Secure Set Union and Bag Union Computation for Guaranteeing Anonymity of Distrustful Participants

Stefan Böttcher and Sebastian Obermeier
Department of Computer Science, University of Paderborn, Fürstenallee 11, Paderborn, Germany
Email: {stb, so}@uni-paderborn.de

***Abstract*— The computation of the union operator for different distributed datasets involves two challenges when participants are curious and can even act malicious: guaranteeing anonymity and guaranteeing security. Anonymity means that the owner of a certain data item cannot be identified provided that more than two participants act. Security means that no participant can underhandedly prevent data items of other participants from being included in the union. We present a protocol for computing both, the set union and the bag union of data sets of different participants that guarantees both properties: anonymity and security even if participants act malicious, i.e.modify messages or change or stop the protocol. We prove the correctness of the protocol and give experimental results that show the applicability of our protocol in a common environment.**

***Index Terms*— distributed databases, multiparty computation, secure anonymous union computation, sovereign information sharing**

I. Introduction

The technique of *sovereign information sharing* [1]–[7] is used for database operations involving sets of sensitive data from different curious partners. The purpose of sovereign information sharing is that only the result of a database operation is revealed, but not the complete database contents of each partner that have been used for the database operation. For example, for intersection computation, only the common data tuples shall be revealed. However, when the union of the data sets $D_1 \dots D_n$, i.e. $D = D_1 \cup D_2 \dots \cup D_n$, must be calculated and returned, there are different requirements that must be fulfilled: anonymity and security.

The anonymity property ensures that it is impossible to identify which data set $D_1 \dots D_n$ contains a certain data item found in the resulting union set D. In other words, it is impossible to identify the owner of a certain data item of the union in scenarios with more than two participants as long as less than $n-1$ partners cheat. Additionally, the sizes $|D_i|$ of the data sets D_i are concealed.

The security property ensures that if one user stops the protocol at an arbitrary point in time, the damage to the other users should be minimized, i.e. the malicious participant should not be able to prevent other participants from contributing their data to the union. This also includes strategies to avoid that one user can steal information and then stop the protocol. Whenever users may act not only curious but also malicious, it is important to detect that users act malicious, i.e. that they modify messages or change the protocol.

However, the approaches [1]–[7] neither guarantee anonymity nor security for the union operator. This motivates the use of our proposed union computation protocol, which is described in this article.

A. Example

As an example, we consider several banks that want to share the names of customers who received loans, but cannot pay the instalments.

Although the banks want to warn each other about customers that are not creditworthy anymore, no bank wants the other banks to know that it was the bank that has lent money to a non-paying customer. Although the number and sum of all outstanding credits of a customer shall be revealed, the outstanding credits of each singular bank shall be hidden. Thus, the banks focus on an anonymized union computation. To make sure that a bank B cannot delete data from the union that is not owned by B itself, a computation that guarantees that all contributed data will appear in the union is necessary. For example, if a bank A is in debt at bank B, A could try to prevent B from contributing A's name to the union.

Whenever the amount of outstanding credits per customer is important, the computation of the bag union ensures that all credits of a customer are revealed and thus can be summed up, while the computation of the set union would only reveal the fact *that* a customer went bust.

Furthermore, we assume that the banks do not fully trust each other, i.e. no bank wants to risk that another bank gets knowledge about its particular outstanding unsafe credits. For this reason, our protocol must take into consideration that participants might cheat and even alter the protocol or fake or modify messages in order to get to know which customers have also non-payed credits at certain competitors.

This paper is based on "Secure Anonymous Union Computation Among Malicious Partners," by S. Böttcher and S. Obermeier, which appeared in the Proceedings of the Second International Conference on Availability, Reliability and Security (ARES 2007), Vienna, Austria, April 2007.

B. Contributions

We present a solution to the problem of computing the union D of different data sets $D_1 \dots D_n$ of the participants $P_1 \dots P_n$ that may act malicious. The particular contributions of our protocol are that our protocol

- does not let participant P_i conclude which other participant P_j contributes a certain data tuple and even keeps $|D_j|$ secret
- prevents that a malicious participant P_i excludes another participant P_j from contributing P_j's data D_j to the union D
- does not depend on the use of a third party
- detects when participants change data, or invent or delete messages
- limits damage to other participants when some participants change or delete data or messages or stop the protocol at an arbitrary point in time

Beyond our previous contribution [8], this article further,

- addresses the problem of computing the *bag* union, which allows to perform further operations on the bag union, e.g. sum computation.
- extends the algorithms proposed in [8] by also computing a secure and anonymous bag union, and provides an overview algorithm
- shows a business application that benefits from the proposed protocol
- defines adversary models and argues why the protocol is resilient against these attacks

Furthermore, we show experimental results on the time that is needed for the secure union computation.

II. Problem Description

Definition II.1 *A data item d belongs to the* set union *of different datasets $D_1 \dots D_n$ if it belongs to at least one data set D_j.*

Definition II.2 *A data item d occurs k times in the* bag union *of different datasets $D_1 \dots D_n$ if d is in total k times contained in the data sets or data bags $D_1 \dots D_n$.*

Definition II.3 Anonymity *means that P_j cannot identify the owner or the owners of a data tuple d besides itself.*

Definition II.4 Security *means that a participant P_k cannot prevent that a data tuple d of participant P_j occurs in the computed set union or in the computed bag union, respectively.*

The problem considered is to provide a secure anonymous computation of set union or bag union, respectively.

