

Spec for SIMAP/Danisco Database and Client Encryption

Ivan Damgård

ver. 1.0

1 Introduction

This document specifies how the database with bids for the SIMAP/Danisco auction is organized, how the client should encrypt bids and how the servers get hold of data from the database.

2 Bids in plaintext

When a buyer talks to the client software, he will give up to some maximum number of bids. Maximum is to be determined later, for now we take 5 as an example. The bids must be decreasing in the sense that if the price gets higher you want to buy at most the amount you wanted to buy at a lower price. We have a predefined price grid containing u potential prices per unit, and any price that is used in a bid must be one of these u possibilities. One way to have the user specify bids is as follows

- The user selects the highest price he is willing to pay, and specifies the quantity he wants to buy (below a certain maximum).
- The user selects a price that must be below the previous one, and a quantity that must be higher than the previous one
- This may go on until the user has given up to 5 bids. But he does not have to give more than one.

It will probably be necessary to give additional explanations and help in the process but we do not specify this here.

After the bid is specified, we have prices $p_1 < p_2 < \dots$, and quantities $m_1 \geq m_2 \geq \dots$. The client applet now translates this such that there is a quantity for every potential price, as follows: we set the quantity to m_1 for every price that is $\leq p_1$, to m_2 for every price p' that is $p_1 < p' \leq p_2$, etc. Quantities above the highest bid price are set to 0. The resulting list of u quantities is called a *bid vector*.

For sellers, we need an additional input, namely a list of contracts that this sellers owns, with identifier and quantity for each contract. This is assumed to be given to our system as a token that is passed from Growcom where users log in before they come to us. The rule is that you can only sell entire contracts, so we can ask for bids by having the user specify some number of prices and for each price say which contracts he wants to sell at that price. This must be increasing, i.e., if the price gets higher you want to sell at least the amount you wanted to sell at a lower price. One way to have the user specify this is as follows

- The user selects a quantity to sell by selecting a subset of the contracts he has, and selects the lowest price he is willing to sell for.
- The user selects a quantity that must be higher than the previous one, i.e., a larger subset of contracts, and a price that must be above the previous one.
- This may go on until the user has given up to 5 bids. But he does not have to give more than one.

After the bids are specified, we have prices $p_1 < p_2 < \dots$ and quantities $m_1 \leq m_2 \leq \dots$, and for each bid a subset of the contracts. The applet now translates this to a bid vector, which is again a list of quantities, one for each price in the grid. The quantities for prices $< p_1$ are set to 0, for prices p' with $p_1 \leq p' < p_2$ are set to m_1 , etc.

We now have a list of u numbers x_1, \dots, x_u . We now add u additional numbers x_{u+1}, \dots, x_{2u} to the bid vector. These specify for each price which of his contracts the seller wants to sell at the given price. So a seller's bid vector contains $2u$ numbers.

For price nr. i , this is done as follows: the seller wants to sell x_i units at this price. Let b_1, b_2, \dots be the bits of x_{i+u} , least significant bit first. Then

$b_j = 1$ if and only if the seller wants to sell his j 'th contract at this price. This refers to the list of contracts above. Thus, it must be the case that

$$x_i = b_1 \cdot \text{size of 1'st contract} + b_2 \cdot \text{size of 2'nd contract} + \dots$$

The prime number p that is used for secret sharing later is of size k bits, so all numbers used here are assumed to be at most $k - 1$ bits, since this guarantees they are less than p . This also means we assume that sellers have at most $k - 1$ contracts.

3 Database Format

The database will have one entry for each bidder, each entry has the following components

- Bidder's name
- Seller or Buyer (a flag)
- A ordered list of the contracts that the bidder has. Only necessary for sellers, as these are only allowed to to sell entire contracts, as mentioned above. The list may be empty for buyers. The list contains identifiers according to whatever is Danisco's standard for identifying contracts.
- The encrypted bid vector. The encryption algorithm is described below.

4 Encryption Algorithm

We let the entire cleartext bid vector be $X = (x_1, \dots, x_v)$, so in the above notation $v = u$ for a buyer, or $v = 2u$, for a seller.

There are 3 servers who will do the calculation of the market clearing price, they have public keys pk_1, pk_2, pk_3 .

