Designing Secure Indexes for Encrypted Databases

Erez Shmueli[1], Ronen Waisenberg[1], Yuval Elovici[1], and Ehud Gudes[2]

[1] Ben-Gurion University of the Negev, Faculty of Engineering, Department of Information Systems Engineering,
Postfach 653, 84105 Beer-Sheva, Israel;
{erezshmu, ronenwai, elovici}@bgu.ac.il

[2] Ben-Gurion University of the Negev, Department of Computer Science,
Postfach 653, 84105 Beer-Sheva, Israel
ehud@cs.bgu.ac.il

Abstract. The conventional way to speedup queries execution is by using indexes. Designing secure indexes for an encrypted database environment raises the question of how to construct the index so that no information about the database content is exposed. In this paper, the challenges raised when designing a secure index for an encrypted database are outlined; the attacker model is described; possible attacks against secure indexes are discussed; the difficulty posed by multiple users sharing the same index are presented; and the design considerations regarding keys storage and encryption granularity are illustrated. Finally, a secure database-indexing scheme is suggested. In this scheme, protection against information leakage and unauthorized modifications is provided by using encryption, dummy values and pooling. Furthermore, the new scheme supports discretionary access control in a multi-user environment.

1 Introduction

Increasingly, organizations prefer to outsource their data center operations to external application providers. As a consequence of this trend toward outsourcing, highly sensitive data is now stored on systems that are not under the data owners' control. While data owners may not entirely trust providers' discretion, preventing a provider from inspecting data stored on their own machines is difficult. For this kind of service to work successfully, it is of primary importance to provide means of protecting the secrecy of the information remotely stored, while guaranteeing its availability to legitimate clients [1].

Communication between the client and the database service provider can be secured through standard means of encryption protocols, such as SSL (Secure Socket Layer), and is therefore ignored in the remainder of this paper. With regard to the security of stored data, access control has proved to be useful, on condition that data is accessed using the intended system interfaces. However, access control is useless if the attacker simply gains access to the raw database data, thus bypassing the traditional mechanisms [2]. This kind of access can

easily be gained by insiders, such as the system administrator and the database administrator (DBA).

Database encryption introduces an additional layer to conventional network and application security solutions and prevents exposure of sensitive information even if the raw data is compromised [3]. Database encryption prevents unauthorized users from viewing sensitive data in the database and, it allows database administrators to perform their tasks without having access to sensitive information. Furthermore, it protects data integrity, as unauthorized modifications can easily be detected [4].

A common technique to speed up the execution of queries in databases is to use a pre-computed index [5]. However, once the data is encrypted, the use of standard indexes is not trivial and depends on the encryption function used. Most encryption functions preserve equality, thus, "Hash" indexes can be used, however information such as the frequencies of indexed values is revealed. Most encryption functions do not preserve order, so "B-Tree" indexes, can no longer be used once the data is encrypted.

Moreover, if several users with different access rights use the same index, each one of them needs access to the entire index, possibly including indexed elements that are beyond his access rights. For example, Google Desktop, allows the indexing and searching of personal computers data. Using this tool, a legitimate user is able to bypass user names and passwords and view personal data belonging to others who use the same computer, since it is stored in the same index [6].

The contribution of this paper is threefold. First, we describe the challenges arising when designing a secure index for an encrypted database. Second, we outline design considerations regarding keys storage and encryption granularity. Third, we present a new indexing scheme that answers most of these challenges.

The remainder of the paper is structured as follows: in section 2, related works are outlined; in section 3, the problem statement is defined; in section 4, design considerations regarding database encryption are described; in section 5, we present a new secure database index; and section 6 presents our conclusions.

