Hiding Traversal of Tree Structured Data from Untrusted Data Stores *

Ping Lin and K. Selçuk Candan

Department of Computer Sciences and Engineering
Arizona State University
Tempe, AZ. 85287

Abstract. With the increasing use of web services, many new challenges concerning data security are becoming critical. Especially in mobile services, where clients are generally thin in terms of computation power and storage space, a remote server can be outsourced for the computation or can act as a data store. Unfortunately, such a data store may not always be trustworthy and clients with sensitive data and queries may want to be protected from malicious attacks. In this paper, we present a technique to hide tree structured data from potentially malicious data stores, while allowing clients to traverse the data to locate an object of interest without leaking information to the data store. The two motivating applications for this approach are hiding (1) tree-like XML data as well as XML queries that are in the form of tree-paths, and (2) tree-structured indexes and queries executed on such data structures. We show that this task is achievable through a one-server protocol which introduces only a limited and adjustable communication overhead. This is especially essential in low bandwidth (such as wireless) distributed environments. The proposed protocol has desirable communication and concurrency performance as demonstrated by the experiments we have conducted.

Keywords: XML, content privacy, access privacy.

1 Introduction

In web and mobile computing, clients usually do not have sufficient computation power or memory and they need remote servers to do the computation or store data for them. Publishing data on remote servers helps improve data availability and system scalability, reducing clients' burden of managing data. With their computation power and large memory, such remote servers are called data stores or oracles. Typically, these data stores can not be fully trusted, for they may be malicious and can make illegal use of information stored on them to gain profits. Clients with sensitive data (e.g., personal identifiable data) may require that their data be protected from such data storage oracles. This leads to encrypted database research [1, 2], in which sensitive data is encrypted, so the content is hidden from the database. It is defined as *content privacy* [3].

* This work is supported by the AFOSR grant #F49620-00-1-0063 P0003.

Lin P. and Selçuk Candan K. (2004).
Hiding Traversal of Tree Structured Data from Untrusted Data Stores.
In *Proceedings of the 2nd International Workshop on Security in Information Systems*, pages 314-323
DOI: 10.5220/0002685203140323

Sometimes not only the data outsourced to a data store, but also queries are of value and a malicious data store can make use of such information for its own benefits. This privacy is defined as *access privacy* [3]. Typical scenarios demanding access privacy include:

- A mineral company wants to hide the locations to be explored when retrieving relevant maps from the IT'department map database.
- In a stock database, the kind of stock a user is retrieving is sensitive and needs to be kept private [4].

This leads to private information retrieval [4] research, which studies how to let users retrieve information from database without leaking (even to the server) the location of the retrieved data item.

Tree structure is a very important data structure and tree-structured data shows itself in many application domains. In this paper, we address outsourcing and hiding of tree-structured data and queries on this data. For this work, we have two motivating applications: (1) hiding XML data that is stored in the form of trees and XML queries in the form of tree paths; (2) hiding tree indexed data and queries for the data.

In this paper, we concentrate on hiding tree structured data and traversal of trees from oracles. Noticing that existing private information retrieval techniques require either heavy replication of the database onto multiple non-communicating servers or large communication costs [4], we give an one-server *tree-traversal* protocol that provides a balance between the communication cost and security requirements. To protect the client from the malicious data store, some tasks (such as traversing the tree-structures) are delegated to client.

This paper: In Section 2 we present a general overview of the framework and the outline of the hidden data access. In Section 3, we discuss how redundancy enables oblivious traversal of a tree structure. In Section 4, we address the underlying technical challenges and provide traversal algorithm. In Section 5 we give a quantitative analysis of the protocol and discuss how to tune the various system and security parameters to optimize the performance. We implement the protocol and analyze experiment results in Section 6. Section 7 discusses the amount of security the protocol can achieve and suggests ways to improve the security of the protocol in the future. Finally, we conclude in Section 8.

2 Overview of the Hiding Framework

In this section, we first give a general overview of the hiding framework. We, then, provide an outline of the proposed hidden data retrieval protocol.

There are three types of entities with different roles in the system: data owners, licensed users, and a data store (oracle). The data owners and licensed users are thin clients (as explained before). A data owner has the right to publish its data on the oracle, and a licensed user has the permission granted by some data owner to retrieve information from the data owner's data storage space in the oracle. The oracle manages data storage spaces, where data and tree structures are stored in a hidden way.

