit* can be intuitively imagined as a directed graph without cycles where each node is described either by an addition/subtraction or by a multiplication operator. Each leaf is a constant.

The secure multi-party simulation of arithmetic circuit evaluation proposed in [BOGW88] exploits Shamir's secret sharing [Sha79]. This sharing is based on the fact that a polynomial $f(x)$ of degree $t-1$ with unknown parameters can be reconstructed given the evaluation of f in at least t distinct values of x, using Lagrange interpolation. Absolutely no information is given about the value of $f(0)$ by revealing the valuation of f in any at most $t-1$ non-zero values of x. Therefore, in order to share a secret number s to n participants $A_1, ..., A_n$, one first selects $t-1$ random numbers $a_1, ..., a_{t-1}$ that will define the polynomial $f(x) = s + \sum_{i=1}^{t-1}(a_i x^i)$. A distinct non-zero number τ_i is assigned to each participant A_i. The value of the pair $(\tau_i, f(\tau_i))$ is sent over a secure channel (e.g. encrypted) to each participant A_i. This is called a (t, n)-threshold scheme. We will assume that all computations are performed in a field $\mathbb{Z}_q$ for some prime number q. Once secret numbers are shared with a (t, n)-threshold scheme, evaluation of an arbitrary arithmetic circuit can be performed over the shared secrets, in such a way that all results remain shared secrets with the same security properties (the number of supported colluders, $t-1$) [BOGW88, Yao82]. For [Sha79]'s technique, one knows to perform additions and multiplications when $t \leq (n-1)/2$. Since any $\lfloor n/2 \rfloor$ participants cannot find anything secret by colluding, such a technique is called $\lfloor n/2 \rfloor$-private [BOGW88]. It is also known how to evaluate with computational securely any arithmetic circuit on additively shared secrets.

Shuffling with mix-nets In [Sil03, Sil04, Sil05c] it is shown how a mix-net can shuffle a vector of shared secrets and can unshuffle a vector of the same size using the inverse permutations. Each participant encrypts his share of each secret using a $(+ \mod q, X)$ public encryption scheme for which it holds the secret key, and sends a vector holding each encrypted share to A_1. The vectors with the encrypted shares are passed along each participant in A, each of the applying the same secret permutation on all vectors. A shared 0 is also added to each sharing of a secret using the homomorphism of the encryption. Each participant will provide the others with a zero-knowledge proof for the correctness of his shuffling (respectively unshuffling).

Shuffling with arithmetic circuits Assume that we have composable multi-party computations [Kil05, DFNT05] for computing:

- $\delta_K(x, y)$: Kronecker's delta returning a shared 1 when $x = y$ and 0 otherwise
- $cmp(x, y)$ returns 1 when $x < y$ and 0 otherwise
- $RS(m, M)$: random secret generator, generating a shared secret in the interval m, M.
- $\vee_{k=1}^{\ell} x_i$: computes a shared secret equal to the result of applying logic $\vee$ on the vector $x_1, ..., x_\ell$ with values $\{0, 1\}$.

3.2 First in Array

The primitive $first([a[1..m]]^{\mathbb{F}}, m)$ can be applied on a vector of m shared secrets $[a[1..m]]^{\mathbb{F}} = [a[1]]^{\mathbb{F}}$,...,$[a[m]]^{\mathbb{F}}$, in $\{0,1\}$, and replaces all its elements with 0, except for the first occurence of a 1. A version of the implementation in [DFNT05], requiring a constant number of rounds, namely 17, and 20m multiplications, is described in the following. Implementations with less multiplications but linear or logarithmic round are straight-forward and are shown in [Sil03, Sil04].

Let $\lambda = \lceil\sqrt{m}\rceil$. First the elements of a are wrapped in a $\lambda \times \lambda$ matrix $b[i,j]$, such that:

$$[b[i,j]]^{\mathbb{F}} = \begin{cases} [a[i\lambda + j]]^{\mathbb{F}} & \text{if } (i-1)\lambda + j \leq m \\ 0 & \text{otherwise} \end{cases}$$

Then compute $[x[i]]^{\mathbb{F}} = \vee_{j=1}^{\lambda}[b[i,j]]^{\mathbb{F}}$ and $[y[i]]^{\mathbb{F}} = \vee_{k=1}^{i}[x[k]]^{\mathbb{F}}$ for $i \in [1..\lambda]$, and compute the row selector:

$$[row[i]]^{\mathbb{F}} = \begin{cases} [y[i]]^{\mathbb{F}} & \text{if } i = 1 \\ [y[i]]^{\mathbb{F}} - [y[i-1]]^{\mathbb{F}} & \text{if } i \in [2..\lambda] \end{cases}$$

Next one computes the selected row $[r[j]]^{\mathbb{F}} = \sum_{i=1}^{\lambda}[row[i]]^{\mathbb{F}} * [b[i,j]]^{\mathbb{F}}$ and the column selector with $[z[i]]^{\mathbb{F}} = \vee_{k=1}^{i}[r[k]]^{\mathbb{F}}$ for $i \in [1..\lambda]$, and

$$[col[j]]^{\mathbb{F}} = \begin{cases} [z[j]]^{\mathbb{F}} & \text{if } j = 1 \\ [z[j]]^{\mathbb{F}} - [z[j-1]]^{\mathbb{F}} & \text{if } j \in [2..\lambda] \end{cases}$$

Finally, the changes in the input vector are performed with: $[b[i,j]]^{\mathbb{F}} = [row[i]]^{\mathbb{F}} * [col[j]]^{\mathbb{F}}$ and b is returned as result. Operations can be optimized to disregard the 0 elements added to b at the beginning.

3.3 Shuffling

It is possible to design an arithmetic circuit for shuffling secrets, using the Algorithm 3. This algorithm uses Algorithm 1 for a permutation of two elements on secret positions in a vector. The random permutation is defined by a random vector computed with Algorithm 2. Unshuffling can be done with the Algorithm 4.

function *Perm (s,i,r,m,M,k*
- $s_i = \sum_{j=m}^{M}(\delta_K(r,j) * s_j)$;
- **for** $j \in (i,k]$ **do**
  - $s_j = s_j + (s_i - s_j) * \delta_K(r,j)$;

Algorithm 1: Permuting element s_i with s_r for a secret value $r \in [m, M]$ in vector s with k shared secrets

This permutation was shown in [Sil05c] to lead to a random shuffling (taken from a uniform distribution). Note that the random vector defining the permutation could have been built allowing each element to belong to any value between 1 and k. This would be computationally more expensive as it would require each call to the procedure $Perm$ to recompute all the elements of the vector to be shuffled (see Algorithm 5).

function *RandomVector(k)*

  for $j = 1$ *to* $k - 1$ **do**

    $r[j] = RS(j, k)$;

  return r;

Algorithm 2: Shuffling a vector s with k shared secrets

function *Shuffle(s,k,r)*

  for $j = 1$ *to* $k - 1$ **do**

    Perm(s,j,r[j],j,k,k);

Algorithm 3: Shuffling a vector s with k shared secrets, and a random vector r obtained with Algorithm 2

function *Shuffle(s,k,r)*

  for $j = k - 1$ *to* 1 **do**

    Perm(s,j,r[j],j,k,k);

Algorithm 4: Un-shuffling a vector s with k shared secrets, when the shuffling was defined by random secret vector r.

function *Shuffle(s,k,r)*

  for $j = 1$ *to* k **do**

    Perm(s,j,r[j],1,k,k);

Algorithm 5: Shuffling a vector s with k shared secrets, and a random vector r where each element is obtained with $RS(1, k)$.

