

Arithmetic circuit for the first solution of distributed CSPs with cryptographic multi-party computations*

Marius-Călin Silaghi
Florida Institute of Technology
msilaghi@cs.fit.edu

Abstract

A large class of problems like meeting scheduling, negotiation, or different types of coordination, can be formulated in terms of agents, variables, and constraints (i.e. predicates) on those variables. Distributed Constraint Satisfaction (DisCSP) is a framework addressing such general problems, namely defined in terms of a set of agents, variables, and constraints that the different agents enforce. General algorithms for DisCSPs yield a basic solution for each of those problems.

Each participant has its own constraint satisfaction problem, private concerns that should remain as secret as possible. Resources may be shared and cause the need for cooperation. Here we consider the case where privacy is an overwhelming requirement and we assume that any majority of the participants are incorruptible. Namely, given n participants, at least an $n/2$ unknown subset of them are trustworthy and not corrupted by attackers. This is a common assumption in cryptographic multi-party-computations, known as a threshold scheme.

This work shows how a solution of a general DisCSP can be found securely by the owners of the problem without appealing to any trusted servers. The constraints are shared with Shamir's secret sharing scheme, transforming the DisCSP into a shared constraint satisfaction problem. An algorithm for such problems is developed.

1 Introduction

Everybody has his private problems. Private concerns can often be formulated as constraint satisfaction problems (CSPs) and then be solved with any of the applicable CSP techniques. But life is more difficult than this: we must share our streets, busses, shops, green spaces, security, civic infrastructure, national research budget, and even our polluted air and ozone layer. We may have to share our tools

and consumables and coordinate on our agendas. One has to find agreements with the others for a solution from the set of possible valuations that satisfy her subproblem.

A distributed constraint satisfaction problem (A, X, D, C) is defined as: a set of agents, A_1, \dots, A_{n-1}, A_n , each of them willing to enforce a corresponding set of private constraints C_1, \dots, C_{n-1} , respectively C_n . A constraint $c_k \in C_i$ is defined by a predicate on a set of variables, V_{c_k} . The sets of variables involved in the constraints of different agents need not be disjoint. The union of all the variables is $X = \{x_1, \dots, x_m\}$ and each variable x_i can take values from an associated domain $D_i = \{v_1^i, \dots, v_d^i\}$, from the set of domains D . Agents want to avoid that others find details about the combinations allowed by the constraints they enforce.

The algorithm proposed here allows the n participating agents to securely find a solution by interacting directly without any external arbiter and without divulging any secrets. A common assumption in some multi-party computations is that an unknown majority of the participating agents are trustworthy and not corrupted by any adversary. Given a problem with n agents, no subset of t malicious agents that follow the protocol (typically called *passive adversaries*), $t < n/2$, can find anything about others' problems except what is revealed by the solution. This threshold scheme applies to our result as well.

Multi-party computations can be 'automatically' compiled for any function that consists only of multiplications and additions. No such function existed so far for solving DisCSPs and we show how to design it. We develop an algorithm that computes securely one solution of a DisCSP, namely the first in the lexicographic order induced by predefined ordering on variables and values. The algorithm is refutation complete. Our protocol is based on Shamir's secret sharing [5] and on cryptographic multi-party-computations [10]. We start our presentation by first introducing these elements.

*Patent pending

2 Public key cryptosystems

A cipher is also referred to as a pair of algorithms E, D together with their keys, K_E, K_D . Given a plaintext m , the ciphertext is $E_{K_E}(m)$. Decryption retrieves the plaintext $m = D_{K_D}(E_{K_E}(m))$.

Shamir's secret sharing We intend to have the participants interact directly (without intermediaries or trusted arbiters) to perform all the computations required for solving this problem. Consider that a secret s should be split and shared among n participants in such a way that any t of them can reconstruct s but no $t-1$ participants could infer anything. The idea of Shamir comes from the observation 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 . This can be done using Lagrange interpolation. Instead, 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 s to n participants A_1, A_2, \dots, A_n , one first selects $t-1$ random numbers a_1, \dots, 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 k_i is assigned to each participant A_i . Each participant A_i is then communicated over a secure channel (e.g. encrypted with E_i) the value of the pair $(k_i, f(k_i))$.