Clients run data encryption algorithms, have initial secret keys for decryption. Encryption algorithms are used to encrypt data and tree structures before sending them to the oracle to ensure that the content of data and the data structure are hidden from the oracle. If clients are accessing an outsourced index tree, they have point- or range-queries. If they are accessing outsourced XML trees, they have query patterns. Query patterns are used to traverse a tree structure along paths described by some regular-like expressions. These tasks are accomplished efficiently by "thin" clients with the help of specialized embedded hardware, such as smartcards, distributed to licensed user by data owners. Smartcards have been used a lot in mobile computing. They are relatively cheap, costing no more than several dollars. Such embedded hardware also helps in solving secret key distribution problem, i.e. by distributing smartcards that contain secret keys, a data owner distributes keys to licensed users[5].

Every time the data owner wants to insert new data into the tree structure or delete a data item from it, the owner

1. encrypts the data with a secret key,
2. walks the index structure in an oblivious manner so that the traversal path is hidden to the data store
3. locates the node of interest (either for insertion or deletion),
4. updates the tree structure by inserting or deleting encrypted index or data nodes in proper positions in the tree, in an oblivious way with respect to the data store.

By walking or updating the tree structure in an oblivious way with respect to the data store, we mean minimizing the leakage of information about the data and the tree structure as much as possible; the details of how to walk and update tree-structures in an oblivious way is described in Section 4.

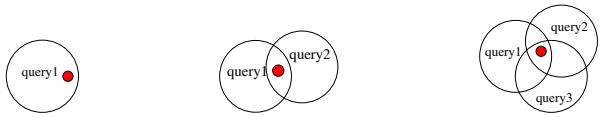
Client traversal of the tree for retrieving information is similar to update as in order to prevent the database server from differentiating between read and write operations, a read operation is always implemented as a read followed by a writing of the contents back.

3 Oblivious Traversal of the Tree Structure

It is obvious to hide the content of the nodes of a tree structure by encrypting them before they are passed to the data store. Consequently their content is already hidden from a malicious store. However, if a client traverses the tree structure in a plain way, the relationships between nodes in the tree, therefore the tree-structure as well as the user's query, are revealed. We propose two adjustable techniques to achieve oblivious traversal of tree structures: *access redundancy* and *node swapping*.

Access Redundancy: Access redundancy requires that each time a client accesses a node, instead of simply retrieving that particular node, it asks from the server a set of randomly selected $m-1$ nodes in addition to the target node. Consequently, the probability with which the data store will guess the intended node is $\frac{1}{m}$. m is a security parameter that is adjustable. We discuss how to choose the value of m in Section 5. We define this set the *redundancy set* of the target node.

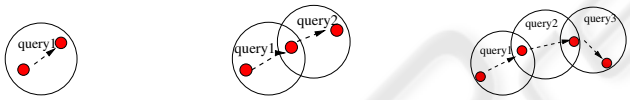
The problem with redundancy sets, on the other hand, is that their repeated use can leak information about the target node. For example, if the root node's address is fixed,



(a) After first access (b) After the second access (c) After the third access

Fig. 1. Leakage of the position of root node of index as a result of repeated accesses

then multiple access requests for the root node reveal its position (despite the use of redundancy) since the root is always in the first `redundancy set` any client asks. By intersecting all the `redundancy set`s, the data store can learn the root node. The situation is depicted in Figure 1. If the root is revealed, the risk that its children may be exposed is high, and so is the case with the whole tree structure.



(a) After the first access (b)After the second access (c) After the third access

Fig. 2. The movement of a node

Node Swapping: Consequently, in order to prevent the server from using an attack based on intersecting repeated or related requests, we have to move nodes each time they are accessed. Preferably, the move should have minimal impact on the tree structure and should not leak information about where a given node is moved to. To achieve this, each time a client needs to access a node from the server, it asks from the server a `redundancy set` consisting of m nodes that includes at least one empty node along with the target node. The client then

```
1. decodes the target,    2. swaps it with the empty,
3. re-encrypt the redundancy set and writes them back
```

Figure 2 shows how this approach prevents information leakage: Figure 2(a) shows that after the first access, the position of the target node is moved (the arrow shows the node's movement). Figure 2(b) and 2(c) show that after the second and the third accesses, the position of the target node is moved again. As shown in Figure 2, during the course of an access, the oracle has the chance to know the position of the node only if the `redundancy set` for the access has little intersection with the set of the previous access so that the position where the node moved to after the previous access is revealed. But since the node moves again once the nodes are written back after the access, such leakage is of no use to the server. In this way, the possible position of the target node is randomly distributed in the data storage space and thus the repeated-access-attack is avoided.

