2009 International Conference on Computational Science and Engineering

Privacy-Preserving Multi-agent Constraint Satisfaction

Thomas Léauté and Boi Faltings
École Polytechnique Fédérale de Lausanne (EPFL)
Artificial Intelligence Laboratory (LIA)
firstname.lastname@epfl.ch

***Abstract*—Constraint satisfaction has been a very successful paradigm for solving problems such as resource allocation and planning. Many of these problems pose themselves in a context involving multiple agents, and protecting privacy of information among them is often desirable. Secure multiparty computation (SMC) provides methods that in principle allow such computation without leaking any information. However, it does not consider the issue of keeping agents' decisions private from one another. In this paper, we show an algorithm that uses SMC in distributed computation to satisfy this objective.**

***Keywords*-distributed constraint satisfaction; privacy; secure multiparty computation; homomorphic encryption.**

I. Introduction

Constraint satisfaction is a very successful paradigm for solving problems such as resource allocation and planning. Often, these problems pose themselves in a setting involving multiple agents. For example, resources such as airport landing slots, railway or pipeline capacity, and communication frequencies are often shared between different users. Furthermore, manufacturing enterprises are increasingly distributed and supply chains should be coordinated among multiple agents. The resulting problems can be modeled as *multi-agent constraint satisfaction problems*.

A common way to solve such problems is to combine all variables and constraints into a single problem that can then be solved by a central server. When agents do not all belong to the same organization, a challenge is then to protect the privacy of the agents' constraints from the server, who could leak this often sensitive information. This is the motivation for *secure multiparty computation* (SMC). The field started with the millionaire problem [1]: two parties each have a secret number and want to compare them without either party learning the other's number. This problem has been generalized to *multiparty computation protocols* (MPC), where participants compute a public function of their private data without revealing it. Multiparty computation protocols have been applied to decide the winner of a large commercial sealed-bid auction without revealing the bids [2].

In principle, multiparty computation can also be applied to multi-agent constraint satisfaction, where the public function is a feasible solution to the constraint satisfaction problem. However, since the function is public, this would mean that the solution would be revealed to all involved agents. For example, all agents would learn what access other agents get to resources, or what the joint plans of other agents are. This is private information that usually should not be revealed.

Another problem is that the complexity of multiparty computation protocols increases significantly with the function complexity, and it is difficult to implement such a complex function as the solution to a constraint satisfaction problem.

A way to enhance the privacy guarantees of multiparty computation is to distribute the computation among the agents so that each agent computes its own part of the solution, and use a multiparty computation protocol that keeps the inputs from other agents hidden.

We show in this paper how such guarantees can be obtained using a well-known distributed algorithm for constraint satisfaction and optimization, *DPOP* [3].

This paper is structured as follows. We first present definitions of multi-agent constraint satisfaction and the privacy notions that can be associated with such a problem, as well as a review of earlier work on distributed constraint satisfaction algorithms and privacy protection. We then present our protocol, called *P^2-DPOP* (Privacy-Preserving DPOP), in three steps. Finally, we consider several open issues that are relevant to this context.

II. Preliminaries

This section recalls preliminary concepts in the fields of distributed constraint satisfaction (Sections II-A to II-D) and secure multiparty computation (Sections II-E and II-F).

A. Distributed Constraint Satisfaction

Definition 1 (DisCSP): A discrete *Distributed Constraint Satisfaction Problem* (DisCSP) is a tuple $< \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{C} >$:

- $\mathcal{A} = \{a_1, \ldots, a_{|\mathcal{A}|}\}$ is a set of agents;
- $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a set of variables. Each variable x_i is controlled by an agent $a(x_i)$;
- $\mathcal{D} = \{d_1, \ldots, d_n\}$ is a set of finite variable domains;
- $\mathcal{C} = \{c_1, \ldots, c_m\}$ is a set of constraints, where each constraint c_i is a function of scope $(x_{i_1}, \ldots, x_{i_l})$, such that $c_i : d_{i_1} \times \ldots \times d_{i_l} \rightarrow \{\texttt{true}, \texttt{false}\}$ assigns `true` to feasible tuples, and `false` to infeasible ones.

A feasible solution is a complete assignment such that the conjunction of all constraints $\bigwedge_{c_i \in \mathcal{C}} c_i = \texttt{true}$, which is the case exactly when the assignment satisfies all constraints.

In the following, we assume that all agents know the number n of variables in the problem, that the graph formed

978-0-7695-3823-5/09 $26.00 © 2009 IEEE
DOI 10.1109/CSE.2009.169

by the constraints is connected (otherwise each sub-problem can be solved independently), and that agents that control variables in the same constraint can communicate securely. We also assume that all constraints are either unary or binary, which is a common assumption as general constraint satisfaction problems can be reformulated into this form. Furthermore, we assume that all agents know an upper bound $D_{\max}$ on the sizes of variable domains, that they add dummy values to their domains to make them exactly that size, and that they have a local constraint that makes all these dummy values inconsistent. This is done so that variables cannot be identified by their domain sizes.

Furthermore, we assume that any constraint is known to all agents that control one of its variables. If an agent has a constraint that needs to be kept private, one can create a private copy of all variables in the constraint that the agent does not own, and link them with equality constraints to the original variables controlled by other agents. We illustrate this method below on a resource allocation example, but keep in mind that it applies to any DisCSP.

B. DisCSP Example: Resource Allocation

A large class of multi-agent constraint satisfaction problems where privacy is desired are resource allocation problems, where a set of resources has to be allocated among different agents. We show below how to model such problems as DisCSPs and what the privacy considerations are.