Multi-party computations Once secret numbers are split and distributed with a (t, n) scheme, distributed computations of an arbitrary agreed function of a certain class can be performed over the distributed shares, in such a way that all results remain shared secrets with the same security properties (the number of supported colluders, $t-1$) [10]. For Shamir's secret sharing [5], when the same distribution of k_i is used for all secrets, the computation of the sum of two secrets is efficiently implemented by summing the corresponding shares of each agent. Multiplication is more complex and recent research has focused on improving its efficiency [3]. In two sentences, multiplying the corresponding shares in each agent generates a valid $(2t-1, n)$ sharing of the multiplied secrets whenever $t \geq (n+1)/2$. Re-sharing with a (t, n) scheme all these products of corresponding shares allows for the computation of a (t, n) sharing of the secret product with only need of additions using Cramer's rule.

Any functions based on additions and multiplications can therefore be *compiled* unto secure cryptographic protocols. This can be done such that any ($<t$) colluding participants that cannot directly find the initial secrets, cannot find anything else than the *official output* of the protocol.

It is possible to compile protocols for parties when less than $n/2$ players cheat, and use intractability assumptions for guaranteeing security [2].

3 Searching for the first solution

As mentioned in previous sections, any algorithm that can be implemented solely with additions and multiplications (without branching with conditions on secrets) can be straightforwardly translated into a secure protocol with threshold schemes. We show such an algorithm for finding the first solution (given a lexicographic order) of a general CSP. It translates automatically to DisCSPs. [6] extend this algorithm to extract randomly a solution.

Imagine we want to solve the CSP $P = (X, D, C)$ where X is a set of variables x_1, x_2, \dots, x_m , C is a set of functions with results in the set $\{0, 1\}$ (the constraints), and D a set of finite discrete domains for X . The domain of x_i is $D_i \in D$, whose values are $v_1^i, v_2^i, \dots, v_d^i$ (i.e. all domains are extended to d values). The solutions of P are the valuations ϵ (of type $\langle (x_1, v_{\epsilon_1}^1), \dots, (x_m, v_{\epsilon_m}^m) \rangle$) with $\sum_{c \in C} c(\epsilon|_{V_c}) = |C|$. Here $\epsilon|_{V_c}$ denotes the projection of ϵ to the variables in V_c .¹

Definition 1 *The first solution of P given a total order \prec_1 on its variables X , and a total order on its values \prec_2 is the first among solution tuples when these are ordered with the lexicographical order induced by \prec_1 and \prec_2 .*

Often, one has to restrict a problem by adding additional constraining functions. We define the union between a problem $P = (X, D, C)$ and a function $c (c : X' \rightarrow \{0, 1\}, X' \subseteq X)$ as the problem $P = (X, D, C \cup \{c\})$.

$$(X, D, C) \cup c \stackrel{\text{def}}{=} (X, D, C \cup \{c\})$$

Let us imagine that we have available a function $\text{satisfiable} : CSP \rightarrow \{0, 1\}$ with the next property (an example is given later):

$$\text{satisfiable}(P) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } P \text{ has a solution} \\ 0 & \text{if } P \text{ is infeasible} \end{cases}$$

We will design now a set of functions: f_1, f_2, \dots, f_m , $f_i : CSP \rightarrow \mathbb{N}$, such that each f_i will return the index of the value of x_i in the first solution (between 1 and d), or 0 if no solution exists.

$$f_i(P) \stackrel{\text{def}}{=} \begin{cases} k & \text{if } P \text{ has the first solution for } x_i = v_k^i \\ 0 & \text{if } P \text{ has no solution} \end{cases}$$

Therefore, we first design the functions $g_{i,1}, g_{i,2}, \dots, g_{i,d}$, $g_{i,j} : CSP \rightarrow \{0, 1\}$.