2 Related Work

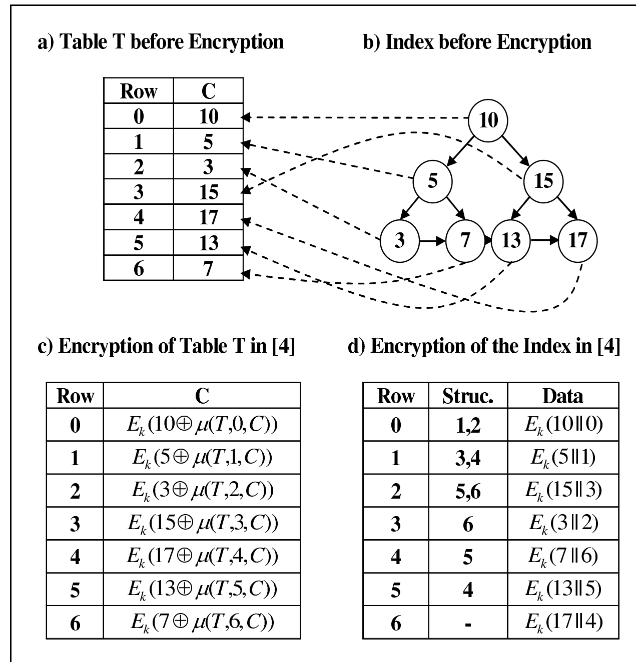
The indexing scheme proposed in [7] suggests encrypting the whole database row and assigning a set identifier to each value in this row. When searching a specific value, its set identifier is calculated and then passed to the server, who, in turn, returns to the client a collection of all rows with values assigned to the same set. Finally, the client searches the specific value in the returned collection and retrieves the desired rows. In this scheme, equal values are always assigned to the same set, so some information is revealed when statistical attacks are applied, as stated in [1].

The indexing scheme in [1] suggests building a B-Tree index over the table plaintext values and then encrypting the table at the row level and the B-Tree at the node level. The main advantage of this approach is that the B-Tree content

is not visible to the untrusted database server. However, only the client can now perform the B-Tree traversal, by executing a sequence of queries. Each query retrieves a node located at a deeper level of the B-Tree.

The indexing scheme provided in [2] is based on constructing the index on the plaintext values and encrypting each page of the index separately. Whenever a specific page of the index is needed for processing a query, it is loaded into memory and decrypted. Since the uniform encryption of all pages is likely to provide many cipher breaking clues, the indexing scheme provided in [8] proposes encrypting each index page using a different key depending on the page number. However, these schemes, which are implemented at the level of the operating system, are not satisfactory, since in most cases it is not possible to modify the operating system implementation. Moreover, in these schemes, it is not possible to encrypt different portions of the database using different keys. The disadvantage of using only one key is discussed in subsection 3.6.

The database encryption scheme in [4] suggests encrypting each database cell with its unique cell coordinates $\mu(T, R, C)$ and each index value concatenated with its unique row identifier, as illustrated in Fig. 1.



Row	C
0	10
1	5
2	3
3	15
4	17
5	13
6	7

Row	C
0	$E_k(10 \oplus \mu(T,0,C))$
1	$E_k(5 \oplus \mu(T,1,C))$
2	$E_k(3 \oplus \mu(T,2,C))$
3	$E_k(15 \oplus \mu(T,3,C))$
4	$E_k(17 \oplus \mu(T,4,C))$
5	$E_k(13 \oplus \mu(T,5,C))$
6	$E_k(7 \oplus \mu(T,6,C))$

Row	Struc.	Data
0	1,2	$E_k(10 \Vert 0)$
1	3,4	$E_k(5 \Vert 1)$
2	5,6	$E_k(15 \Vert 3)$
3	6	$E_k(3 \Vert 2)$
4	5	$E_k(7 \Vert 6)$
5	4	$E_k(13 \Vert 5)$
6	-	$E_k(17 \Vert 4)$

Fig. 1. Database and index encryption as described in [4]

Figure 1 illustrates the database and index encryption as described in [4]. The use of cell coordinates for the encryption of the database table and of row identifiers for the index entries, ensures that there is no correlation between the indexed values and the database ciphertext values. However, this indexing scheme is not resistant to tampering attacks.

The encryption function suggested in [9] preserves order, and thus allows range queries to be directly applied to the encrypted data without decrypting it. In addition it enables the construction of standard indexes on the ciphertext values. However, the order of values is sensitive information in most cases and should not be revealed.

In [10], a smart card with encryption and query processing capabilities is used to ensure the authorized and secure retrieval of encrypted data stored on untrusted servers. Encryption keys are maintained on the smart card. The smart card can translate exact match queries into equivalent queries over encrypted data.

In [11], the security of databases stored on smart cards is explored. However, retrieval performance is not the focus of their work and it is not clear how much of their techniques applies to general-purpose databases not stored on smart cards, as stated in [9].

3 The Problem Statement

3.1 The Attacker Model

The attacker can be categorized into three classes: *Intruder* - A person who gains access to a computer system and tries to extract valuable information. *Insider* - A person who belongs to the group of trusted users and tries to get information beyond his own access rights. *Administrator* - A person who has privileges to administer a computer system, but uses his administration rights in order to extract valuable information [10].