III. Assumptions and Requirements

We have the following assumptions regarding each participant's behavior:

- Participants contribute the data that they want to contribute, i.e. a participant P_i that wants to hide some of its data can exclude this data from its data set D_i before the protocol starts.
- Participants may change messages.
- Participants may stop the protocol execution at every point in time.
- Participants exchange public keys securely before the protocol starts.
- Each honest participant stops the protocol when cheating is detected and notifies the other participants.

Our protocol must fulfill the following requirements:

- Compute the set union or the bag union $D = D_1 \cup D_2 \dots \cup D_n$ of the data sets $D_1 \dots D_n$
- Do not allow the owner of D_j, i.e. P_j, to conclude which participant P_k owns a data item d.
- Conceal size information $|D_j|$ for all participants P_k with $k \neq j$.
- Conceal how many parties own a certain data item for set union computation.
- Do not use or trust third parties.
- Detect cheating of participants, i.e.
  - detect message manipulation,
  - detect faulty decryption, and
  - detect message suppression.
- Furthermore, if a participant stops the protocol execution or is excluded due to cheating, this must either result in no exposure of data tuples at all, or the union of the remaining participants' data must be completely revealable. After excluding malicious participants and restarting the protocol, the remaining participants should not be able to change their contributed data or add new data if data has already been decrypted, e.g. when using voting schemes.
- Finally, before revealing any data of the union, each participant must be able to verify that its contributed data will be considered in the revealed union during a verification phase. Furthermore, each participant must be able to check whether the final revealed union of all data corresponds to the (encrypted) data on which all participants agreed before, i.e. it must be able to check whether each other participant acted fair.

Note that the requirement for anonymity especially means that it is not possible to detect whether a participant contributed any data at all. Thus, our protocol explicitly tolerates participants that only want to learn the union, but do not contribute any data.

IV. Solution

We first give an overview of the key ideas of our protocol, and then explain the details of each algorithm used in our protocol. Section V proves the correctness of our algorithm.

A. Overview

Our solution can be structured into three phases, where each phase implements different key ideas to fulfill the requirements:

The Anonymous Exchange Phase ensures the anonymized cyclic exchange of encrypted data records. Depending on whether the set union or the bag union shall be computed, the initialization of this phase is different. A key idea of this initialization is to add dummy data that can be identified as such after decryption in order to hide the number of data items that have been contributed by a participant.

In order to avoid that other participants can conclude where a data item originated, a second key idea is to randomly select either an own data tuple, or an already received data tuple for sending it to the next participant. This ensures that the receiving participant cannot conclude whether the received data tuple originated from the sending participant or from any other participant. Furthermore, to prevent that participants may successfully group together and observe the incoming and outgoing data of a single participant to detect the data that originated from this participant, a third idea is that each data tuple is sent along an exchange cycle that is determined by the data tuple itself.

The Verification Phase verifies that no participant is excluded from receiving the union. The goal is to check whether participants cheated during the cyclic exchange phase, i.e. to detect data suppression or manipulation. Therefore, first, each participant checks whether he received its own tuples in the cyclic exchange phase, and each honest participant, as soon as he detects cheating, stops the protocol and notifies the other participants about the reason.

Whenever the participant has verified that all of its tuples are contained in the encrypted union, it computes the result of a hash function hash() to the lexicographically ordered union D. Second, each participant sends a part of the calculated hash value hash(D) to all other participants. Only if all participants have the same union data and thus the same hash value, this phase is successful.

The Decryption Phase ensures a verifiable decryption of each encrypted record. The key idea is that after a participant P_i decrypted the union with its own key, i.e. it takes away a layer of decryption, the next participant P_{i+1} encrypts the data with P_i's public key in order to check whether the decryption function was applied correctly.

To simplify our algorithm descriptions, we omit to repeat the following details for each phase:

- Whenever a message must be sent from participant P_i to participant P_j, the message is encrypted with P_j's public key and signed with P_i's private key.
- Encryption does not rely on random variables, i.e. encrypting a value d with public key pk always results in the same ciphered text.
- The used encryption mechanism is commutative, i.e. the application order of multiple encryption and decryption functions does not matter.

B. Anonymous Exchange Phase

Depending on whether the set union or the bag union should be computed, the initialization of the Anonymous Exchange Phase differs. Algorithm 1 shows the Set Union Initialization Phase. First, an amount of dummy tuples is created for each participant P_i (line 1). Then, it is shuffled with the real ownData (line 2) and stored into the sendQueue to hide the size information $|D_i|$ of the data contributed by P_i. To be able to identify the dummy records later on, each dummy record starts with the word "dummy" and a random value is appended. Each entry of the sendQueue is sequentially encrypted with the public keys of all participants (line 4 - 6).

Algorithm 1 Set Union Initialization Phase for P_i

```
1: dummy = createDummyRecords()
2: sendQueue = new Queue( shuffle(dummy, ownData) )
3:                  ▷ shuffle tuples and store them in send queue
4: for k=1 to n do
5:     encrypt each j ∈ sendQueue with publicKey(P_k))
6: end for
7: return sendQueue()
```

Algorithm 2 shows the initialization phase for the Bag Union Computation. As in the set union, the participants create dummy records which are used for anonymization. However, in order to ensure that the resulting data in the sendQueue is unique among all participants, in the bag union initialization, each data item is concatenated with a separator "–%–" and sufficiently long randomly created value (line 3 - 5).