The model uses two sets of variables. y_a is controlled by the agent offering resource y and takes value 1 if it allocates resource y to agent A and 0 otherwise. a_y is a variable controlled by agent A and constrained to be equal to y_a. Three types of constraints exist on these variables:

1) Each resource y can be assigned to at most one agent, so for any variable pair (y_a, y_b), the pair of assignments $(1, 1)$ is infeasible.
2) Each agent has private constraints on the feasible combinations of resource assignments: if agent A wants at least one of resources y and z, then $c_A(a_y = 0, a_z = 0)$ is `false`, and `true` in all other cases.
3) All variables y_a and a_y must have equal values.

Constraints of types 1 and 2 must be kept private to the corresponding agents, for they would otherwise reveal important competitive information to competing agents. They can in fact be seen as constraints internal to the agents that define them (Figure 1). The only inter-agent constraints are of type 3. They need to be kept private to the agents that control the corresponding variables, but do not reveal any other private information to these agents about each other.

C. Privacy in DisCSP

DisCSP can encode a wide variety of problems and there are consequently many different types of privacy that could be important, depending on the application. In this paper, we assume that all participants honestly follow the protocol.

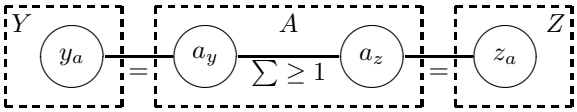

Figure 1. Constraint graph for a simple example: agent A wants at least one of resources y and z, respectively controlled by agents Y and Z.

This is a standard assumption for DisCSP, and is motivated by the fact that any deviation from the protocol could lead to producing an infeasible solution, which we assume is against every agent's interests. Rather than addressing the issue of adversarial behaviors, the techniques presented in this paper aim to provide strong guarantees on the leak of information about any particular agent to other participants. Participants can collude with each other: in fact, several variables in the problem might actually be owned by the same agent.

An important limitation on privacy is that the fact that a certain variable value is part of a consistent solution to the DisCSP will invariably leak some information. For example, if two variables belonging to different agents are connected by an equality constraint, each agent can infer the value of the other's variable from the value of its own variable. Therefore, any privacy guarantee will have to specifically exclude all information that can be inferred from knowledge of the final solution. We call such information *semi-private*:

Definition 2: Any information an agent can infer from the knowledge of the values that its variables take in any consistent solution is called *semi-private* information.

While excluding semi-private information, we distinguish four types of privacy guarantees (Definitions 3 to 6).

Definition 3: An algorithm preserves *agent privacy* when no agent learns the identity of the agent controlling any variable x_j that does not share a constraint with another variable x_i controlled by this agent. For example in Figure 1, agent Y cannot discover the identity of agent Z.

Definition 4: An algorithm preserves *topology privacy* if no agent learns about the existence of either constraints or cycles of constraints that do not involve at least one variable it controls. For example, agent Y cannot find out that agent A has a constraint over another resource z.

Definition 5: An algorithm preserves *constraint privacy* if an agent does not learn the feasibility or infeasibility of any particular tuple in a constraint that does not involve any variable it controls, except for semi-private information. For example, Y cannot find out that A wants y or z.

Definition 6: An algorithm preserves *decision privacy* if no agent can learn the values of any variable that it does not control in the final solution, except for semi-private information. For instance, Y cannot discover the outcome of any decision that Z makes in the chosen solution.

D. Algorithms for Distributed Constraint Satisfaction

The first algorithm for DisCSP was *asynchronous backtracking* [4], based on backtrack search. Since that time,

there have been a variety of search-based algorithms such as *AWC* [4], *AAS* [5], *AFC* [6], *ADOPT* [7], and *OptAPO* [8]. As was pointed out in [9], algorithms based on backtrack search leak problem information through their solution times. An alternative paradigm where computation time is independent of the consistency of particular solutions is dynamic programming, in particular the *DPOP* algorithm [3] that is the basis for the method in this paper.

Most earlier work on privacy in DisCSP has focussed on constraint privacy, and addressed the issue of how to measure the loss of privacy entailed by different algorithms. Franzin et al. [12] measure privacy loss as the reduction in entropy of other agents' preferences. Maheswaran et al. [13] develop a framework called *valuations of possible states* that measures privacy loss as the degree to which the possible states of other agents are reduced. Greenstadt [14] uses this framework to analyze privacy loss of DPOP and ADOPT.

While it is useful to *measure* privacy loss, in practice it is important to give guarantees that certain information is not revealed by an algorithm. Brito [15], [16] has developed methods to reduce privacy loss in search and in particular to protect privacy of a constraint from agents involved in it. Greenstadt [17] proposes the *SSDPOP* algorithm that uses cryptographic secret sharing to eliminate a major source of privacy loss in DPOP, but does not entirely eliminate it. The *P-DPOP* algorithm by Faltings *et al.* [18] guarantees partial privacy by obfuscating messages with large random numbers. However, the algorithm leaks agents' decisions to other agents that are involved in common constraints, and the obfuscation is not cryptographically secure.

The approach we propose in this paper is somewhat similar to those in [10], [9], [11]. However, these approaches violate agent and value privacy, as agents can learn the final decisions of other agents, even including non-neighboring agents. In our protocol, each agent only discovers a feasible assignments to its own variable using a separate multiparty computation, and does not learn the assignments to other variables (except for semi-private information).