Node swapping requires *re-encryption* of nodes before they are re-written to the server. Re-encryption should employ a new encryption scheme/key, the reason is as follows: if the same encryption scheme is used, by comparing the content of nodes in

the `redundancy set` after rewriting with their original content, the server can easily identify the new position of the node. This means that a client has to identify how each node is encrypted. We achieve this by adding a new field which contains the secret key for that particular node. This field is always encrypted using a single/fixed secret key.This way, the client can decrypt this field to learn how to decrypt the rest of the node.

4 Hidden Tree Traversal Algorithm

To implement oblivious traversal of tree structure, some critical issues have to be solved:

- After moving one node, in order to maintain the integrity of the tree structure, the parent's pointer to this node has to be updated accordingly. How can this be performed without revealing parent-child relationships on the tree structure?
- How to keep consistency of a tree structure when there are many clients access it concurrently?
- How can we choose the values of various system parameters, such as the amount of redundancy m?

In this section, we provide techniques to address the first two of these challenges, and provide hidden retrieval algorithms based on them and the underlying protocol. In Section 5, we will discuss the choice of system parameters in greater detail.

Maintaining Parent/Child Relationships: As to the challenge of maintaining node/parent-node relationships after node swapping, we propose the following solution: find the empty node to be swapped with the child node and update the parent node correspondingly before actually moving the child node. This way, parents are always updated considering the future locations of their children.

Concurrency Control without Deadlocks: The proposed protocol will be applied to web-based mobile computing environments with large number of clients. In order to keep consistency of the tree structure with many clients accessing tree structures simultaneously, proper concurrency control must be used at server's side. There has been intensive study about index locking so that maximum concurrency is achieved with the integrity of tree structure preserved [6–8]. Since there is no pure read operation in the scheme (each node, after being read, should be written back), only exclusive locks are needed. To prevent deadlocks, we organize nodes in a data owner's data storage space into d levels.Each level of a data owner's data storage space requires an empty node list to maintain empty nodes at this level. Client always asks for locks of parent level nodes before asking for locks of child level nodes, and it always asks for locks of nodes belonging to the same level in some predetermined order (e.g. in the order of ascending node ids). In this way, all nodes in a data owner's data storage area are accessed by all clients in a fixed predetermined order. This ensures that circular waits can not occur, hence deadlocks are prevented.

In Figure 3, we provide the pseudo code of the oblivious traversal algorithm. The time complexity for this algorithm is O($d \times m$), with d denoting the depth of tree storage space and m denoting the `redundancy set` size, and the space complexity for it is O(m).

[Oblivious traversal algorithm]
Input: feature values of target data and the identifier of the data owner.
Output: pointer to the node that contains the data if there exists one; or null pointer.

1. lock and fetch the fixed public entry node to the data store, let it be PARENT, find the root, let it be CURRENT.
2. select a `redundancy set` for the CURRENT, lock nodes in the set, let the empty node in the set be EMPTY.
3. update the PARENT's pointer to refer to the EMPTY, release locks on the PARENT level.
4. swap the CURRENT with the EMPTY.
5. if CURRENT contains the data, return CURRENT
   else
   let CURRENT be PARENT, find the child node to be traversed next, let it be CURRENT, repeat 2,3,4,5.

Fig. 3. Oblivious traversal algorithm

5 Identifying Appropriate Values for the System Parameters: Hiding a Single Query

Choosing the appropriate design parameter values for a hiding system depends on various system constraints, including the acceptable communication cost and the required degree of hiding. Let us model a data owner's data storage space as d levels. Suppose the tree structure is an l-level tree. Then, the following parameters and constraints have to be considered:

- the maximum probability, δ, for the server to be able to find the actual node that the client is asking from a `redundancy set`. We have: $\frac{1}{m} \leq \delta$.
- the maximum probability, λ, for the server to find the path along which a client walks the tree structure. We have: $\frac{1}{m^l} \leq \lambda$.
  We emphasize here that although it is easy for the data store to guess the target node from the `redundancy set` if m is small, it becomes much harder to guess the parent-child relations between sequential node accesses. And the probability to discover a path is reduced exponentially with the increase of length of the path, hence should be slim even with a small value of m.
- the total communication cost ε clients are allowed to make for each data retrieval. We have: $((read(m) + write(m)) \times l \leq \varepsilon$, here $read(m)$/$write(m)$ denotes communication cost to read/write m nodes from the server.
- a node may contain multiple data points. We denote the node size, i.e. the number of data points a node is able to contain, as s. Value of s can be determined by considering the following:
  Let c denote the function of one round-trip communication cost for data points to be received from and sent to the server, e and d denote the encryption and decryption

cost function , w and r denote the write and read cost function. Theoretically, they are linear functions. Then :

```
total_cost_for_data_retrieval
=   tree_depth * m *(communication + decryption + encryption +
                          read + write cost_per_node)
=   l * m *( c(s) + d(s) + e(s) + r(s) + w(s) );
```

As node size s increases, tree depth l decreases while costs per node increases. If all other parameters are known, we can calculate *optimal* node size to minimize the total cost. However, as s increases, the probability for the data store to find a path, which is $\frac{1}{m^l}$, increases. Therefore, the value of s should be carefully chosen to ensure that security requirement is satisfied and the total cost is minimized as much as possible.

Note that most of the above constraints are linear, and an appropriate parameter setting can be easily identified using efficient algorithms.

6 Experiment Results

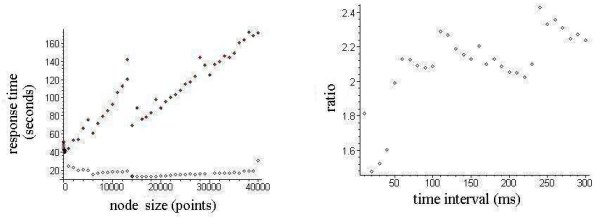
To validate the protocol, we simulated the protocol and conducted some experiments to test the protocol. The computing environment consisted of a Linux server acting as a data store and a 1.0Ghz/256M laptop generating client requests. They were connected via a Wireless LAN system. We implemented a 2 dimensional k-d tree as the index structure due to its simplicity. This simple structure enables us to observe experiment results more effectively.

In the paper, we do not experiment with range queries as we focus on path traversal. We point out that using this protocol, range queries can be implemented as multiple path traversals without deadlocks. We generated 40000 data points that were uniformly distributed in the region (0,0) to (1000000, 1000000), and stored them into a data storage space with capacity 30000 nodes. The size of `redundancy set`, m, is set to 8.

Response time and node size We executed a set of experiments to show the relationship between node size and response time, i.e., the time between a client sending a data retrieval request and getting the response.

Figure 4(a) shows the experiment result. In this figure, there are two sets of results. The dark points denote the results of experiments with encryption/decryption implemented by software. This set of results shows that when node size is set to around 50 data points, the minimum response time (about 38s), is achieved. This phenomenon verifies the theoretic observation that there must exist an *optimal* node size (Section 5). Considering the probability for the malicious server to find the path (we denote it as path probability, which is a function of page size, $\frac{1}{m^{log(\frac{num}{s})}}$, here m is the redundancy parameter, num is the total number of data points stored, s denotes node size.), suitable node size can be chosen to satisfy security requirements and minimize response time.

The set of white points depicts experiments with efficient hardware encryption/decryption. From the result, we found that encryption and decryption constitute heavy cost and with assistant hardware, response time can be greatly reduced to about 8s.



(a) response time and node size

(b) ratio of sequential process and concurrency control

Fig. 4. Experiment result

To compare our protocol with one-server Private Information Retrieval (PIR) technique [4], we also simulated PIR by transferring the whole database to a client. The simulation was conducted in the same computing environment (same linux sever, same laptop, same Wireless LAN connection). It takes about 3643s to finish transferring. We can claim that our protocol is much more efficient.

Another interesting phenomenon we observe from Figure 4(a) is that although the two sets of points have big difference in their values, they have similar zigzag pattern. This shows that the discontinues and sharp varieties in response time values are mainly determined by other costs (communication cost $c(s)$, write cost $w(s)$, read cost $r(s)$) than encyption/decryption (Section 5).

We also notice that response time for the set of black points has a strong tendency to increase with the node size, while it does very slightly for white points. This can be explained by the significant parts encryption/decryption play in the total cost and their linear increase with the node size (Section 5).