3.4 MPC-DisCSP4

In [Sil05b] we have proposed a multi-party computation technique, called MPC-DisCSP4, that extracts a random solution of a distributed CSP. MPC-DisCSP4 uses general multi-party computation building blocks. General multi-party computation techniques can solve securely certain functions, one of the most general classes of solved problems being the arithmetic circuits. A distributed CSP is not a function. A DisCSP can have several solutions for an input problem, or can even have no solution. Two of the three reformulations of DisCSPs as a function (see [SR04]) are relevant for MPC-DisCSP4:

i A function DisCSP^1() returning the first solution in lexicographic order, respectively an invalid valuation τ when there is no solution.

ii A probabilistic function DisCSP() which picks randomly a solution if it exists, respectively returns τ when there is no solution.

For privacy purposes only the 2^{nd} alternative is satisfactory. DisCSP() only reveals what we usually expect to get from a DisCSP, namely *some* solution. DisCSP^1() intrinsically reveals more [SR04]. MPC-DisCSP4 implements DisCSP() in five phases:

1. Share the secret parameters of the input DisCSP using Shamir's secret sharing. The value of each publicly possible assignment (allocation) is securely evaluated.

2. The shared DisCSP problem is shuffled in a cooperative way, reordering values (and eventually variables), with a permutation that is not known to anybody [Sil05c].

3. A version of DisCSP^1() where the operations performed by agents are independent of the input secrets (to avoid leaking the secrets), is executed by simulating arithmetic circuits evaluation with the technique in [BOGW88].

4. The solution returned by DisCSP^1() at Step 3 is translated into the initial problem formulation using a transformation that is inverse of the shuffling at Step 2 [Sil05c].

5. Construct the solution from its secret shares.

It is also possible and very simple to find all solutions [HCN$^+$01]. However, when only a single solution is needed, this leaks a lot of information. At Step 3, MPC-DisCSP4 requires a version of the DisCSP^1() function whose cost is independent of the input, since otherwise the users can learn things like: *The returned solution is the only one, being found after unsuccessfully checking all other tuples, all other tuples being infeasible.* Since the used DisCSP^1() has to be independent of the problem details, its cost is exponential (at least as long as nobody proves P=NP).

Note that other alternative techniques are available, notably MPC-DisCSP1 [Sil03], MPC-DisCSP2 [SM04], and MPC-DisCSP3 [Sil04]. We call them generically MPC-DisCSPx. In this paper we only address multi-party computations without trusted servers. A family of secure solvers based on trusted servers is proposed in [YSH02b].

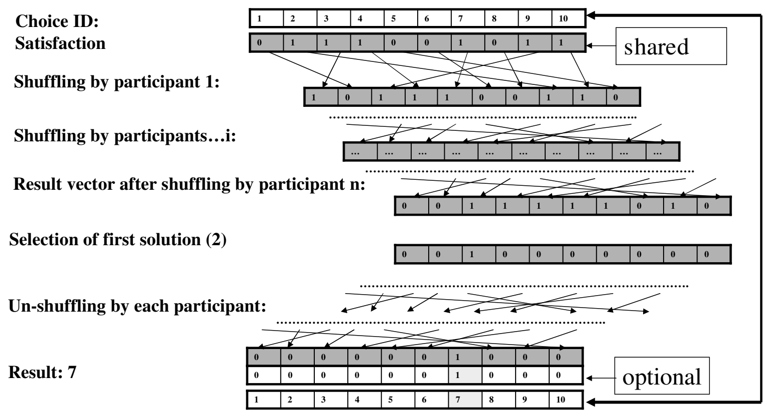

Figure 1: MPC-DisCSP4 using mix-nets

3.5 Hiding existence of solution

When no solution is found, all the participants learn that each alternative is infeasible. For certain problems this leak of secrets may be considered unacceptable and a *don't know* answer is prefered to learning the infeasibility. But the *don't know* answer is believable only if the algorithm may indeed miss some solutions. An algorithm for missing the solution with some predefined probability p is described in [Sil05b]. It consists of computing a solution using a MPC-DisCSPx algorithm and then setting the assignments in the result to the invalid value 0 with a probability p.

3.6 Stochastic algorithms

In the CSP world it is known that complete algorithms are ineffective for hard problem instances. For large problems, most applications apply stochastic search procedures. With stochastic search, only a subset of the search space is analyzed. Typical examples of stochastic search are based on some type of hill climbing. With hill-climbing the solver starts with a random alternative and searches the neighbouring search space for solutions.

4 Simulated Annealing for Secure Optimization

Once the secret constraints of a distributed CSP are shared and shuffled with the technique of MPC-DisCSP1 [Sil03, Sil05c], one can try to search a feasible solution of the shuffled problem using some hill-climbing. The same considerations and procedures apply if the problem is shuffled with a mixnet obtained from the one in [YSH02b] by replacing the encryption scheme with a $(+,\times)$-homomorphic version (E.g., Paillier with shared secret key, or the version of ElGamal of the form $E_{a,y,g,p}(m,r) = \langle g^r \mod p, a^m y^r \mod p\rangle$).

The quality of an alternative will normally be evaluated securely (since we do not tipically want to reveal individual constraints even if they were shuffled - as it would lead to an important privacy loss). The total weight (or number of conflicting constraints) for an alternative ϵ is computed with $q(\epsilon, P) = \sum_{c \in C} c(\epsilon)$. The revelation of the quality will be relatively expensive for both versions (based on either secret sharing or homomorphic encryption). Therefore, this suggests to use stochastic algorithms that are lazy in evaluating the qualities of new tuples. Such a technique is Simulated Annealing (Algorithm 6).

procedure *SSA* **do**
    Shuffle DisCSP using secret sharing or additive encryption homomorphism;
    Select random alternative (tuple) t;
    for *decreasing 'temperature' T* **do**
        change randoly the value of one variable obtaining t';
        compute securely and then reveal $\Delta = q(t') - q(t)$;
        /*alternatively reveal q(t') to detect termination when the optimum is known*/;
        /*or securely compute and reveal only $cmp(q(t'), q(t))$, if it returns 1*/;
        if $\Delta < 0$ **then**
            t=t'
        else
            t=t' with probability $e^{-\frac{\Delta}{T}}$
    Unshuffle the results;

Algorithm 6: Secure Simulated Annealing (minimization)

Similar to the technique in [Sil02, YSH02b], the Secure Simulated Annealing algorithm may reveal undesired statistical information about some secrets via the knowledge of the shuffle search space. However, specific exact information about a secret may only be inadvertently revealed only for problems with very special patterns. In the following we concentrate on algorithms guaranteed not to reveal anything else besides the solution.