E. Secure Multiparty Computation

Cryptography provides solutions for secure multiparty computation that can in principle solve constraint satisfaction problems with total privacy. This field focusses in particular on interactions between a small number of agents, such as scheduling a single meeting [19], the millionaire's problem of comparing numbers without revealing them [1], or secure auction protocols [20]. These techniques can in principle be extended to more complex scenarios such as constraint satisfaction, but their complexity quickly becomes unmanageable. For example, [9], [11] use cryptographic circuits. Other algorithms perform search and protect values with homomorphic encryption, in particular [10], [21]. However, such search-based algorithms can leak information through the computation time required to find a solution.

F. Manipulating Cooperatively Encrypted Booleans

For propagating consistency information, we need an encryption that encodes whether a tuple is consistent, that can only be read with a secret key, and that allows the following operations even without the secret key:

- the AND of an encrypted and a cleartext boolean;
- the OR of two encrypted booleans.

A simple scheme for such multiparty computation can be constructed on the basis of *ElGamal encryption* [22], which is a homomorphic public key cryptography system based on the intractability of the Diffie-Hellman problem [23]. It works as follows. Let p and q be large primes so that q divides $p-1$. G_q denotes Z_p^*'s unique multiplicative subgroup of order q. All computations are modulo p unless otherwise stated. Let $g \in G_q$ be a publicly known element. An *ElGamal key pair* consists of the private key $x \in Z_q$ and the public key $y = g^x$. Here p, g, y are publicly known and x is private. A message $m \in G_q$ is encrypted as follows:

$$E(m) = (\alpha, \beta) = (my^r, g^r) \tag{1}$$

where $r \in Z_q$ is an arbitrary random number chosen by the encrypter. The message is decrypted by computing:

$$\frac{\alpha}{\beta^x} = \frac{my^r}{(g^r)^x} = m.$$

An interesting feature of ElGamal encryption is that it allows to *randomize* an encrypted value to generate a new encryption of the same value that has no similarity with the original value. Randomizing $E(m)$ in Eq. (1) yields

$$E^2(m) = (\alpha y^{r'}, \beta g^{r'}) = (my^{r+r'}, g^{r+r'})$$

which still decodes to m. This can be used to make it impossible to gather information by merely comparing messages.

We represent `false` by 1, and `true` by a value $z \neq 1$, which allows us to compute the AND and OR operations:

- AND operation with a cleartext boolean:

$$E(m) \wedge \texttt{false} = E(1) \qquad E(m) \wedge \texttt{true} = E(m)$$

- OR operation over two encrypted booleans:

$$E(m_1) \vee E(m_2) = E(m_1 \cdot m_2) = (\alpha_1 \cdot \alpha_2, \beta_1 \cdot \beta_2)$$

In our algorithm, we use a shared ElGamal key (x, y) that is generated cooperatively by all agents.

- Distributed Key Generation: The ElGamal key pairs (x_i, y_i) of n nodes can be combined in the following fashion to obtain the shared key pair (x, y):

$$x = \Sigma_{i=1}^n x_i \qquad y = \Pi_{i=1}^n y_i$$

- Distributed Decryption: If each node publishes its decryption share β^{x_i}, the message m can be decrypted:

$$\frac{\alpha}{\Pi_{i=1}^n \beta^{x_i}} = \frac{\alpha}{\beta^x} = m$$

Procedure: CONSISTENCYPROP(received matrix $m(\cdot)$)
Require: local feasibility matrix $m_x(x, \cdot)$
1: Agree on codenames for x with each neighbor
2: $m'_x(x, \cdot) \leftarrow$ APPLYCODENAMES$(m_x(x, \cdot))$
3: $m(x, \cdot) \leftarrow m(\cdot) \wedge m'_x(x, \cdot)$
4: **if** x is not the root variable **then**
5: **if** the value x_0 of x has already been decided **then**
6: $m(\cdot) \leftarrow m(x = x_0, \cdot)$
7: **else**
8: $m(\cdot) \leftarrow \bigvee_x m(x, \cdot)$
9: $m(\cdot) \leftarrow E(m(\cdot))$
10: `self.parent`.TOPREVIOUS(`self`, CONSISTENCYPROP$(m(\cdot))$)
11: **else**
12: $x_0 \leftarrow$ FEASIBLEVALUE$(m(x, \cdot), 1, |d_x|)$
13: Add local constraint $x = x_0$
14: `self.index` $\leftarrow 0$

Figure 2. Finding a feasible value for the first variable in the ordering.

III. OVERALL P^2-DPOP ALGORITHM

The P^2-DPOP algorithm proceeds as follows.

1) The algorithm does not assume that any pair of two agents can communicate directly, which would be a breach of agent privacy. Instead, it only assumes that an agent can communicate directly with its neighbors, i.e. agents with which it shares constraints. However, the algorithm requires that agents be capable of sending messages to non-neighboring agents; it is therefore necessary to first establish a proper *communication structure* that routes messages to their final recipients.
2) While establishing this communication structure, the agents also construct a *linear ordering* of all variables, in which the first variable is called the *root*. Agents also decide on a common ElGamal key for encryption and cooperative decryption of messages.
3) The agents then repeatedly permute the ordering (and the communication structure) so that each variable becomes the root in turn. At each iteration, a feasible assignment is computed for the root variable, using distributed, secure multiparty computation.

The following sections describe the algorithm in more details. To motivate the requirements for the communication structure, Section IV first describes the distributed procedure to compute a feasible assignment for the current root variable. Section V then presents how the communication structure is obtained and used to route messages, and how the variable ordering is chosen. The permutation of the ordering to compute a feasible assignment to all variables in turn is explained in Section VI. For simplicity, we will refer to each variable as a virtual agent that exchanges messages with other variables.

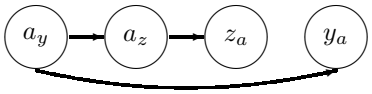

Figure 3. One possible DFS variable ordering rooted at variable a_y.

