# ENSURING THE INTEGRITY OF ENCRYPTED DATABASES IN THE DATABASE-AS-A-SERVICE MODEL*

**Hakan Hacıgümüş**
*IBM Almaden Research Center*
*650 Harry Road, San Jose, CA 95120, USA*
hakanh@acm.org

Bala Iyer
*IBM Silicon Valley Lab.*
*San Jose, CA 95141, USA*
balaiyer@us.ibm.com

Sharad Mehrotra
*Department of Information and Computer Science*
*University of California, Irvine, CA 92697, USA*
sharad@ics.uci.edu

**Abstract** In the database-as-a-service model, a service provider hosts the clients' data and allows access to the data through the Internet. Database-as-a-service model offers considerable benefits to organizations with data management needs by allowing them to outsource their data management infrastructures. Yet, the model introduces many significant challenges, in particular that of data privacy and security. Ensuring the integrity of the database, which is hosted by a service provider, is a critical and challenging problem in this context. We propose an encrypted database integrity assurance scheme, which allows the owner of the data to ensure the integrity of the database hosted at the service provider site, in addition to the security of the stored data against malicious attacks.

*This work was supported in part by NSF grant CCR 0220069 and an IBM Ph.D. Fellowship.

**Keywords:** database, security, privacy, integrity, authentication, encryption, cryptography, e-services, application service provider model

# 1. Introduction

Rapid advances in networking and Internet technologies have fueled the emergence of the "software-as-a-service" model, also referred as the Application Service Provider (ASP) model, for enterprise computing. Successful examples of commercially viable software services include rent-a-spreadsheet, electronic mail services, general storage services, disaster protection services. *Database-as-a-Service* (DAS) model [11, 8, 9], inherits all the advantages of the ASP model by allowing organizations to leverage data management solutions provided by the service providers, without having to develop them on their own. It alleviates the need for organizations to purchase expensive hardware and software, deal with software upgrades, and hire professionals for administrative and maintenance tasks. Instead, the new model allows third party database service providers the capability to seamlessly host data from diverse organizations and take over these tasks. By outsourcing, organizations can concentrate on their core competencies instead of sustaining a large investment in data management infrastructures. Increasingly, large companies are outsourcing their IT departments and sometimes their entire data centers to specialized service providers [6, 5].

NetDB2 system [11], which is being developed at IBM in cooperation with University of California, Irvine, is an instantiation of DAS model. The system has been deployed on the Internet and under constant use for over two years.

Among the others, *data privacy* and *security* are the most significant challenges in the DAS model. In the DAS model, the user data is stored at the service provider (ASP) site. The ASP stores the client's data and the client poses queries against that database and the ASP (or the server) responds back to the client with results of the queries. Most companies and individuals view their data as an asset. The theft of intellectual property already costs organizations great amount of money every year [4]. Therefore, first, the owner of the data needs to be assured that the data is protected against malicious attacks from outside of ASP. In addition to this, recent studies indicate that 40% of those attacks are perpetrated by the insiders [4]. Hence, the second and more challenging problem is the privacy of the data when even the ASP itself is not trusted by the owner of the data. First problem is examined in [11] and the second one is studied in [8], which explores how SQL queries can be executed over encrypted data.

In this paper, we look at another important issue that arises in the context of the second problem stated above. Although the client's data is protected against both outsiders and the ASP with data encryption techniques, how can the client ensure the integrity of the database, which belongs to the client but is under the control of the ASP? Our preliminary work on this issue appears in [10]. That work does not provide efficient schemes to ensure the table-level integrity. Here we present efficient incremental techniques to ensure the table-level integrity and discuss the applicability of the schemes in real database environments. The previous work also fails to assess the performance implications of the integrity assurance schemes. In this paper, we experimentally evaluated the performance of the schemes we propose by using the standard benchmark queries.