5 Secure Stochastic Search

Let us finally detail our proposed techniques for tractable secure stochastic search, allowing to address hard problems. The idea is that only a subset of T alternatives from, the search space will be explored. This could be achieved by adding a public constraint that removes the remaining search space. However, to ensure privacy in case of failure (that the infeasibility of this particular sub-space is not revealed), we propose to take advantage of the shuffling of the whole problem. We select the subspace to be explored from the shuffled problem. This hides the exact search subspace that is analyzed and the only secret leaked in case of failure is that there are T infeasible alternatives (but they are not known).

5.1 Secure Stochastic Search with Mix-nets

Each MPC-DisCSPx solving algorithm using mixnets can be modified into a corresponding secure stochastic search protocol that will be called Stochastic Multi-Party Computation for Distributed CSPs (SMPC-DisCSPx). Each SMPC-DisCSPx differs from the corresponding MPC-DisCSPx by the fact that only the first T tuples of the shuffled search space are used to compute the shuffled solution. Each stochastic solver is parametrized by the number T of alternatives to be explored (T beeing smaller or equal to the size of the search space). To be noted that a stochastic solver can be seen as a generalization of the corresponding complete solver, which is obtained when T equals the size of the search space.

SMPC-DisCSP4 For example, SMPC-DisCSP4 is shown in Algorithm 7. A version that explicits how to exploit an efficient "first in array" primitive is given in Algorithm 8.

```
function SMPC-DisCSP4(T,(X,D,C))
    for i=1 to k do
        S[i]=∏_{φ∈C} φ(ε_i);
    SHUFFLE(S) //using the mixnet;
    h[1]=1;
    for i=2 to T do
        h[i]=h[i-1]*(1-S[i-1]);
        S[i]=S[i]*h[i];
7.1     /* S[i]=S[i]*cmp(RS(0,q-1),p*q)// fine tuning*/;
7.2 /* S[T]=S[T]*cmp(RS(0,q-1),p*q)// fine tuning*/;
    UNSHUFFLE(S);
7.3 set solution S to 0 with probability p; //optional;
    return S // the solution can be extracted from S as in [Sil05b];
```

Algorithm 7: SMPC-DisCSP4 for solving a CSP (X, D, C) with k alternatives allowed by the public constraints, and exploring T alternatives.

```
function SMPC-DisCSP4(t,(X,D,C))
    for i=1 to k do
        S[i]=∏_{φ∈C} φ(ε_i);
    Shuffle(S) //using the mixnet;
    first(S,t); // call "first in array" primitive;
    Un-Shuffle(S);
    // the solution can be extracted from S as in [Sil05b];
    return S;
```

Algorithm 8: SMPC-DisCSP4 for solving a CSP (X, D, C) with k alternatives allowed by the public constraints, and exploring t alternatives.

SMPC-DisCSP4 requires $k(c-1)$ multiplications of secrets to build the vector S and $2T$ multiplications of secrets to select the solution. Also, the shuffling and unshuffling require each $O(kn^2)$ expensive operations, $O(kn)$ for each participant. While SMPC-DisCSP4 leads to a reduction with up to $2k$ multiplications of secrets, the complexity remains the same, dictated by the shuffling.

One can allow agents to avoid revealing if that there exist T alternatives that are not solutions, by enabling the optional cancelation of the solution with probability p at Line 7.1. This cancelation of solution can be done with the technique in [Sil05b]).

It can be noted that the probability that a solution is lost can be fine tuned (e.g. for the application in [Sil05b]) by discarding the alternatives with probability p. This can be done by uncommenting either Line 7.1 or Line 7.2 in the Algorithm 7. In Algorithm 8 this can be done more efficient by multiplying each element $[x[i]]^{\mathbb{F}}$ in the algorithm "first in array" with $cmp(RS(0,q-1),p*q)$ in one round and λ multiplications.

SMPC-DisCSP1 The stochastic algorithm obtained from MPC-DisCSP1 is more successful, and is sketched in Algorithm 9.

```
function SMPC-DisCSP1(T,(X,D,C))
    SHUFFLE(X,D,C) //using the mixnet;
    for i=1 to T do
        S[i]=∏_{φ∈C} φ(ε_i);
    F=DisCSP1(T,(X,D,C),S);
    UNSHUFFLE(F); // Unshuffle each vector in F separately;
    set solution F to 0 with probability p; //optional;
    return F;
```

Algorithm 9: SMPC-DisCSP1 for solving a CSP (X,D,C) with k alternatives and exploring T alternatives.

DisCSP1 (Figure 2) is the arithmetic circuit proposed in [Sil03], with the only modification that function `satisfiable()` only integrates the first T tuples (rather than the whole search space). The result F returned by DisCSP1 is a set of vectors, one for each variable. A vector contains shared 0s on all positions, except for a 1 on the position corresponding the the value of the corresponding variable in the found solution. If there is no solution, then all elements of the vectors are 0.

The cost of SMPC-DisCSP1 is only $O(T(md+c))$ multiplications of secrets. Of these, $T(c-1)$ are used to compute S. DisCSP1 computes `satisfiable` md times, each of them requiring at most $O(T)$ multiplications. The cost of shuffling in SMPC-DisCSP1 can be small even for large and hard problems, if the maximum constraint arity (number of involved variables) is small.