¹For some extensions [6] all constraints are extended (e.g. with redundant variables) to obtain a constraint hyper-graph that is symmetric (in worst case each constraint will involve all variables, and constraints of a given agent can be composed).

$$g_{i,j}(P) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } (P \cup_{k<i} \Lambda_{k,P}^*) \text{ has solution for } x_i = v_j^i \\ 0 & \text{if } (P \cup_{k<i} \Lambda_{k,P}^*) \text{ is infeasible for } x_i = v_j^i \end{cases}$$

$g_{i,j}(P)$ is 1 if and only if the problem obtained by adding to the CSP P the function

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

that selects the value v_j^i for x_i , and $\forall k, 0 < k < i$, the functions $\Lambda_{k,P}^*$

$$\Lambda_{k,P}^*(\epsilon) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } x_k = v_{f_k(P)}^k \text{ in valuation } \epsilon \\ 0 & \text{if } x_k \neq v_{f_k(P)}^k \text{ in valuation } \epsilon \end{cases} \quad (2)$$

instantiating previous variables to their values in the first solution, is satisfiable. $v_{f_k(P)}^k$ is the $f_k(P)^{th}$ value of x_k , the value that x_k takes in the first solution. Namely, a simple implementation of $g_{i,j}$ is:

$$g_{i,j}(P) = \text{satisfiable}(P \cup \Lambda_i^j \cup_{k<i} \Lambda_{k,P}^*) \quad (3)$$

This is a recursion and is possible by first computing the value of the first variable in the first solution, f_1 , based on $g_{1,j}$ as described next. Based on the result one can compute all $g_{2,j}$. Then, one can now compute f_2 . The recursion continues with all $g_{i,j}$ that help in computing f_i for all i up to m .

To help computing f_j , we define the functions $t_{j,1}, t_{j,2}, \dots, t_{j,d}, t_{j,i} : CSP \rightarrow \{0,1\}$. $t_{j,k}(P)=1$ if and only if in a chronological backtracking search tree **no** satisfiable subtree exists under any node $x_j=v_k^j, k < i$, when previous variables are assigned according to the values in the first solution:

$$t_{j,i}(P) = \prod_{0 < k < i} (1 - g_{j,k}(P)) \quad (4)$$

Functions $t_{j,i}$ are obtained incrementally as follows:

$$t_{j,1}(P) = 1 \quad (5)$$

$$t_{j,i}(P) = t_{j,i-1}(P) * (1 - g_{j,i-1}(P)) \quad (6)$$

Once $t_{j,i}$ have been computed for all i , one can compute the index of the value of x_j in the first solution, namely f_j :

$$f_j(P) = \sum_{i=1}^d i * (g_{j,i}(P) * t_{j,i}(P)) \quad (7)$$

Lemma 1 *The functions g and f given by Equations 3 and 7 correspond to their definition.*

Proof The properties can be checked recursively starting with $g_{1,k}$ and f_1 . After computing $g_{i,k}$ and f_i one can check $g_{i+1,k}$ followed by f_{i+1} , etc...

Implementing satisfiable(): Let $SS(P)$ be the ordered set of all tuples in the Cartesian-product of the domains of $P = \langle X, D, C \rangle$. Each function c in the set of constraints C can be transformed (by adding redundant parameters and reordering them) to a function, $G_c : SS(P) \rightarrow \{0,1\}$. The secret parameters of the computation are the various values $c(\epsilon)$ where ϵ are tuples in the domain on which the corresponding c is defined. Let us define the function $p, p : SS(P) \rightarrow \{0,1\}$, defined as $p(\epsilon) = \prod_{c \in C} G_c(\epsilon)$.

procedure satisfiable(P) do

1. $\epsilon = \text{first tuple}; a=0; b=1;$
2. loop: $a = a + p(\epsilon) * b$
3. if (problem space exhausted), then terminate and return a .
4. $b = b * (1 - p(\epsilon))$
5. $\epsilon = \text{next tuple};$
6. goto loop

Algorithm 1: satisfiable(P).

function value-to-unary-constraint1(v, M)

1. Jointly, all agents build a vector $u, u = \langle u_0, u_1, \dots, u_M \rangle$ with $4M-2$ multiplications of shared secrets by computing:
 1. the shared secret vector: $\{x_i\}_{0 \leq i \leq M}, x_0=1, x_{i+1}=x_i * (v-i)$
 2. the shared secret vector: $\{y_i\}_{0 \leq i \leq M}, y_M=1, y_{i-1}=y_i * (i-v)$
then, $u_k = \frac{1}{k!(M-k)!} (v-k+1)x_k y_k$, where $0! \stackrel{\text{def}}{=} 1$.
2. Return u .