We view the integrity problem in two dimensions. First, when the client receives a record from the sever, how can the client ensure the integrity of the record? That is, how can the client verify that the data has not been changed in an unauthorized way? Second, the client needs to assure the integrity of whole table stored at the server site. This problem is more pronounced in dynamic environments, where the update rate of the database is high. In such environments, the client needs efficient mechanisms, which will require minimal computational resources that are limited at the client site. While entailing minimal computational resources, preferred solution is a mechanism that can be automated requiring minimum human involvement. In this paper, we present such a solution to ensure the integrity of hosted databases.

Integrity problems may arise both from malicious or non-malicious circumstances. Malicious threats may originate from misbehaving server or some other adversary who breaks into the system. Active and replay (restore) attacks, as described in [12], are the typical examples for those kinds of threats. Non-malicious threats can also have many sources. One example for those is system failures. ASP may experience a system breakdown and may not be able to recover all user data from on-line and/or archive sources. In those cases, the client does not have any verification mechanism to detect the integrity of the original data. In the course of NetDB2 project, we have particularly observed the need for addressing data integrity problems raised from second group of sources, namely; non-malicious threats.

To address these issues, we propose two-level encrypted database integrity scheme, which consists of *Record-Level Integrity* and *Table-Level Integrity* concepts. Those are developed in the context of the DAS model and described in Section 3.2 and Section 3.3, respectively. We note that, the techniques we present in this paper are not specific to either the DAS

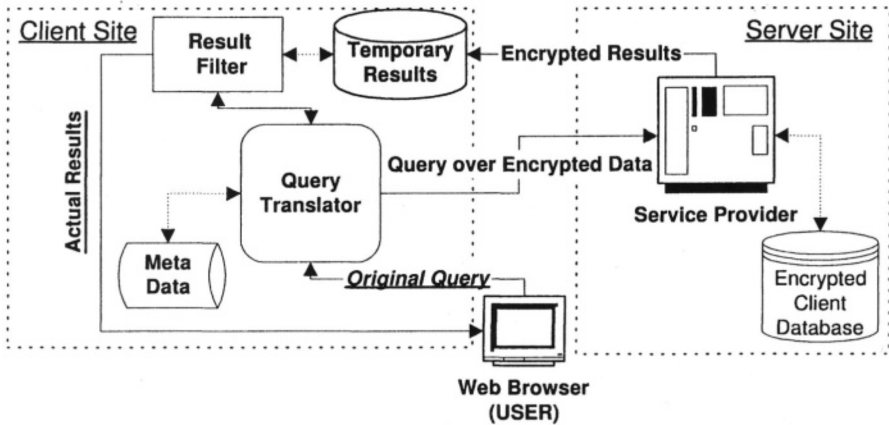


*Figure 1.* Database-as-a-Service architecture

model or the encrypted databases but have wider applicability to any relational data that is subject to integrity issues. Once implementation of these techniques is in place, the system can automatically detect the integrity violations and intrusions.

The rest of the paper is organized as follows. Section 2 provides background on the DAS model and the encrypted database storage model. Section 3 presents our solution to encrypted database integrity by discussing the record-level and the table-level integrity techniques. Section 4 gives our experimental results on queries from the TPC-H benchmark. We conclude the paper in Section 5.

## 2. Background

### 2.1. Database-as-a-Service Model

The system we use in this study is based on the architecture proposed and described in [8] and [9]. The basic architecture and the control flow of the system are shown in Figure 1. It is comprised of three fundamental entities. A *user* poses the query to the client. A *server* is hosted by the service provider that stores the encrypted database. The encrypted database is augmented with additional information (which we call the index) that allows certain amount of query processing to occur at the server without jeopardizing data privacy. A *client* stores the

*Table 1.* Relation *emp*

| *eid* | *ename* | *salary* | *addr* | *did* |
|---|---|---|---|---|
| 23 | Tom | 70K | Maple | 40 |
| 860 | Mary | 60K | Main | 80 |
| 320 | John | 23K | River | 35 |
| 200 | Sarah | 55K | River | 10 |