The stochastic techniques can be adapted to the corresponding secure optimization techniques based on secure distributed weighted CSPs, MPC-DisWCSPx [SM04], leading to algorithms that we will denote SMPC-DisWCSPx. When using such optimization techniques for applications like Generalized Vickerey Auctions (GVA), one

$$
\begin{aligned}
p(\epsilon, P) &= \prod_{c \in C} c(\epsilon) \\
\texttt{satisfiable}(P) &= cmp(0, \sum_{\epsilon_i \in [\epsilon_1 \ldots \epsilon_T]} p(\epsilon_i, P)) \\
g_{i,j}(P) &= \texttt{satisfiable}(P \cup \{x_i = j\} \cup_{k<i} (x_k = f_k(P))) \\
f_j(P) &= \sum_{i=1}^{|D_j|} i * (g_{j,i}(P) * \delta_K(0, \sum_{k<i} g_{j,k}(P)))
\end{aligned}
$$

Figure 2: Arithmetic circuit DisCSP1 for a CSP $P = (X, D, C)$. The result is the vector of vectors $\{\{\delta_K(f_i, j)\}_{j \in [1..|D_i|]}\}_{i \in [1..m]}$. Versions with other primitives appear in [Sil03, Sil04]

needs to make sure that the same subset of the search space is explored by each instance of the optimization algorithms. Remember that in GVA one computes the Clarke taxes using the value of the best allocation for several different settings (excluding each bidder in the computation of his tax).

Remark 1 *Therefore the same shuffling has to be used for all these instances of optimization. Also the same parameter T has to be used for each instance optimization (at each given total weight of the solution).*

At each optimization instance that excludes a bidder one only needs the weight of the optimal solution, and only `satisfiable(P)` *is computed there for each given total weight.*

5.2 Constant round primitives in SMPC-DisCSP1

Constant round primitives can improve SMPC-DisCSP1. Namely *symmetric functions* can be used for reducing the number of rounds for the function `satisfiable`, *first in array* can be used to get rid if the t iteration, and unbounded fan-in multiplications can be used to compute S (needed in $p()$) in constant rounds. The obtained changes are depicted in Figures 3,10

procedure `satisfiable`*(t,P)* **do**
  return $\vee_{k=1}^{t} [p(\epsilon_k, P)]^{\mathbb{F}}$

Algorithm 10: Function `satisfiable`: returns 0 or 1 function on whether at least one of the first t tuples in the search space of P is a solution to P, where ϵ_k is the k^{th} tuple.

$F_j[k] = first([g_{j,1}(P)..g_{j,|D_j|}(P)], |D_j|)$
$p(\epsilon, P) = \prod_{\phi \in P} \phi(\epsilon)$
$g_{i,j}(P) = \texttt{satisfiable}(P \cup \Lambda_i^j \bigcup_{k<i} \Lambda^*_{k,P}, t)$

$$\Lambda_i^j(\epsilon) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } x_i = v_j^i \text{ in valuation } \epsilon \\ 0 & \text{if } x_i \neq v_j^i \text{ in valuation } \epsilon \end{cases}$$

$$\Lambda^*_{k,P}(\epsilon) \stackrel{\text{def}}{=} F_k[\epsilon_{|x_k}] = \begin{cases} 1 & \text{if } x_k = v^k_{f_k(P)} \text{ in valuation } \epsilon \\ 0 & \text{if } x_k \neq v^k_{f_k(P)} \text{ in valuation } \epsilon \end{cases}$$

Figure 3: DisCSP1(t, P): arithmetic circuit for MPC-DisCSP1. It returns the set of vectors $F_j, \forall j \in [1..m]$. $\epsilon_{|x_k}$ stands for the value of x_k in ϵ.

5.3 Secure Stochastic Search with arithmetic circuits

The secure stochastic algorithms based on mix-nets suffer from the fact that the cost of shuffling remains the same as for the non-stochastic complete approaches. This was particularly negative in the case of SMPC-DisCSP1 where the cost of the shuffling is the main cost.

This problem is reduced in algorithms with shuffling based on arithmetic circuits. Namely, with shuffling based on arithmetic circuits one does not need to compute the whole shuffling. With SMPC-DisCSP4, it is possible to only compute the first T elements of the shuffled problem (see Algorithms 11, 12), and 13).

function *Shuffle(s,k,r,T)*
  for $j = 1$ *to* T **do**
    Perm(s,j,r[j],j,k,k);

Algorithm 11: Shuffling a vector s with k shared secrets, and a random vector r obtained with Algorithm 2

function *Shuffle(s,k,r,T)*
  for $j = T$ *to* 1 **do**
    Perm(s,j,r[j],j,k,k);

Algorithm 12: Un-shuffling a vector s with k shared secrets, when the shuffling was defined by random secret vector r.

It can be noted that in secure stochastic algorithms based on arithmetic circuits we succeed to reduce the cost of shuffling and unshuffling from $O(k^2)$ to $O(kT)$ multiplications of secrets. With this improvement the complexity of SMPC-DisCSP4ac decreases, but remains high since k is large for hard problems (can be exponential in the problem size).

function *SMPC-DisCSP4ac(T,(X,D,C))*

for *i=1 to k* **do**

S[i]=$\prod_{\phi \in C} \phi(\epsilon_i)$;

R=RandomVector(T);

SHUFFLE(S,k,R,T) //using the mixnet;

$first(S,t)$; // "first in array" algorithm;

for *i=T+1 to k* **do**

S[i]=0;

UNSHUFFLE(S,k,R,T);

set solution S to 0 with probability p; //optional;

return S// the solution can be extracted from S as in [Sil05b];

Algorithm 13: SMPC-DisCSP4ac, solving a CSP (X, D, C) with k alternatives allowed by the public constraints, and exploring T alternatives.

In conclusion the most appropriate algorithm for Stochastic Search is SMPC-DisCSP1 which has polynomial space requirements and whose computational (time) complexity can be bounded to low values being linear in T and in the problem size.

SMPC-DisCSP4ac (with arithmetic circuits) has a time complexity significantly smaller than MPC-DisCSP4 ($O(k(T + c))$ versus $O(k^2)$). This implies that the size of the problems solvable with SMPC-DisCSP4' is larger than the size solvable with MPC-DisCSP4, which had the best complexity among complete algorithms.

Remark 2 (SMPC-DisCSP1ac) *Arithmetic circuit shuffling for SMPC-DisCSP1 works by separately permuting each domain (with a separate random vector for each of them). The improvement that can be brought is to only compute the permuted constraint elements that are part of the first T tuples.*

The shuffling for SMPC-DisCSP1 is not expensive. Therefore possible improvements in versions based on arithmetic circuit shuffling are less significant, not changing the time complexity.

6 Optimization Problems

It is known that, in general, Constraint Optimization Problems (COP) are NP-hard. Existing arithmetic circuits for secure protocols solving such problems are exponential in the number of variables, n. Using variable elimitation techniques [Dec90] COPs can be solved with computation that is exponential only in the induced-width of the Depth First Search tree (DFS) of the constraint graph [PF05b], i.e. smaller than n. We show how to construct an arithmetic circuit with this property and solving any COP. For forest constraint graphs, this leads to a linear cost secure solver.

