Trustworthy Data from Untrusted Databases

Rohit Jain
Department of Computer Sciences, Purdue University
West Lafayette, IN, USA
jain29@cs.purdue.edu

Sunil Prabhakar
Department of Computer Sciences, Purdue University
West Lafayette, IN, USA
sunil@cs.purdue.edu

ABSTRACT

Data are often stored at untrusted database servers. The lack of trust arises naturally when the database server is owned by a third party, as in the case of cloud computing. It also arises if the server may have been compromised, or there is a malicious insider. Ensuring the trustworthiness of data retrieved from such untrusted database is of utmost importance. Trustworthiness of data is defined by faithful execution of valid and authorized transactions on the initial data. Earlier work on this problem is limited to cases where data are either not updated, or data are updated by a single trustworthy entity. However, for a truly dynamic database, multiple clients should be allowed to update data without having to route the updates through a central server.

In our previous work [5], we proposed solutions to establish authenticity and integrity of data in a dynamic setting where the clients can run transactions directly on the database server. Our solution provides provable authenticity and integrity of data with absolutely no requirement for the server to be trustworthy. Our solutions also provide assured provenance of data. In this demonstration, we present a working prototype of our solution built on top of Oracle with no modifications to the database internals. We show that system can be easily adopted in an existing databases without any internal changes to the database. We also demonstrate how our system can provide authentic provenance.

Categories and Subject Descriptors
H.2 [**Information Systems**]: Database Management

General Terms
Security

Keywords
Query Authentication, Data Integrity, Outsourced Databases

1. INTRODUCTION

Ensuring the integrity and authenticity of data is of ever increasing importance as data are generated by multiple sources, often outside the direct control of fully trusted entities. Increasingly, data are subjected to environments which can result in invalid (malicious or inadvertent) modifications to the data. Such possibilities clearly arise when we outsource data management services, e.g. in the cloud computing setting, where we lack complete control over the hardware and software running at the cloud servers. Even when the data are maintained on trusted servers, the data may get modified by a malicious insider or an intruder that manages to compromise the server or the communication channels. Currently, users have no recourse but to trust the server or rely on legal agreements. Even with such agreements, it is difficult for a user to discover, let alone prove, corruption of data or any foul play by the server. In these situations, *can we be ensured that the data retrieved from an untrusted server are trustworthy (i.e., the data and retrieved values have not been tampered or incorrectly modified)?*

In this demo, we show how it is possible to force an untrusted (relational) database server to provide trustworthy data. This is achieved by engaging the server in a protocol that makes it impossible for it to hide unfaithful execution. Earlier work in this domain assumed that the data were not modified at the untrusted server – all updates were authenticated by the data owner and then sent to the database server. The authenticity of any data that is part of the database was established directly by the data owner. In a dynamic database setting this is an unacceptable assumption. A typical database supports a large number of authorized clients that run transactions directly on the database. Updates to the data are made through these transactions. It is infeasible for the database owner to determine the correct updates for each transaction. The validity of these updates (i.e, what items are modified, and their new values) is determined by the faithful execution of a transaction semantics over the latest valid state. *How can the database owner be assured that the (untrusted) server is indeed correctly executing all transactions?*

Many applications may also require, e.g., due to regulatory compulsions, to keep the provenance of updates to the database. This can be particularly important to check if malicious activity occurred in the past. In addition to these requirements from the data owner's perspective, there is an additional requirement from the service provider. The server should be able to prove its innocence if it has faithfully executed all transactions.