Table I
AGENT Y'S LOCAL FEASIBILITY MATRIX FOR $y_a = a_y$.

	$a_y = 0$	$a_y = 1$
$y_a = 0$	`true`	`false`
$y_a = 1$	`false`	`true`

Table II
MESSAGE SENT BY VARIABLE y_a TO VARIABLE z_a

(a) in cleartext

$a_y = 0$	$a_y = 1$
`true`	`true`

(b) encrypted

$[a_y = 0]_{y_a}$	$[a_y = 1]_{y_a}$
$E_1(z)$	$E_2(z)$

IV. FINDING A FEASIBLE VALUE FOR ONE VARIABLE

This section describes the method used to obtain a feasible value for just one variable (variable a_y in our example), while maintaining all four types of privacy outlined earlier. The method we present assumes that we have first constructed a linear ordering of all variables (Figure 3) and a communication structure such that each agent can send messages to the agent controlling the preceeding variable in the ordering, while preserving agent and topology privacy. Sending messages to an agent that is not one's direct neighbor is taken care of by the method TOPREVIOUS() that is described in Section V.

The procedure is presented in Figure 2. Like for the other distributed algorithms in this paper, we write pseudocode in terms of *remote method invocation* rather than in terms of message exchange. Before starting the procedure CONSISTENCYPROP(), each variable x computes its *local feasibility matrix* $m_x(x, \cdot)$, which is the join of all constraints over x that involve only variables preceding x in the ordering. The procedure then iteratively aggregates and propagates encrypted feasibility matrices, starting from the last variable, *projecting out* variables along the way, until the root variable obtains a feasibility matrix involving only itself.

A. Consistency Propagation on an Example

Consider the example in Figure 1, in which we want to compute a feasible assignment to variable a_y. Assume the linear variable ordering in Figure 3. The procedure is initiated by agent Y, which projects its variable y_a out of its local feasibility matrix (Table I) by row-wise OR of the entries (line 8). This yields the message in Table II(a).

If it sent this message to variable z_a, agent Y would reveal agent A's interest in resource y. To protect this information, it encrypts the message using codenames (line 2) and the encryption scheme from Section II-F (line 9). The resulting

Table III
JOIN OF MESSAGE IN TABLE II(B) WITH AGENT Z'S LOCAL MATRIX.

	$[a_y=0]_{y_a}$		$[a_y=1]_{y_a}$	
	$z_a=0$	$z_a=1$	$z_a=0$	$z_a=1$
$[a_z=0]_{z_a}$	$E_1(z)$	$E_3(1)$	$E_2(z)$	$E_3(1)$
$[a_z=1]_{z_a}$	$E_3(1)$	$E_1(z)$	$E_3(1)$	$E_2(z)$

Table IV
ENCRYPTED MESSAGE SENT BY VARIABLE z_a TO VARIABLE a_z.

	$[a_y=0]_{y_a}$	$[a_y=1]_{y_a}$
$[a_z=0]_{z_a}$	$E_4(z)$	$E_5(z)$
$[a_z=1]_{z_a}$	$E_6(z)$	$E_7(z)$

Table V
ENCRYPTED MESSAGE SENT BY VARIABLE a_z TO VARIABLE a_y.

	$[a_y=0]_{y_a}$	$[a_y=1]_{y_a}$
$[a_y=0]_{a_z}$	$E_9(z)$	$E_{10}(z)$
$[a_y=1]_{a_z}$	$E_{11}(z)$	$E_{12}(z)$

encrypted message is presented in Table II(b). We use the notation $E_i(.)$ to indicate encryption based on some random variable $r_i \in Z_q$. The use of two different random numbers yields two different representations for the two `true` entries in Table II(a), which renders the receiving agent Z incapable of inferring that the two entries are equal. We also use the notation $[x = x_0]_y$ to denote the use of a codename agreed upon with variable y, and that only x and y can decode.

Upon reception of the message, agent Z joins it with its local feasibility matrix (which is essentially the same as in Table I, but for variables a_z and z_a) using the AND operator on each pair of entries (line 3). The resulting joint matrix is presented in Table III. The agent finally generates its output message (Table IV) by projecting out variable z_a (by column-wise OR of the entries, line 8), and randomizing the entries (line 9) to hide the equality of some of the entries.

Variable a_z then joins the received message with its local feasibility matrix corresponding to $a_y + a_z \geq 1$ (line 3), projects itself out (line 8), and randomizes the entries (line 9), to obtain the message in Table V that it sends to variable a_y. Finally, a feasible value for a_y is extracted from the message (line 12) using the dichotomy algorithm in Figure 4. This algorithm performs repeated collaborative decryptions of consistency values using the DECRYPT() function, which circulates the cyphertext through all agents in the ordering using repeated calls to TOPREVIOUS(), each agent i raising the β to the power of its secret key x_i.

B. Privacy in Consistency Propagation

This section argues that the consistency propagation protocol used to compute a feasible value for the root variable preserves the four types of privacy defined in Section II-C. Formal proofs are beyond the scope of this paper; we only provide intuitions, first for constraint and decision privacy,

Procedure: FEASIBLEVALUE$(m(x = x_1 \dots x_d), l, u)$
1: **if** $l = u$ **then**
2: **if** DECRYPT$(m(x_l)) =$ **true then return** x_l
3: **else return null**
4: **else if** DECRYPT$(\bigvee_{i=l\dots\lfloor\frac{l+u}{2}\rfloor} m(x_i)) =$ **true then**
5: **return** FEASIBLEVALUE$(m(x), l, \lfloor\frac{l+u}{2}\rfloor)$
6: **else**
7: **return** FEASIBLEVALUE$(m(x), \lceil\frac{l+u}{2}\rceil, u)$

Figure 4. Procedure to find a feasible value in the root's message $m(x)$.

and then for agent and topology privacy. We also propose a variant of the algorithm that sacrifices some topology privacy to produce smaller messages. Finally, we briefly discuss the algorithm's resistance to collusion.