Combinatorial optimization is an important operation in many problems. One important formalism for modeling combinatorial optimization is the constraint optimization problem (COP). A constraint optimization problem (X,D,C) is defined by a set of variables, $X = \{x_1, ..., x_m\}$, with domains from $D = \{D_1, ..., D_m\}$, and a set of weighted constraints $C = \{\phi_0, ..., \phi_m\}$, each such constraint ϕ_i specifying a distinct cost associated with each assignment of values to a subset X_i of X.

An assignment is a pair $\langle x_i, v\rangle$ where $v \in D_i$. A solution of the COP is a tuple of assignments ϵ with values for each variable in X such that the sum of the weights associated by the constraints in C to ϵ is maximized (minimized). Without loss of generality we assume that by optimal solution we understand the solution with maximal weight. If we denote the projection of a tuple ϵ on a set of variables X_i by $\epsilon_{|X_i}$, then the solution is:

$$\underset{\epsilon}{\text{argmax}} \sum_{\phi_i \in C} \phi(\epsilon_{|X_i})$$

A distributed COP (DCOP) arises when some constraints are functions of secrets own by some agents from a set $A = \{A_1, ..., A_n\}$. Without loss of generality we assume that ϕ_0 is the only public constraint and that X_m, the set of variables in ϕ_m, contains besides x_m only variables x_i with $i < m$. Note that such a formulation can be obtained from any DCOP by building a Depth First Search (DFS) tree, introduced later and combining the constraints such that there remains a single constraint per variable (with his ancestors in the tree).

Our work employs the following secure multi-party computation techniques:

- polynomial secret sharing [Sha79]: Each participant k out of n participants receives $\langle s\rangle_k^t = s + \sum_{i=1}^t a_i k^i$, where a_i is a secret random number. The secret can be reconstructed with the collaboration of $t+1$ participants using $s = \sum_{k=1}^{t+1} l_{k,t}$, where $l_{k,t}$ are the corresponding Lagrange coefficients.

- addition of shared secrets [BOGW88]: $\langle s_1 + s_2\rangle_k^t = \langle s_1\rangle_k^t + \langle s_2\rangle_k^t$

- resharing shared secrets: To reshare a secret $\langle s\rangle^t$ with another threshold t', each share $\langle s\rangle_k^t$ is shared with $(t'+1, n)$-polynomial sharing scheme.

- multiplication of shared secrets [BOGW88]: $\langle s_1 * s_2\rangle_k^{2t} = \langle s_1\rangle_k^t * \langle s_2\rangle_k^t$

- arithmetic circuit evaluation with additive secret sharing [Gol04]: Each participant $k, k > 1$ out of n participants receives $[s]_k = a_i$ where a_i is a random number. Participant 1 gets $[s]_1 = s - \sum_{i=2}^n a_i$. The secret could be reconstructed with $s = \sum_{k=1}^n [s]_k$. Addition of additively shared secrets is done with $[s_1 + s_2]_k = [s_1]_k + [s_2]_k$. Multiplication is done using oblivious transfers [Gol04].

- secure test [DFNT05]: $\delta(x)$ returns 1 if $x = 0$ and 0 otherwise.

- secure Kronnecker's δ [Kil05]: $\delta_K(x, y) = \delta(x - y)$ returns 1 if $x = y$ and 0 otherwise.

- secure comparison [DFNT05]: $cmp(x, y)$ returns 0 if $x < y$ and 1 otherwise.

- secure max: $\max(x, y) = cmp(x, y) * (x - y) + y$.

6.1 Optimization Background

DCOPs have been addressed with various techniques that differ both in efficiency and in their privacy guarantees. The former techniques seeking the strongest privacy guarantees are based on secure multiparty computation and scan several times the whole search space, i.e. Cartesian product of domains in D, once for each possible total weight [SM04]. An optimization protocol specialized on generalized Vickrey auctions and based on dynamic programming is proposed in [YS04] and is significantly more efficient, but does not randomize the selection of the solution, needed for reaching the highest level of privacy [SR04]. DPOP, a dynamic programming algorithm for solving (D)COPs was proposed in [PF05b] and consists of a Viterbi-like combination of a maximization and decoding [Vit67]. The algorithm in [PF05b] can also be seen as a clever heuristique for variable elimination [Dec90], or as a parallelization of ADOPT [MTSY03], and is based on a different concept of privacy [SF02].

6.1.1 Variable Elimination

Variable Elimination is a principled technique for complexity reduction in COPs. It consists of replacing all the constraints (objective functions) linked to a variable chosen for elimination by the projection of their composition on the remaining variables. A heuristique for selecting the variables to be eliminated next is provided by the DFS tree [PF05b].

6.1.2 DFS tree

The primal graph of a COP is the graph having the variables as nodes and having an arc for each pair of variables linked by a constraint [Dec03]. A Depth First Search (DFS) tree associated to a COP is a spanning tree generated by the arcs used for visiting once each node during some depth first traversal of its primal graph. DFS trees were first successfuly used for Distributed Constraint problems in [CDK00]. The property exploited there is that separate branches of the DFS-tree are completely independent once the assignments of common ancestors are decided.

Definition 4 (neighbor nodes) *The nodes directly connected to a node in a primal graph are said to be its* neighbors.

In Figure 4.a, the neighbors of x_3 are $\{x_1, x_5, x_4\}$.

Definition 5 (ancestor nodes) *The* ancestors *of a node are the nodes on the path between it and the root of the DFS tree, inclusively.*

In Figure 4.b, the ancestors of x_2 are $\{x_5, x_3\}$, while x_3 has no ancestors.

Definition 6 (descendants nodes) *The* descendants *of a node are its children in the DFS tree, as well as the children of any other of its descendants.*

In Figure 4.c, the descendants of x_3 are $\{x_1, x_4, x_2\}$, while x_2 has no descendants.

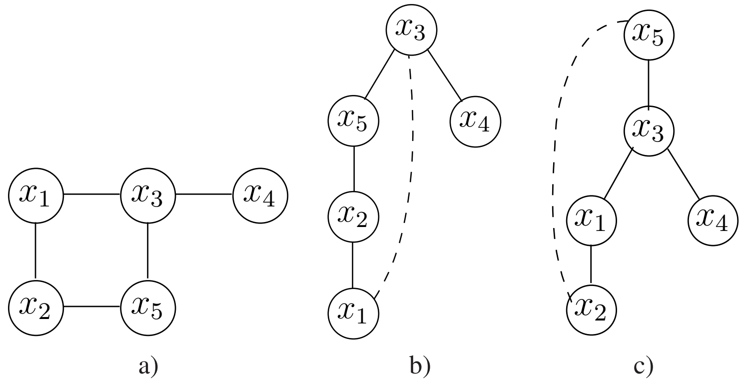

Figure 4: For a COP with primal graph depicted in (a), two possible DFS trees are (b) with induced width with 2 and (c) with induced width 2.

