Appendix B

Problems with a secure test operator

The computer operation that is the most frequently used in solving CSP-related problems is the "test" (consistency checks). Often, centralized algorithms are evaluated solely with the number of tests (consistency checks) that they make.

Many existing cryptographic protocols are targeted to the computation of functions based on addition and multiplication. Since solving a CSP with only this two operators is too expensive (Chapter 15), we would naturally like to also have a secure "test" operator that would allow the branching typically used in backtracking and local search.

I will describe here some alternatives that I tried in obtaining such an operator. None of these alternatives has an acceptable level of security. However, describing these negative results is important since they are not yet known among other researchers.

A famous researcher blocked for many years the world's research of the now successful field of neural networks because he did not believe it could work. Similarly, by not knowing the different trade-offs of competing techniques for privacy, one risks to block prematurely one or the other still promising research directions.

B.1 How could one perform "tests" without losing information?

A test for a set of variables x taking values v has the form:

$$\textbf{if } P(x,v) = 0 \textbf{ then do call } A\textbf{, else do call } B.$$

According to the Definition 15.2 taken from (Chaum *et al.* 1988a), an attacker, *Eve*, should not be able to find $P(x,v)$, but only the final result returned by A or B.

Remark B.1 *Whenever* Eve *sees the branch, A or B, on which the search continues, she can infer the value of $P(x,v)$.*

In our case, if all the agents have to execute a branch in their computation, they all know which branch they take.

5 *How could we use the "test" operator and still do not let Eve find anything about $P(x,v)$.*

A natural research direction, that I noticed and tried simultaneously with other researchers, is to try to shuffle x and v. It should be done in such a way that the tested predicate is some $P(x',v')$ and nobody can find the relation between x and x', respectively v and v'. Even if one cannot stop Eve from finding the value $P(x',v')$, she will not be able to interpret it its semantic.

In the following I do this using the Merritt election protocol. The approaches presented here can fail against some types of statistical attacks.

B.2 Compiling the Backtracking

Here we will discuss some compilations for backtracking that are not satisfactory despite their attractiveness. The basic trusted party solution for DisCSPs is to gather all the problems on a computer and to solve it with an algorithm, e.g. backtracking or local search. Cryptographic protocols typically try to simulate the run of a trusted party and perform its computations over the distributed shares of the parameters.

B.2.1 Secure backtracking

Cryptographic protocols typically try to simulate the run of a trusted party and perform computations over the distributed shares of the parameters (Goldwasser & Bellare 1996). In this direction, I will first recall the typical solution to DisCSPs with a trusted party, and then I describe tentative solutions for compiling its steps unto cryptographic protocols.

The first step of a compilation of a trusted party solution for DisCSPs consists in compiling the transfer of the local problems to the trusted party. The local problem of an agent A_i in DisCSPs can be considered with acceptable generality to consist of:

- a domain D_i for a local variable x_i, and
- a constraint C_i involving x_i and variables of lower priority agents.

The elements of a domain D_i are v_i^1, v_i^2,.... Each agent A_i has therefore to share D_i and C_i.

For a problem with discrete domains, the feasibility (0,1) of a tuple can be shared as a number (e.g. with Shamir's schema). The protocol known and followed by each participant is the secure backtracking[1] where a current variable s_i is being instantiated to a current value v_i^k. All the agents owning shares of the tuples of constraints for the assignment (s_i, v_i^k) with active instantiations of lower priority variables $s_j, j < i$, cooperate to check the corresponding tuples. In case of success, the next chosen variable is s_{i+1}. In the case a failure is found, the agents proceed to the value v_i^{k+1}. If no such value exists all the agents backtrack to the variable s_{i-1} when s_{i-1} exists, or announce failure.

The lesson from the previous description is that even if the feasibility of the constraint tuples are securely shared, they could be reconstructed from the trace of the search. To force the use of such a protocol with "test" operations, as discussed in the previous section, a first trick that comes to mind, is to shuffle the variables and values such that nobody could understand to whom they belong.

Shuffling techniques can be obtained as shown in (Merritt 1983)'s election protocol. I explain first this algorithm for values with one index, as described in (Goldwasser & Bellare 1996), and slightly rephrased for the current application and notations.