Algorithm 2 Bag Union Initialization Phase for P_i

```
1: dummy = createDummyRecords()
2: usedData = new Queue()
3: for each j ∈ ownData do
4:     usedData.add( concatenate( j, "–%–", drawRandom()))
5: end for
6: sendQueue = new Queue( shuffle(dummy, usedData) )
7:                  ▷ shuffle tuples and store them in send queue
8: for k=1 to n do
9:     encrypt each j ∈ sendQueue with publicKey(P_k))
10: end for
11: return sendQueue()
```

Then as in the set union, the data is merged and shuffled with the dummy tuples and encrypted (lines 6 to 10).

After protocol initialization, the cyclic exchange phase starts, which is shown in Algorithm 3.

To prevent that incoming and outgoing data tuples are monitored and data origin is inferred from this, each data tuple is sent on a different cycle which prevents monitoring even if malicious participants group together for the following reason. Each message that is sent is encrypted with the receiver's public key. The function computeRecipient(tuple) (Algorithm 3, line 21) assigns to each data tuple a different cycle along which it is sent. Therefore, tuples are not passed on the same cycle, which would imply that all tuples are always sent to the *same* next participant, but for each tuple a *different* next participant is calculated. For this purpose, each participant P_j sends an encrypted tuple to that participant that is

determined by the function computeRecipient(tuple), which takes the tuple as input. This function computes a permutation of the n participants which depends on the concrete data tuple. A possible implementation of the function computeRecipient(tuple) is the following: Hash the tuple into n numbers of the domain $\mathbb{Z}_n$, i.e. $s_1 \dots s_n$. To compute a permutation, start with the sequential order of all participants and shift participant P_j by s_j units to the right.

TABLE I.
DIFFERENT EXCHANGE CYCLES FOR DIFFERENT TUPLES

Tuple	Encr. Tuple	Shift	Permutation
Smith, John	a4fa3 ...	1,3,1,2	$P_1P_3P_2P_4$
Miller, Tom	hu3kj ...	1,2,0,2	$P_1P_4P_3P_2$

Table I demonstrates the computation of different exchange cycles for $n = 4$ participants. The second column contains the data tuples encrypted with the public keys of all four participants. The "shift" column is computed by applying a hash function on the second column, and the result of this is cut into n pieces and each piece is transformed to $\mathbb{Z}_4$. The resulting values indicate the number of positions by which a participant is shifted. The first permutation results from the following operations $P_1P_2P_3P_4$ (initial), $P_2P_1P_3P_4$ (P_1 shifted 1 position), $P_1P_3P_4P_2$ (P_2 shifted 3 positions), $P_1P_4P_3P_2$ (P_3 shifted 1 position), $P_1P_3P_2P_4$ (P_4 shifted 2 positions). Whenever a participant receives a tuple that it has not seen before, it calculates the permutation, locates its own position within the permutation, and sends the tuple to the next participant when the algorithm selects this tuple.

The primary goal of Algorithm 3 is to collect and distribute the n times encrypted tuples from all participants to all other participants in a way that hides the real owner of a data tuple and eliminates duplicates. Therefore, each participant sends each n times encrypted data tuple exactly once to the next recipient indicated by the computed permutation path.

After computing the queue of tuples to be sent by P_i, P_i creates a new queue, sendQueueOther for storing received data tuples (line 2). Furthermore, a receive thread is started (line 4) that stores the received data at a random position into sendQueueOther (line 13). However, a data item that is already present in one of the queues sendQueue or sendQueueOther is not inserted a second time (line 12).

First, a random amount of encrypted dummy tuples of the sendQueue is taken (line 29) and sent to the next participant by using the sendTuple procedure. This method marks each tuple that it is going to send (line 20) to avoid loops, which would occur due to a repeated sending of the same tuples again and again. Then, it computes the recipient of the tuple (line 21) as explained before. Furthermore, the algorithm computes the difference of the number of sent and received data tuples, and waits until this difference drops below a specified threshold value (line 22 - 23). This synchronization avoids that a participant flushes its own sendQueue too fast, i.e.

Algorithm 3 Cyclic Exchange Phase for Participant P_i

```
 1: procedure EXCHANGE PHASE(sendQueue)
 2:     sendQueueOther = new Queue()          ▷ store rcv. tuples
 3:     receivedTuples = 0; sentTuples = 0
 4:     RECEIVETHREAD.START( )
 5:     SENDTHREAD.START( )
 6: end procedure
 7:
 8: procedure RECEIVETHREAD.RUN()
 9:     repeat
10:         tuple = receiveTupleFromOtherParticipant()
11:         receivedTuples++
12:         if tuple ∉ (sendQueue ∪ sendQueueOther) then
13:             sendQueueOther.addAtRandomPosition(tuple)
14:         end if
15:     until no tuple is received for time t > abort time
16: end procedure
17:
18: procedure SENDTUPLE(tuple)
19:     sentTuples++
20:     markTupleAsSentInItsQueue(tuple)
21:     sendMessage(tuple, computeRecipient(tuple))
22:     while (sentTuples - receivedTuples ) > threshold do
23:         wait() for receivedTuples increase
24:     end while
25: end procedure
26:
27: procedure SENDTHREAD.RUN()
28:     for random(1 . . . ..n) times do
29:         tuple = SendQueue.getUnmarkedDummyTuple()
30:         SENDTUPLE(tuple)
31:     end for
32:     while ∃ unmarkedTuple ∈ sendQueue do
33:         if random(1 . . . #participants) == j) then
34:             SENDTUPLE(sendQueue.getUnmarkedTuple())
35:         else
36:             tuple = sendQueueOther.getUnmarkedTuple()
37:             if tuple ≠ NULL then SENDTUPLE(tuple) end if
38:         end if
39:     end while
40:     repeat                                  ▷ forward data
41:         tuple = sendQueueOther.getUnmarkedTuple()
42:         if tuple ≠ NULL then SENDTUPLE(tuple) end if
43:     until no tuple is received for time t > abort time
44: end procedure
```

without having received other participants' tuples. Since this would give malicious participants the possibility to draw conclusions regarding the data origin, our algorithm prevents this behavior.