1) Constraint and Decision Privacy: We claim the consistency propagation method maintains constraint privacy: consistency values cannot be learned without cooperative decryption, and the propagation looks the same independently of the constraints. The root variable only finds out which of its values are consistent, which is semi-private information since it can be inferred from a solution. Furthermore, it does not violate decision privacy since only the first agent computes its value and does not reveal it to others.

2) Agent and Topology Privacy: Observe first that if the constraint graph consists of a single branch (as in Figure 1), and the ordering is such that any two consecutive variables are neighbors in the constraint graph, for instance the ordering (y_a, a_y, a_z, z_a), then agents only learn about their direct neighbors and so both agent and topology privacy are also guaranteed during the consistency propagation phase.

However, privacy problems arise from *backedges* in the linear ordering, such as the one between variables a_y and y_a in Figure 3. The leaf of the backedge then needs to transmit to its predecessor in the ordering a message that refers to another agent elsewhere in the constraint graph (namely, the root of the backedge). For instance, agent Y needs to send the message in Table II(a) to agent Z, containing references to A that Y would want to hide from Z.

This is achieved by identifying the corresponding variable's name and values by codenames only known to the agents at both ends of the backedge. In this way, agents in between do not know what variable the message refers to, nor the possible values for this variable. The codenames are generated by the root of the backedge and communicated through a secure channel to the agent at the other end. If a variable is the root of several backedges, it communicates different codenames to each corresponding agent; the rationale for this is given in the following paragraph. Agents also pad their domains to a maximum size so that they cannot be identified by their variable domain sizes.

3) Complexity vs. Privacy Tradeoff: The message in Table V contains two dimensions referring to the same

variable a_y, encoded with two different codenames. In a slight variant of the protocol, a_y could have communicated the same codename to y_a and a_z. Variable a_z would have then been able to merge the two dimensions into a one-dimensional message. More generally, communicating the same codename to all neighbors would reduce the maximum number of dimensions in a message from a value equal to the largest number of backedges over any given variable, to a value equal to the *induced width* of the ordering. This would however sacrifice topology privacy: a_z would learn that another variable has a constraint with a_y, and could hence discover a cycle without being part of it.

4) Collusion Resistance: In the consistency propagation procedure, only the root is allowed to initiate cooperative decryptions of the entries in the feasibility matrix it receives. However, if it colluded with another agent, it could use its decryption power to decrypt the matrix received by this other agent. To prevent this, the algorithm only allows the root to initiate $\lceil \log_2 D_{\max} \rceil$ cooperative decryptions, where $D_{\max}$ is the size of the variable domains. This is exactly the number of decryptions the root needs to find a feasible value for its variable (Figure 4).

V. Anonymously Establishing Communication

The procedure previously described assumes that there is a linear ordering of variables and any agent is able to communicate with the agent controlling the preceding variable in the ordering. This section describes the algorithms that construct such a structure, based on DFS trees.

Figure 5 shows the procedure InitializeOrder() that constructs an initial order and communication structure. Agents first cooperatively elect one variable as the root using an anonymous leader election protocol (Section V-A). They then use the procedure DFSOrder() to traverse the constraint graph in depth-first order from the chosen root to generate a DFS ordering of the variables. Each variable stores this ordering in the form of a vector that is encrypted with its ElGamal key, and that is later used to select each variable in turn as the root of the ordering (Section VI).

A. Anonymous Leader Election

The problem of leader election has been widely studied in the literature. Provided each variable has a unique identifier, a simple procedure is to apply a function (such as raising to a power in a finite field) to the identifier, and then distributedly select the variable with the largest transformed identifier. A distributed algorithm that only requires knowledge of an upper bound on the number of variables is described in [18].

B. Anonymous DFS Ordering

Variables are ordered in a linear sequence according to their position in a DFS traversal of the constraint graph, starting from the root, using the function DFSOrder(). Each variable x records:

Procedure: InitializeOrder()
Require: total number of variables n
1: `self.key` $\leftarrow$ new ElGamal key
2: `self.order` $\leftarrow [E(1), \ldots, E(1)]$ n times
3: `self.shift` $\leftarrow$ random integer in $[1, n]$
4: `self.visited` $\leftarrow$ **false**
5: `self.index` $\leftarrow 0$
6: **if** ElectLeader() $=$ **true then**
7: DFSOrder(`self`, 1)
8: **else**
9: **wait until** `self.index` > 0
10: `self.order`[`self.index`] $\leftarrow E(z)$ with $z \neq 1$

Procedure: DFSOrder($parent, index$)
1: **if** `self.visited` $=$ **true then**
2: **return** $index - 1$
3: `self.parent` $\leftarrow parent$
4: `self.index` $\leftarrow index$
5: `self.pp` $\leftarrow \emptyset$
6: `self.children` $\leftarrow \emptyset$
7: `self.visited` $\leftarrow$ **true**
8: $openList \leftarrow$ all neighboring variables except $parent$
9: **while** $openList \neq \emptyset$ **do**
10: $neighbor \leftarrow$ Pop($openList$)
11: $i \leftarrow neighbor$.DFSOrder(`self`, $index + 1$)
12: **if** $i = index$ **then**
13: `self.pp` $\leftarrow$ `self.pp` $\cup \{neighbor\}$
14: **else**
15: `self.children` $\leftarrow$ `self.children` $\cup \{neighbor\}$
16: $index \leftarrow i$
17: **return** $index$

Figure 5. Algorithms used to construct the communication structure and the variable ordering.

