Privacy Preserving Cloud Transactions

Debmalya Biswas[1] and Krishnamurthy Vidyasankar[2]

[1] Nokia Research Center, Lausanne, Switzerland
debmalya.biswas@nokia.com
[2] Dept. of Computer Science, Memorial University, St. John's, NL, Canada
vidya@mun.ca

Abstract. Cloud computing as an outsourced data storage platform is becoming popular by the day with multiple service offerings by companies such as Amazon, Microsoft, among others. However to gain widespread enterprise level acceptance, cloud providers still need to address the following significant challenges: (a) Data confidentiality: Organizations do not trust cloud providers with their confidential data. One way of ensuring data confidentiality is of course to store data in encrypted form. This in turn requires that data be decrypted before each access. Further, decryption should be enabled only to authorized users. (b) Transactions: Consistent, concurrent and reliable access to data should be provided via transactions, as with traditional databases. (c) User confidentiality: It should be possible to keep the identity of the users executing transactions confidential from the 'owners' of data and vice versa.

In this work, we aim to integrate the above objectives providing transactional guarantees over encrypted outsourced data. More specifically, we propose concurrency control protocols for the different cloud sharing configurations, with different confidentiality requirements. Experimental results are given to validate the scalability of the proposed protocols.

1 Introduction

Cloud computing storage platforms are rapidly gaining widespread acceptance, especially among the small/medium range business organizations. To further gain enterprise level acceptance and completely eliminate the need for local databases, the following properties still clearly need to be provided:

- Data confidentiality: Organizations do not trust cloud providers with their confidential data. So any data stored in the cloud ideally needs to be in encrypted form.
- Transactions: Consistent, concurrent and reliable access to data should be provided via transactions, as with traditional databases.
- User confidentiality: It should be possible to keep the identity of the users executing transactions confidential from the 'owners' of data and vice versa.

In general, performing operations on encrypted data is much more expensive than doing so on cleartext data. This is due to the performance overhead of

R. Meersman et al. (Eds.): OTM 2012, Part II, LNCS 7566, pp. 511–526, 2012.

having to encrypt (decrypt) data before storing (accessing) any data. On the same lines, providing transactions in a cloud computing environment is still an open research problem, especially so on encrypted data. Our objective in this work is thus to explore approaches providing scalable transactions on encrypted outsourced data.

In a cloud computing environment, different data sharing configurations are possible based on the capabilities/requirements of the participating organizations. We consider a distributed scenario with multiple clients sharing data via the cloud, with portions of the shared data owned by the different clients. The clients may also cache some data at their local sites, so some replication is also possible. Given this, different parameters can be considered sensitive depending on the level of confidentiality/anonymity required for a configuration. The parameters include:

- Data values are clearly the most sensitive, and at the very least need to be protected from the cloud provider.
- Data owners may restrict read/update access of their data items to specific clients.
- Data owners may wish to be anonymous to the transaction owners (the clients initiating transactions to process data), and vice versa. So there might be a need to protect the identity of data/transaction owners.
- Finally, the data access permissions themselves may be sensitive, e.g. a client requesting permission to process multiple data items (within a transaction) need only be aware of the overall decision; no need to know the response (accept/reject) of each individual data owner.

The system model including possible configurations is formalized in Section 4. *The main contribution of this work is to provide transactional guarantees for the different cloud data sharing configurations* (Section 5). In Section 5.2, we consider the scenario where only 'update' permissions need to be requested from the data owners. In this context, we give concurrency control protocols for the different confidentiality requirements:

- starting with the base protocol where only the data values need to be protected,
- then adding the requirement of keeping the data owners confidential as well, and
- finally the scenario where the responses to requests for permissions also need to be kept confidential.

The scenario is then extended to also accommodate 'read' restrictions in Section 5.3. Solutions for the different confidentiality requirements integrate a host of encryption primitives including symmetric encryption, proxy re-encryption and homomorphic encryption (introduced briefly in Section 3.2). We give experimental results in Section 6 to study the effect of encryption primitives in the protocols on overall performance and abort rates. Related work is discussed in Section 2. Finally, Section 7 concludes the paper and provides some directions for future work.

2 Related Works

Transactions [1] have a long established history of providing reliability and concurrent access to distributed data storage systems. Some works which have considered transactions in a cloud computing environment include [2,3]. However, providing transactions on encrypted data (as is a growing requirement with more and more data stored on third party untrusted cloud servers) is clearly a novel aspect of this work.