After the dummy tuples have been sent (line 28 - 31) and while the sendQueue contains tuples that have not been sent yet, i.e. unmarked tuples (line 32), a random number is drawn (line 33), and used to determine whether an own tuple or a tuple of a different participant is sent next. With a chance of $\frac{1}{\#\text{participants}}$, an unmarked data tuple of the sendQueue is sent and marked as sent by the procedure sendTuple (line 34). Otherwise, a tuple of the queue sendQueueOther is taken and sent (line 36). When the while-loop is completed, the sendQueue does not contain unmarked items anymore, but the sendQueueOther may contain new and unmarked data tuples, and tuples can be received later. These tuples must also be forwarded to the other participants (line 40 - 43). In order to avoid that participants can draw conclusions on the number of provided data tuples, our algorithm prevents the notification of other participants when the sendQueue is empty. Thus, the algorithm terminates when for a sufficiently long time no data tuple

has been received.

Since each participant sends by chance an own tuple or somebody else's tuple, a participant P_j cannot definitely determine whether the received tuple originates from the previous participant of the permutation cycle, or from another participant of the permutation cycle. Furthermore, the tuple owner may appear at any position within the permutation cycle.

C. Verification Phase

Algorithm 4 is used for verifying that no participant cheated in Algorithm 3, e.g. by suppressing or manipulating another participant's data tuples. If no participant cheated, all participants have the same data tuples and the contributed data of each participant, which can be found in the sendQueue, is found in the resulting union set D (line 1). After that, each participant sorts the received data lexicographically (line 2), and generates a hash value by applying a hash function (line 3) on which all participants agreed prior to protocol execution. To prove that a participant has not cheated, it cuts out a participant specific substring of the complete hash value (line 4) and broadcasts it (line 5). After receiving all other participant's substrings (line 6), a comparison between the concatenated received hash-substrings (line 7) and the self-generated hash value is done (line 8). If no participant has cheated, these values correspond to each other.

Algorithm 4 Verification Phase for Participant P_j and a Set D of All Received Data Tuples

```
1: if D does not contain all tuples of the sendQueue then error
2: D = sortLexicographically(D)
3: hash = hash(D)
4: proof = hash.substr( fromPos: ((j − 1) · ⌈ |hash| / #participants ⌉) + 1,
     toPos: j · ⌈ |hash| / #participants ⌉)
5: broadcast(proof)
6: while not received all other proofs do wait()
7: completeProof = proofOf(P1)+proofOf(P2)+...+proofOf(Pn)
8: if completeProof == hash then ok else error
```

Example IV.1 *The following example demonstrates the concept of the verification:*

$$\textit{Hash} = \underbrace{\texttt{l9j3sflkj2}}_{P_1}\underbrace{\texttt{ki87fre3nj}}_{P_2}\ldots\underbrace{\texttt{8jht42dr4s}}_{P_n}$$

Participant P_1 broadcasts the first part of the union's hash value, P_2 the second part, etc.

D. Decryption Phase

The Decryption Phase (Algorithm 5) is started only if the prior Verification Phase has been successfully completed.

The participant P_1 is the first to decrypt the complete union and send it to P_2 (line 3). P_i with $i > 1$ then receives all data tuples of the union (line 5), and checks whether P_{i-1} decrypted correctly (line 6) by encrypting the received plain text with the public keys of all participants that decrypted before, and comparing whether

Algorithm 5 Decryption Phase for Participant P_j and Union D

```
 1: procedure DECRYPTION
 2:     if j=1 then
 3:         sendData(decrypt(D), To: P2)
 4:     else
 5:         receivedUnion = receiveTuples(from: Pj−1)
 6:         CHECKDECRYPTION(receivedUnion, j − 1)
 7:         sendData(decrypt(receivedUnion), To: Pj+1)
 8:     end if
 9:     receivedUnionPlain = receiveTuples(from: Pj−1)
10:     CHECKDECRYPTION(receivedUnionPlain, j)
11:     sendTuples(receivedUnionPlain, To: Pj+1)
12: end procedure
13:
14: procedure CHECKDECRYPTION(ReceivedTuples, k)
15:     decryptCheck = ReceivedTuples
16:     for i=1 to k do
17:         encrypt each j ∈ decryptCheck with publicKey(Pi)
18:     end for
19:     if not decryptCheck equals (D) then error
20: end procedure
```

the encrypted values correspond to those values that P_i received (line 14 - 20). If this test is successful, P_i decrypts the tuples, i.e. one more layer of the encryption is decrypted, and sends them to the next participant P_{i+1} (line 7), where P_n sends to P_1. After one cycle, participant P_1 will be the first that receives the plain text and executes line 9. Again, P_1, checks for correct decryption and forwards the plain text of the union to the next participant. In the end, all participants have learned the union.