- its `index` in the ordering;
- its `parent`: the variable it was reached from;
- its `children`: the variables whose parent is x;
- its *pseudoparents* `pp`: the variables that precede x in the ordering and that are linked to x through a constraint; we call such constraints with pseudoparents *backedges*.

The resulting structure enables a variable to send a message to its predecessor in the ordering, even if it does not know its identity. This is performed by the procedure ToPrevious() (Figure 6), which routes the message through the DFS tree back to the variable that was visited just before the sender variable.

C. Building the Communication Structure on an Example

Each variable runs InitializeOrder() (Figure 5), following which it initially sets itself to *unvisited* (line 4). It then calls ElectLeader() (line 6), and the variable that discovers itself as the leader (in the example in Figure 3, it is variable a_y) triggers the DFS traversal of the constraint

Procedure: TOPREVIOUS($caller, procedure$)
1: **if** $caller = first($`self.children`$)$ **then**
2: **invoke** `self.`$procedure$
3: **else**
4: $child \leftarrow$ last child in `self.children` before $caller$
5: $child$.TOLASTLEAF($procedure$)

Procedure: TOLASTLEAF($procedure$)
1: **if** `self.children` $= \emptyset$ **then**
2: **invoke** `self.`$procedure$
3: **else**
4: $last($`self.children`$)$.TOLASTLEAF($procedure$)

Figure 6. Algorithms to call a procedure on the previous variable.

graph (line 7). Variable a_y first sets its index in the vector `self.order` to 1 (procedure DFSORDER(), line 4), and then sets itself to *visited* (line 7). Then, it recursively calls the procedure DFSORDER() on each of its neighbors (line 11), starting with variable a_z. Variable a_z sets its parent to a_y (line 3) and its index to 2, and recursively calls the procedure on its only remaining *open* neighbor z_a, which sets its parent to a_z and its index to 3.

Since z_a has no more open neighbors (line 9), the control returns to a_z, which learns the index $i = 3$ of the last visited variable (line 11). Had this index been equal to its own index (line 12), this would have meant that the current variable $neighbor = z_a$ had already been visited (line 2), and a_z would have concluded that z_a is a pseudo-parent (line 13). Instead, its marks it as its child (line 15). The control then returns to a_y, which marks a_z as its child and moves on to its next open neighbor y_a, and repeats the procedure.

Finally, all variables set to $E(z)$ the element in the vector `self.order` corresponding to their index (see procedure INITIALIZEORDER(), line 10). As a result, each variable has an element $E(z)$ in a different place in its vector `self.order`, and, if they decoded the first entry of this vector, only the root variable would obtain `true`. This property will be used later to re-root the ordering (Section VI).

D. Communication along the DFS Structure on an Example

Let us now illustrate how the communication structure is used in the procedure CONSISTENCYPROP() (Figure 2) to recursively pass the control to the previous variable in the ordering. In order to pass the control to its previous variable, y_a calls the procedure TOPREVIOUS() (Figure 6) on its parent a_y. Since y_a is not a_y's first child (line 1), y_a instructs its previous child a_z (line 4) to transfer the control to the last leaf in its subtree (line 5), which is variable z_a. When it is z_a's turn to transfer the control to its previous variable, it calls TOPREVIOUS() on its parent a_z, which concludes from the fact that z_a is its first child that it can keep the control. The same happens when passing the control from a_z back to the root variable a_y.

Procedure: P^2-DPOP()
Require: total number of variables n
1: INITIALIZEORDER()
2: **while** `self.order` $\neq \emptyset$ **do**
3: `self.visited` $\leftarrow$ **false**
4: SHIFTORDER(`self.order`, n)
5: $isRoot \leftarrow$ DECRYPT(POP(`self.order`))
6: **if** $isRoot \neq 1$ **then**
7: DFSORDER(`self`, 1)
8: `self.parent`.TOPREVIOUS(`self`, CONSISTENCYPROP([`true`])
9: **else**
10: **wait until** `self.index` $= 0$

Procedure: SHIFTORDER($order, round$)
1: **if** $round > 0$ **then**
2: $order \leftarrow$ circular shift of $order$ by `self.shift`
3: $order \leftarrow E(order)$
4: `self.parent`.TOPREVIOUS(`self`, SHIFTORDER($order, round - 1$))
5: **else**
6: `self.order` $\leftarrow order$

Figure 7. P^2-DPOP: Privacy Preserving DPOP.

E. Privacy in DFS Construction and Communication

The procedure to generate the communication structure satisfies agent privacy: variables only send messages to neighbors, and these messages do not contain any information about other agents in the problem. Topology privacy is also guaranteed, as agents only learn about constraints or cycles of constraints that involve variables they control.

Similarly, once the structure has been established, the communication mechanism does not leak any more privacy, since variables only send messages to neighbors. In particular, sending a message to the previous variable can be performed without knowing its identity, and only requires forwarding the message to your parent or to a child.

VI. COORDINATED COMPUTATION OF ALL VALUES

The P^2-DPOP algorithm is described in Figure 7. The agents compute feasible values for their variables in turn using procedure CONSISTENCYPROP(). Consistency across iterations is ensured by the fact that once a value has been decided for a variable, its assignments are restricted to that value in the following iterations. Note that this does not leak any information about these values since the computation is carried out on randomized and encrypted values. This is the main difference with earlier work [10], [9], [11], [18], in which the algorithm would leak the chosen feasible value for a variable to other agents, thereby violating value privacy.