All of the above attackers can use different attack strategies: *Direct storage attacks* - Attacks against storage may be performed by accessing database files following a path other than through the database software, by physical removal of the storage media or by access to the database backup disks. *Indirect Storage attacks* - An adversary can access schema information, such as table and column names, metadata, such as column statistics, and values written to recovery logs in order to guess data distributions. *Memory attacks* - An adversary can access the memory of the database software directly [9] (The last one is usually protected by the Hardware/OS level).

3.2 Information Leakage

According to [4], a secure index in an encrypted database should not reveal any information on the database plaintext values. We extend this requirement, by categorizing the possible information leaks:

Static leakage - Gaining information on the database plaintext values by observing a snapshot of the database at a certain time. For example, if the index is encrypted in a way that equal plaintext values are encrypted to equal ciphertext values, statistics about the plaintext values, such as their frequencies can easily be learned.

Linkage leakage - Gaining information on the database plaintext values by linking a database value to its position in the index. For example, if the database value and the index value are encrypted in the same way (both ciphertext values are equal), an observer can search the database ciphertext value in the index, determine its position and estimate its plaintext value.

Dynamic leakage - Gaining information about the database plaintext values by observing and analyzing the changes performed in the database over a period of time. For example, if a user monitors the index for a period of time, and if in this period of time only one value is inserted (no values are updated or deleted), the observer can estimate its plaintext value based on its position in the index.

3.3 Unauthorized Modification

In addition to the passive attacks that *monitor* the index, active attacks that *modify* the index should also be considered. Active attacks are more problematic, in the sense that they may mislead the user. For example, modifying index references to the database rows may result in queries returning erroneous set of rows, possibly benefiting the adversary.

Unauthorized modifications can be made in several ways: *Spoofing* - Replacing a ciphertext value with a generated value. *Splicing* - Replacing a ciphertext value with a different ciphertext value. *Replay* - Replacing a ciphertext value with an old version previously updated or deleted [11].

3.4 Structure Perseverance

When applying encryption to an existing database, it would be desirable that the structure of the database tables and indexes is not modified during the encryption. This ensures that the database tables and indexes can be managed in their encrypted form by a database administrator as usual, while keeping the database contents hidden. For example, if a hash index is used and the values therein do not distribute equally, performance might be undermined, and the DBA might wish to replace the hash function. In such a case, the DBA needs to know structure information, such as the number of values in each list, but does not need to know the values themselves.

3.5 Performance

Indexes are used in order to speed up queries execution. However, in most cases, using encrypted indexes causes performance degradation due to the overhead of decryption. Indexes in an encrypted database raise the question of how to construct the index so that no information about the database content is revealed, while performance in terms of time and storage is not significantly affected.

3.6 Discretionary Access Control (DAC)

In a multi-user (discretionary) database environment each user only needs access to the database objects (e.g., group of cells, rows and columns) needed to perform his job.

Encrypting the whole database using the same key, even if access control mechanisms are used, is not enough. For example, an insider who has the encryption key and bypasses the access control mechanism can access data that are beyond his security group. Encrypting objects from different security groups using different keys ensures that a user who owns a specific key can decrypt only those objects within his security group [15]. Following this approach, different portions of the same database column might be encrypted using different keys. However, a fundamental problem arises when an index is used for that column as illustrated in Fig. 2.

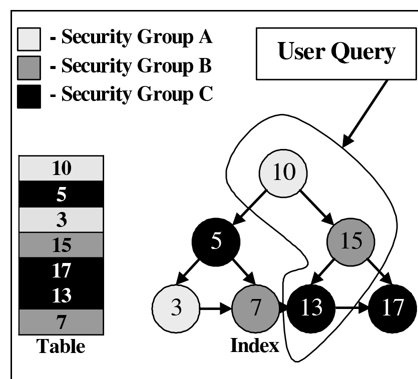


Fig. 2. An Indexed Column Encrypted using Different Keys.

Figure 2 illustrates an index that is queried by users who belong to different security groups. Each one of them needs access to the entire index, possibly to indexed elements, which are beyond their access rights. The same problem arises when the index is updated.

4 Design Considerations

4.1 Key Storage

One important issue in any encrypted database is that of keys storage [1, 2, 12]. Several alternatives were proposed in the literature:

Storing the encryption keys at the server side - The server has full access to the encryption keys. All computations are performed at the server side.