*Table 2.* Encrypted version $emp^S$ of relation *emp*

| *RID* | *etuple* | $eid^{id}$ | $ename^{id}$ | $salary^{id}$ | $eid^f$ | $ename^f$ |
|---|---|---|---|---|---|---|
| 1 | =*?Ew@R*((¡¡=+,-. . . | 2 | 19 | 81 | @R*. . . | ¡(*. . . |
| 2 | b*((¡¡(*?Ew@=l,r. . . | 4 | 31 | 59 | E:q!,. . . | =+,. . . |
| 3 | w@=W*((¡¡(*?E:,j. . . | 7 | 59 | 22 | @Aw*. . . | ¿(]. . . |
| 4 | ffTi* @=U(¡?G+,a. . . | 8 | 49 | 59 | hRs(. . . | ?@Q=. . . |

data at the server. Client[1] also maintains *metadata* for translating user queries to the appropriate representation on the server, and performs post-processing on server query results. From the privacy perspective, the most important feature is, the client's data is always stored in encrypted form at the server site. The server never sees the unencrypted form of the data, and executes the queries directly over encrypted data without decrypting it. We fully studied the query processing techniques, which allow this process, in [8].

## 2.2. Encrypted Database Storage Model

We briefly summarize how the client's data stored at the server in an encrypted fashion.[2] Following this we introduce our extensions to the model to implement data integrity techniques.

For each relation $R(A_1, A_2, \ldots, A_n)$, we store, on the server, an encrypted relation: $R^S(etuple,\ P_1^{id}, P_2^{id},\ \ldots, P_i^{id},\ F_1^f, F_2^f, \ldots, F_j^f)$, where $1 \leq i \leq n, 1 \leq j \leq n$. Here, an *etuple* stores an encrypted string that corresponds to a tuple in relation *R*. Each attribute $P_i^{id}$ stores the partition index for corresponding attribute $A_i$ that will be used for

[1]Often the client and the user might be the same entity.

[2]We will not repeat here all of the details of storage model, since they are thoroughly discussed in [8]. Rather, we only provide necessary notations to explain the constructs we develop in this work.

query processing at the server. $F_j^f$ represents the encrypted form of the attribute value of corresponding attribute $A_j$.

For example, consider a relation *emp* given in Table 1 that stores the information about the employees. The *emp* table is mapped to a corresponding table, shown in Table 2, at the server: $emp^S(RID$, *etuple*, $eid^{id}$, $ename^{id}$, $salary^{id}$, $eid^f$, $ename^f)$.

RID represents record identifier, which we will use in Section 3.3. The second column *etuple* contains the string corresponding to the encrypted tuples in *emp*. For instance, the first tuple is encrypted to "=*?Ew@R*((¡¡=+,-. . . " that is equal to $\mathcal{E}_k$(1,23, *Tom*, 70*K*, *Maple*, 40), where $\mathcal{E}$ is a deterministic encryption algorithm with key $k$. Any deterministic encryption technique such as AES [1], Blowfish [14], DES [7] etc., can be used to encrypt the tuples. The third column corresponds to the index on the employee ids.[3] The sixth column represents the individually encrypted values employee ids.

# 3. Encrypted Database Integrity

We study the integrity of a database at two different granularity levels, 1) Integrity of the records, 2) Integrity of a table. We call the former as *Record-Level Integrity* and the latter as *Table-Level Integrity*.

## 3.1. Preliminaries

In this subsection we provide necessary definitions and concepts that we use in the rest of the section. Our definitions are based on [12, 15].

- A *hash function* is a function $h$, which has, as a minimum, the following properties: 1) *Compression*, meaning $h$ maps an input $x$ of arbitrary finite bitlength to an output $h(x)$ of fixed bitlength $n$ and 2) *Ease of computation*, meaning given $h$ and an input $x$, $h(x)$ is easy to compute.
- A *one-way function* is a function $f$ that for each $x$ from the domain of $f$, it is easy to compute $f(x)$, but it is computationally infeasible to find any $x$ such that $y = f(x)$. A hashfunction $h(x)$ is *collision resistant* if it is computationally infeasible to find any two distinct inputs $x, x'$ where $h(x) = h(x')$.
- The *manipulation detection codes (MDCs)* are one-way collision resistant hash functions that provide a *representative image* or *hash* of a message.