The closest and only known works (to the best of our knowledge) related to "transactions on encrypted data" are [4,5]. [4] considers a slightly different use-case where the same data is replicated (cached) among multiple clients, and the cloud server is only responsible for providing transactional guarantees based on stored transaction logs. It can thus be considered as solving a sub-problem of the set of problems we wish to address in this work. [5] considers the integrity of lock-based concurrency controls provided by untrusted cloud providers. The challenge is that in a centralized locking protocol, a malicious cloud provider may re-arrange the transaction order favoring some clients to maintain its QoS or for other monetary gains. The problem is implicitly overcome in our protocols as we consider (non-lock based) optimistic concurrency controls where conflict detection is performed in a distributed fashion involving the data owners (i.e. the conflict detection decision is not taken independently by the cloud provider).

From a privacy/security perspective, previous works [6,7] have considered privacy preserving approaches to outsource data storage to third party providers. However [6,7] mainly focus on searching over outsourced encrypted data, while our focus is on providing reliability/consistency guarantees over encrypted data.

3 Preliminaries

3.1 Transactions

Transactions [1] allow grouping a sequence of operations within Begin and Commit/Abort operations with ACID (Atomicity, Consistency, Isolation, Durability) properties.

- Atomicity: ensures that all data operations in a transaction are either performed in their entirety or none at all. In the event of a failure during execution, the transaction is aborted leading to rollback of all uncommitted changes (if any).
- Isolation: To improve performance, often several transactions are executed concurrently. Isolation necessitates that the effects of such concurrent execution are equivalent to that of a serial execution [8]. This property is provided by the concurrency control protocols.
- Consistency: Each transaction moves the system from one consistent state to another. Atomicity and isolation together ensure consistency.
- Durability: The effects of the committed transactions should survive any system failures.

We focus on the A, C and I properties here. Durability is assumed to be guaranteed by the storage medium. In a distributed setting, with a transaction processing data belonging to different clients, there might be a need to get feedback from multiple clients before the transaction can be committed. This is achieved using distributed commit protocols, such as the commercially available 2 Phase Commit (2PC) protocol, or more advanced ones such as [9,10].

3.2 Cryptographic Primitives

In this section, we give a brief introduction to the cryptographic primitives used in this paper.

Symmetric encryption refers to a shared secret key among multiple parties, which is used for both encryption and decryption. In practice, the encryption and decryption keys are either identical, or a simple transformation between the two keys. Let sym_k be the shared symmetric key. Then, $v = DEC(sym_k, ENC(sym_k, v))$ denotes the corresponding encryption/decryption operations of a data value v.

Proxy re-encryption [11] allows a ciphertext for A to be re-encrypted into a ciphertext for B (can be decrypted using B's secret key). The envisioned application of such an encryption scheme is delegation, e.g. an employee can delegate his confidential encrypted emails to his secretary, without any need to forward his secret key. We specify a proxy re-encryption scheme $PRE(KGEN, ENC, PRK, RENC, DEC)$ formally as follows:

- $KGEN(1^k)$ outputs a public-private key pair: (A_{pb}, A_{pr}).
- $ENC(A_{pb}, m)$ outputs c_{A1}, the message m encrypted under public key A_{pb}.
- $PRK(A_{pr}, B_{pb})$ outputs a re-encryption key $rk_{A\to B}$ that allows ciphertexts generated using A's public key to be decrypted by B's private key.
- $RENC(rk_{A\to B}, c_{A1})$ outputs the ciphertext c_{B2} generated by re-encrypting c_{A1} under $rk_{A\to B}$.
- $DEC(B_{pr}, c_{B2})$ decrypts c_{B2} using B_{pr}, returning the message m.

A construction based on bilinear maps of the above scheme is given in [6].

Homomorphic encryption schemes allow arithmetic operations to be performed locally on the plaintext values, based on their encrypted values (ciphertext). A homomorphic encryption scheme $HE = (ENC_H, DEC_H)$ satisfies the following property for two positive integers a and b:

$$DEC_H(ENC_H(a) \times ENC_H(b)) = a + b$$

The homomorphic encryption scheme is public-key, i.e. any party can perform the encryption operation $ENC()$ (by itself). A construction of such a scheme is given in [12].

4 System Model

We consider the following distributed storage system. The database consists of a sequence of $(location, value)$ pairs, denoted (l, v). Such data can be processed

using the following operations: We assume that all locations are initialized to 'null'.

- select$(l_1, \cdots, l_n)$ returns the corresponding values $v_1, \cdots, v_n$ of the location parameters.
- delete$(l_1, \cdots, l_n)$ sets the values of locations $(l_1, \cdots, l_n)$ to 'null'.
- update$((l_1, v_1), \cdots, (l_n, v_n))$ has the semantics that the values of locations $(l_1, \cdots, l_n)$ are updated to $(v_1, \cdots, v_n)$ respectively.
- insert$((l_1, v_1), \cdots, (l_n, v_n))$ is similar to update, except that it would be updating 'null' to the specified values $(v_1, \cdots, v_n)$.