Merritt's election protocol was initially meant for accounting the votes during elections and ensures the privacy of the relation vote-elector by shuffling the indexes of the votes. The shuffling is obtained by a chain of permutations (each permutation being the secret of an election center) on the encrypted votes.

Merritt's shuffling protocol with one index Each agent A_i publishes a public key E_i of a corresponding private key. E_i can be used with two parameters, $E_i(s, r)$ where s is the secret and r is a random number for randomizing the result. Alternatively, E_i can be used with a single parameter, $E_i(s)$ when the encryption is not randomized (r is a constant).

Each agent that announces a value v for an index j chooses a random number h_j and computes: $E_1(E_2(...E_n(v, h_j))) = y_{n+1,j}$. The y values are posted in order from A_n to A_1. Each agent A_i, for each $y_{i+1,j}$, chooses a random number $r_{i,j}$ and broadcasts $y_{i,j'} = E_i(y_{i+1,j}, r_{i,j})$. Given a random permutation π_i of the integers [1..n], (secret of A_i), then $j' = \pi_i(j)$.

$$y_{1,j'''} = E_1(E_2(...E_n(y_{n+1,j}, r_{n,j})..., r_{2,j'}), r_{1,j''})$$

[1] Any other known centralized technique can be similarly adapted.

The shuffled value can be found immediately by two decryption rounds in the order $A_1 \rightarrow A_2 \rightarrow ... \rightarrow A_n$.

Merritt's shuffling protocol with several indexes The values that have to be shared are:

- values of the form (v, i, j), denoting v as the j's value of x_i.
- feasibilities of tuples like $(1/0, (var_1, val_1), (var_2, val_2), ..., (var_n, val_n))$.

Actually, all the values of a domain could be permuted together in one chain of messages. The real values v are actually not needed when the constraints are represented extensively. In this case, it is enough to announce the existence of the values with a unary constraint, and each value is uniquely referred with its index. A chain of agents that compose their secret permutations is referred to as a *Merritt chain*.

The agents have to use the same permutation for indexes of variables, and reserved permutation for indexes of values of the same variable. The number of random permutations that have to be stored by each agent is therefore $n + 1$ where n is the number of variables. Let the maximum domain size be d.

For a value with t pairs of indexes, the modifications to Merritt's algorithm are straightforward. Each agent A_i publishes a public key E_i of a corresponding private key. Each agent that announces a value v for the pairs of indexes $((j_1, u_1), ..., (j_t, u_t))$ chooses a random number h_j and computes: $E_1(E_2(...E_n(v, h_j))) = y_{n+1,(j_1,u_1),...,(j_t,u_t)}$. The y values are posted in order from A_n to A_1. Each agent A_i, for each $y_{i+1,(j_1,u_1),...,(j_t,u_t)}$, chooses a random number $r_{i,j}$ (j stands for the tuple $(j_1, u_1), ..., (j_t, u_t)$) and broadcasts $y_{i,(j'_1,u'_1),...,(j'_t,u'_t)} = E_i(y_{i+1,(j_1,u_1),...,(j_t,u_t)}, r_{i,j})$. Given random permutations $\pi_i, \pi_{i_1}, ..., \pi_{i_n}$ of the integers [1..n], respectively [1..d] (these permutations are secrets of A_i), then $j'_k = \pi_i(j_k)$ and $u'_k = \pi_{i_{j_k}}(u_k)$.

$$y_{1,(j'''_1,u'''_1),...,(j'_t,u'_t)} = E_1(E_2(...E_n(y_{n+1,(j_1,u_1),...,(j_t,u_t)}, r_{n,j})..., r_{2,j'}), r_{1,j''})$$

The shuffled feasibility values of constraint tuples is found immediately by two decryption rounds in the order $A_1 \rightarrow A_2 \rightarrow ... \rightarrow A_n$.

Cryptographic backtracking with Merritt's protocol (SBMP) After A_n decodes the feasibility of the tuples of the initial DisCSP, they are associated to the permuted indexes obtained after encoding by A_1. A_n and A_1 can distribute the result (e.g. sharing it between agents, or sending it to a single (eventually external) agent), obtaining a new (shared) CSP.

With the described protocol, the initial discrete DisCSP is shuffled and a CSP is obtained. The solving algorithm can be run by all the agents as a secure protocol over the shuffled CSP, or it can be run by some single agent which broadcasts the result. If the search is run by a single agent called *solver agent*, it can be easily proven that:

Once the result is obtained, it can be translated back by inverting the Merritt's protocol, such that all the permutations are undone. This is straightforward by one traversal through the agents $A_1 \rightarrow A_2 \rightarrow ... \rightarrow A_n$.

