Efficient integrity checking for untrusted database systems

Anderson Luiz Silvério[1],
Supervised by Ronaldo dos Santos Mello[1] and Ricardo Felipe Custódio[1]

[1] Programa de Pós-Graduação em Ciência da Computação
Universidade Federal de Santa Catarina
Florianópolis – SC – Brazil

{anderson.luiz,ronaldo,custodio}@inf.ufsc.br

Level: MSc
Admission: March 2012
Qualifying exam: June 2013
Conclusion: March 2014 (expected)
Steps completed: literature review, preliminary solution and evaluation
Future steps: complete solution and evaluation

Abstract. *Unauthorized changes to database content can result in significant losses for organizations and individuals. As a result, there is a the need for mechanisms capable of assuring the integrity of stored data. Meanwhile, existing solutions have requirements that are difficult to meet in real world environments. These requirements can include modifications to the database engine or the usage of costly cryptographic functions. In this paper, we propose a technique that uses low cost cryptographic functions and is independent of the database engine. Our approach allows for the detection of malicious data update operations, as well as insertion and deletion operations. This is achieved by the insertion of a small amount of protection data into the database. The protection data is calculated by the data owner using Message Authentication Codes. In addition, our experiments have shown that the overhead of calculating and storing the protection data is minimal.*

Keywords: Data Integrity, Outsourced Data, Untrusted Database

1. Introduction

Database security has been studied extensively by both the database and cryptographic communities. In recent years, some schemes have been proposed to check the integrity of the data, that is, to check if the data has not been modified, inserted or deleted by an unauthorised user or process. These schemas often try to resolve one of the following aspects of the data:

- *Correctness*: From the viewpoint of data integrity, correctness means that the data has not been tampered with.
- *Completeness*: When a client poses a query to the database server it returns a set of tuples that satisfies the query. The completeness aspect of the integrity means that all tuples that satisfy the posed query are returned by the server.

In trying to assure database integrity, many techniques have been proposed, such as in [Kamel, 2009; Li et al., 2006]. However, most of them rely on techniques that require modification of the database kernel or the development of new database management systems. Such requirements make the application of the integrity assurance mechanisms in real-world scenarios difficult. This effort becomes more evident when we consider adding integrity protection to already deployed systems.

Most of the remaining work is based on authenticated structures [Di Battista and Palazzi, 2007; Heitzmann et al., 2008; Miklau and Suciu, 2005], such as Merkle Hash Trees [Merkle, 1989] and Skip-Lists [Pugh, 1990]. These works are simpler to put into practice, since they don't require modifications to the kernel of the DBMS. However, the use of authenticated structures limits their use to static databases. Authenticated structures are not efficient in dynamic databases because the structure must be recalculated for each update.

We assume the data owner outsources its data to an untrusted database server. This assumption is particularly interesting nowadays with cloud storages and a global market. Increasingly, businesses are outsourcing their data in order to reduce the maintenance costs of its infrastructure, to increase scalability and to simplify the growth of the infrastructure [Samarati and Capitani, 2010].

This scenario is preferable for the data owner in terms of operational costs, but by storing data in outsourced database systems, there are no mechanisms to guarantee the security of the outsourced data. That means the remote server can read, modify, insert and remove information. We consider the server to be vulnerable to external attacks, such as the case of an attacker gaining access to the server and performing malicious modifications to the stored data, or internal attacks, where someone from the personnel can be coerced to perform such modifications. Although we cannot prevent such attacks from happening in reality, our goal is to create the means to detect such attacks.

The primary contribution of our work is the design, implementation and performance evaluation of Message Authentication Codes (MACs) to provide the *correctness* aspect of data integrity for a client that stores his/her data in an untrusted relational database server. We first present a simple technique to allow the client to detect updates to the stored data and insertion of data by an attacker. Additionally, we introduce a new algorithm, called Chained-MAC (CMAC), to allow the client to detect the deletion of data.

The remainder of this paper is divided into five sections. In Section 2 we discuss related work. In Section 3 we describe in details our techniques for providing integrity assurance. In Section 4 we analyse the performance impact of our proposed method and section 5 presents our final considerations and future works.