A transaction t_i is a grouping of the data manipulation operations defined above. We characterize transactions by their read and write sets, where read set r_i is a union of the location parameters in the select operations in transaction t_i. On the same lines, write set w_i is a union of the location parameters in the insert, delete and update operations in transaction t_i.

The system consists of the following entities (illustrated in Fig. 1):

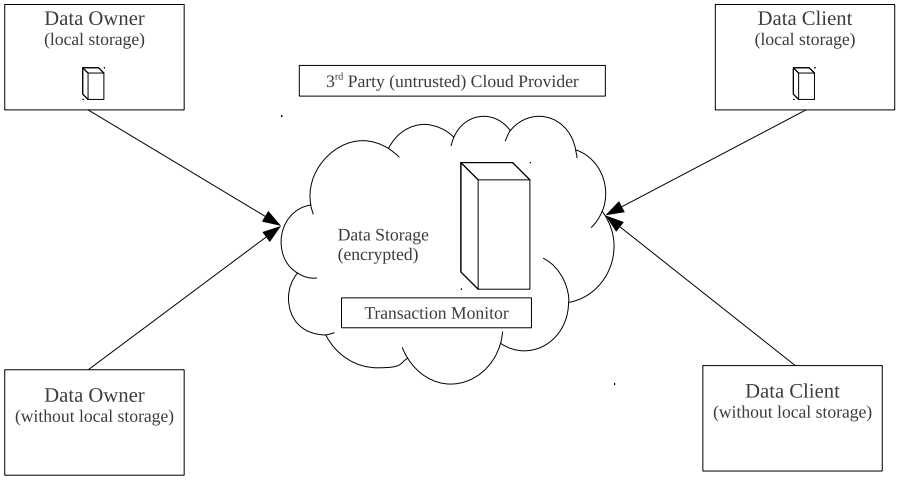

Fig. 1. System architecture

- Clients: Clients act as data processing clients capable of executing transactions. Depending on the scenario, clients can store the whole database locally, only the data they own, or just act as "dumb" terminals downloading the needed data from a remote server/client before executing each operation, etc. Broadly, we consider both scenarios where clients act as peers (with equal data processing capabilities) and also scenarios where some clients are mainly responsible for creating/controlling data and others with querying data.

  Clients are denoted as $C_1, C_2, \cdots$. For a transaction t_i, the client C_j initiating t_i is also referred to as the transaction owner $C_j = C(t_i)$.

– Server: We refer to the third party cloud storage provider as the server S. The data storage capabilities and requirements of the server vary from keeping a copy of the whole database to only the transaction log. In addition, the server also acts as a global co-coordinator for distributed protocols such as concurrency control, commit, etc.

For such a system architecture, it is clear that many scenarios are possible depending on which data is stored where, the capabilities of the clients, the concurrency protocol used (pessimistic/optimistic), etc. A specific (and simplified) use-case scenario of the above system configuration is when users would like to share their personal data (generated on their phones/computers) with advertisers. Here users are the data owners who only perform update operations (upload their data). Advertisers on the other hand are data clients restricted to performing only select queries on the database. The cloud provider acts as the intermediate broker facilitating storage and control, without itself having access to data.

From a privacy perspective as well, different parameters can be considered sensitive depending on the level of security required for a configuration, the clients themselves may wish to remain anonymous to others, and so on. The parameters include:

– Data values: are clearly the most sensitive and need to be protected from the untrusted third party storage provider S as the minimum required level of security. In the scenario the entire database is not accessible to all clients, we also differentiate between the client who owns data at location l_i (referred to as the data owner $C_j = O(l_i)$), and the clients who can read and update location l_i. Let $\mathcal{C}_r(l_i)$ and $\mathcal{C}_w(l_i)$ refer to the set of possibly overlapping set of clients capable of reading and updating the data at location l_i, respectively. The sets $\mathcal{C}_r(l_i)$ and $\mathcal{C}_w(l_i)$ are defined and maintained by the client $O(l_i)$.
– Transaction and data owners: Transaction owners, i.e. the client $C_i = O(t_j)$ initiating a transaction t_j may wish to be kept anonymous from the owners $O(l_k)$ of locations $l_k \in (r_j \cup w_j)$, and vice versa.
– Decision parameters: Parameters which might need to be protected also include the decisions taken in a distributed commit/concurrency control protocol, e.g. the fact that client C_i decided NOT to commit in the polling phase of a 2PC protocol only needs to be known to the co-coordinator (and not to the other clients $C_j \neq C_i$). We give any additional notations required for the distributed commit/concurrency protocols in the sequel.