We propose to demonstrate a prototype implementation of our solution [5] to the problem of ensuring transactional integrity over an untrusted database server. The system allows any user to deploy our solution to an existing database. Once deployed, the system allows database clients to execute transactions and verify the integrity of transaction execution. The database server is able to provide prove-

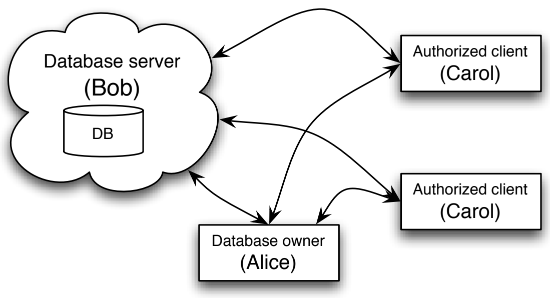

Figure 1: The various entities involved: The database owner (Alice); The database (cloud) server(Bob); and Authorized clients (Carol).

nance of data, and be able provide proof for trustworthiness of it. To the best of our knowledge, this is the first implementation to address this problem.

The rest of this paper is organized as follows. Section 2 presents our model, our requirements for a trustworthy database, and key ideas for our solution. A discussion of the implementation of the system and different components of the solution is presented in Section 3. Section 4 describes our demostration scenario. Section 5 discusses related work, and Section 6 concludes the paper and presents future directions.

2. OUR SOLUTION

In this section we briefly describe our model, our requirements for a trustworthy database, and our protocols. For more details, please refer to our paper [5].

2.1 Model

We consider a model in which multiple clients can run transactions on the server concurrently. These transactions can run search queries or modify the data as well. Figure 1 shows the various entities in our model. There are three main entities involved: **Alice**, the database *owner*; **Bob**, the *(untrusted) database server* that will host the database; and **Carol**, the client(s) that will access the database (may include Alice) from the server. Clients are authorized by Alice to independently authenticate themselves with the server.

Bob is interested in hosting Alice's database (possibly in return for a fee) and will thus make efforts to ensure that Alice is convinced about the fidelity of the hosting. Bob has control over the hardware and software that runs the database. He has unconditional read and write access to the data, and can intercept all queries posed to the database and their results, and may even modify the stored data or the results sent back to Carol. *Note that our assumptions about Bob are minimal.* In most settings, the server is likely to be at least semi-honest – i.e., it will not maliciously alter the data or query results. However, due to poor implementation, failures, over commitment of resources, or other reasons, some loss of data or updates may occur. Given the lack of direct control over the server, Alice and Carol need to be ensured that the database is operated faithfully – i.e., all the transactions are executed correctly and the data are not maliciously modified by Bob or an attacker.

2.2 Correctness and Completeness

We begin by discussing the use of Merkle Trees to prove correctness. And then further discuss a variant of it, MB-tree, which we use to prove completeness. We will use MB-tree as a building block for our overall solution. Correctness requires that any data in the query result is indeed part of the latest state of the database. Merkle Hash Tree can be used to establish correctness of query results. A Merkle Hash Tree (MHT), or Merkle Tree, is a binary tree with labeled nodes. We represent the label for node n as $\Phi(n)$. If n is an internal node with children n_{left} and n_{right}, then

$$\Phi(n) = h(\Phi(n_{left})||\Phi(n_{right})) \tag{1}$$

where $||$ is concatenation and h is a one-way hash function. Labels for leaf nodes as the hash of the tuple represented by that leaf. The root label is called 'Proof'.

Initially, MHT is created on top of the database table. Alice stores only the root hash value ($Proof$) to authenticate query results. To prove correctness of a tuple, i.e., to verify that a tuple existed in the database, Alice can ask Bob for some extra data from MHT and compute the root hash label. If the computed root hash label is the same as that she stored initially, she is convinced about the correctness of the tuple.

Completeness requires that all data items that should have been part of the query result are indeed present in the query result. Correctness and completeness combined establish the correntness of read-only queries. Merkle Hash Trees can be extended to use B+ trees instead of a binary tree [7]. Merkle B+ trees (MBT) can be used to verify both correctness and completeness. To prove completeness of a range query, Bob provides extra data with which Alice can verify that tuple values just before, and just after, the query results were indeed outside the query range. Alice can also verify that no data are missing from the query result and the returned values are indeed part of the database. For more details, please refer to [5, 7].