S-attack on SBMP Now I describe an attack on SBMP that I call S-attack. An attacker can find the secrets as follows. The main feeble point of the described SBMP that distributes the shuffled problem to the participants is that the initial DisCSP and the shuffled CSP are isomorphic. Therefore, an agent that succeeds to build the shuffled CSP (e.g. from the trace of the backtracking) and discovers a homomorphism/isomorphism between his constraints and the constraints of the obtained CSP, can recover the identity of several variables, and this way a certain amount of privacy is lost. One or several such agents that cooperate can recover the whole initial DisCSP.

Example 2.33 *An attacker A_e (Eve) can introduce in the system two binary constraints:*

- *q_1 is a constraint on x_i and x_j such that all the tuples are* **feasible** *except for e_{k_1} tuples, $e_{k_1} = 1$.*

- q_2 *is a constraint on* x_i *and* x_k *such that all the tuples are* **infeasible** *except for* e_{k_2} *tuples,* $e_{k_2} = 1$.

There is only little probability that another agent has such constraints. Therefore, after all permutations, A_e *is able to recognize constraints* $q^{p_{t_1}}(x^{p_{t'_1}}, x^{p_{t'_2}}), q^{p_{t_2}}(x^{p_{t'_1}}, x^{p_{t'_3}})$ *as corresponding to* q_1 *respectively* q_2 *— they have only one feasible, respectively only one infeasible tuple.*

Based on the previous observation, A_e *can infer that the final composed permutation contains* ($i \rightarrow p_{t'_1}$, $j \rightarrow p_{t'_2}$, $k \rightarrow p_{t'_3}$).

More than this, the composed permutation of the values involved in the tuples that are different is also revealed directly.

Now, several other constraints with different patterns (number of feasible/infeasible) tuples can be submitted by the same attacker or by other attackers and the inferences can be easily combined to reveal most secrets.

Remark B.2 *Revealing the composed permutation can lead to the revelation of the initial constraints from the permuted ones. While the interest of the agents in variables may be known, their constraints can be revealed.*

To increase the intractability for finding isomorphisms, all the domains of all the variables have to be extended to d to avoid the identification of the variables by their domain size. Also it is interesting to use a secure technique of CSP reformulation to change the set of variables and further transform the configuration of the constraints before search.

Remark B.3 *If the shuffled problem is not restricted to trusted third parties, SBMP is not even 1-private.*

B.2.2 Cryptographic search for continuous domains

Actually, my first effort was concerned with symbolically represented problems. While it was not more successful than SBMP, I mention it for the completeness of the description.

When C_i is a symbolic predicate over reals, its sharing can be done by adding redundant operations (e.g. $+x - x$) and then distributing the resulting operations to different agents.

Example 2.34 *When* C_i *defined as* $x_1 + x_2 = 2$ *has to be shared to 4 agents,* A_i *can transform* C_i *in* $((x_1^2 + x_2) + (x_2^2 + x_1)) - (x_2^2 + x_1^2 - 3) = 5$. *Now, assuming that variables* x_{10}, x_{11}, x_{12} *are reserved for reformulations,* C_i *can be distributed as:*
$x_{10} = x_1^2 + x_2$,
$x_{11} = x_2^2 + x_1$,
$x_{12} = x_2^2 + x_1^2 - 3$,
$x_{10} + x_{11} - x_{13} = 5$

The description of the constraints can be encoded as a string with local indexes for variables. The constraint description, the variables and the bounds of the domains can then be shuffled with Merritt's protocol as described for discrete domains. The shuffled string description of the constraints is distributed and enforced by some agents and the (cryptographic) search can proceed over the shuffled numeric CSP. The feeble points are similar to the ones for discrete problems, namely the isomorphism can be detected by some agents.

Example 2.35 *The constraint* $x_{10} = x_2^2 + x_1^2 - 3$ *can be represented as the tuple ("v_1=v_2^2+v_3^2-3",10,2,1). "v_1=v_2^2+v_3^2-3" is the string description of the constraint.*

The numeric domains are represented as tuples of the form:
(list of intervals, index of the variable).