[3] Details of creation of those index values can be found in [8].

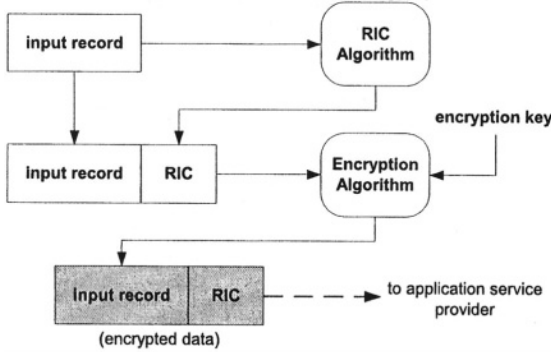


*Figure 2.* Record-level integrity with RICs

• The *data integrity,* in general, can be defined as a property, which guarantees that the data has not been manipulated in an unauthorized manner since the time it was created by an authorized spurce. MDCs provide this level data integrity in combination with data encryption.

As we will see, we need to expand this definition to satisfy data integrity requirements in the DAS model. This definition only provides data integrity for individual records stored in the database at the server site. In addition to that, we want to ensure the integrity of tables in an efficient way.

## 3.2. Record-Level Integrity

The record-level integrity represents that the content of a record has not been manipulated in an unauthorized manner. Although it may not be apparent, data encryption does not provide data integrity automatically. The owner of the decryption key can decrypt the encrypted messages, which were encrypted with the same key. But this does not guarantee that the encrypted message has not been manipulated by the adversary. The discussion of how encrypted messages can be manipulated undetectably can be found in [12]. This motivates the need for data integrity measures over encrypted data.

To provide record-level integrity we propose a scheme based on *Record Integrity Codes (RICs).* RICs are specially computed representative images for each record with certain security and uniqueness measures. Figure 2 shows the procedure that provides record-level data integrity. The client has a record $r$ that will be inserted into the database, which is maintained by the application service provider or simply the server. The client first computes the hash code of the record $H = h(r)$ by using a

hashing algorithm, which produces *Record Integrity Code* (RIC). This can be any algorithm, which satisfies the security requirements given before. Here we use MDCs for this purpose. After this step, the client concatenates the hash code $H$ with the original record text $r$ and encrypts them together by using any deterministic encryption algorithm $\mathcal{E}$ with secret key $k$, i.e., the client computes ciphertext $C = \mathcal{E}_k(r \parallel h(r))$, where $\parallel$ represents concatenation. The client inserts ciphertext $C$ as an *etuple* into the database.

Whenever the client requests a record, the server sends back the corresponding *etuple* in encrypted form. To verify the integrity of the record, the client first decrypts the *etuple* recovering $r'$ and $H'$, which is the RIC, parts. Since only the client has the secret key $k$ for encryption algorithm no one else can decrypt. Then the client independently computes $h(r')$ of received record $r'$ and compares that with the hash code $H'$. If they are equivalent, this verifies that the received record is authentic and has data integrity, i.e., has not been manipulated in an unauthorized manner.

## 3.3. Table-Level Integrity

In the previous section we discussed mechanisms that enable the client to test the integrity of individual the records returned by the server. In this section, we study schemes, which allow the client to validate the integrity of the table(s).

The integrity of a table may be compromised by an adversary by modifying, adding, or deleting some or all of the records. An unauthorized modification on records can be detected by the record-level integrity techniques, when they are queried by the client. However, the record-level integrity is not enough to detect unauthorized addition and/or deletion of records in any case.

To successfully detect such modifications, we propose a mechanism, which creates a signature for each table. Those signatures can be thought of the fingerprints of the corresponding tables. They are created, updated, and stored at the client site in a secure way. Hence, an adversary cannot re-compute the signature and verification of the signature the would detect the unauthorized modifications. Considering our focus on security, the signatures should be resilient against various cryptographic attacks.