If one of the CHECKDECRYPTION calls (line 6 and 10) is not successful, the participant that detects the cheating will announce that the previous participant P_{j-1} has not decrypted correctly and has cheated, and thus P_{j-1} is excluded in a second run of the Union Protocol. If P_j wrongfully accused P_{j-1} of cheating, P_{j-1} itself can broadcast the decryption to prove that it acted fair.

To ensure that the tuples of the fair participants can still be decrypted but not changed anymore like it would be possible if the complete algorithm would be restarted, the following optional check of Algorithm 6 can be included after the restarted protocol has finished.

Algorithm 6 Optional Tuple Check for Restarted Algorithm After Excluding a Cheating Partner P_c

```
1: procedure CHECK(plainUnionNew, encryptedUnionOld)
2:     return {t ∈ plainUnionRestart |
3:             encrypt(t, keys(P1 ... Pn)) ∈ EncryptedUnionOld}
4: end procedure
```

Algorithm 6 prevents participants from adding data to the union when the algorithm is restarted and the participants may know parts of the union as follows. First, it restarts Algorithms 1 to 5 of the protocol and encrypts the plain union data received by this repeated protocol execution with the public keys of all participants including the cheaters (line 3). Second, it selects only those tuples of plainUnionRestart that have been in the encrypted union encryptedUnionOld of the first protocol run (line 2-3). Tuples of plainUnionRestart that have no corresponding encryption in encryptedUnionOld

have been added in the second execution of the union algorithm, and should not be considered. Encrypted tuples of encryptedUnionOld that have no corresponding plain text in the union plainUnionRestart of the restarted protocol are either tuples of the cheating participant P_c, or the owner of the data deleted its own data. However, a cheating participant cannot prevent that the data of another fair playing participant P_j is excluded from the union if the fair playing participant P_j wants these data to be considered within the union.

The complete protocol can be found in Algorithm 7. Depending on whether the set union or the bag union must be computed, Algorithm 1 or Algorithm 2 is executed for initialization. After the successful execution of Algorithm 3, each participant owns the encrypted union (line 6). Only when the Verification Phase is successful, tuples are decrypted by Algorithm 5 (line 8). If cheating is detected within this phase, the cheater is expelled, and the complete protocol is restarted (line 12). However, since some participants may already know the encrypted union, only those tuples that have been in the initial union are considered (line 13 - 14).

Algorithm 7 Protocol Overview

```
 1: if SetUnion then
 2:     sendQueue = Algorithm 1
 3: else
 4:     sendQueue = Algorithm 2
 5: end if
 6: EncryptedUnion = Algorithm 3 (sendQueue)
 7: if Algorithm 4 (sendQueue, EncryptedUnion) is ok then
 8:     PlainUnion = Algorithm 5 (Union)
 9:     if Algorithm 5 is ok then
10:         return plainUnion
11:     else
12:         plainUnionRestart = Algorithm 7(w/o cheater)
13:         return Algorithm 6 (plainUnionRestart,
14:                                              EncryptedUnion)
15:     end if
16: end if
```

V. CORRECTNESS

We define the behavior of several adversary models and argue why these malicious attacking models do not harm our protocol. In the following, we call the cyclic exchange phase of our protocol, i.e. Algorithm 3, "Phase 1", and the decryption phase, i.e. Algorithm 5 and Algorithm 6, "Phase 2".

The observer inspects all messages, but does not contribute any data. He wants to learn the union and the data that is contributed by certain participants.

The phase 1 deleter deletes some messages that he is requires to route in the cyclic exchange phase in order to reduce the number of data items in the union.

The phase 1 modifier modifies data within the cyclic exchange phase without knowing what he is actually modifying.

The phase 2 deleter stops the protocol in phase 2 after he learned about data items.

The phase 2 modifier tries to modify the message in phase 2 after he learned about the plain text in order to let other participants see a different union than he sees.

Lemma V.1 *The origination of an encrypted data tuple d containing relevant information is disguised during and after the execution of Algorithm 3 if participants do not form cheating groups.*

Proof: The first data tuple that is received by a participant P_j is a dummy tuple, this does not violate any privacy. The second tuple that is received may originate, with a chance of $\frac{1}{n}$, by P_{previous}. The chance that this tuple originates from a different participant is therefore $\frac{n-1}{n}$. A participant that receives a data tuple can therefore not exactly know where a certain tuple originated. ■

Even if participants group together in order to cheat, and monitor the incoming and outgoing data of a certain participant P_j in order to get to know all data tuples originating at P_j, *all* remaining participants must group together to learn of P_j's data since P_j may send encrypted messages to all participants depending on the permutation. This, however, would also be possible when all participants but P_j run the union algorithm a second time on the data of $\{P_1 \dots P_n\} \backslash P_j$ and compare with the union $\{P_1 \dots P_n\}$ to which P_j contributed its data. Therefore, this kind of deanonymizing that appears when participants group together is inevitable for the union operator.

Lemma V.2 *For set union computation, the number of participants owning a certain data tuple d is disguised during and after the execution of Algorithm 3.*

Proof: Assume, two or more participants P_j and P_k own the data tuple d. The encryption of d results in the same encrypted tuple d_e and in the same permutation of participants, i.e. the same cycle where the tuple is sent. Let P_j be the first participant who sends the tuple d_e. Each participant that receives the tuple and has not seen d_e before, forwards d_e some time when the random function chooses d_e. However, a participant P_k that owns d_e as well will not add the received tuple d_e to any send queue, since P_k's own data item – which is equal to d_e – is already in the sendQueue or has already been sent. Therefore, a tuple d_e is only sent *once* by each participant, no matter how many participants own the tuple. ■

The only information that may be leaked is that a participant P_j that has a tuple d may get to know that some other participant P_k has the tuple as well, but it will not know how many other participants also have d.