Storing encryption keys at the client side - The client never transfers the keys to the server and is solely responsible for performing all encryption and decryption operations. When the database server has no access to the encryption keys, most computations cannot be performed at the server side, since they require decryption.

Keys per session - The database server has full access to the encryption keys during the session but does not store them on disk. This ensures that during the session, the user transaction can be performed entirely at the server side.

Table 1 summarizes the dependency between the trust in the server and the keys storage. If we have no trust in the database server, we would prefer to keep the encryption keys at the client side only. In cases where the database server itself is fully trusted, but its physical storage is not, we can store the keys at the server side in some protected region.

Table 1. Keys Storage vs. Trust in Server

	Server Side	Keys per Session	Client Side
Absolute	+	+	+
Partial	-	+	+
None	-	-	+

4.2 Encryption Granularity

Index encryption can be performed at various levels of granularity: single values, nodes, pages or whole index. When choosing the level of granularity, the following should be considered (see Table 2):

Information Leakage - The higher the level of encryption granularity, the less information is revealed. *Single values* level encryption of the index reveals sensitive information, such as frequencies of the index values. *Whole Index* level encryption ensures that information about the indexed data cannot be leaked, since the index is encrypted as one unit.

Unauthorized Modifications - Encryption at higher levels of granularity makes it harder for the attacker to tamper with the data. *Single values* level encryption of the index allows an attacker to switch two ciphertext values without being noticed. *Whole Index* level encryption implies that a minor modification to the encrypted index has a major effect on the plaintext index and can easily be detected.

Structure Perseverance - Higher levels of encryption granularity conceal the index structure. *Whole Index* level encryption changes the structure of the index

since the basic element of reference is changed from a single value to the entire index. *Single values* level encryption of the index preserves its structure.

Performance - Finer encryption granularity affords more flexibility in allowing the server to choose what data to encrypt or decrypt. *Whole Index* level encryption requires the whole index to be decrypted, even if a small number of index nodes are involved in the query. *Single values* level encryption of the index enables decryption of values of interest only.

Table 2. Comparing Different Levels of Encryption Granularity

	Information Leakage	Unauthorized Modifications	Structures Perseverance	Performance
Single values	Worst	Worst	Best	Best
Nodes	Low	Low	Medium	Medium
Pages	Medium	Medium	Low	Low
Whole Index	Best	Best	Worst	Worst

Better performance and preserving the structure of the database cannot be achieved using pages or whole index encryption granularity. However, special techniques can be used in order to cope with unauthorized modifications and information leakage, when single values or nodes granularity encryption are used.

In our scheme, which is presented in the remainder of this paper, we assume that the encryption keys are kept per session and that the index is encrypted at the single values level of granularity.

5 A New Secure Database Index

In this section, a secure database index, encrypted at the single values level of granularity is suggested. Best performance and structure perseverance are simply obtained since single values granularity encryption is used. Information leakage and unauthorized modifications are protected against using encryption, dummy values and pooling. Finally, a technique that supports discretionary access control in a multi-user environment is presented.

5.1 Encryption

Let us assume that a standard index entry is of the form:

$$(V_{trc}, IRs, ER) \tag{1}$$

Where:
V_{trc} - An indexed value in table t, row r and column c.
IRs - The internal reference (references between index entries)

ER - The external reference (reference to the database row).

An entry in the proposed secure index is defined as follows:

$$\left(E_k\left(V_{trc}\right), IRs, E'_k\left(ER\right), MAC_k\left(V_{trc} \parallel IRs \parallel ER \parallel SR\right)\right) \tag{2}$$

Where:
k - An encryption key.
E_k - A nondeterministic encryption function.
E'_k - An ordinary encryption function.
SR - The entry self reference.
MAC_k - A message authentication code function.

The E_k function The implementation of E_k introduces a tradeoff between *static leakage* and *performance* (see Table 3). If E_k is a non-deterministic encryption function (that is, equal plaintext values are encrypted to different ciphertext values), statistics such as the frequencies and distribution of values are concealed, but comparing index values requires their decryption. On the other hand, if E_k is an order preserving encryption function, some information about the index values is revealed (e.g., their order) but it is possible to compare values without the need to decrypt them.