Algorithm 2: Transforming secret value $v \in \{0, 1, 2, \dots, M\}$ to a shared secret unary constraint. This is a multi-party computation using the shares of secret v .

Now we can finally design an implementation for `satisfiable` (see Algorithm 1).

$$\text{satisfiable}(P) = \sum_{\epsilon_i \in SS(P)} (p(\epsilon_i) \prod_{k < i} (1 - p(\epsilon_k)))$$

Proposition 1 *Given the previous definitions of the functions p , `satisfiable`, $g_{i,j}$, and f_i , and a problem P , the vector $\{v_{f_i(P)}^i\}_{i \in [1..m]}$ defines a solution of P (the first one).*

Proof Immediate from the definition of the functions f .

To avoid storing all the tuples in memory, the function `satisfiable` is computed similarly with the functions $g_{i,j}$, namely by using two temporary values.

Remark 1 *The computation of each $f_i(P)$ requires only additions and multiplications of the secrets. There exist some branches in loops but they do not involve evaluations*

of secrets (they are equivalent to completely unfolded versions).

Remark 2 Whenever an element of the vector $\{f_i(P)\}_{i \in [1..m]}$ is 0, the computation can be stopped since P is infeasible (Algorithm 3, step 2).

Remark 3 To compute the whole vector $\{f_i(P)\}_{i \in [1..m]}$, some constraints based on secrets also have to be dynamically shared according to Shamir's technique (see Equations 1, 2, 3). This is done according to Algorithm 2.

The secure algorithm obtained by compiling the computation of $\{v_{f_i(P)}^i\}_{i \in [1..m]}$ to a secure multi-party computation with threshold schemes (see previous section) is referred to as SecureSatisfaction. The steps that any agent has to follow here are given in Algorithm 3.

procedure SecureSatisfaction1 do

1. Securely distribute to each agent A_j (e.g. by encryption) Shamir shares of the feasibility of each local tuple ϵ_k^i of A_i : $(\epsilon_k^i, s_{i,k}^j)$, $s_{i,k}^j$ is A_j 's share of the secret $c(\epsilon_k^i)$.
2. Jointly compute `satisfiable(P)` (using Algorithm 1). If P is not satisfiable (result 0), terminate with failure.
3. $j = 1$
4. Compute in parallel all $g_{j,k}$ (Eq. 3). The functions Λ_j^k are publicly known, therefore one simply sets $p(\epsilon)$ to 0 when $\Lambda_j^k(\epsilon)$ is 0, and disregards Λ_j^k otherwise.
Compute $t_{j,k}(P)$ for all k , from 1 to $|D_j|$ (Eq. 5).
5. Compute $f_j(P)$ (Eq. 7).
6. if $j = m$, then terminate algorithm:
 - 6a For all k , let $f_k(P)$ be revealed to the owners of the x_j variable (agents that have functions involving x_j). This is done by reconstructing the corresponding secrets from their shares with Shamir's technique.
7. The shared secret $f_j(P)$ will be transformed in a unary constraint extensively represented by a vector f_j' of size d . Its $f_j(P)^{th}$ element $f_j'[f_j(P)]$ is 1 and all the other elements are 0. This is achieved by the call `value-to-unary-constraint(f_j(P)-1, d-1)`, followed by multiplying each element of the returned vector by $\frac{1}{((-1)^{\sigma_j}(\sigma_j!)^2)}$ where $\sigma_j = (d-1)$. The function `value-to-unary-constraint` doing this is shown in Algorithm 2. The obtained vector is used to evaluate $\Lambda_{j,P}^*(\epsilon)$ in future steps 4 by returning the k -th element of the vector for a parameter tuple ϵ having $x_j = v_k^j$.
8. $j = j + 1$
9. goto step 4

Algorithm 3: Algorithm followed by each agent A_i for finding a solution satisfying conditions where g functions are computed in parallel. It is possible to also compute them sequentially with lower space complexity.

