13

Different Search Strategies on Encrypted Data Compared

Richard Brinkman

University of Twente
The Netherlands

Summary. When private information is stored in databases that are under the control of others, the only possible way to protect it, is to encrypt it before storing it. In order to efficiently retrieve the data, a search mechanism is needed that still works over the encrypted data. In this chapter an overview of several search strategies is given. Some add meta-data to the database and do the searching only in the meta-data, others search in the data itself or use secret sharing to solve the problem. Each strategy has its own advantages and disadvantages.

13.1 Why Should We Search in Encrypted Data?

In a chapter about *searching in encrypted data* we should first ask ourselves the questions:

- Why should we want to protect our data using encryption?
- Why not use access control?
- Why should we want to search in encrypted data?
- Why not decrypt the data first and then search in it?

Access control is a perfect way to protect your data as long as you trust the access control enforcement. And exactly that condition often makes access control simply impossible.

Consider a database on your friend's computer. You store your data on his computer because he has bought a brand new large-capacity hard drive. Furthermore, he leaves his computer always on, so that you can access your data from everywhere with an Internet connection. You trust your friend to store your data and to make daily backups. However, your data may contain some information you do not want your friend to read (for instance, letters to your girlfriend). In this particular setting you cannot rely on the access control of your friend's database, because your friend has administrator privileges. He can always circumvent the access control or simply turn it off.

Fortunately, you read a book about cryptography a few years ago, and decide to encrypt all your sensitive information before storing it in the database.

Now you can use your friend's bandwidth and storage space without fearing that he is reading your private data.

Happy as you are, you keep on going storing more and more information. However, the retrieval of it gets harder and harder. In the situation before you encrypted your data you were used to send a precise query to the server and to retrieve only the information you needed. But in the current situation you cannot make the selection on the server. So, for each query you have to download the whole database and do the decryption and querying on your own computer. Since you have a slow Internet connection, you get tired of waiting for the download to finish. Of course, you can send your encryption key to your friend's database and ask it to do the decryption for you, but then you end up in almost the same situation as you started with. If the database can decrypt your data, your friend can read it.

Fortunately, there are mechanisms that solve the sketched problem. Some of the techniques will be explained in the remainder of this chapter.

13.2 Solutions

This section gives some solutions to the problem stated in the previous section. They all have one thing in common: the data are encrypted and stay encrypted for the time it resides on the server. The goal is to prevent the server (and everyone having access to it) from learning the data it is storing, the queries that are asked and the answers it gives back.

13.2.1 Index-Based Approach

Relational databases use tables to store the information. Rows of the table correspond to records and columns to fields. Often hidden fields or even complete tables are added to act as an index. This index does not add information; it is only used to speed up the search process. Hacıgümüş et al. [1, 2, 3] use the same idea to solve the problem of searching in encrypted data. To illustrate their approach we will use the example table shown in Table 13.1, which is stored on the server as shown in Table 13.2.

The first column of the encrypted table contains the encryptions of whole records. Thus $etuple = E(id, name, salary)$, where $E(\cdot)$ is the encryption

Table 13.1. Plain text *salary* table

id	*name*	*salary*
23	Tom	70000
860	Mary	60000
320	John	50000
875	Jerry	5600

Table 13.2. Encrypted *salary* table

etuple	id^S	$name^S$	$salary^S$
010101011 ...	4	28	10
000101101 ...	2	5	10
010111010 ...	8	7	2
110111101 ...	2	7	1

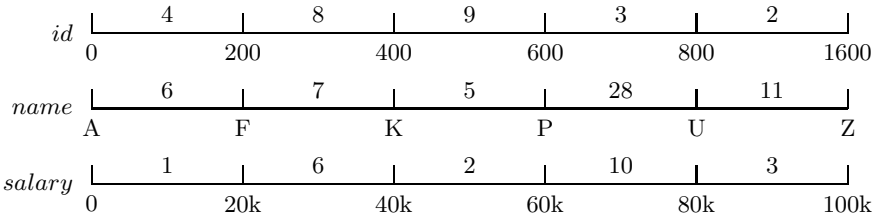


Fig. 13.1. Partitioning of the *id*, *name* and *salary* fields.

function. The extra columns are used as an index, enabling the server to prefilter records. The fields are named similar to the plain text labels, but are annotated with the superscript S which stands for server or secure. The values for these fields are calculated by using the partitioning functions drawn as intervals in Fig. 13.1. The labels of the intervals are chosen randomly. For example, consider John's salary. It lies in the interval $[40k, 60k\rangle$. This interval is mapped to the value two which is stored as the $salary^S$ field of John's record. It is the client's responsibility to keep these partitioning functions secret.