2.3 Transactional Integrity

Transactional integrity requires that each transaction is executed against a consistent database state that reflects the execution of all previously committed transactions in the commit order. Our solution is built upon the fact that although the database at any time is in flux and contains inconsistent states, they are built to ensure that the following simple definition of consistency is satisfied. Figure 2 shows this graphically. The initial state of the database (DB_0) is considered to be consistent. The correct (ACID) execution of a transaction (T_i) over a consistent database state (DB_{i-1}) takes the database to a new consistent state (DB_i). All the reads of T_i come from DB_{i-1}. At commit, the updates generated by T_i are applied to DB_{i-1} to produce the next consistent state DB_i. Based on this observation, the verification of a transaction execution can be done by establishing:

- Correctness and completeness of the values read by the transaction, and that the values came from the previous consistent state.

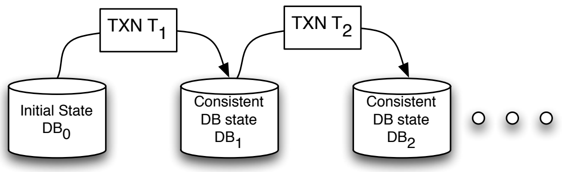

Figure 2: A simplified view of database consistency

- The transaction was faithfully executed on the read values, and the generated updates were correct.
- The updates were applied correctly to the previous consistent state to produce the next consistent state.

Alice maintains the chain of DB commit states to ensure that no valid transaction is dropped and that no invalid transaction is inserted by the server.

In the next section, we discuss our system implementation details where we also discuss how transactions are executed and verified.

3. SYSTEM DETAILS

In this section, we first present some implementation details of our system, including auxilary data structures used to allow verification of transactional integrity. Then, we describe components of our system that solve the problem.

3.1 Implementation Details

The system has been built on top of Oracle without making any modifications to the internals. The protocols are implemented in the form of database procedures using Pl/SQL. Three database procedures were created, *Select*, *Insert*, and *Update* to respectively select, insert, and update tuples in the database table. A transaction is formed using a combination of these three procedures. Three more database procedures were created, SelectVerify, InsertVerify, and UpdateVerify to verify a select, insert, and update respectively. Verification of a transaction is done using the corresponding combination of these procedures. Another database procedure, *Initialize*, is created to deploy the system on a user table. Transactions are executed with strict serializability.

Three user interfaces, ServerUI, OwnerUI, and ClientUI, have been created to demonstrate the role of the server, data owner, and clients respectively. The data owner and clients interact with the database using their respective UIs. The UIs are implemented using Python.

A user table *uTable* is composed of multiple attributes including $tupleID$ and A. At the time of deployment, *Initialize* creates an MB-tree on attribute A of *uTable*. We implement the MB-tree in the form of a database table, *uTableMBT*. Table 1 describes the different tables and indexes used in our system. Each tuple in *uTableMBT* represents a node in the tree. A better way to maintain the MB-Tree would be to use the B+ index trees of the database. However, that will require internal modifications to the index system of the database. We leave that for future work. An MB-tree node is identified by a unique id. Each node stores keys in the range $[key\_min, key\_max)$. $level$ denotes the height of the node from the leaf level, i.e., leaf nodes have $level = 0$, and the root has the highest level. The $keys$ field stores the keys of the node, and the $children$ field stores the corresponding child ids and labels. Finally, $Label$ stores the label of the node.