Example 1 Take a DisCSP with $n=3$, $d=2$, and $m=2$. A_1 (secretly) does not want $((x_1=v_1^1), (x_2=v_1^2))$ and A_2 (secretly) does not want $((x_1=v_2^1), (x_2=v_2^2))$... For simplicity, 0 is always shared as $3x+0$ and 1 as $4x+1 \pmod 7$. A tuple $((x_i=v_a^i), (x_j=v_b^j))$ is denoted in the following by $(\begin{smallmatrix} i & j \\ a & b \end{smallmatrix})$.

During step 1 in Algorithm 3, A_2 generates: $((\begin{smallmatrix} 1 & 2 \\ 1 & 1 \end{smallmatrix}), E_1(3))$, $((\begin{smallmatrix} 1 & 2 \\ 1 & 2 \end{smallmatrix}), E_1(5))$, $((\begin{smallmatrix} 1 & 2 \\ 2 & 1 \end{smallmatrix}), E_1(5))$, $((\begin{smallmatrix} 1 & 2 \\ 2 & 2 \end{smallmatrix}), E_1(5))$, that he sends to A_1 . A_2 also generates for itself, $((\begin{smallmatrix} 1 & 2 \\ 1 & 1 \end{smallmatrix}), 6)$, $((\begin{smallmatrix} 1 & 2 \\ 1 & 2 \end{smallmatrix}), 2)$, $((\begin{smallmatrix} 1 & 2 \\ 2 & 1 \end{smallmatrix}), 2)$, $((\begin{smallmatrix} 1 & 2 \\ 2 & 2 \end{smallmatrix}), 2)$, and for A_3 : $((\begin{smallmatrix} 1 & 2 \\ 1 & 1 \end{smallmatrix}), E_3(2))$, $((\begin{smallmatrix} 1 & 2 \\ 1 & 2 \end{smallmatrix}), E_3(6))$, $((\begin{smallmatrix} 1 & 2 \\ 2 & 1 \end{smallmatrix}), E_3(6))$, $((\begin{smallmatrix} 1 & 2 \\ 2 & 2 \end{smallmatrix}), E_3(6))$.

A_1 generates for itself: $((\begin{smallmatrix} 1 & 2 \\ 1 & 1 \end{smallmatrix}), 5)$, $((\begin{smallmatrix} 1 & 2 \\ 1 & 2 \end{smallmatrix}), 5)$, $((\begin{smallmatrix} 1 & 2 \\ 2 & 1 \end{smallmatrix}), 5)$, $((\begin{smallmatrix} 1 & 2 \\ 2 & 2 \end{smallmatrix}), 3)$, and also the shares to be delivered to A_2 and A_3 , $((\begin{smallmatrix} 1 & 2 \\ 1 & 1 \end{smallmatrix}), E_2(2))$, $((\begin{smallmatrix} 1 & 2 \\ 1 & 2 \end{smallmatrix}), E_2(2))$, $((\begin{smallmatrix} 1 & 2 \\ 2 & 1 \end{smallmatrix}), E_2(2))$, $((\begin{smallmatrix} 1 & 2 \\ 2 & 2 \end{smallmatrix}), E_2(6))$, $((\begin{smallmatrix} 1 & 2 \\ 1 & 1 \end{smallmatrix}), E_3(6))$, $((\begin{smallmatrix} 1 & 2 \\ 1 & 2 \end{smallmatrix}), E_3(6))$, $((\begin{smallmatrix} 1 & 2 \\ 2 & 1 \end{smallmatrix}), E_3(6))$, $((\begin{smallmatrix} 1 & 2 \\ 2 & 2 \end{smallmatrix}), E_3(2))$...