A *neighboring ancestor* of a node is any node that is both a neighbor in the primal graph and an ancestor in the used DFS tree. The induced width of a DFS tree is given by the the number of ancestors that are neighboring him or some of its descendants Two examples of DFS trees for a COP primal graph are shown in Figure 4. It can be noted that trees with different induced widths can be obtained. Several heuristiques had been used in the past for building DFS trees with reduced width, a prefered optimal technique consisting of a branch & bound procedure.

The ancestors of x_i are all the nodes of the path between root of the tree and x_i, inclusively the root. The descendants of x_i are all the nodes for which x_i is an ancestor. We use the following notation. Let:

- F_x be the parent of x (in Figure 4.b $F_{x_5} = x_3$)
- S_x be the children of x (in Figure 4.c $S_{x_3} = \{x_1, x_4\}$)
- P_x be the neighbor ancestors of x (in Figure 4.c $P_{x_2} = \{x_1, x_5\}$)
- G_x be the induced parents of x, i.e., ancestors that are neighbors for x or for some descendant of x (in Figure 4.c $G_{x_1} = \{x_3, x_5\}$, since x_5 is the neighboring ancestor of the descendant x_2)

In this work we do not address heuristiques for building DFS trees, but consider that such a tree is provided.

6.1.3 DFS-based Variable Elimination

A heuristique for selecting the order to eliminate variables based on exploiting the DFS-tree is proposed in [PF05b]. The idea is that before eliminating a node in the DFS tree one should first eliminate its children. In a centralized approach, such an order could be generated by either a postorder traversal or a reversed level-order traversal. This heuristiq guarantees that the arity of the largest constraint that will be added to the problem (and therefore the complexity of the algorithm) is bounded by the distance between two neighbors in the DFS tree. This is bounded by the depth of the tree and potentially much smaller than n. The advantage of this heuristique is that the quality of an elimination order can be easily evaluated.

6.1.4 Secure Optimization

A secure optimization algorithm for DCOP is proposed in [SM04]. It chooses randomly one of the values with the optimal value and reveals the total weight of the solution and the corresponding assignments only if desired and only to agreed participants. To ensure random selection of the solution, shuffling of values is done prior to solving. The result of the computation will be unshuffled. In [Sil04] it is shown how to make the selection with a uniform random distribution. However, the complete versions of these techniques are always exponential in the size of the search space (as defined by the public constraints).

Remark 3 *The fact that a mix-net leads to a random solution only hold for an agent (or a set of agents) if the 'other' agents shuffle all variables.*

If at least one agent does not shuffle a variable, this do not hold for the sub-group that shuffle, and the benefit of the shuffling is completely lost

Theorem 1 *All agents must be involved in shuffling each variable's domain of a problem.*

Proof. If some agent A_i does not shuffle a variable x_j, then all the other agents can make a coalition, find the order on x_j with which the problem was solved, and learn a set of values of x_j for which there is no optimal solution. Eliminating x_j for those values leads to a projection of constraint on the remaining variables, X', from which they can infer bounds on A_i's costs on those remaining variables X' (lower bounds at minimization respectively upper bounds at maximization). □ The techniques in [Sil05c] achieve the required type of shuffling/unshuffling.

In the following we show how to formulate arithmetic circuits for securely computing an optimal solution of a DCOP using DFS-based Variable Elimination. As in the non-secure version, the algorithm has two parts, un (upward) dynamic programming step, and a (downward) decoding step.

6.2 Arithmetic Circuits

The data structure we employ as well as their usage is depicted graphically in Figure 5. For each node x_i there is a separate set of data structures $W_{F_i}^{x_i}, W_{x_i}^{x_i}, W_{x_i}$. These data structures are accessed by indexing with partial assignment of variables (and are implementable as multi-dimensional matrices).

- $W_{x_i}^{x_i}$ holds ϕ_i, namely a local costs associated to each assignment of x_i and neighboring ancestors of x_i.

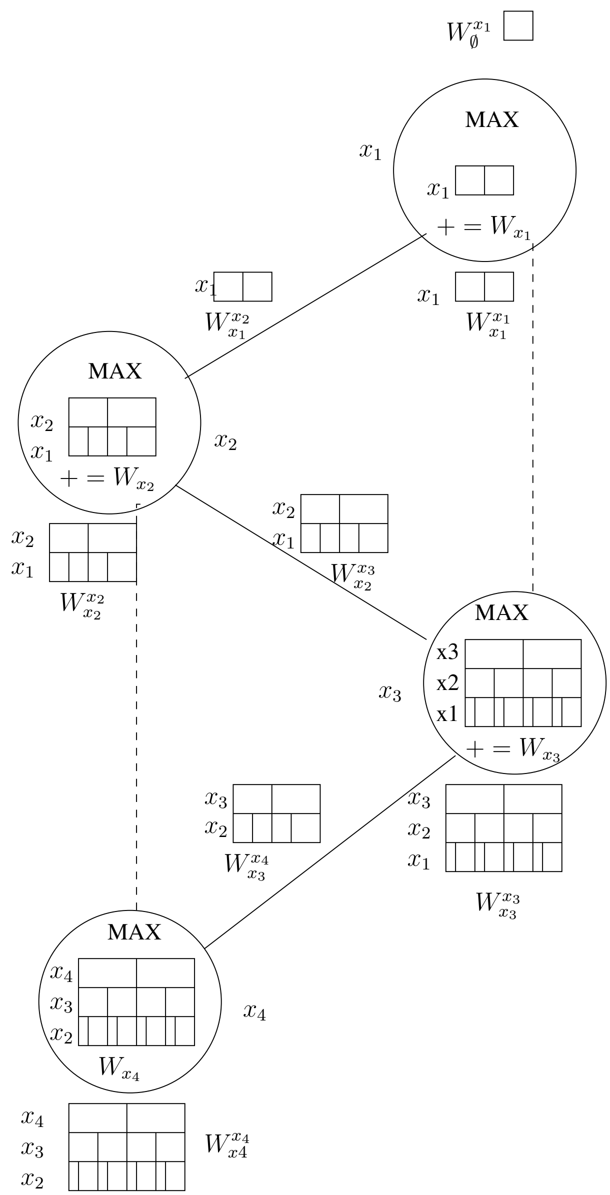

Figure 5: Data structures in the Upward computation.

- W_{x_i} holds the cummulated cost associated by ϕ_i and the projection of the constraints of all its children to each assignment of x_i and induced parents of x_i, $G_{x_i} \cup \{x_i\}$.

- $W^{x_i}_{F_i}$ holds the cummulated cost associated by ϕ_i and the projection of the constraints of all its children to each assignment of induced parents of x_i, G_{x_i}.