Tables *uTableHistory* and *uTableMBTHistory* are used to store the provenance of tuples in the tables *uTable* and *uTableMBT* respectively. In the start, *Initialize* duplicates data from *uTable* and *uTableMBT* to *uTableHistory* and *uTableMBTHistory* respectively. When a tuple is modified in *uTable* or *uTableMBT*, the new tuple value is inserted in the corresponding history table. For example, when a new tuple is created in *uTable* by a transaction with transaction ID *tID*, an entry is added to *uTableHistory* with the value of the new tuple and transactionID as *tID*. Table *uTableMBT* is updated at the time of transaction commit. At the time of a commit, the transaction modifies the MB-tree to update the proof. The updated node values are inserted into *uTableMBTHistory* with transactionID *tID*. Updates to the MB-tree are made level by level, beginning at the leaves and working to the root. Once the root is updated, the transaction is committed.

3.2 Main Components

There are three main components of our solution. *Initialization* describes the initial setup that Alice and Bob do to start accepting transactions. *Transaction Execution* describes the process different entities go through to execute a transaction which can be verified in future as described in the last component, *Verification*.

3.2.1 Initialization

In the start, Alice sends the user tables, *uTable*, to Bob. Bob runs the *Initialize* procedure to setup the database. *Initialize* creates *uTableMBT*, *uTableHistory*, and *uTableMBTHistory*. If Bob and Alice agree upon the *Proof*, Bob starts processing transactions from clients.

3.2.2 Transaction Execution

In our system a transaction is defined as a database procedure. Transactions are created using a combination *Select*, *Insert*, and *Update*. At the time of commit, each transaction is given a transaction ID tID_i, which defines the commit order of the transaction. *Insert* and *Update* modify the data and other auxiliary tables as explained before. At the end of the execution, Bob informs Carol of the transaction output and the final database state, i.e., tID_i, old proof $Proof_{i-1}$, and the new proof $Proof_i$. Bob and Carol both independently inform Alice about tID_i, $Proof_{i-1}$ and $Proof_i$, so Alice could ensure that the transaction was indeed executed on the current database state. For more details please refer to [5].

3.2.3 Verification

For verification of a transaction with tID_i, $Proof_{i-1}$ and $Proof_i$, Bob needs to prove the correctness and completeness of the values read by the transaction against $Proof_{i-1}$. Carol computes (using ClientUI) the updates that the transaction would generate when executed on the given read values. Bob then provides a proof that all and only updates generated by the transaction were used to produce $Proof_i$. Since Bob stores the provenance of each tuple, it can search the provenance data to find the values that the transaction read and wrote.

With the help of provenance data, a client can verify old transactions as well, even though the data has been updated after that.

4. DEMONSTRATION CONTENT

The demostration of our solutions will be performed using our implementation on Oracle. The demonstration will show the applicability of our solutions to an existing database. We will show the efficiency of the system using different sample transactions and show how our system can handle a dynamic database scenario where updates can arrive from multiple clients. With the use of three UIs described before, the audience will be able to visualize and interact with the system. The demonstration will mainly emphasize on the following key points:

Ease of deployment: We will demonstrate the ease with which our system can be adopted in an existing database to verify transactional integrity. We will also show the construction time of de-

Table 1: Relations and Indexes in the database

Table	Attributes	Indexes
uTable	tupleID, A, Version#	A
uTableHistory	TransactionID, tupleID, A, Version#	(tupleID, TransactionID)
uTableMBT	id, level, Label, keys, children, key_min, key_max	id, (key_min, key_max, level)
uTableMBTHistory	TransactionID, id, level, Label, keys, children, key_min, key_max	(TransactionID, id)
Transaction	id, query, finalLabel	id

ploying the system on a sample relational database with a varying number of tuples.

Transaction Execution and Verification: We will demonstrate the efficiency of executing different sample transactions. These sample transactions will show how our system handles inserts, updates and selects. A visualization of different steps involved in the transaction execution at the server will be shown as well. With this visualization the audience will be able to understand the working of our system and the overheads involved. We will demonstrate that our system can handle concurrent transactions and verifications. Users will be able to verify an old transaction from the transaction history. The demonstration will show how the system ensures correctness and completeness of query results and verifies that the transaction was executed faithfully. The roles of each involved entity (the server, the data owner, and the clients) and the overheads incurred by them will be demonstrated as well.