Table 3. The Tradeoff between Security and Performance for E_k implementation

	Security	Performance
Nondeterministic	High	Worst
Equality Preserving	Medium	Low
Order Preserving	Low	Medium
No Encryption	Worst	High

We suggest using a non-deterministic E_k. A possible implementation of E_k follows:

$$E_k(x) = E''_k(x||r) \tag{3}$$

Where:
k - An encryption key.
E''_k - An ordinary encryption function.
r - A random number with a fixed number of bits.

Using the above implementation of E_k there is no correlation between $E_k(V_{trc})$ and the corresponding column ciphertext value (random numbers are used before encryption) and thus *linkage leakage* attacks are eliminated.

The MAC_k function Most commercial databases implement indexes like tables (as heap files). In this implementation, index entries are uniquely identified using the pair: *page id* and *slot number* [5] (in our notations SR and IR).

Message authentication codes (MAC) are used to protect against unauthorized modifications of messages. They mix the message cryptographically under a secret key, and the result is appended to the message. The receiver can then recompute the MAC and verify its correctness. It should be impossible for an attacker to forge a message and still be able to compute the correct MAC without knowing the secret key.

In our scheme, we use a MAC_k function to protect the index entries against unauthorized modifications. *Spoofing* attacks are eliminated, since the MAC value depends on V_{trc}, and once $E_k(V_{trc})$ is tampered with, V_{trc} will not match the V_{trc} used in the MAC. *Splicing* attacks are eliminated since the MAC value depends on SR and trying to substitute two encrypted index entries will be detected, since SR would not match the SR used in the MAC. *Replay* attacks can be eliminated by adding a new dimension, that of time, to each index node. This enables the validity of the node version to be verified, just as SR was used in order to verify its logical location.

The MAC value added to each index entry causes data expansion and thus, its size introduces a tradeoff between security and data expansion.

Evaluating a Query The following pseudo code illustrates a query evaluation using the encrypted index [3]:

```
INPUT:
     A table: T
     A column: C
     A value: V
     A query: SELECT * FROM T WHERE T.C>=V

OUTPUT:
     A collection of row-ids.

X := getIndex(T, C).getRootNode();
While (not X.isLeaf()) Do
    If (not x.isValid())
        Throw IllegalStateException();
    Else
        If X.getValue()<V Then
             X := X.getRightSonNode();
        Else
             X := X.getLeftSonNode();
        End If;
    End If;
End While;
```

[3] The encrypted index is assumed to be implemented as a binary tree. However, the pseudo code can be easily be generalized to handle a B-Tree implementation.

```
RESULT := {};

While X.getValue()<V Do
     X := X.getRightSiblingNode();
End While;

While X is not null Do
     RESULT := RESULT union {X.getRowId()};
     X := X.getRightSiblingNode();
End While;

Return RESULT;
```

While *isLeaf*, *getRightSonNode*, *getLeftSonNode* and *getRightSiblingNode* methods relate to the index structure and their implementation does not change, *getValue* and *getRowId* are implemented differently so that encryption and decryption support is added. The new method, *isValid*, verifies the index entry integrity using the MAC value.

Performance can be furthermore improved, if entries' verification is performed periodically on the entire index and not as part of each index operation.

5.2 Using Dummy Values and Pooling

In order to cope with *dynamic leakage* attacks, we need to reduce the *level of confidence* an adversary has about the effect of new inserted data on the database indexes. There is a trade-off between how much of the index is updated and how much information an adversary is able to learn [13]. In this subsection, we propose two techniques for reducing the adversary level of confidence:

Dummy values - We can insert dummy values to the index with each insertion made by the user, and thus reduce the level of confidence. However, inserting dummy values with each insertion results in data expansion. The number of dummy values added in each insertion determines the level of confidence an adversary has about the position of a value within the index.

Pooling - The use of pooling in order to improve performance of insertions to database indexes was suggested by [14]. We suggest the use of pooling for security reasons. We define a fixed size pool for each index holding the new inserted values. Only when the pool is full, will the indexes be updated with these values. Furthermore, the extraction of values from the pool should be done in a random order, since it makes it difficult to link the extracted values and their corresponding inserted values. When a query is to be executed, we first need to search the pool, and then to search the rest of the index. The pool size determines the level of confidence an adversary has about the position of a value within the index. Note that a full scan has to be performed on the pool whenever the index is used. Thus, the size of the pool is a privacy-performance trade-off. Using a pool size that has space complexity of $O(\log |table\ size|)$ will not affect the time complexity of the queries.