We will assume that a pseudorandom function F is available. One can initialize F with a key K . We then assume that F can take an index i as input, where $1 \leq i \leq v$ and outputs a pseudorandom k -bit value $F_K(i)$.

For the calculation of the servers, we use a fixed prime p , of size k bits. We also define 3 polynomials of degree 1, as follows: $f_1 : f_1(0) = 1, f_1(1) = 0$, $f_2 : f_2(0) = 1, f_2(2) = 0$, and $f_3 : f_3(0) = 1, f_3(3) = 0$.

To encrypt X , do the following:

1. Choose 3 random keys K_1, K_2, K_3 .
2. Compute encryptions $E_{pk_1}(K_2, K_3), E_{pk_2}(K_1, K_3), E_{pk_3}(K_1, K_2)$.
3. Initialize $F_{K_1}, F_{K_2}, F_{K_3}$, if required.
4. For $i = 1 \dots v$, compute:

$$y_i = F_{K_1}(i) + F_{K_2}(i) + F_{K_3}(i) + x_i \bmod p.$$

The final encryption is: $E_{pk_1}(K_2, K_3), E_{pk_2}(K_1, K_3), E_{pk_3}(K_1, K_2), y_1, \dots, y_v$

5 Processing at the server side

When reading the data base, the servers want to create normal secret shares of the values x_1, \dots, x_v that are encrypted, either because they are going to compute on the values, or because the values are to be revealed. The following allows the servers to do this for any subset of the values x_i .

The input is therefore a set I of indices of values that we want to access and the encrypted data, $E_{pk_1}(K_2, K_3), E_{pk_2}(K_1, K_3), E_{pk_3}(K_1, K_2), y_1, \dots, y_v$. Each server ℓ , $\ell = 1, 2, 3$ does the following:

1. Decrypt the j 'th ciphertext from the input to get K_1, K_2, K_3 , except for K_ℓ .
2. Initialize F_{K_j} for $j \neq \ell$, if required.
3. For each $i \in I$, compute a share of x_i as

$$share_{i,j} = y_i - F_{K_1}(i) \cdot f_1(j) - F_{K_2}(i) \cdot f_2(j) - F_{K_3}(i) \cdot f_3(j) \bmod p$$

keeping in mind that since $f_\ell(\ell) = 0$, it does not matter that you do not know $F_{K_\ell}(i)$.

It is of course important for efficiency that if you need to do this for large set I , you call the above procedure once for the entire set and not many times for each index in I – since each call costs public key decryption and inits of F .

6 Some details

The choice of the K_j 's needs to be unpredictable to outsiders, any secure random generator that is available to the applet is fine, no one else needs to regenerate them.

$F_K(i)$ can be implemented as $AES_K(i)$ where, say, the least significant k bits are used as output. Note that AES keys can be processed initially such that encryption is faster afterwards. It may be a significant advantage if this can be exploited in the applet. This is also the reason for mentioning initialization of F explicitly above. Of course, if we can get an AES implementation that runs in native code on the client, we can be much faster.

Standard crypto providers for Java usually have built-in functions for “key enveloping”, i.e. encrypting keys such as (pairs of) AES keys under RSA, we may as well use those directly.

An alternative is to use the built-in Secure Random class in the mode where the seed you give determines the output completely. In this case $F_K(i)$ is computed by first initializing Secure Random with a seed determined from K and i , in such a way that the seed is guaranteed to be different for different i . It may be sufficient to just concatenate K and i . Then we extract k bits from the generator, and define this to be $F_K(i)$. With this alternative, there is no global initialization step, we have to initialize every time we compute an output. This may slow us down, but on the other hand, there is a good chance that this will run in native code on almost any machine, and so may be faster than AES.

7 Choice of parameters

From the data known at the time of writing, it seems that we will need to compute on quantities that are no more than 32 bits. For this, a 64 bit prime p will certainly be sufficient, so $k = 64$ seems a reasonable choice. This means we are implicitly assuming that sellers have no more than 63 contracts, but this also seems safe at the moment. The current suggestion is to have a price grid with 4000 prices, i.e., $u = 4000$ in the above.