The procedure P^2-DPOP() involves the following steps: the algorithm first initializes, for each variable, a vector `order` that identifies a common permutation for all

Table VI
VALUES OF `self.order` FOR EACH VARIABLE, IN CLEARTEXT.

	`shift`	init	iter 1	iter 2	iter 3	iter 4
y_a	2	$[z,1,1,1]$	$[1,1,1,z]$	$[z,1,1]$	$[1,1]$	$[1]$
a_y	1	$[1,z,1,1]$	$[z,1,1,1]$	$[1,1,1]$	$[1,1]$	$[1]$
a_z	3	$[1,1,z,1]$	$[1,z,1,1]$	$[1,z,1]$	$[1,z]$	$[z]$
z_a	1	$[1,1,1,z]$	$[1,1,z,1]$	$[1,1,z]$	$[z,1]$	$[1]$
root	$\Sigma = 7$	-	a_y	y_a	z_a	a_z

agents by having an encrypted $z \neq 1$ in a different position for each agent, and 1 in all other positions. This is constructed by all agents in INITIALIZEORDER() (Figure 5).

The algorithm then loops through all n agents; each checks if it is the root, i.e. if its `order` vector starts with a $z \neq 1$ (lines 5 and 6). If so, it constructs a variable ordering with its variable x at the root, and computes a feasible value for x using the procedure CONSISTENCYPROP(). After n iterations, all variables have obtained a value.

At each iteration, the procedure SHIFTORDER() is used to permute the `order` vector so as to choose a new root for the ordering. This procedure passes the vector through the cycle of variables, each variable shifting it by its random shift (line 2) and randomizing the result (line 3). This results in the vector being shifted by the same amount, independently of the starting point, and is thus used to shift all vectors in lockstep. At each iteration, the first element of the vector is taken off so that the agent cannot become the root twice.

If the problem is infeasible, no agent finds consistent values for its variables, and on termination all agents know that there is no consistent solution. It is possible to improve the algorithm efficiency for this case by directly propagating this information from the first agent, but in the interest of clarity we do not consider this simple improvement here.

A. Walk through an Example

Table VI shows the values of the `self.order` vectors during the course of the algorithm. It assumes that ELECTLEADER() chose y_a as the root of the initial communication structure. Before entering the loop on line 2, the vectors are the ones in the column labelled *init*. After circularly shifting all vectors by each variable's `shift`, variable a_y becomes the root during the first iteration. All vectors are then truncated and shifted again, yielding variable y_a as the new root. The procedure is repeated twice, so that z_a and a_z can in turn become roots.

B. Privacy Guarantees

The privacy properties of each iteration alone were discussed in previous sections. When it comes to making each variable the root in sequence, the algorithm preserves agent and topology privacy, as the messages exchanged contain encrypted vectors, and a non-root variable cannot discover the new root. Randomizing the vectors each time prevents from making inferences from one iteration to the next.

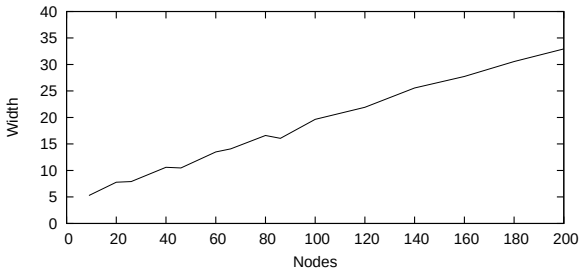

Figure 8. Average width versus the number of nodes in the problem.

VII. EXPERIMENTAL EVALUATION

We ran experiments to check the scalability of the algorithm. We used a 32-bit ElGamal encryption with 8 bytes for every encrypted value (4 bytes for α and for β). For a problem with width 6, and 8 possible values for each variable, the memory requirement would be 2 MB in terms of largest message size. In practice, we could go up to width 8 (25 variables) with a limit of 128 MB on the maximum message size.

VIII. OPEN ISSUES

A. Proving Honest Propagation

In secure multiparty computation, agents may be tempted to not execute the protocol correctly and manipulate the outcome in their favor. Because of privacy, such manipulation is hard or impossible to detect. Particular attention is therefore paid to ensure the non-manipulability of the protocols by malicious agents. This problem could be addressed for example by incentive mechanisms such as in *M-DPOP* [24] or by verification of consistency in the computation by comparing and selectively decrypting messages.

B. Using DFS Trees for Propagation

The encryption scheme we use in this paper allows us to compute the OR of two encrypted booleans, and the AND of an encrypted boolean with a cleartext boolean, but not the AND of two encrypted booleans. This makes it necessary to use a linear variable ordering: an agent would not know how to join multiple encrypted feasibility matrices received from multiple successors in a more complex ordering. Using a pseudo-tree ordering like in *P-DPOP* [18] would yield better performance: the largest consistency message would contain a number of dimensions equal to the *treewidth* of the pseudo-tree, rather than to the induced width of the corresponding linear ordering. Figure 8 shows how this induced width increases with the number of variables, for random pseudo-trees of constant treewith 4. This shows how the use of a pseudo-tree ordering would yield smaller messages; however it would require a fully homomorphic encryption scheme.

IX. Conclusion

Many circumstances require a set of agents to collectively take decisions in a coordinated way. Constraint satisfaction is a general and widely used formalism for this purpose. However, in general it requires agents to give up the privacy of their constraints and decisions. While secure multiparty computation addressed the privacy of constraints, it does not preserve privacy of decisions.