In addition to that, we make note of another important issue regarding the computation of the signatures. The client should re-compute the signature when he inserts/deletes a record from a table. This is an extra overhead for the client and can be significant in terms of system resources

at the client site and network traffic. Undoubtedly, a highly preferred alternative would be a scheme, which allows incremental updates on the signatures instead of requiring the processing of all of the existent records.

**3.3.1 Incremental Signatures.** To achieve the incremental updates on signatures, we make use of specialized schemes, XOR MACs, introduced by Bellare et al. [3, 2], in the context of incremental cryptography [2]. The main idea is; when there is a change in a document $D$, to construct mechanisms to reflect "only the changes" on the cryptographic transformation (in our case the signatures) of the document $D$ without processing all the document from scratch. Hence, such an incremental algorithms run time would be proportional to the number of changes but not proportional to the document length. Formal definition of XOR scheme we use in this study is given in [3] as follows:

$f_1$ is a pseudorandom function (PRF) with secret key $k_1$ and $f_2$ is a pseudorandom permutation (PRP) with secret key $k_2$. $Rand(\cdot)$, called randomizer, is an algorithm, which given a string $x$ picks a random $k$-bit string $r$ and returns $x \| r$. The message that will be authenticated is $D = D_0 \| D_1 \| \ldots \| D_n \| D_{n+1}$,where $D_0$ is a special start symbol and $D_{n+1}$ is a special end symbol. Then the tag of the message is computed in three steps:

1 Randomize: Let $R_i = Rand(D_i) : 0 \leq i \leq n$

2 Chain: Let $h = \oplus_{i=0}^{n} f_1(R_i, R_{i+1})$

3 Tag: Let $T = f_2(h)$

In our system setup, $D_i$'s are defined as the database records $r_i$ of the table and the tag $T$ corresponds to the signature $S$ of the table. [3] discusses possible instantiation alternatives for PRF $f_1$, such as using DES, MD5, or both. The instantiation of $f_2$ can be handled in a similar way as a block-cipher can be viewed as a PRP. For simplicity, we assume that size of a database record is equal to block size of the block-cipher. If record size is larger, then it can be processed as multiple blocks. If it is smaller than the block size, the standard padding techniques can be used. Security of the scheme also analyzed thoroughly in [3] against the various types of attacks.

**Insert:** We use the record identifiers (RIDs) to keep track of the index of the records $r_i$ in the algorithm. Note that, although the RIDs are stored in the clear at the server, they are also included in the *etuples* in the encrypted form. (See Section 2.2) Therefore, even the server manipulates the RIDs, the client can always recover the original ones

by decrypting the *etuples*. We assume that the RIDs are always non-decreasing unique numbers. (Most of commercial database systems already provide table definitions to create this type of ids.)

Assume that the current signature of a table is $S$ and the RID of the last record is $i$. If the client inserts a new record $r_{i+1}$ (i.e., RID=$i+1$), then the new signature $S'$ is computed as follows:

1 The client recovers hash value $h$ as $h = f^{-1}(S)$

2 The client moves the special end record one position forward by assigning $R_{i+2} = R_{i+1}$ (Note that here $R_{i+1}$ is not the randomized form of the new record being inserted but the randomized form of the special end record.)

3 The client computes $R_{i+1} = Rand(r_{i+1})$

4 The client updates the hash value by:

$$h' = h \oplus f_1(R_i, R_{i+2}) \oplus f_1(R_i, R_{i+1}) \oplus f_1(R_{i+1}, R_{i+2})$$

5 The client computes the new signature $T' = f_2(h')$

**Delete:** Deletions are reflected on the signature in an incremental way as follows. Assume that the current signature of a table is $S$ and the client wants to delete a record $r_i$ (i.e., RID=$i$), then the new signature $S'$ is computed as follows:

1 The client recovers hash value $h$ as $h = f^{-1}(S)$

2 The client updates $h$ by:

$$h' = h \oplus f_1(R_{i-1}, R_{i+1}) \oplus f_1(R_{i-1}, R_i) \oplus f_1(R_i, R_{i+1})$$

3 The client computes the new signature $T' = f_2(h')$

Incrementing requires five calls to the PRF. This is a significant saving as compare to computing the signature from scratch, which would require one call for each record in the table.