On the upward path in the DFS tree, for each node x_i one computes for each assignment S of the induced parents G_x:

$$W^{x_i}_{F_i}[S] = \max_{v \in D_i}(W^{x_i}_{x_i}[\langle x_i, v\rangle \cup S_{|P_{x_i}}] + \sum_{y \in S_{x_i}} (W^{y}_{x_i}[(S \cup \{\langle x_i, v\rangle\})_{|G_y}])$$

The value $W^r_{\emptyset}$, where r is the index of the root node, is the weight of the optimal solution. The steps of computation are detailed in the Algorithm 14.

procedure *Upward(x_i)* **do**
  foreach *($y \in S_{x_i}$)* **do**
    Upward(y);
  foreach *tuple ϵ for $G_{x_i} \cup \{x_i\}$* **do**
    $W_{x_i}[\epsilon] = W^{x_i}_{x_i}[\epsilon_{|\{x_i\} \cup P_{x_i}}] + \sum_{y \in S_{x_i}}(W^{y}_{x_i}[\epsilon_{|G_y}]$;
  foreach *tuple ϵ for G_{x_i}* **do**
    $W^{x_i}_{F_i}[\epsilon] = \max_{v \in D_i}(W_{x_i}[\epsilon \cup \langle x_i, v\rangle])$;

Algorithm 14: Arithmetic circuit for the upward (dynamic programming) step. At the first call, the parameter x_i is the root, r, of the DFS tree.

6.2.1 Unsecure decoding of solution

If we also want to reaveal the assignment with the optimal value, it can be done with the following arithmetic circuit, for the downward path (e.g., level-order traversal from root).

The value of the whole tree, V_r, is the shared secret $W^r_{\emptyset}$ computed at the upward step. At variable x_i with subtree value V_{x_i}, for each tuple ϵ with value W' in the structure W_{x_i} at assignments equal to the ones selected at previous levels, compute and reveal $\delta_K(W, W')$. If the result is 1, the corresponding assignment of x_i is selected, and the corresponding values V_y in $W^y_{x_i}$ for each child variable $y \in S_{x_i}$ is selected as value of the subtree with root y.

The problems with this approach, of revealing the solution, is that the algorithm cannot be used in cases where the solution of the COP is only an intermediary computation, e.g. [Sil05a]. We fix this in the following algorithm.

6.2.2 Secure decoding of solution

NOTE that a more complete and accurate version of this subsection appears in [SFP06]! In particular, the version in this subsection does not treat the case where there are several optimal solutions, which is addressed in [SFP06].

The data structures proposed for the secure downward computation (decoding) are again similar for each node. At node x_i we store:

- d_{x_i} is a shared secret vector of $|D_i|$ boolean values, each indicating the selection of the corresponding value of D_i. Only one value can be set to true.
- V_{x_i} is the shared secret weight associated to the selected optimal solution by the subtree with root x_i.

After performing the upward computation of dynamic programming, we can decode the solution securely during a preorder or level-order traversal of the tree. On visiting each node x_i, a procedure is run to compute securely the shared secret assignment of x_i using the shared secret assignments of the induced parents, d_p, for $p \in G_{x_i}$, and the shared secret selected weight of this variable, V_{x_i}. The procedure also computes the inputs for the next recursive procedure calls, at the descendents of x_i, namely the shared secret selected weight V_y of each child $y \in S_{x_i}$. The following sums are over all tuples ϵ of assignments for $G_{x_i} \cup \{x_i\}$.

$$V_y = \sum_{\epsilon} (\prod_{p \in G_{x_i}} d_p[\epsilon_{|p}]) \delta_K(V_{x_i}, W_{x_i}[\epsilon]) W^y_{x_i}[\epsilon_{|G_y}]$$

$$d_{x_i}[v] = \sum_{\epsilon_{|\{x_i\}}=v} (\prod_{p \in G_{x_i}} d_p[\epsilon_{|p}]) \delta_K(V_{x_i}, W_{x_i}[\epsilon])$$

procedure *Downward(CSP)* **do**
    while $x_i \leftarrow$ *get_InOrder_Next(DFS(CSP))* **do**
        foreach $v \in D_i$ **do**
            $d_{x_i}[v] = \sum_{\epsilon_{|\{x_i\}}=v} (\prod_{p \in G_{x_i}} d_p[\epsilon_{|p}]) \delta_K(V_{x_i}, W_{x_i}[\epsilon])$;
        foreach $y \in S_{x_i}$ **do**
            $V_y = \sum_{\epsilon} (\prod_{p \in G_{x_i}} d_p[\epsilon_{|p}]) \delta_K(V_{x_i}, W_{x_i}[\epsilon]) W^y_{x_i}[\epsilon_{|G_y}]$;

Algorithm 15: Arithmetic circuit for the downward step.

The computation steps required by the downward phase are detailed in Algorithm 15.

At the end of this computation, the vectors d_{x_i} hold a shared unary constraint allowing a single value for x_i, namely the one in the optimal solution. These unary constraints can then be unshuffled [Sil05c].

6.3 Complexity Analysis

The *Upward* step is called once for each variable x_i and the number of operations for each variable is linear in the number of elements of W_{x_i}, i.e., exponential in $|G_{x_i}| + 1$. The total cost for the upward step is $O(nd^{g+1})$, where d is the maximum size of a domain of a variable, and g is the maximum value for $|G_{x_i}|$, i.e. the induced width of the used DFS tree.

In the *Downward* step there exists a *while* loop for each variable x_i and each such cycle has two summations for each element in W_{x_i}, each term having $|G_{x_i}| + 1$ multiplications. The total cost for the downward step is $O(ngd^{g+1})$.

Therefore, the total complexity of the secure version is $O(ngd^{g+1})$. If the downward version with immediate revelation of assignments in solutions is used, then the complexity is only $O(nd^{g+1})$. If shuffling of constraints and unshuffling of solution vectors d_x are used to randomize the selection of the solution, then the cost of the shuffling is also added [Sil05c].

6.4 Extensions and Applications

A version of Secure Stochastic Optimization can be obtained by trimming the intermediary data structures at each node on the *Upward* phase. Something similar was proposed for non-cryptographic techniques in [PF05a].

Remark 4 *Note that sorting the set of weigths before trimming (in a beam-search like approach) is possible, but only for some applications. E.g., in auctions one has to ensure that the same tuples survive the prunning at each different optimization subproblem appearing in the Clarke tax computations [Sil05a].*

An immediate application for secure DCOPs is in performing the intermediary optimizations steps for Clarke tax in generalized Vickrey auctions, and related auction clearance mechanisms. If a secure stochastic version is used, then one has to use the same prunning (with the same surviving tuples) for each different optimization task on a given problem, as explained in [Sil05a].

7 Conclusions

In this work we have proposed a new family of secure solvers for distributed Constraint Satisfaction Problems (disCSPs). While most existing techniques were complete and inapplicable to large instances, the new techniques can be used to address large problems.