During step 2 in Algorithm 3 the agents jointly compute by multi-party multiplication (see Section 2) a sharing of the following secrets: $p(\begin{smallmatrix} 1 & 2 \\ 1 & 1 \end{smallmatrix}) = 0$, $p(\begin{smallmatrix} 1 & 2 \\ 1 & 2 \end{smallmatrix}) = 1$, $p(\begin{smallmatrix} 1 & 2 \\ 2 & 1 \end{smallmatrix}) = 1$, $p(\begin{smallmatrix} 1 & 2 \\ 2 & 2 \end{smallmatrix}) = 0$. They are used in Algorithm 1 where we have four loops with the following states in its step 3: $(a = 0, b = 1)$; $(a = 1, b = 1)$; $(a = 1, b = 0)$; $(a = 1, b = 0)$.

Now Algorithm 3 enters a cycle that will be run two times: In step 4 the agents compute jointly shares of the secrets: $g_{1,1} = 1$, $g_{1,2} = 1$. The agents also compute sharing of secrets: $t_{1,0} = 1$, $t_{1,1} = 0$.

Agents can now compute at step 5 in Algorithm 3: $f_1(P) = 1 * 1 * 1 + 2 * 0 * 0 = 1$.

At step 6 in Algorithm 3 one applies Algorithm 2: `value-to-unary-constraint(1-1,2-1)` that translates the shared secret $f_1(P)$ into a vector of shared secrets: $\langle 1, 0 \rangle$ by first computing $\langle 0, -1 \rangle$, then in a second step $\langle 1 * (-1) * 1, (-1 + 1) * (-1 - 1) * (-1 + 1) \rangle$ obtaining $\langle -1, 0 \rangle$ and at the end by scalarly dividing with $\sigma_1 = -1$. This shares $\Lambda_{1,P}^*$.

In the following a new loop is started with step 4 by computing shared secrets: $g_{2,1} = 0$, $g_{2,2} = 1$. The agents also compute sharing of secrets: $t_{1,0} = 1$, $t_{1,1} = 1$.

Agents can compute at step 5 in Algorithm 3: $f_2(P) = 1 * 0 * 1 + 2 * 1 * 1 = 2$. Now the solution is recovered unto interested agents: $f_1(P) = 1$, $f_2(P) = 2$.

Alternative Implementation of satisfiable Let $q(\epsilon) = \sum_{c \in C} c(\epsilon)$. A solution of a CSP (X, D, C) is a valuation ϵ with $q(\epsilon) = |C|$. So, one can (less efficiently) compute:

$$p(\epsilon) = (q(\epsilon) - |C| + 1) \frac{\prod_{i=0}^{|C|-1} (q(\epsilon) - i)}{|C|!}$$

To find a solution where exactly x_0 constraints are satisfied:

$$p(\epsilon) = (q(\epsilon) - x_0 + 1) \frac{\prod_{i=0}^{x_0-1} (q(\epsilon) - i) \prod_{i=x_0+1}^{|C|} (i - q(\epsilon))}{x_0! (|C| - x_0)!} \quad (8)$$

Notably, to find a solution where the maximum number of constraints are satisfied, one can insert in X a variable x_0 with domain $|C|..1$ and use its current value in Eq. 8.

Theorem 2 *The described technique offers t -privacy (No collusion of less than t attackers can learn anything else than the final solution, and what can be inferred from it).*

Proof The technique is based on the evaluation of a set of functions consisting solely of additions and multiplication. It has been proven in [10, 2] that the compilation to multi-party computations of such a technique is t -private.

procedure GenerateAndSecureTest do

- each A_i securely distributes shares of the feasibility of each local tuple t_k^i to each A_j : (t_k^i, s_k^j) .
- **foreach** tuple t in Cartesian product of domains
if $(\prod_{t_k^i \in t} c(t_k^i)=1) // \text{ or } \sum_{t_k^i \in t} (c(t_k^i)-1)=0$
return t

Algorithm 4: Generate and Secure Test

Generate and Secure Test From the classic Generate and Test algorithm it is straightforward to obtain an algorithm that returns the first solution and whose only divulged information results from the knowledge of the solution, namely that there is no solution that is ordered lexicographically before the found solution. This protocol is given in Algorithm 4.