5 Privacy Preserving Concurrency Control Protocols

Distributed concurrency control protocols have been widely considered in literature [1]. The novelty here is the additional confidentiality requirements with respect to the different parameters as discussed earlier. We start with a base model (described in [4]) considering the minimum privacy requirements, and incrementally add restrictions outlining the modifications (if any) required to provide transactional guarantees in those scenarios.

5.1 Base Model

Data configuration: All clients $C_i \in \mathcal{C}$ maintain a copy of the whole database with the server S keeping only a log of the transactions (mainly for durability reasons).

Concurrency Control: Any client C_i can execute a transaction. In the pessimistic case, C_i requests S for a lock before initiating a transaction t_j. S processes the transaction requests sequentially, leading to only one client executing a transaction at a time. To improve performance, the authors also consider an optimistic variant where clients first execute the transactions on their local database copies, and then send their read and write sets to the server for conflict detection. If a conflict is detected, C_i needs to rollback any updates performed on its local database copy.

Sync distributed database/propagate updates: On successful commit of t_j, C_i logs details of any updates performed as part of t_j (i.e. w_j entries along with their corresponding values) at S. S is then responsible for propagating the updates to other clients $C_k \neq C_i$ in a lazy fashion.

Privacy: The only privacy requirement here is to keep the transaction data values protected from S. As such, transaction log entries are encrypted using a symmetric key pre-shared among the clients before storing them at S.

In the sequel, we only consider optimistic control protocols as they are more relevant for our use-cases.

5.2 Update Restrictions

Data configuration: All clients can read all data items. The data configuration with respect to the type of data stored by each entity is similar to the base model.

Privacy: Here, in addition to protecting the transaction data values from S, we also consider "update" restrictions, i.e. not all clients can update all data items. Thus, for each location l_i, client $C_j = O(l_i)$ is responsible for regulating the clients $C_k \neq C_j$ capable of updating l_i. Note that the update restrictions considered here are run-time restrictions, i.e. the set of clients that can update a location l is not fixed over time.

Single Confidential Owner. We first consider the simplified scenario where (i) all data items processed as part of a transaction t_i (i.e. $w_i \cup r_i$) are owned by the same client C_O.

Privacy. The additional privacy requirement here is that data owners remain unknown to the transaction owners.

Given this, a privacy preserving optimistic concurrency control protocol satisfying the above requirements is given in Algorithm 1.

Algorithm 1. Single Confidential Owner

1. *Initial setup*: Let sym_k denote the symmetric key shared among all the clients.

2. (Transaction owner) Client C_I executes transaction t_i on its local database, leading to the read and write sets: r_i and w_i respectively.
3. C_I sends the set of location-(encrypted)value pairs $(l_i, c_i = ENC(sym_k, v_i))$ to S, which then forwards them to the owner C_O (with the additional piece of information that C_I is the transaction owner).
4. C_O decrypts the received values $v_i = DEC(sym_k, c_i)$ and performs the following checks:
   - *Permissions*: Check if C_I is allowed to update the locations in w_i.
   - *Conflict detection*: Check if the values v_i corresponding to locations $l_i \in r_i$ sent by C_I are the same as the values of those locations in its database. If yes, then there is no conflict and t_i can be committed. Otherwise, there is a conflict and t_i needs to be aborted. *Note that conflict detection here can be performed based on just the read set values.*
5. Depending on the outcome of the permissions and conflict detection checks above, Commit or Abort t_i can be performed as follows:
   - *Abort*: Transaction t_i can be aborted by a simple response from C_O (relayed via S) notifying the same to C_I, which then rollbacks any updates performed on the locations in w_i from its local database.
   - *Commit*: C_O can commit the transaction t_i on C_I's behalf by:
     (a) updating the locations $l \in w_i$ to the corresponding values v_i (sent by C_I) in its local database.
     (b) notifying the transaction owner C_I and S that transaction t_i has been successfully committed.
     (c) The server S on receiving the commit notification, updates its own local database, and sends the updates lazily to the remaining clients $C_j (\neq C_O$ and $\neq C_I)$. Clearly, sooner the updates reach other clients, less will be the number of aborts due to failed conflict detection checks. This compromise between propagating updates more frequently to increasing number of aborts is studied experimentally in Section 6.
     (d) Each client C_j on receiving the pairs $(l_i \in w_i, c_i)$ from S, decrypts the values $v_i = DEC(sym_k, c_i)$ and updates the corresponding locations l_i in their local databases.

Security Guarantees: Security of the above protocol can be defined in terms of standard semantic security. Data values are protected from the (untrusted) cloud provider S as it lacks keys required to decrypt the encrypted c_i values.