2. Related work

The major part of integrity verification found in literature is based on authenticated structures. Namely, Merkle Hash Trees [Merkle, 1989] and Skip-Lists [Pugh, 1990].

Li et al. [Li et al., 2006] present the Merkle B-Tree (MB-Tree), where the B^+-tree of a relational table is extended with digest information as in an Merkle Hash Tree (MHT). The MB-Tree is then used to provide proofs of correctness and completeness for posed queries to the server. Despite presenting an interesting idea and showing good results in their experiments, their approach suffers from a major drawback. To deploy this approach, the database server needs to be adapted as the B^+-tree needs to be extended to support an MHT. Such modifications may not be feasible in real world environments, especially those that are already in use.

Di Battista and Palazzi [Di Battista and Palazzi, 2007] propose to implement an authenticated skip list into a relational table. They create a new table, called *security table*, which stores an authenticated skip list. The new table is then used to provide assurance of the authenticity and completeness of posed queries. This approach overcomes the requirement of a new DBMS, present in the previous approach. While only a new table is necessary within this approach, its implementation can be done as a plug-in to the DBMS. However, the experimental results are superficial. It is not clear what is the actual overhead in terms of each SQL operation. Moreover, their experiments show that the overhead increase as the database increases, while in our approach the overhead is constant in terms of the database size.

Miklau and Suciu [Miklau and Suciu, 2005] implement a hash tree into a relational table, providing integrity checks for the data owner. The data owner needs to securely store the root node of the tree. To verify the integrity, the clients need to rebuild the tree and compare the root node calculated and stored. If they match, the data was not tampered with. Despite using simple cryptographic functions, such as hash, the use of trees compromises the efficiency of their method. A tuple insert using their method is 10 times slower than a normal insert, while a query is executed 6 times slower. In our experiments, presented in section 4, we show that the naive implementation of our method is as good as their method.

E. Mykletun et al. [Mykletun et al., 2006] study the problem of providing *correctness* assurance of the data. Their work is most closely related to what we present in this paper. They present an approach for verifying data integrity, based on digital signatures. The client has a key pair and uses its private key to sign each tuple he/she sends to the server. When retrieving a tuple, the client uses the correspondent public key to verify the integrity of the retrieved tuple. This work was extended by Narasimha and Tsudik [Narasimha and Tsudik, 2005] to also provide proof of *completeness*.

The motivation of the authors to use digital signatures is to allow integrity checking in multi-querier and multi-owner models. Therefore, for multi-querier and multi-owner models, their work is preferable. On the other hand, if the querier and the data owner are the same, our work can provide integrity assurance more efficiently.

3. Contributions

To achieve a low cost method to provide integrity and authenticity, we propose to perform the cryptographic operations on the client side (application), using Message Authentication Codes (MAC) [Bellare et al., 1996; Krawczyk et al., 1997]. The implementation consists of adding a new column to each table. This new column stores the output of the MAC function applied to the concatenation of the attributes (all columns, or a subset of them) of the table. The function also utilises a key, which is only known by the application. The value of the MAC column is later used to verify integrity.

The use of a MAC function ensures the integrity of the INSERT and UPDATE operations. However, the table is still vulnerable to the unauthorized deletion of rows. To overcome this issue, we propose a new algorithm for linking sequential rows, called "Chained-MAC (CMAC)". The result of the CMAC is then stored into a new column. The value of this new column, given a row n, a key k, and MAC_n as the MAC value of the row n, is calculated as follows:

$$CMAC_n = MAC(k, (MAC_{n-1} \oplus MAC_n)) \tag{1}$$

The use of CMAC provides an interesting property to the data stored in the table where it is used. When used, the CMAC links the rows in a way that an attacker cannot delete a row without being detected, since he does not have access to the secret key to produce a valid value to update the CMAC column of adjacent rows.

Despite linking adjacent rows, any subset of the first and last rows can be deleted without being detected. This is possible because the first row has no previous row and the last row does not have a subsequent row to be linked with. To overcome this issue, we propose changing the CMAC to a circular method. That is, for the first row, the $n-1$-th row to be considered will be the n-th row (i.e. the last row). With this change, if the last row is deleted, the integrity check will fail for the first row. Similarly, since the first row now has a predecessor, integrity checks can start at the first row (in the regular mode it would always start in the second row).