We have proposed stochastic versions for each of the complete secure multi-party algorithms MPC-DisCSP1 and MPC-DisCSP4, based on shuffling with mixnets or with arithmetic circuit. MPC-DisCSP1 is remarkable for its polynomial space requirements while MPC-DisCSP4 for its low time complexity and for the uniform distribution in selecting solutions.

The new versions only explore a subset of the search space of the problem, subset whose size is specified as a parameter. We have thus analyzed in detail three newly obtained versions: SMPC-DisCSP1, SMPC-DisCSP4, and SMPC-DisCSP4ac.

As its complete counterpart, SMPC-DisCSP1 requires only polynomial space. Unexpectedly, the versions obtained from MPC-DisCSP4 are much less appropriate for addressing large problems, but maintain the desirable property of selecting solutions with a uniform distribution. Among SMPC-DisCSP4 and SMPC-DisCSP4ac, the latter (based on arithmetic circuits) presents the largest speed-up in comparison to its complete version. The algorithm of choice for tackling large problems are therefore the ones based on MPC-DisCSP1 (SMPC-DisCSP1 and SMPC-DisCSP1ac), and their time complexity is linear in the problem size and in a parameter deciding the size of the explored search space.

References

[BOGW88] M. Ben-Or, S. Goldwasser, and A. Widgerson. Completeness theorems for non-cryptographic fault-tolerant distributed computating. In *STOC*, pages 1–10, 1988.

[CDK00] Z. Collin, R. Dechter, and S. Katz. Self-stabilizing distributed constraint satisfaction. *Chicago Journal of Theoretical Computer Science*, 2000.

[Dec90] R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *AI'90*, 1990.

[Dec03] Rina Dechter. *Constraint Programming*. Morgan Kaufman, 2003.

[DFNT05] I. Damgård, M. Fitzi, J. B. Nielsen, and T. Toft. How to split a shared number into bits in constant round and unconditionally secure. Cryptology ePrint Archive, Report 2005/140, 2005. `http://eprint.iacr.org`.

[FMW01] E.C. Freuder, M. Minca, and R.J. Wallace. Privacy/efficiency tradeoffs in distributed meeting scheduling by constraint-based agents. In *Proc. IJCAI DCR*, pages 63–72, 2001.

[Gol04] O. Goldreich. *Foundations of Cryptography*, volume 2. Cambridge, 2004.

[HCN+01] T. Herlea, J. Claessens, G. Neven, F. Piessens, B. Preneel, and B. Decker. On securely scheduling a meeting. In *Proc. of IFIP SEC*, pages 183–198, 2001.

[Kil88] J. Kilian. Founding cryptography on oblivious transfer. In *Proc. of ACM Symposium on Theory of Computing*, pages 20–31, 1988.

[Kil05] E. Kiltz. Unconditionally secure constant round multi-party computation for equality, comparison, bits and exponentiation. Cryptology ePrint Archive, Report 2005/066, 2005. `http://eprint.iacr.org`.

[MTSY03] P.J. Modi, M. Tambe, W.-M. Shen, and M. Yokoo. An asynchronous complete method for distributed constraint optimization. In *AAMAS*, Melbourne, 2003.

[NZ05] K. Nissim and R. Zivan. Secure discsp protocols - from centralized towards distributed solutions. In *DCR05 Workshop*, 2005.

[PF05a] Adrian Petcu and Boi Faltings. Approximations in distributed optimization. In *Principles and Practice of Constraint Programming CP 2005*, 2005.

[PF05b] Adrian Petcu and Boi Faltings. A scalable method for multiagent constraint optimization. In *IJCAI*, 2005.

[SF02] M.-C. Silaghi and B. Faltings. A comparison of DisCSP algorithms with respect to privacy. In *AAMAS-DCR*, 2002.

[SFP06] M.-C. Silaghi, B. Faltings, and A. Petcu. Secure combinatorial optimization using dfs-based variable elimination. In *Symposium on AI and Maths*, January 2006.

[Sha79] A. Shamir. How to share a secret. *Comm. of the ACM*, 22:612–613, 1979.

[Sil02] M.-C. Silaghi. *Asynchronously Solving Distributed Problems with Privacy Requirements*. PhD Thesis 2601, (EPFL), June 27, 2002. `http://www.cs.fit.edu/~msilaghi/teza`.

[Sil03] M.-C. Silaghi. Solving a distributed CSP with cryptographic multi-party computations, without revealing constraints and without involving trusted servers. In *IJCAI-DCR*, 2003.

[Sil04] M.-C. Silaghi. Meeting scheduling system guaranteeing n/2-privacy and resistant to statistical analysis (applicable to any DisCSP). In *3rd IC on Web Intelligence*, pages 711–715, 2004.

[Sil05a] M. Silaghi. Using secure discsp solvers for generalized vickrey auctions, complete and stochastic secure techniques. In *IJCAI05 DCR Workshop*, 2005.

[Sil05b] M.-C. Silaghi. Hiding absence of solution for a discsp. In *FLAIRS'05*, 2005.

[Sil05c] M.-C. Silaghi. Zero-knowledge proofs for mix-nets of secret shares and a version of ElGamal with modular homomorphism. Cryptology ePrint Archive, Report 2005/079, 2005. `http://eprint.iacr.org`.

[SM04] M.-C. Silaghi and D. Mitra. Distributed constraint satisfaction and optimization with privacy enforcement. In *3rd IC on Intelligent Agent Technology*, pages 531–535, 2004.

[SPF05] M.C. Silaghi, A. Petcu, and B. Faltings. Secure combinatorial optimization using dfs-based variable eliminations. Technical Report CS-2005-15, FIT, 2005.

[SR04] M.-C. Silaghi and V. Rajeshirke. The effect of policies for selecting the solution of a DisCSP on privacy loss. In *AAMAS*, pages 1396–1397, 2004.

[Vit67] A.J. Viterbi. Error bounds for convolutional codes and an asymtotically opti mum decoding algorithm. *IEEE Trans. on Information Theory*, 13(2):260–267, 1967.

[Yao82] A. Yao. Protocols for secure computations. In *FOCS*, pages 160–164, 1982.

[YS04] M. Yokoo and K. Suzuki. Generalized Vickrey Auctions without Third-Party Servers. In *FC04*, 2004.

[YSH02a] M. Yokoo, K. Suzuki, and K. Hirayama. Secure distributed constraint satisfaction: Reaching agreement without revealing private information. In *CP*, 2002.

[YSH02b] M. Yokoo, K. Suzuki, and K. Hirayama. Secure distributed constraint satisfaction: Reaching agreement without revealing private information. In *Proc. of the AAMAS-02 DCR Workshop*, Bologna, July 2002.