Multiple Confidential Owners. We now consider the extended scenario where data items processed by a transaction t_i are owned by different clients, i.e there exists a pair of locations $l_j, l_k \in (r_i \cup w_i)$ such that $O(l_j) \neq O(l_k)$. We initially consider that the *Permissions/Conflict detection* check outcomes are not confidential, and hence do need to be protected from S. The additional privacy requirements above can be accommodated by modifying Step 5 onwards of the optimistic concurrency control protocol above as given in Algorithm 2.

Algorithm 2. Multiple Confidential Owners

5. Each data owner C_O sends the outcome $d_O = 0(Commit)/1(Abort)$ of its *Permissions/Conflict detection* checks to S (acts as the co-ordinator here).
6. S takes a decision based on the received decision d_O values, and notifies C_I of the commit/abort decision. Depending on the outcome, the following steps need to be undertaken:
   - *Abort*: C_I rollbacks any updates performed on the locations in w_i from its local database.
   - *Commit*: S updates its local database with the location-(encrypted)value pairs (l_i, c_i) received earlier from C_I, and sends the updates lazily to the other clients $C_j \neq C_I$).
   - Each client C_j on receiving the pairs $(l_i \in w_i, c_i)$ from S, decrypts the values $v_i = DEC(sym_k, c_i)$ and updates the corresponding locations l_i in their local databases.

Note that it may still be possible for a data owner C_O to commit t_i on C_I's behalf, if the other involved data owners are allowed to send their outcomes to C_O (assuming the identity of data owners involved in a transaction and their respective outcomes are not confidential among each other).

Confidential Decisions. Finally, we add the privacy requirement that the *Permissions/Conflict detection* check outcomes are also confidential, i.e. which owner decided what also needs to be protected from S and C_I. We accommodate this privacy requirement based on a homomorphic encryption scheme $HE = (ENC_H, DEC_H)$ (introduced in Section 3.2) as given in Algorithm 3. As with Algorithm 2 (to avoid redundancy), Algorithm 3 also outlines the modifications needed Step 5 onwards of Algorithm 1 (i.e. Steps 1-4 follow from Algorithm 1).

Algorithm 3. Multiple Confidential Owners with Confidential Decisions

5. Each data owner C_O sends the outcome $d_O = 0(Commit)/1(Abort)$ of its *Permissions/Conflict detection* checks, encrypted as $c_O = ENC_H(d_O)$ to S.
6. S on receiving the encrypted decision values c_{O_i} from all involved data owners $C_{O_i} | i = 1, \cdots n$, computes $d = DEC_H(c_{O_1} \times \cdots \times c_{O_n})$. If $d = 0$, then S notifies the commit decision to C_I, otherwise $(d > 0)$ abort. The remaining steps to actually commit/abort t_i follow as before.

Security Guarantees: The use of homomorphic encryption ensures that only the combined decision (of all involved owners) becomes known to S, and not the individual replies by each owner.

5.3 Read Restrictions

Privacy: In addition to the update restrictions considered in the previous section, here we also consider read restrictions. This implies that for each location l_i,

client $C_j = O(l_i)$ is also responsible for regulating the clients $C_k \neq C_j$ capable for reading l_i.

Data configuration: A weak interpretation here would be consider the same data configuration as before with all data cached at all clients. With this configuration, imposing read restrictions requires the only additional step of also getting read permissions for all locations $l_i \in r_i$ of transaction t_i from $C_O(l_i)$.

On the other hand, if we assume that the read set can be estimated (at least conservatively) without actually executing the transaction, then we can envision a data configuration where data is only stored at the outsourced (cloud) storage provider S. That is, there is no need anymore for clients C_i (including $C_O = O(l_i)$) to cache l_i.

Collusion: The above data configuration and privacy requirements leads to the additional challenge of preventing *collusion* between S and the clients. With 'read' restrictions, only specific clients can access (read) a location l_i. Given this, and assuming all data is stored as symmetrically encrypted ciphertexts at S, a malicious S can collude with client $C_j \notin \mathcal{C}_r(l_i)$, giving C_j access to the data value at l_i. Thus, symmetric encryption is no longer sufficient in this configuration.

In the sequel, we give a privacy preserving concurrency control protocol based on *proxy re-encryption* to overcome the above challenges. We outline the protocol for the "Single Confidential Owner" scenario, with the extensions to "Multiple Owners" and "Confidential Decisions" scenarios provided as before.