Furthermore, we conducted a set of experiments to show the effect of concurrency control. In this set of experiments, 50 retrieval requests for independently selected random data points were launched out one by one at varying frequency from every 10ms to every 300ms. In the experiment results, we found no deadlocks. We also found that the total time to finish all the requests was much less than letting the server process those retrievals sequentially. To give a sample result, when requests were launched out every 20ms, the total time required to finish them was 734.8s, and the time to process them sequentially was 1442.9s. Figure 4(b) gives the ratio of the time required to process sequentially and the time required by our protocol with concurrency control. We can see that the ratio is about 2. This means we gain 100% saving with the concurrency control. Figure 4(b) also shows that this ratio increases with the time interval. This is consistent with the common knowledge that the efficiency of the data store reduces with more clients accessing trees at the same time.

7 Future Work on Hiding Correlated Queries

The protocol should be able to protect queries and tree data structure from a polynomial time server. To study the security guarantee the protocol provides, suppose that the server keeps a history of all `redundancy sets` users retrieved, and the server

tries to infer about queries and data by statistic analysis of the history. We define each `redundancy set` a **call**, and the history a **view** of the server. The amount of security is defined as:

1. For any two different queries Q_1 and Q_2 posed in the view, the distribution of their sequences of calls are indistinguishable in polynomial-time.
2. For any two queries Q_1 and Q_2 posed in the view, it is hard to tell if they are identical or not by observing their sequences of calls.

If the data storage space is randomly initialized, queries are uniformly posed, tree nodes will always be uniformly distributed in each layer of the data storage space. So for two different queries, if their query path lengths are equal, the distribution of their sequences of calls are identical, hence indistinguishable in polynomial-time; if their query path lengths are not equal, clients can execute dummy calls at deeper levels to always make the same number of calls. We are currently studying how to improve the protocol when queries are not uniformly distributed.

As to the second security requirement, if two identical queries are posed consecutively without any interfering calls, their calls at the same level will always intersect, hence intersections will give some hint about identical queries. We are also currently studying how to improve the protocol by methodically introducing intersections between non-identical queries to make intersections independent from identical queries.

8 Conclusion

In this paper, we propose a simple, adaptive and deadlock free protocol to hide tree structured data and traversal of it from a data store. Since a lot of data such as XML has a tree structure and queries can be expressed as traversal paths, this protocol can be utilized to hide such data and queries. Compared with existing private information retrieval techniques [4, 9], our protocol does not need replication of databases and it requires less communication, and is thus practical. We provide an example how to apply it to hide XML documents and tree path based queries. Finally, we conduct experiments and observe that the proposed techniques achieve hiding without generating unacceptable concurrency problems.

Acknowledgement

We thank Dr Rida A. Bazzi for his helpful comments on this paper.

References

1. Hacigümüs, H., Iyer, B.R., Li, C., & Mehrotra, S.(2002) Executing SQL over Encrypted Data in the Database-Service-Provider Model, Proceedings of 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002. pp. 216-227.

2. Oracle Corp.,Database Security in Oracle8i, 1999. Retrieved Febuary 26, 2004, from http://otn.oracle.com/depoly/security/oracle8i/index.html.
3. Smith, S. W., & Safford, D.(2001). Practical Server Privacy with Secure Coprocessors. IBM Systems Journal, Vol. 40, No. 3. pp.683-695.
4. Chor, B., Goldreich, O., Kushilevitz, E., & Sudan, M.(1995). Private Information Retrieval, Proceeding of 36th IEEE Conference on the Foundations of Computer Sciences, Milwaukee, Wisconsin, USA, October 23-25, 1995. pp. 41-50.
5. Bouganim, L., & Pucheral, P.(2002). Chip-secured Data Access: Confidencial Data on Untrusted Servers, Proceedings of 28th Very Large Data Bases Conference, Hongkong, China, 2002. pp.131-142.
6. Bayer, R., & Schkolnich, M.(1977). Concurrency of Operations on B-Trees, *Acta Informatica*, Vol. 9, pp. 1-21.
7. Mohan, C.(1996). Concurrency Control and Recovery Methods for B+-Tree Indexes: ARIES/KVL and ARIES/IM, In Kumar, V.(Ed.) Performance of Concurrency Control Mechanisms in Centralized Database Systems, Prentice-Hall 1996, pp. 248-306.
8. Mohan, C.(2002). An Efficient Method for Performing Record Deletions and Updates Using Index Scans, Proceedings of 28th Very Large Data Bases Conference, Hongkong, China, 2002.pp.940-949.
9. Chor, B., Gilboa, N., & Naor, M.(1997). Private Information Retrieval by Keywords, Technical Report TR CS0917. Technion Israel, 1997.