Lemma V.1 and V.2 ensure that **the observer** cannot identify the data tuples contributed by a certain participant. However, the observer's behavior to reject to contribute any own data except dummy tuples is tolerated by the anonymity assumption.

Lemma V.3 *After the successful execution of Algorithm 4, (a) each participant's encrypted data is found in the resulting union set, and (b) each participant received the same data.*

Proof: If some participant would not find its data in the union, it would stop the algorithm and never decrypt something. In this case, Algorithm 4 would not be successful, therefore (a) is fulfilled.

(b) is fulfilled since each participant contributes a part of the union's hash value. If a participant P_j would have a different union set, the resulting hash value and corresponding substring that P_j calculated would extremely unlikely match the concatenated hash value calculated by another participant P_i. Since different hash values indicate that cheating occurred, the protocol would stop without revealing any plain text data. ■

Lemma V.3 ensures that the occurrence of a **phase 1 deleter** and **phase 1 modifier** is detected in the verification phase, and thus no tuple will be decrypted.

Lemma V.4 *If a participant P_c stops protocol execution in Phase 2, i.e. after the Verification Phase was successful, all tuples (except the tuples of P_c) are decryptable, and no further tuples can be provided anymore.*

Proof: If a participant P_c cheats and stops the protocol during the Decryption Phase, P_c is identified and the complete protocol is started again without P_c. To ensure that the other participants' data is still decryptable, i.e. the union verified in algorithm 4 can be decrypted (except the tuples of P_c), only those tuples that can be found in the first *and* in the second run of the protocol are considered (Algorithm 7, line 12 - 14). This ensures that in the second run, participants cannot change their own tuples or add new tuples. ■

Lemma V.4 ensures that the **phase 2 deleter** cannot harm the data of other participants. A refused decryption of the phase 2 deleter does not lead to data loss since each participant can identify its own data. Thus, the restart of the protocol having the deleter expelled will reveal the same data excluding the data of the phase 2 deleter. Although the phase 2 deleter may have learned the union in this case, the behavior of the phase 2 deleter leads to the same result as the observer's behavior does.

The **phase 2 modifier** will be identified by the next participant in the decryption phase, since the proof of correct decryption will fail (Algorithm 5, line 5).

If a participant combines multiple attacks, the first of its attacks could be detected as described, and the attacking participant is excluded.

VI. Experimental Results

We have implemented our protocol as a prototype in order to evaluate what impact the size of the union and the number of participants have on the exchange speed.

We have used Java as programming language and the Bouncycastle Cryptographic Provider [9] to encrypt and decrypt each tuple by using RSA encryption. Furthermore, we have used SHA-1 as hash function for verifying that each participant has the same data.

To generate the test set, we have extracted a part of the Paderborn telephone directory containing about 6,000 entries, each entry containing between 150 and 300 bytes.

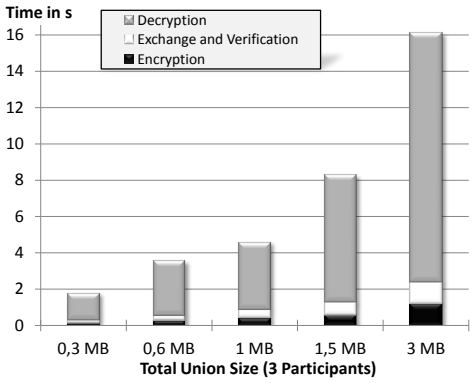

Figure 1. Varying union size and constant number of 3 participants. ©2007 IEEE

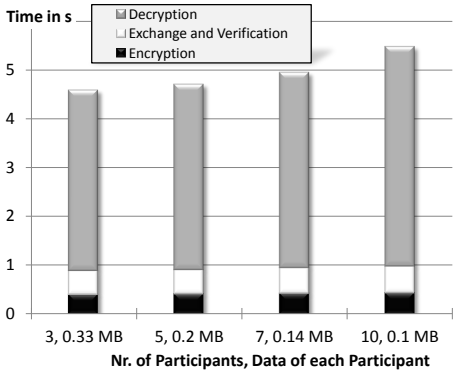

Figure 2. Varying number of participants and 1MB constant union size ©2007 IEEE

In total, the set has a size of approximately 1MB. When we vary the number of participants, we split this data into n parts of equal size, so each participant owns $\frac{1}{n}$th of the 1MB union. The experiments have been performed on Intel Core2Duo 6700@2,66Ghz machines with 4 GB RAM and have been repeated 20 times to get the average execution times.

Figure 1 shows the time needed for each phase when we vary the total size of the union, but hold the number of participants constant to three. The decryption phase takes very long compared to the other phases because it uses the RSA algorithm, which is known to be time consuming. However, our investigation focuses on the impact of an increasing union size and of an increasing number of participants on the total time. Our solution does not rely on RSA, therefore, cryptographic public-key algorithms like elliptic curve algorithms [10] that are known to be faster can speed up the encryption and decryption phase. To conclude this experiment, we can see that the time needed is linear to the union size.