Algorithm 4. Single Confidential Owner with Read Restrictions

1. *Initial setup*: Each client C_i runs $KGEN(1^k)$ to generate its public-private key pair: $(C_{i_{pb}}, C_{i_{pr}})$. Each data owner $C_O = O(l_i)$ also generates a re-encryption key $rk_{C_O \to C_j} = PRK(C_{O_{pr}}, C_{j_{pb}})$ for each client $C_j \in \mathcal{C}_r(l_i)$, and sends them to S. We consider all data hosted at S, with the data value v_i corresponding to each location l_i stored in encrypted form $c_i = ENC(C_{O_{pb}}, v_i)$.
2. (Transaction owner) Client C_I sends the transaction specification (r_i, w_i) of its transaction t_i to server S.
3. S forwards them to the respective owners $C_O = O(l_i \in (r_i \cup w_i))$.
4. (Data Owner) C_O performs the *Permissions* and *Conflict detection* checks as follows:
   - *Permissions*: Check if C_I is allowed to read and update the locations in r_i and w_i, respectively.
   - *Conflict detection*: For a location $l_i \in (r_i \cup w_i)$, if there exists another concurrently executing transaction t_j such that $l_i \in w_j$, then t_i is aborted. To perform the above check, C_O keeps a record of the last allowed transaction t_j details (r_j, w_j) for which it is yet to receive a Commit notification (Step 5d). Otherwise, there is no conflict detected.

   C_O notifies the outcome of the above checks to S.
5. Depending on the outcome, S proceeds as follows:
   - *Fail*: S sends the abort message to C_I. C_I can possibly retry the same transaction after some time.

– *Permit*: For each $l_i \in r_i$, S re-encrypts the (encrypted) value $c_i = ENC(C_{O_{pb}}, v_i)$ stored in its local database as follows:
$rc_i = RENC(rk_{C_O \to C_I}, c_i)$
and forwards rc_i to C_I along with $key_O = C_{O_{pb}}$.
  (a) C_I decrypts the received values $v_i = DEC(C_{I_{pr}}, rc_i)$, and executes t_i updating the locations $l_i \in w_i$.
  (b) On completion of t_i, C_I locally commits and sends the Commit notification to S consisting of the location, (updated and encrypted) values pairs: $(l_i, c_i = ENC(key_O, v_i))$ for each $l_i \in w_i$.
  (c) For each received (l_i, c_i) pair, S updates the locations l_i in its database with ciphertext c_i. Finally, S forwards the commit notification to C_O.
  (d) C_O on receiving the Commit notification of transaction t_i, sets the corresponding "currently executing transaction" record to 'Null'.

Security Guarantees: Security of the above protocol can also be defined in terms of standard semantic security. Only client C_I can decrypt ciphertexts rc_i [6]. Data is also protected from the (untrusted) cloud provider S as it lacks keys required to decrypt both encrypted (c_i) and re-encrypted (rc_i) values.

Performance. Interestingly, the above concurrency control protocol can be considered as a pessimistic one in the sense that that it does not allow conflicting transactions to execute in parallel. From a performance perspective, this implies that while a transaction initially might encounter delays in getting the needed approval to start execution from the data owner, it saves with respect to the overhead of never having to perform a local rollback (on abort). The protocol could be made somewhat optimistic by requesting read permissions for the read set first, and then following the protocol for update permissions.

6 Experimental Results

To simulate our motivational scenario, we have developed a multi-party transactional framework. The framework is implemented in Java with currently 4 clients interacting with a cloud (data) server hosted in the Nokia internal cloud environment. (The clients, at least the ones owning data, listen for incoming requests as well to process data access requests.)

We have implemented the 'update restrictions with optimistic concurrency control' protocols as described in Section 5.2 (Algorithms 1-3). Transaction load is simulated by clients executing transactions after random delays. The number of operations in a transaction and their type is also chosen in a random fashion (using the Java Random class). To negate the effect of using random parameters in the experimental results, we have presented results taken as an average of 5 runs. We actually did not observe a lot of variation among the readings for the different runs.

The implementation is available here[1]. Instructions to setup and run the code are given in the ReadMe file of each folder.

[1] `https://sites.google.com/site/debmalyabiswas/research/Cloud_Trans.zip`

6.1 Encryption Overhead

Fig. 2 gives the encryption overhead, or the increase in overall execution time as a result of the encryption primitives. In the *Symmetric* scenario (implementation of Algorithm 2), the data values are encrypted using a pre-shared symmetric key by the clients. Symmetric key encryption is implemented using the Triple DES Encryption (DESede) algorithm provided by javax.Crypto class. Symmetric encryption for the data values coupled with homomorphic encryption to keep their decisions confidential (Algorithm 3) forms the basis of the scenario *Symmetric+Homomorphic*. Homomorphic encryption is provided by the open source Java implementation *thep*[2].