Each agent needs only one permutation function, namely the one for indexes of variables, as values are not permuted.

Enhancing intractability To enhance intractability of recognizing one's numeric constraint, each agent A_i in Merritt's shuffling schema must further split each received string encoding of a constraint in more constraints by its secret heuristic. A set of indexes of variables can be reserved in

1. Each agent splits its own constraint in k constraints with new variables, and send each encrypted split to the first agent of each of the k chains.
2. A passage through the Merritt chains performs shuffling with an additional encryption.
3. the constraints are decrypted by two inverse traversals through the corresponding chains.
4. the shuffled plain constraints are passed through the Merritt chains where they are warped (Remarks B.4 and B.5)
5. Each constraint output from an encoding chain will be distributed to an agent (or stored and enforced by the agents that are the last in the chains).
6. The shuffled problem is solved.
7. The results of the search can be transformed back to the initial variables by inverse passages through any of the Merritt's chains.

Figure B.1: The SBMMC protocol.

advance for these needs. The transformations can include injective transformations of the received variables by operations like scaling, inverting, etc. The initial variables can even be completely transformed and discarded.

Example 2.36 *A constraint tuple ("v_1=v_2^2+v_3^2-3",10,2,1) can be transformed into ("v_1=v_2^4+4v_2+v_3^2+1",10,15,1)*
by transforming x_2 to $(x_{15}+2)^2$, where x_{15} is introduced and x_2 is discarded from the resulting y.

The domains have to follow the transformations of the corresponding variables, such that they would no longer be recognized in the end.

Remark B.4 *To allow that domains are transformed, one can pass them in plain, encrypting only their index by permutations. This can however follow a first shuffling where the plain constraints are separated from their indexes in an encrypted form.*

However, cryptographic techniques that allow the transformations to be performed on encrypted values also exist, and can be used.

Remark B.5 *To allow the agents on the Merritt chain to split other's constraints, the constraints must be passed in plain (not encrypted) along Merritt's chains that compute the permutations of the indexes. This can however follow a first shuffling where the plain constraints are separated from their indexes in an encrypted form.*

Remark B.6 *To avoid that A_n succeeds to find out all the split constraints of each agent, k parallel chains of shuffling can be used.*

Example 2.37 *$A_{kn+1}, ..., A_{k(n+1)}$ share for each n their permutation functions and transformations of initial variables. A set of k Merritt chains are defined by the sets of agents $A_i, A_{i+k}, ...$, for i<k.* [2]

Example 2.38 *When $n = 3m$, $A_n \rightarrow ... \rightarrow A_2 \rightarrow A_1$ is replaced by three chains:*
$A_{3m} \rightarrow ... \rightarrow A_6 \rightarrow A_3$
$A_{3m-1} \rightarrow ... \rightarrow A_5 \rightarrow A_2$
$A_{3m-2} \rightarrow ... \rightarrow A_4 \rightarrow A_1$

The protocol is described in Figure B.1.

[2]When n cannot be divided by k, the remaining agents can form a shorter chain that is a branch of an initial chain. The agent at the separation point can split the incoming plain constraints and send the split pieces along the two branches.

Remark B.7 *The y values of each constraint string encoding are not broadcasted but rather they are only sent along the chains. Encrypted communication is required to avoid that some agent finds all constraints permuted by all the agents sharing permutation functions with him.*

With assumptions of intractability of isomorphic/homomorphic recognition of one's constraint, the obtained protocol for numeric constraints is secure to any $min(k-1, n/k-1)$ colluders. This protocol is called Secure Backtracking with Multiple Merritt chains (SBMMC).

S-attack in SBMMC An attacker of SBMMC can act by submitting symbolic constraints with very particular structures (e.g. a high number of exponentials). The structures can then be easily recognized even in fragmented and warped versions and the permutation is detected.

Moreover, whenever all the agents at the same level on the Merritt chains collude, they can compose their knowledge on the constraint fragments and the recognition of initial constraints and corresponding permutations becomes even much easier.

B.2.3 Discrete domains with SBMMC

A comment somebody made about the S-attack against SBMP revealed the power of dynamically complicating the constraint matrix. That discussion helped to realize that an algorithm more robust to S-attack than SBMP can be obtained by using SBMMC on discrete constraints, too. Unfortunately, even this version is not secure against S-attack.