We have shown an algorithm, P^2-DPOP, that uses distributed computation together with a simple secure multiparty computation protocol to protect both types of privacy. It is useful for scenarios where agents all have the same interests, for example to avoid conflicts in their assignments.

We believe that such methods can be made significantly more powerful by incorporating stronger mechanisms for multiparty computation. In particular, such methods may allow verification of honest behavior and thus allow application to problems with self-interested agents that might otherwise manipulate the process. On this basis, it could also be possible to extend the methods to distributed optimization.

Acknowledgment

The authors would like to thank Nikhil Garg for his contribution in implementing the algorithm and producing some experimental results, and Adrian Petcu for his supervision.

References

[1] A. C. Yao, "Protocols for secure computations," in *Proceedings of SFCS'82*, 1982, pp. 160–164.

[2] P. Bogetoft, D. L. Christensen, I. Damgård, M. Geisler, T. Jakobsen, M. Krøigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, M. Schwartzbach, and T. Toft, "Secure multiparty computation goes live," Cryptology ePrint Archive, Report 2008/068, February 2008.

[3] A. Petcu and B. Faltings, "DPOP: A Scalable Method for Multiagent Constraint Optimization," in *Proceedings of the 19th Intl Joint Conf. on AI (IJCAI'05)*, 2005, pp. 266–271.

[4] M. Yokoo and K. Hirayama, "Algorithms for distributed constraint satisfaction: A review," *Autonom. Agents and Multi-agent Sys. (JAAMAS)*, vol. 3, no. 2, pp. 185–207, June 2000.

[5] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings, "Asynchronous search with aggregations," in *Proceedings of the AAAI and IAAI Conferences*, 2000, pp. 917–922.

[6] A. Meisels and R. Zivan, "Asynchronous forward-checking on DisCSPs," in *Proceedings of the DCR'03 Workshop*, 2003.

[7] P. J. Modi, W.-M. Shen, M. Tambe, and M. Yokoo, "ADOPT: Asynchronous distributed constraint optimization with quality guarantees," *AI Journal*, vol. 161, pp. 149–180, 2005.

[8] R. Mailler and V. R. Lesser, "Solving distributed constraint optimization problems using cooperative mediation," in *Proceedings of AAMAS'04*, vol. 1, 2004, pp. 438–445.

[9] M.-C. Silaghi and D. Mitra, "Distributed constraint satisfaction and optimization with privacy enforcement," in *Proc. Intl Conf. on Intelligent Agent Tech. (IAT'04)*, 2004, pp. 531–535.

[10] K. Suzuki and M. Yokoo, "Secure generalized Vickrey auction using homomorphic encryption," in *Proc. of the Intl Conf. on Financial Crypto. (FC'03)*, vol. 2742, 2003, pp. 239–249.

[11] M.-C. Silaghi, B. Faltings, and A. Petcu, "Secure combinatorial optimization simulating DFS tree-based variable elimination," in *Proc. 9th Intl Symp. on AI and Math*, 2006.

[12] M. S. Franzin, E. C. Freuder, F. Rossi, and R. Wallace, "Multi-agent constraint systems with preferences: Efficiency, solution quality, and privacy loss," *Computational Intelligence*, vol. 20, no. 2, pp. 264—286, May 2004.

[13] R. T. Maheswaran, J. P. Pearce, E. Bowring, P. Varakantham, and M. Tambe, "Privacy loss in distributed constraint reasoning: A quantitative framework for analysis and its applications," *JAAMAS*, vol. 13, no. 1, pp. 27–60, 2006.

[14] R. Greenstadt, J. P. Pearce, and M. Tambe, "Analysis of privacy loss in distributed constraint optimization," in *Proc. of the 21st National Conf. on AI (AAAI06)*, 2006, pp. 647–653.

[15] I. Brito and P. Meseguer, "Distributed forward checking," in *Proceedings of CP'03*, vol. 2833, 2003, pp. 801–806.

[16] ——, "Distributed forward checking may lie for privacy," in *Proceedings of the CP-DCR'07 Workshop*, 2007.

[17] R. Greenstadt, B. Grosz, and M. D. Smith, "SSDPOP: Using secret sharing to improve the privacy of DCOP," in *Proceedings of the CP-DCR'07 Workshop*, 2007.

[18] B. Faltings, T. Léauté, and A. Petcu, "Privacy guarantees through distributed constraint satisfaction," in *Proc. of the Intl Conf. on Intelligent Agent Tech. (IAT08)*, 2008, pp. 350–358.

[19] T. Herlea, J. Claessens, B. Preneel, G. Neven, F. Piessens, and B. D. Decker, "On securely scheduling a meeting," in *Proc. Intl Conf. on Information Sec. (SEC'01)*, 2001, pp. 183–198.

[20] F. Brandt, "Fully private auctions in a constant number of rounds," in *Proc. of FC'03*, vol. 2742, 2003, pp. 223–238.

[21] M. Yokoo, K. Suzuki, and K. Hirayama, "Secure distributed constraint satisfaction: Reaching agreement without revealing private information," *Artificial Intelligence*, vol. 161, no. 1–2, pp. 229–245, January 2005.

[22] T. Elgamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE Transactions on Information Theory*, vol. 31, no. 4, pp. 469–472, July 1985.

[23] Y. Tsiounis and M. Yung, "On the security of Elgamal-based encryption," in *PKC'98*, vol. 1431, 1998, pp. 117–134.

[24] A. Petcu, B. Faltings, and D. C. Parkes, "M-DPOP: Faithful distributed implementation of efficient social choice problems," *JAIR*, vol. 32, pp. 705–755, 2008.