Provenance: In our solution, a client can verify the provenance of any selected tuple. The client can ask the server to verify the provenance of the tuple as well. In the demonstration, the audience will be able to arbitrarily check the provenance trustworthiness of any tuple. The system will show the verification process with a visual aid.

5. RELATED WORK

The problem of ensuring the authenticity of query results from an untrusted (e.g., outsourced) database has been explored by several researchers [3, 7, 8, 9, 10, 11]. In most of these works, it is assumed that a single, central entity executes the updates. This is an unreasonable assumption for many applications. Only limited work has been done for the situation where multiple clients can update the data [9]. To the best of our knowledge, no work has been done towards transactional integrity with multiple clients.

[7] offers an example of a single updater solution. It proposes an embedded Merkle tree (EMB tree) for query correctness and completeness. An EMB tree is an embedded B+-tree similar to Merkle Hash tree. The root hash of the tree is made available to clients. With the help of this root hash, clients can prove the correctness and completeness of their query results. Updates are performed only by the data owner and the updated root label is then distributed to the clients. Much work has been done towards data provenance and tamper-proofing of data [1, 2, 4, 6]. While most works focus on storing and querying provenance [1, 6], some have considered the problems of privacy and trustworthiness of data provenance [2, 4].

6. CONCLUSIONS AND FUTURE WORK

In this demonstration we presented a system that addresses the problem of ensuring the authenticity and integrity of dynamic transactional database hosted on an untrusted server where the data owner may not have any direct control over the database. To the best of our knowledge, this problem has not been identified in earlier work. Our solution makes it possible to detect any failures on the part of the server to faithfully host a transactional database with multiple independent clients. Furthermore, we demonstrate that our system provides assured provenance for the data managed by the untrusted server. These solutions are the first to address the problem of transactional integrity in an untrusted database. We demonstrate that the solutions are easy to implement in an existing database system (Oracle) without making any changes to the internals of the DBMS. Our results show that we are able to remove the need to trust the server and provide support for independent clients at a cost comparable to earlier work that does not provide either of these guarantees.

We believe that the efficiency of the solutions can be further improved by modifying the internals and also developing proof structures that have better disk performance (e.g., using GiST like indexes). We plan to explore these issues in future work. For this work, we considered strict serializability. We also plan to work towards finding efficient extentions of our system to support weaker isolation levels.

7. REFERENCES

[1] A. P. Chapman, H. V. Jagadish, and P. Ramanan. Efficient provenance storage. In *SIGMOD*, 2008.

[2] S. B. Davidson, S. Khanna, S. Roy, J. Stoyanovich, V. Tannen, and Y. Chen. On provenance and privacy. In *ICDT*, 2011.

[3] P. T. Devanbu, M. Gertz, C. Martel, and S. G. Stubblebine. Authentic third-party data publication. In *DBSec*, 2000.

[4] R. Hasan, R. Sion, and M. Winslett. Introducing secure provenance: Problems and challenges. In *StorageSS*, 2007.

[5] R. Jain and S. Prabhakar. Trustworthy data from untrusted databases. In *ICDE*, 2013.

[6] G. Karvounarakis, Z. G. Ives, and V. Tannen. Querying data provenance. In *SIGMOD*, 2010.

[7] F. Li, M. Hadjileftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *SIGMOD*, 2006.

[8] E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. In *NDSS*, 2004.

[9] M. Narasimha and G. Tsudik. Authentication of outsourced databases using signature aggregation and chaining. In *DASFAA*, 2006.

[10] H. Pang, A. Jain, K. Ramamritham, and K. lee Tan. Verifying completeness of relational query results in data publishing. In *SIGMOD*, 2005.

[11] S. Singh and S. Prabhakar. Ensuring correctness over untrusted private database. In *EDBT*, 2008.