After the first chain of permutations on encrypted constraints, the constraint are decrypted. The permuted plain splits of constraints are further permuted and split as typical in the third phase of SBMMC.

The peculiarity of the discrete version I propose for SBMMC consists in the splitting technique for discrete technique.

Example 2.39 *An unary constraint q,* $1,0,0,1,0,1,1,0$*, can be split in two constraints whose composition maintains the initial semantic. E.g., q can be split into* $q_1 = 1,0,1,1,1,1,1,0$ *and* $q_2 = 1,1,0,1,0,1,1,1$.

It can be noted that any value ruled out by q, is ruled out either by q_1 *or by* q_2*. The technique can be applied identically to any n-ary constraint, where the tuples in the n-ary constraint are treated as values in a unary one.*

S-attack on discrete SBMMC When an agent A_i introduces in the system a constraint having exactly one infeasible tuple, if exactly one constraint split has the same property, A_i detects with a certain high probability the connection between initial variables and variables of the constraint. If a second constraint sharing one variable with the first constraint has e.g. exactly 2 infeasible tuples, the permutation of the shared variable is revealed with 100% certainty. Therefore SBMMC is secure only when the number of infeasible tuples in submitted constraints is relatively high.

Improvement to SBMMC on discrete problems One of the most recent improvements that I noticed and propose to discrete SBMMC, is to allow agents in the Merritt chains to introduce new variables, as it was done in symbolic SBMMC. This was initially not obvious to do with discrete constraints. A simple solution that I propose is to simply increase the arity of the treated constraints. The use of this additional technique allows a more secure treatment of constraints with low number of infeasible tuples.

Remark B.8 *When a discrete constraint is extended on a new direction, infeasible tuples can be added for some values of the added variable as long as there exists a value of the added variable where a solution can be found for the space where infeasible variables are added. A coherent policy (invalidated parts of the search space) has to be maintained for all constraints extended with the same variable.*

S-attack At the beginning of step 4 in SBMMC, the first agents in the warping chains (e.g. A_n) can detect isomorphisms between initial fragments of constraints and their shuffled version. It can therefore discover the secrets at this stage.

B.2.4 Related work

(Yokoo *et al.* 2002)[3] described an independent related result that also consists of a shuffling technique. It applies to distributed valued discrete CSPs. While that work did not mention and did not start from the Merritt election protocol for shuffling, it proposes a custom technique, somewhat related to Merritt's shuffling technique that we employed.

Concerning the integration of the constraint shuffling in search, this related work is different from SBMP in that it uses a set of external trusted agents that shuffle constraints and perform the search separately.

Remark B.9 *Their solution is better than a classical trusted party solution by the fact that the trusted party has to cooperate with at least a participant in order to discover the secrets. While similarly to SBMP, that technique is not safe to coalitions of attackers, due to the use of trusted parties it is 1-private.*

The other differences between (Yokoo *et al.* 2002) and SBMP are the fact that (Yokoo *et al.* 2002) always dynamically decodes the constraints at each check, it uses valued CSPs and therefore it can apply its technique to optimization problems, but do not consider and do not apply to intensionally (symbolically) represented constraints.

Another peculiarity of the technique in (Yokoo *et al.* 2002) is that the constraints are grouped for each value of a variable and the encrypted index of the value is distributed with the group. This value will be decrypted by the sender once a solution is found, in order to interpret the solution. This is different from our case where the value is obtained by a backward chain of permutations.

A contribution of the proposed work described in this chapter with respect to the related work appearing simultaneously, is the discovery of the feebleness of this kind of techniques, the "S-attacks".

B.2.5 Research directions

Secure asynchronous/parallel backtracking

Remark B.10 *Any multi-threaded centralized algorithm can be compiled as an asynchronous secure protocol. Parallel backtracking algorithms can be similarly used.*

This seems to be an appreciated research direction.[4] However, its motivation is not yet clear.

B.3 Summary

In this chapter we analyze techniques based on the Merritt election protocol for securely shuffling, decomposing, splitting and warping numeric and discrete constraints. These techniques are not fully secure against collusions of agents and a set of attacks called S-attacks is shown for each of the techniques.

[3] I was offered its submitted version, after the submission of this thesis.

[4] After I first submitted the previous remark, an anonymous reviewer mentioned that he also believes it to be a promising topic.