It is important to notice that the introduction of the CMAC brings a new requirement: the table must be ordered by some attribute. However, in real world scenarios, all tables have a primary key, and all the main DBMS orders the tables in terms of the primary key. Therefore, the requirement for a ordered table of the CMAC does not have a big impact to the deployment of our technique in real world scenarios.

3.1. Verifying the integrity of a table

To verify the integrity of a row with the MAC column, the application must calculate the MAC of that row and compare it with the value of the MAC column. The row can be considered as not modified if the calculated MAC is equal to the stored MAC. Applying this comparison to each row of a table will ensure the integrity of this table against insertion and modification attacks. As stated earlier, the use of the MAC does not provide a means to verify the integrity of a table against unauthorized deletions. In this case, the CMAC column should be used. To verify the integrity of a table with the CMAC column, the application must check the integrity of each pair of sequential registries of the table. That is, a Table T has not been modified (unauthorized) if:

$$\forall t_{n-1}, t_n \in T : t_n.CMAC = CMAC(k, t_{n-1}, t_n) \tag{2}$$

4. Preliminary Experimental Evaluation

To assess the efficiency of our techniques we implemented a tool to evaluate the performance of using HMAC, as the MAC function, and CMAC. The prototype was implemented using the C programming language and the OpenSSL library. The DBMS used was MySQL database and the experiments were performed on a machine running both MySQL server and client application. The machine had Intel Core 2 Quad CPU Q8400 with 4Mb cache, at 2.66GHz, 4GB RAM 800Mz, and 320Gb disk, SATAII, 16Mb cache, 7200RPM, running an Ubuntu 11.04 32-bit operating system with OpenSSL 0.9.8d and MySQL 5.1. Additionally, we used the SHA-1 hash function to calculate the HMAC with a 256-bit long key.

We considered different scenarios to evaluate the performance of the proposed techniques. For each scenario, we executed the workload a thousand times over a table with 10 thousand tuples of random values. All the results shown below are the average of these executions. In all scenarios we focus on evaluating the amount of time spent on the operations of INSERT, UPDATE, DELETE and SELECT, performed under four distinct conditions:

1. Without security mechanisms;
2. Using HMAC only;
3. Using both HMAC and CMAC;
4. Using both HMAC and CMAC in the circular mode.

In the first scenario, we focused on measuring and comparing the execution times for the INSERT operation under each specified condition. The results show that the baseline took 42,3 μs, while the HMAC took 47 μs, 90% of which is spent on the server side and 10% on the client side. The scenario with the use of CMAC executed in 118,3 μs, with 91% of the time spent on the server and 9% on the client. The CMAC in the circular mode executed in 331,7 μs, where 72% is executed by the server and 28% by the client.

The CMAC in the circular mode can be optimized if the client stores a small amount of data. The major reason for the difference between the regular mode and the circular mode of the CMAC is that in the circular mode we need to retrieve and update additional rows. If the client stores the first row locally, we eliminate one query, reducing the execution time from 331,7 μs to 236,5 μs.

In the second scenario, we focused on measuring and comparing the execution times for the UPDATE operation under each specified condition. The results show that the baseline took 127,6 μs, while the HMAC took 134 μs, 95% of which is spent on the server side and 5% on the client side. The CMAC (both in regular and circular mode) executed in 381,9 μs, with 80% of the time spent on the server and 20% on the client. The reason that the execution time for the CMAC in the regular and circular mode are the same is because they execute the exact same operations.

We can also optimize the CMAC for the UPDATE operation if we consider that some values are available on the client side at the moment of the operation. In this case, when updating a row n, we need the MAC and CMAC of the $n + 1$-th and the $n - 1$-th rows. If these rows are available on the client side at the moment of the update, the execution time is 204,5 μs.

In the third scenario, we focused on measuring and comparing the execution times for the DELETE operation under each specified condition. The baseline executed in 51 μs and when using the HMAC to delete a row, there is no additional cost since there is no extra operations to be performed. On the other hand, the CMAC (both in regular and circular mode) executed in 186,5 μs, with 96% of the time spent on the server and 4% on the client. As we have shown for the UPDATE operation, the CMAC in the regular and circular mode have the exact same operations and therefore the overhead is the same.