Execution time (in secs)			
No. of transactions	No encryption	Symmetric	Symmetric + Homomorphic
100	80	84	123
200	145	151	242
500	350	360	562
1000	701	725	1105
5000	3492	3554	5491

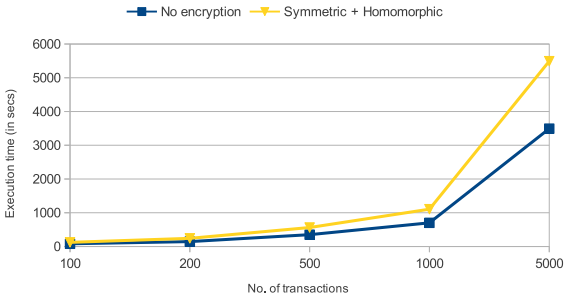

Fig. 2. Performance comparison of the protocols using different encryption primitives

From Fig. 2, we note that the overhead of symmetrically encrypting data values is minimal, with almost similar execution timings for the scenario without any encryption. Homomorphic encryption does seem to have a noticeable impact on the performance, however this increase is still linear with respect to the number of transactions, i.e. 1105secs for 1000 transactions to $1105 \times 5 \approx 5491$secs for 5000 transactions. This shows the scalability of the proposed protocols even while using homomorphic encryption in conjunction with symmetric encryption.

[2] `http://code.google.com/p/thep/`

6.2 Periodic Propagation

Periodic propagation in our context refers to the regular propagation of updates by the server to clients, other than the transaction owner performing the updates. The frequency or interval after which updates are propagated is configurable. Other than the obvious effect of periodic updates propagation on the communication overhead, here we are more interested in its effect on the abort rate (the number of aborted transactions vs. the overall number of transactions).

Towards this end, Fig. 3 plots the Abort rate vs. Frequency of updates propagation for 5000 transactions with 'No encryption'. As expected, the number of aborts increases as the propagation interval increases. For instance, the propagation interval needed to achieve $\approx 50\%$ abort rate is 20secs, which is the average execution time of $15 \approx (20 \times 3495/5000)$ transactions (the execution timings are taken from Fig. 2), i.e. it is sufficient to propagate updates once for every 15 transactions to achieve a 50% abort rate.

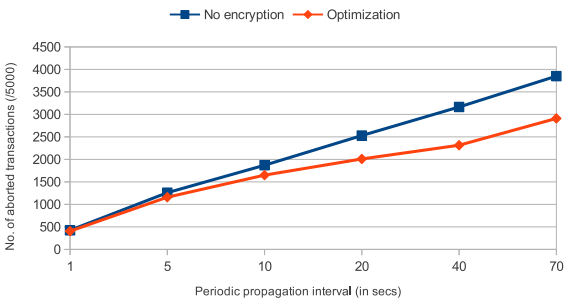

Fig. 3. Abort rate vs. Propagation interval

Recall that the primary reason for aborts in our protocols is that transactions were executed locally on outdated values (which would happen less with more frequent propagation of updates). To overcome this, we studied the effect of the following optimization on the abort rate: 'Send the updated (current) values as well while sending (or propagating via S) the abort notifications to the transaction owner C_I'. Refer to the 'Optimization' scenario in Fig. 3. As the propagation interval increases, the optimization led to $\approx 30\%$ decrease in abort rate (for both propagation intervals 40 and 70secs).

6.3 Abort Rate

Here, we extend the study of effect on abort rate to also consider scenarios with encryption - Fig. 4. We know (Fig. 2) that with encryption, the average

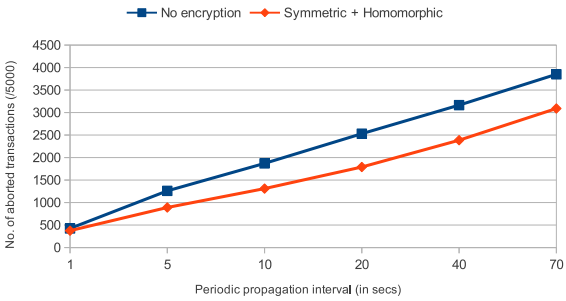

Fig. 4. Abort rate vs. Propagation interval vs. Encryption

transaction execution time increases. The underlying intuition then is to study the effect of frequency of updates propagation on the abort rate, as the execution time increases. We note that for higher execution time scenarios, a lower frequency of periodic propagation is sufficient to achieve the same abort rate. For instance (Fig. 4), to achieve a 50% abort rate in the 'No encryption' scenario, updates need to be propagated every 20secs. This is in comparison to a 40secs update propagation interval needed to achieve the same abort rate in the 'Symmetric + Homomorphic' scenario.

From another angle, let us consider the effect in terms of the transactions committed successfully (rather than the aborted ones). Let times Avg_N and Avg_S denote the average execution times of a successfully committed transaction in the 'No encryption' (Algorithm 2) and 'Symmetric+Homomorphic' (Algorithm 3) scenarios respectively. Then, we define the

$$\text{Success Ratio} = \frac{Avg_N}{Avg_S}$$

The Success ratio vs. Periodic propagation interval graph is plotted in Fig. 5. We can observe that the increase in execution time as a result of encryption (Fig. 2) is offset by the increasing abort rate. In other words, thinking only in terms of successfully committed transactions, the ratio of their average execution times steadily decreases to the point of reaching equality (70secs) as the propagation interval increases.