**3.3.2 Application of the Incremental Signatures.** We have presented the incremental signature computation formally. The scheme uses the RIDs as record indexes, which are also employed in the computation steps. Now, we will discuss the realization of the scheme in database applications.

The insertion operation does not pose any difficulty. Since the RIDs are always incrementing unique numbers, the client inserts a new record

with a new RID and updates the signature of the table. On the other hand, deletion operation requires some more work.

In typical database applications, the users usually don't query the records with their RIDs. As an example, if we want to delete a record of an employee whose employee id is 123 from `employee` table, a typical query would be:

```
DELETE FROM employee WHERE eid = 123
```

To execute this deletion, the client runs two queries to obtain the RIDs and the *etuples* required to compute the updated signature.

The first query retrieves the RID of the record being deleted as follows:

```
SELECT rid,etuple FROM employee WHERE eid^f = ℰ(123)
```

Let us assume that the RID of the corresponding record is 345, then the second query retrieves the RIDs of two records, whose RIDs are the immediate predecessor and successor of the RID of the record being deleted.[4]

```
(SELECT rid,etuple FROM employee
   WHERE rid < 345 ORDER BY rid DESC FETCH FIRST ROW ONLY)
UNION
(SELECT rid,etuple FROM employee
   WHERE rid > 345 ORDER BY rid ASC FETCH FIRST ROW ONLY)
```

After running this query, the client would have enough information to update the signature of the table.

We used equality predicate in the WHERE clause of the query for this example. We note that, if the query has inequality predicates then the approach should be different. In that case, the client could use partition indexes (See Section 2.2) instead of the field-level encrypted values to obtain the necessary information.[5] What the client needs are RIDs and corresponding *etuples* in any case. Therefore, the queries should be reformulated accordingly to retrieve that information by utilizing partition indexes.

We also note that SQL update operations can be implemented as the combination of delete and insert.

---

[4]SQL queries given below have been executed on IBM DB2 v8.1.

[5]Query processing with partition indexes is fully discussed in [8].