Figure 2 shows the time that is needed for each phase of the algorithm execution having a constant union size but a varying number of participants. This means, for an increasing number of participants, the amount of data each participant contributes decreases. We can see that the total time of the algorithm execution is almost independent of the number of participants. This can be explained as follows: Each participant receives and sends only the amount of data of the resulting union size. Having

an increasing number of participants in total, a single participant still sends each tuple only once to the next calculated participant. This means, the parallelism in the exchange phase increases to the same extend as the overall data transfer.

To summarize the experiments, we have seen that the exchange time is almost independent of the number of participants, and grows linearly to the total union size.

VII. Related Work

Several multiparty computation approaches have been proposed in order to implement database operators that reveal no more than the result, e.g. [1]–[7]. These approaches are called *sovereign information sharing*, since each participant will not reveal more data than shown in the query result. However, current approaches have the shortcoming that participants must act fair and are not assumed to change the protocol. For example, algorithms for computing the intersection of two databases [1]–[4] reveal the intersection only to a *single* participant and assume that this participant will send the intersection to the second partner as well. Our approach focuses on participants who may try to cheat and stop the protocol at an arbitrary point in time, e.g., after they received the union of the data. Thus, our assumed participant behavior follows [11], [12], which propose a solution for intersection computation that reduces the damage in case that participants cheat or stop the protocol.

[7], [13] propose a special co-processor hardware for the encryption and the calculation of sovereign joins. However, special purpose hardware must be bought and participants have to trust the manufacturer of the hardware chip.

[14] proposes a technique for privacy preserving query processing using third parties, but does not consider malicious partners. The use of third parties, which is also proposed by [15], [16], involves the risk that third parties behave malicious in terms of deleting data or forming cheating groups, which may be the case if control over third parties is reached by a computer virus or trojan horse.

In addition, approaches tackling the sovereign information sharing problem do not consider data anonymity, i.e. to keep the origin of each data tuple secret. When the union operator must be applied, this property is especially relevant in order to preserve the origin of the data.

Data anonymity for voting and election mechanisms is also a topic in the cryptographic community, e.g. [17]–[20]. However, these mechanisms are proposed for counting the number of votes for a previously fixed set of possible candidates, and not for the database union operator. [19], e.g., proposes a voting protocol in combination with a broadcast protocol. The approach is comparable with a physical black box containing padlocks. Every time a participant casts his vote, he removes a padlock until in the end all votes are visible to the last voter, who passes the result in addition to a voter's proof to each participant. However, this protocol relies on the fact that the last participant must play fair. If he dislikes the result, he could stop the protocol execution and prevent all other participants from learning the result. To solve this problem, [18] proposes a special trusted *voting authority* that always plays fair, which is in fact a trusted-third party. In comparison, our protocol works correctly without such a trusted authority.

If an untrusted third party is available, [21], [22] show how to use this untrusted third-party for fair data exchange. A fair exchange can be guaranteed for items belonging to the categories revocable or generatable. However, since database content is information that is in many cases neither revocable nor generatable, the proposal of [21], [22] to revoke the information does not work. In contrast, our protocol does not rely on a certain item category; it is useful for non-revocable and non-generatable items as well.

Other cryptographic approaches focus on *receipt-freeness*, which means that a voter must not be able to prove that a particular vote was casted by him in order to prevent "vote buying" or vote extortion [20], [23]. Therefore, vote mechanisms should have no means to construct a receipt that allows a participant to prove the content of his vote to a third party. However, within our scenario, participants may act malicious and change other participants' data if they get in contact with them. Thus, each participant must be able to stop the complete protocol in case his encrypted data was changed or deleted before it is possible to decrypt the first vote.

To guarantee anonymity, *mix-networks* have been proposed by [24], [25]. However, the voting authority and the mix-networks could manipulate by aborting the complete voting process in case of an unpopular result. Additionally, the authority may publish a "trend" before it stops the protocol in order to make participants change their vote in a renewed voting when the participants know such a trend. Furthermore, when practical applications within business environments come into consideration, third parties are often rejected due to political reasons. In comparison, our approach does not need any third party authority and is also usable for sets with previously unknown data.

Our approach computes the union of different participants' databases and guarantees that size information like $|D_j|$ is concealed. It also guarantees that a participant P_k is not able to conclude which other databases P_j contain a certain data tuple d of the computed union in scenarios with more than two independent participants.

To summarize, our protocol does not rely on third parties and it allows to detect whether participants act malicious, i.e. whether a participant changes the protocol and modifies messages. If a participant stops the protocol, either no tuple of the union is revealed, or the tuples of the fair playing participants may still be revealable, but not changeable. This ensures that malicious participants cannot prevent the union from containing other participants' data.

VIII. Summary and Conclusion

Union computation among multiple participants that may behave malicious involves two challenges that have been addressed in this article: data anonymity and malicious participant behavior.

We have presented both, a set union and a bag union computation protocol, both of which preserve data tuple anonymity and detect cheating of participants. Furthermore, our approach allows each participant to verify the occurrence of its data within the union before a single data item of the union is revealed in plain text. Our algorithm consists of three phases, namely the exchange phase to guarantee anonymity, the verification phase that detects cheating, and the decryption phase that ensures that all participants get to know the computed union by a verifiable decryption of the unified data in a distributed fashion.

In order to prove the correctness of our protocol, we have defined several adversary models, and have shown that our protocol is resilient against these kinds of attacks.

Our experimental results have shown that the execution time of the algorithm mostly depends on the size of the union. Since increasing union sizes mean increasing gain for each individual participant, the additional time for growing union sizes is still reasonable and linearly scaling, like the experimental results pointed out.