To summarize, the performance overhead as a result of encryption can be considered to have some pros as well in terms of reducing the communication overhead while maintaining the same abort rate, or equivalently, reducing the abort rate for the same communication overhead.

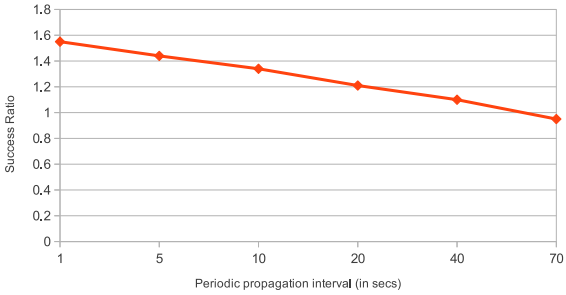

Fig. 5. Success Ratio vs. Propagation interval

7 Conclusion

In this work, we studied the problem of providing transactional guarantees over encrypted data stored in an untrusted cloud environment. We showed how different encryption primitives can be used to provide different levels of confidentiality/anonymity guarantees to the enterprises storing their data in the cloud. We further provided concurrency control protocols that allow such encrypted data to be accessed in a consistent and concurrent fashion. Together, they provide the properties much needed for wider acceptability of the cloud computing environment, especially among large enterprises.

In future, we would like to also accommodate the scenario where the cloud server itself is managed in a distributed fashion. From an implementation perspective, the proxy re-encryption primitive still needs to be integrated in our protocols. Currently, the only known open source implementation[3] is in C++, and the Java-C++ interfacing was distorting our performance results. So we plan to develop a Java based implementation of the proxy re-encryption primitive for our work.

Acknowledgments. We would like to thank Hemant Saxena for developing the prototype used in the experiments in Section 6. We would like to thank the anonymous referees for their helpful suggestions that helped to improve the work in this paper considerably. Krishnamurthy Vidyasankar's work is supported in part by the Natural Sciences and Engineering Research Council of Canada individual research grant 3182.

References

1. Weikum, G., Vossen, G.: Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery. Morgan Kaufmann Publishers (2011)

[3] `http://spar.isi.jhu.edu/prl/`

2. Kossmann, D., Kraska, T., Loesing, S.: An evaluation of Alternative Architectures for Transaction Processing in the Cloud. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 579–590 (2010)
3. Levandoski, J., Lomet, D., Mokbel, M., Zhao, K.: Deuteronomy: Transaction Support for Cloud Data. In: Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR), pp. 123–133 (2011)
4. Williams, P., Sion, R., Shasha, D.: The Blind Stone Tablet: Outsourcing Durability to Untrusted Parties. In: Proceedings of the Network and Distributed System Security Symposium (NDSS) (2009)
5. Tan, C., Liu, Q., Wu, J.: Secure Locking for Untrusted Clouds. In: Proceedings of the IEEE International Conference on Cloud Computing (CLOUD), pp. 131–138 (2011)
6. Biswas, D., Haller, S., Kerschbaum, F.: Privacy-Preserving Outsourced Profiling. In: Proceedings of the IEEE International Conference on E-Commerce Technology (CEC), pp. 136–143 (2010)
7. Camenisch, J., Kohlweiss, M., Rial, A., Sheedy, C.: Blind and Anonymous Identity-Based Encryption and Authorised Private Searches on Public Key Encrypted Data. In: Jarecki, S., Tsudik, G. (eds.) PKC 2009. LNCS, vol. 5443, pp. 196–214. Springer, Heidelberg (2009)
8. Vidyasankar, K.: Serializability. In: Encyclopedia of Database Systems, pp. 2626–2632 (2009)
9. Vidyasankar, K., Vossen, G.: A Multi-Level Model for Web Service Composition. In: Proceedings of the IEEE International Conference on Web Services (ICWS), pp. 462–469 (2004)
10. Biswas, D., Vidyasankar, K.: Spheres of Visibility. In: Proceedings of the IEEE European Conference on Web Services (ECOWS), pp. 2–13 (2005)
11. Ivan, A., Dodis, Y.: Proxy Cryptography Revisited. In: Proceedings of the Network and Distributed System Security Symposium (NDSS) (2003)
12. Paillier, P.: Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In: Stern, J. (ed.) EUROCRYPT 1999. LNCS, vol. 1592, pp. 223–238. Springer, Heidelberg (1999)