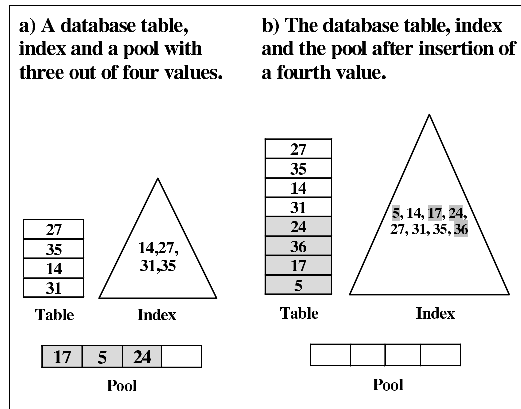


Fig. 3. A Database Index Using Pooling.

Figure 3 illustrates a database index using pooling. Figure 3a illustrates the database table, index and pool after the insertion of three values: 17,5,24 where the pool size is four values. Figure 3b illustrates the database table, index and pool after the insertion of a fourth value: 36, that fills the pool. The values in the pool are then extracted in random order and inserted into the database table and index.

5.3 Supporting DAC

If indexes are used only by one user or if they are never updated, it is possible to maintain a local index for each user. Securing indexes stored locally is relatively easy. However, such local indexes do not work well in a multi-user environment, since synchronizing them is difficult. Thus, it is necessary to store the indexes in one site, such as the database server, and share them between users.

As mentioned in subsection 3.6, a fundamental problem arises when multiple users share the same encrypted index and each user has different access rights. We suggest a simple but elegant solution to this problem: split the index into several sub-indexes where each sub-index relates to values in the column encrypted using the same key. A similar approach for disseminating XML documents was proposed in [15].

Figure 4 illustrates sub-indexes where each sub-index relates to values in the column encrypted using the same key. In order to evaluate a query, only ciphertext values with the same access right are queried. All the values in a sub-index belong to the same security group, and thus the problem that is illustrated in Fig. 2 is eliminated.

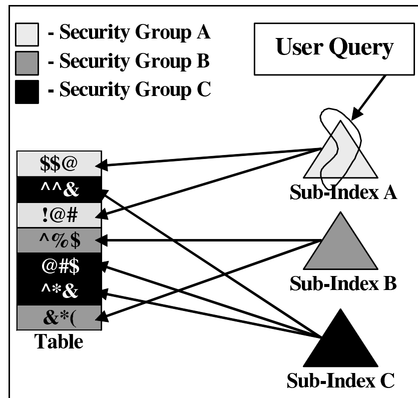


Fig. 4. An Encrypted Database Column and its Corresponding Sub-Indexes.

Figure 5 illustrates how a query is executed using sub-indexes. A secure session between the user and the database server is created (step 1). The user supplies his encryption keys[4] (step 2). During the secure session, the user submits queries to the server (step 3). The server uses the encryption keys in order to find the set of indexes that the current user is entitled to access[5] (step 4). The query is executed on the set of indexes found (step 5). The result set is returned to the user (step 6).

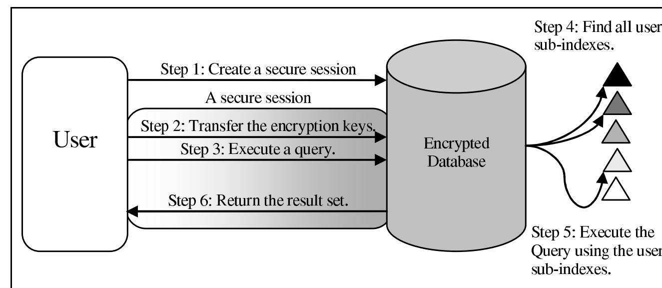


Fig. 5. Query Execution Using Sub-Indexes.

[4] The encryption keys can be supplied using smart card architecture.
[5] The database server can maintain a directory that maps the hash of a given encryption key to the corresponding sub-index.

6 Conclusions

In this paper, we outlined the challenges raised when designing a secure index for an encrypted database. The challenges include: prevention of information leakage; detection of unauthorized modifications; preserving the structure of the index; and supporting discretionary access control; while performance in terms of time and storage is not significantly affected. In addition, two design considerations, keys storage and encryption granularity, were discussed. For each design consideration, we proposed several alternatives and elaborated on the tradeoffs between them.