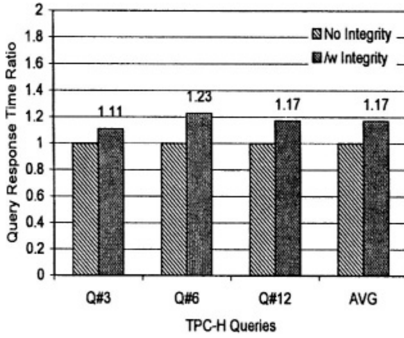


*Figure 3.* Server response time

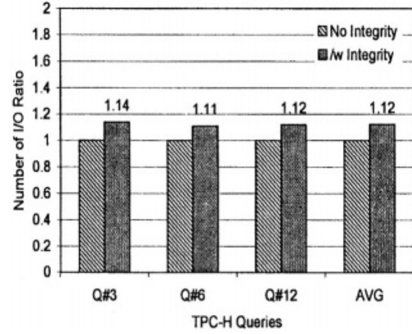


*Figure 4.* Server I/O performance

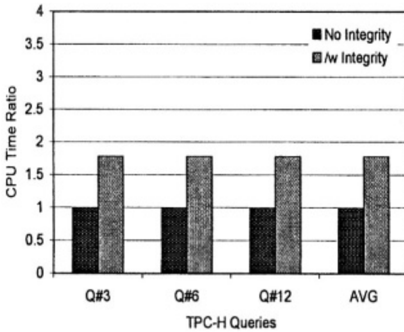


*Figure 5.* Client response time

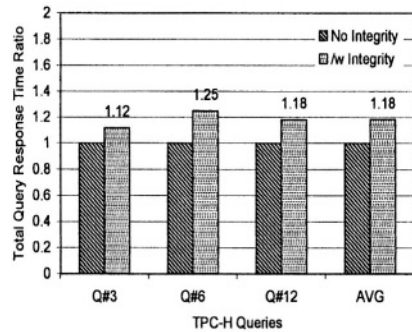


*Figure 6.* Total query response time

## 4. Experimental Evaluation

We have conducted experiments to evaluate the overhead introduced by the record-level integrity schemes that we have presented. Our results showed that the overheads are not significant. We used the standard TPC-H benchmark [16] queries, specifically Q#3, Q#6, Q#12 from TPC-H suite. The TPC-H database was created in scale factor 1, which corresponds to 1GB in size. To implement the integrity schemes, we used the AES block cipher [1] as the encryption algorithm, and the MD5 message digest [13] as the one-way hash function. The experiments were executed on a server, which had a Pentium III-750MHz CPU, 256MB RAM, Windows 2000 OS, and IBM DB2 UDB v8.1.

To provide the detailed analysis, we report the results for the server side queries, the client side queries, and the total query elapsed times. In all figures, we compare two cases, namely; the case, where there is

no integrity scheme is deployed and the case, where the record-level integrity schemes we have presented are implemented.

Figure 3 shows the overhead in server side query response times. On the average the increase is 17%. Figure 4 presents the increase in the number of I/Os executed at the server site. The increase, 12% on the average, is mainly due to the increase in the tuple sizes. The tuple sizes increase as a result of inclusion of the RICs into the records and it is reflected to server side query response time.

Figure 5 shows the measurements for client side query CPU time. It increased, constantly for all queries, by 70%. The overhead is due to the increased number of bytes that are decrypted and the validation of the integrity of the tuples by the client. The total query response time, which is presented in Figure 6, showed 18% increase on the average, which did not constitute significant overhead.

## 5. Conclusions

We have studied the crucial problem of encrypted database integrity in the context of database-as-a-service model. We have proposed two-level encrypted database integrity scheme, which consists of *Record-Level Integrity* and *Table-Level Integrity* concepts, as a solution to this problem. Our scheme is combined with encrypted database storage model. Consequently, the resultant system provides the security of the stored data against the malicious attacks as well as the database integrity features, which ensure the authenticity and the validity of the data stored at the service provider site.

## References


[1] AES. Advanced Encryption Standard. *National Institute of Science and Technology, FIPS 197,* 2001.

[2] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental cryptography: The case of hashing and signing. *Lecture Notes in Computer Science,* 839:216–233, 1994.

[3] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental cryptography and application to virus protection. In *STOC,* pages 45–56, 1995.

[4] Computer Security Institute. CSI/FBI Computer Crime and Security Survey. *http://www.gocsi.com,* 2002.

[5] ComputerWorld. J.P, Morgan signs outsourcing deal with IBM. Dec. 30, 2002.

[6] ComputerWorld. Business Process Outsourcing. Jan. 01, 2001.

[7] DES. Data Encryption Standard. *FIPS PUB 46, Federal Information Processing Standards Pub.,* 1977.

[8] **H. Hacıgümüş,** B. Iyer, C. Li, and S. Mehrotra. Executing SQL over Encrypted Data in Database Service Provider Model. In *Proc. of ACM SIGMOD,* 2002.

[9] H. Hacıgümüş. Privacy in Database-as-a-Service Model. *Ph.D. Thesis, Department of Information and Computer Science, University of California, Irvine,* 2003.

[10] H. Hacıgümüş, B. Iyer, and S. Mehrotra. Encrypted Database Integrity in Database Service Provider Model. In *Proc. of Certification and Security in E-Services (CSES'02), IFIP $17^{th}$ World Computer Congress,* 2002.

[11] H. Hacıgümüş, B. Iyer, and S. Mehrotra. Providing Database as a Service. In *Proc. of ICDE,* 2002.

[12] D. R. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1997.

[13] R. Rivest. The MD5 Message-Digest Algorithm. *RFC 1321,* 1992.

[14] Bruce Schneier. Description of a new variable-length key, block cipher (blowfish), fast software encryption. In *Cambridge Security Workshop Proceedings,* 1994.

[15] D. R. Stinson. *Cryptography: Theory and Practice.* CRC Press, 1995.

[16] TPC-H. *Benchmark Specification,* `http://www.tpc.org/tpch`.