The value of a tuple is computed by multiplying (or summing) the values of the applicable predicates. The fact that the tuple is a solution is then verified by checking whether the value of the tuple equals 1.

Remark 4 *Generate and Secure Test is more efficient than the Secure Satisfaction1.*

However, *Generate and Secure Test* reveals more information than the *Secure Satisfaction1* (and than what is required). Namely the *Generate and Secure Test* reveals to all agents the values of all the variables in the first solution and the needed effort. This cannot be escaped due to the fact that the enumeration of all the tuples up to the first solution is performed jointly by all the agents and can be exploited for inferring secrets.

Discussion While algorithms like *Generate and Secure Test* is faster than *Secure Satisfaction1*, they can be shown to provide a strictly weaker security when the technique is extended to the generation of a random solution rather than of the first solution in lexicographic order [6]. For example, if the whole search space is exhausted before the solution is generated from the last tuple with *Generate and Secure Test*, then each agent can infer that all tuples he accepts and were not selected are refused by the other participants. This secret lose cannot occur with extensions built on *Secure Satisfaction1*.

4 Conclusions and Alternatives

Privacy has been recently stressed in [8, 4, 9, 7, 11] as an important goal in SisCSP algorithm design. We presented a technique where agents that need to cooperate and whose problems can be modeled as CSPs can find a first solution without any leak of additional information about their constraints. If one did not specifically ask for the *first* solution but for *any* solution, the fact of offering the first solution reveals a set of tuples that are forbidden by the secret constraints and therefore reveals about these secrets something not asked. In a distinct work we show how due to its invariant cost the technique proposed here can be extended to also avoid this potential information leak.

The proposed algorithm is the first such technique requiring no need of trusted servers (To be noted that this cherished property was the historically claimed driving force behind the development of public key cryptography [1]). It is assumed that all adversaries are passive (they follow the protocol) and that only a minority of the participants may be corrupted by any attacker. It is also assumed that the set of variables involved in the CSP of each agent are public knowledge. Most of these assumptions can be relaxed with different trade-offs and this is an important research field.

References

- [1] W. Diffie and M. Hellman. Multiuser cryptographic techniques. *IEEE Transactions on Information Theory*, 1976.
- [2] O. Goldreich, S. Micali, and A. Wigderson. Proofs that yield nothing but their validity and a methodology of cryptographic protocol design. In *Proc. of 27th IEEE FOCS*, pages 174–187, Toronto, 1986.
- [3] M. Hirt, U. Maurer, and B. Przydatek. Efficient secure multi-party computation. In *Advances in Cryptology - ASIACRYPT'00*, volume 1976 of *LNCSS*, pages 143–161, 2000.
- [4] P. Meseguer and M. Jiménez. Distributed forward checking. In *DCS*, 2000.
- [5] A. Shamir. How to share a secret. *Comm. of the ACM*, 22:612–613, 1979.
- [6] M. Silaghi. Solving a distributed csp with cryptographic multi-party computations, without revealing constraints and without involving trusted servers. In *IJCAI-DCR*, 2003.
- [7] M. C. Silaghi and B. Faltings. A comparison of DisCSP algorithms with respect to privacy. In *AAMAS-DCR*, 2002.
- [8] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Asynchronous search with private constraints. In *Proc. of AA2000*, pages 177–178, Barcelona, June 2000.
- [9] R. Wallace and E. Freuder. Constraint-based multi-agent meeting scheduling: Effects of agent heterogeneity on performance and privacy loss. In *DCR*, pages 176–182, 2002.
- [10] A. Yao. Protocols for secure computations. In *FOCS*, pages 160–164, 1982.
- [11] M. Yokoo, K. Suzuki, and K. Hirayama. Secure distributed constraint satisfaction: Reaching agreement without revealing private information. In *CP*, 2002.