We can use the same idea presented for the UPDATE operation to improve the efficiency of the CMAC. In the naive implementation, before deleting a row n, we execute a select query to retrieve the $n + 1$-th and the $n - 1$-th rows. Considering that these rows are available on the client side at the moment of the delete, the execution time is reduced from 186,5 μs to 105,2 μs.

Finally, in the last scenario, we focused on measuring and comparing the execution times to check the integrity during the SELECT operation under each specified condition. A SELECT query, without verifying the integrity (the baseline), took 18,4 μs. To verify the integrity of the HMAC the client needs to recalculate the HMAC and compare it to the one retrieved from the server. This operation executed in 22,5 μs, due to the calculation of the HMAC. When using the CMAC, the client needs to retrieve the HMAC of the previous row and recalculate both the HMAC and CMAC. These extra operations increase the execution time to 54 μs. However, if we consider that the previous row is available on the client side, the execution time is reduced to 27,6 μs.

5. Final remarks

This paper proposes secure and efficient methods for providing integrity for relational database systems. Our methods focus on strategies for detecting unauthorised actions (insertions, deletions and updates) from a vulnerable database server.

Prior work either requires modifications in the database implementation or uses inefficient cryptographic techniques (for example, public key cryptography). The requirement of modifying the core of a database system makes the deployment of these methods difficult in real world scenarios. Thus, one significant advantage of our method is that it is DBMS-independent and can be easily deployed in existing environments. Another advantage of our method is that we focused on using more simple and efficient cryptographic algorithms to provide integrity checks.

To complete this work we'll first improve the experimental evaluation, comparing our approach with prior work. Additionally, we'll investigate and propose methods regarding the management of the secret key, used to generate the MAC and CMAC values. That is, we need methods to allow the change of the secret key, in case of a simple key update and/or a key compromise.

References

Bellare, M., Canetti, R., and Krawczyk, H. (1996). Keying hash functions for message authentication. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pages 1–15, London, UK, UK. Springer-Verlag.

Di Battista, G. and Palazzi, B. (2007). Authenticated relational tables and authenticated skip lists. In *Proceedings of the 21st annual IFIP WG 11.3 working conference on Data and applications security*, pages 31–46, Berlin, Heidelberg. Springer-Verlag.

Heitzmann, A., Palazzi, B., Papamanthou, C., and Tamassia, R. (2008). Efficient integrity checking of untrusted network storage. In *Proceedings of the 4th ACM international workshop on Storage security and survivability*, StorageSS '08, pages 43–54, New York, NY, USA. ACM.

Kamel, I. (2009). A schema for protecting the integrity of databases. *Computers & Security*, 28(7):698–709.

Krawczyk, H., Bellare, M., and Canetti, R. (1997). HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational). Updated by RFC 6151.

Li, F., Hadjieleftheriou, M., Kollios, G., and Reyzin, L. (2006). Dynamic authenticated index structures for outsourced databases. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 121–132, New York, NY, USA. ACM.

Merkle, R. C. (1989). A certified digital signature. In Brassard, G., editor, *CRYPTO*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer.

Miklau, G. and Suciu, D. (2005). Implementing a tamper-evident database system. In *Proceedings of the 10th Asian Computing Science conference on Advances in computer science: data management on the web*, ASIAN'05, pages 28–48, Berlin, Heidelberg. Springer-Verlag.

Mykletun, E., Narasimha, M., and Tsudik, G. (2006). Authentication and integrity in outsourced databases. *ACM Transactions on Storage*, 2(2):107–138.

Narasimha, M. and Tsudik, G. (2005). Dsac: integrity for outsourced databases with signature aggregation and chaining. In *Proceedings of the 14th ACM international conference on Information and knowledge management*, CIKM '05, pages 235–236, New York, NY, USA. ACM.

Pugh, W. (1990). Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676.

Samarati, P. and Capitani, S. D. (2010). Data Protection in Outsourcing Scenarios : Issues and Directions. *ACM Symposium on Information, Computer and Communications Security*.