A secure database index encrypted at the single values level of granularity was suggested. Performance and structure perseverance are simply obtained since single values granularity encryption is used. We used encryption, dummy values and pooling in order to prevent information leakage and unauthorized modifications. Finally, in order to support discretionary access control in a multi-user environment, we suggested splitting the index into several sub-indexes, where each sub-index relates to values in the column encrypted using the same key.

Table 4 summarizes the challenges and solutions that were suggested throughout this paper.

Table 4. Summary of Challenges and Solutions

Challenge	Solution
Static Leakage	E_k is nondeterministic
Linkage Leakage	Different encryption functions for the index and the table
Dynamic Leakage	Dummy Values and Pooling
Spoofing	MAC (V_{trc} - the indexed value)
Splicing	MAC (SR - the node self reference)
Replay	MAC (Version)
Structure Perseverance	Single values granularity; Structure data is not encrypted
Performance	Single values granularity; Periodic verification
DAC	Sub-Indexes

References

1. Damiani, E., De Capitani diVimercati, S., Jajodia, S., Paraboschi, S. and Samarati, P.: Balancing Confidentiality and Efficiency in Untrusted Relational DBMSs. CCS03, Washington (2003) 27-31.
2. Iyer, B., Mehrotra, S., Mykletun, E., Tsudik, G. and Wu, Y.: A Framework for Efficient Storage Security in RDBMS. E. Bertino et al. (Eds.): EDBT 2004, LNCS 2992 (2004) 147-164.

3. Davida, G.I., Wells, D.L., and Kam, J.B.: A Database Encryption System with subkeys. ACM Trans. Database Syst. 6 (1981) 312-328.
4. Elovici, Y., Waisenberg, R., Shmueli, E., Gudes, E.: A Structure Preserving Database Encryption Scheme. SDM 2004, Workshop on Secure Data Management, Toronto, Canada, August, (2004).
5. Ramakrishnan, R and Gehrke, J.:Database Management Systems. McGraw-Hill (2000).
6. Spring, T.: Google Desktop Search: Security Threat? http://blogs.pcworld.com/staffblog/archives/000264.html, October, (2004);
7. Hacigms, H., Iyer, B., Li, C., and Mehrotra, S.: Executing SQL over encrypted data in the database-service-provider model. In Proc. of the ACM SIGMOD'2002, Madison, USA (2002).
8. Bayer, R. and Metzger, J.K.: On the Encipherment of Search Trees and Random Access Files. ACM Trans Database Systems, Vol. 1 (1976) 37-52.
9. Agrawal, R., Kiernan, J., Srikant, R. and Xu, Y.: Order Preserving Encryption for Numeric Data. In Proc. of the ACM SIGMOD'2004, Paris, France (2004).
10. Bouganim, L. and Pucheral, P.: Chip-secured data access: confidential data on untrusted servers. In Proc. of the 28th Int. Conference on Very Large Data Bases, Hong Kong, China (2002) 131-142.
11. Vingralek, R.: Gnatdb: A small-footprint, secure database system. In Proc. of the 28th Int'l Conference on Very Large Databases, Hong Kong, China, August (2002) 884-893.
12. Hore, B., Mehrotra, S. and Tsudik, G.: A Privacy Preserving Index for Range Queries. In Proc. of the 30th International Conference on Very Large Data Bases, Toronto, Canada(2004) 720-731.
13. Song, D.X., Wagner, D. and Perrig, A.: Practical Techniques for Searches on Encrypted Data. In Proc. of the 2000 IEEE Security and Privacy Symposium, May (2000).
14. Jermine, C. Datta, A. and Omiecinski, E.: A Novel Index Supporting High Volume Data Warehouse Insertions. In Proc. of the 25th Int. Conference on Very Large Data Bases, Edinburgh, Scotland (1999) 235-245.
15. Bertino, E. and Ferrari, E.: Secure and Selective Dissemination of XML Documents. ACM Transactions on Information and System Security Vol. 5 No. 3 (2002) 290-331.
16. Denning, D.E.: Cryptography and Data Security. Addison-Wesley, Massachusetts (1982).
17. Menezes, A., Van Oorschot, P. and Vanstone, S.: Handbook of Applied Cryptography. CRC Press (1996).
18. National Bureau of Standards. Data Encryption Standard. FIPS, NBS (1977).
19. Database Encryption in Oracle9i$^{\mathrm{TM}}$. An Oracle Technical White Paper (2001).