Thus, our contribution is useful for many problems that require a union computation of different data sources, and must rely on both data anonymity and security.

References

[1] R. Agrawal, A. Evfimievski, and R. Srikant, "Information sharing across private databases," in *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM Press, 2003, pp. 86–97.

[2] C. Clifton, M. Kantarcioglu, X. Lin, J. Vaidya, and M. Zhu, "Tools for privacy preserving distributed data mining," 2003.

[3] M. Freedman, K. Nissim, and B. Pinkas, "Efficient private matching and set intersection," in *Advances in Cryptology — EUROCRYPT 2004.*, 2004.

[4] B. A. Huberman, M. Franklin, and T. Hogg, "Enhancing privacy and trust in electronic communities," in *ACM Conference on Electronic Commerce*, 1999, pp. 78–86.

[5] L. Kissner and D. X. Song, "Privacy-preserving set operations." in *Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference*, 2005, pp. 241–257.

[6] R. Agrawal and E. Terzi, "On honesty in sovereign information sharing." in *10th International Conference on Extending Database Technology*, Munich, Germany, 2006, pp. 240–256.

[7] R. Agrawal, D. Asonov, M. Kantarcioglu, and Y. Li, "Sovereign joins," in *Proceedings of the 22nd International Conference on Data Engineering, Atlanta, USA*, 2006.

[8] S. Böttcher and S. Obermeier, "Secure anonymous union computation among malicious partners," in *ARES '07: Proceedings of the The Second International Conference on Availability, Reliability and Security*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 128–138.

[9] "Bouncy castle open source cryptographic package," http://www.bouncycastle.org, 2006.

[10] V. S. Miller, "Use of elliptic curves in cryptography," in *Lecture notes in computer sciences; 218 on Advances in cryptology—CRYPTO 85*. New York, NY, USA: Springer-Verlag New York, Inc., 1986, pp. 417–426.

[11] S. Böttcher and S. Obermeier, "Sovereign information sharing among malicious partners." in *Secure Data Management*, ser. Lecture Notes in Computer Science, W. Jonker and M. Petkovic, Eds., vol. 4165. Springer, 2006, pp. 18–29.

[12] S. Obermeier and S. Böttcher, "Secure computation of common data among malicious partners," in *International Conference on Security and Cryptography (Secrypt), Barcelona, Spain*, 2007.

[13] S. W. Smith and D. Safford, "Practical server privacy with secure coprocessors," *IBM Syst. J.*, vol. 40, no. 3, pp. 683–695, 2001.

[14] F. Emekci, D. Agrawal, A. E. Abbadi, and A. Gulbeden, "Privacy preserving query processing using third parties," in *Proceedings of the 22nd International Conference on Data Engineering, ICDE, Atlanta, USA*, 2006.

[15] S. Ajmani, R. Morris, and B. Liskov, "A trusted third-party computation service," MIT, Tech. Rep. MIT-LCS-TR-847, 2001.

[16] N. Jefferies, C. J. Mitchell, and M. Walker, "A proposed architecture for trusted third party services," in *Cryptography: Policy and Algorithms*, 1995, pp. 98–104.

[17] J. D. Cohen and M. J. Fischer, "A robust and verifiable cryptographically secure election scheme," in *FOCS85*. Portland: IEEE, 1985, pp. 372–382.

[18] A. Kiayias and M. Yung, "Self-tallying elections and perfect ballot secrecy," in *Public Key Cryptography — 5th International Workshop on Practice and Theory in Public Key Cryptosystems*, 2002, pp. 141–158.

[19] J. Groth, "Efficient maximal privacy in boardroom voting and anonymous broadcast," in *Financial Cryptography, LNCS 3110*, 2004, pp. 90–104.

[20] J. Benaloh and D. Tuinstra, "Receipt-free secret-ballot elections (extended abstract)," in *Proceedings of the 26th annual ACM symposium on Theory of computing*. New York, NY, USA: ACM Press, 1994, pp. 544–553.

[21] M. K. Franklin and M. K. Reiter, "Fair exchange with a semi-trusted third party (extended abstract)," in *ACM Conference on Computer and Communications Security*, 1997, pp. 1–5.

[22] N. Asokan, M. Schunter, and M. Waidner, "Optimistic protocols for fair exchange," in *Proceedings of the 4th ACM conference on Computer and communications security*. New York, NY, USA: ACM Press, 1997, pp. 7–17.

[23] M. Hirt and K. Sako, "Efficient receipt-free voting based on homomorphic encryption," in *Eurocrypt*, ser. Lecture Notes in Computer Science, vol. 1807, 2000.

[24] M. Abe, "Mix-networks on permutation networks," in *ASIACRYPT '99: Proceedings of the International Conference on the Theory and Applications of Cryptology and Information Security*, 1999, pp. 258–273.

[25] Y. Desmedt and K. Kurosawa, "How to break a practical MIX and design a new one," *Lecture Notes in Computer Science*, vol. 1807, 2000.

Stefan Böttcher is a professor of computer science at the University of Paderborn, Germany. He works in the areas of databases, XML, query optimization, mobile transactions, access control, security and privacy.

Sebastian Obermeier received his Diploma in computer science from the University of Paderborn, Germany, in 2005.

He is currently a doctorate candidate at the University of Paderborn and a fellow of the International Graduate School "Dynamic Intelligent Systems". His research interests include database security and privacy, as well as distributed transaction execution within mobile ad-hoc networks.