Querying the data is performed in two steps. First, the server tries to give an answer as accurately as it can. Second, the client decrypts this answer and postprocesses it. For this two-stage approach it is essential that the client splits a query Q into a server part Q^S (working on the index only) and a client part Q^C (which postprocesses the answer retrieved from the server). Several splittings are possible. The goal is to reduce the workload of the client and the network traffic. In order to have a realistic query example, let us first add a second table containing addresses to the database. The plain *address* table is shown in Table 13.3. It is stored encrypted on the server as shown in Table 13.4.

Table 13.3. Plain text *address* table

id	*street*
23	4th avenue
860	Owl street 4
320	Downing street 10
875	Longstreet 100

Table 13.4. Encrypted *address* table

etuple	id^S	$street^S$
110111100 ...	4	5
110111110 ...	2	2
000111010 ...	8	6
001110110 ...	2	3

As an example we choose the following SQL query:

```
SELECT street
FROM address, salary
WHERE address.id=salary.id AND salary<55000
```

SQL is a descriptive query language. It does not dictate the database *how* the result should be calculated (like a programming language does) only *what* the result should be. The database has freedom in the sequence of operations e.g., selection (σ), projection (π), join ($\bowtie$), etc. In this case the optimal evaluation is the one drawn in Fig. 13.2.

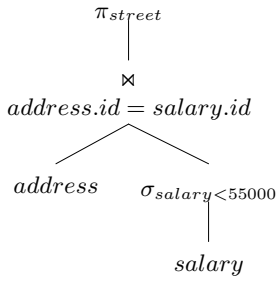


Fig. 13.2. Optimal query evaluation on uncrypted data

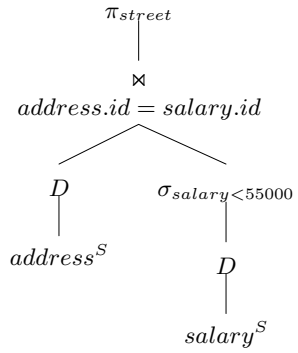


Fig. 13.3. Inefficient evaluation on encrypted data

The direct translation of the query tree to the encrypted domain is by simply decrypting the tables first (operation D) and then continuing with the standard evaluation (see Fig. 13.3). It clearly calculates the correct result but misses our goal of reducing network bandwidth and client computation. The operators should be pushed below the decryption operator D as much as possible. In Fig. 13.4 the selection on the salary is pushed below the decryption. Notice that the selection $\sigma^S_{salary^S \in \{1,6,2\}}$ also returns salaries between 55,000 and 60,000, so the client-side selection $\sigma_{salary<55000}$ cannot be left out. After the client selection is pulled above the join (not shown), the join can be pushed below the decryption as shown in figure 13.5.

The original strategy as described in [2] has two drawbacks: it cannot handle aggregate functions like SUM, COUNT, AVG, MIN and MAX very well and frequency analysis attacks are possible.

In a follow-up paper [4] the authors extend the method described in this section with privacy homomorphisms [5], allowing operations like addition and multiplication to work on encrypted data directly without the need to decrypt first.

The second drawback of the original method is dealt with by Damiani et al. [6]. Instead of using an encrypted invertable index, they use a hash function that is designed to have collisions. This way, an attacker has no certainty that two records are equal when they have the same index. This makes frequency analysis harder. As a down side, the efficiency drops when the security increases.

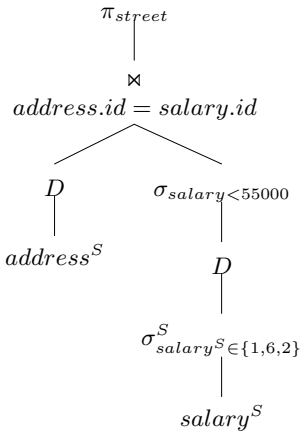


Fig. 13.4. Selection pushed down

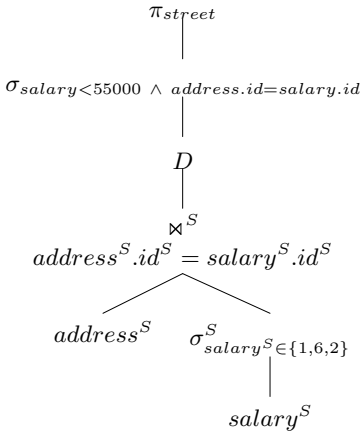


Fig. 13.5. Efficient evaluation on encrypted data

13.2.2 Search *in* the Encrypted Data

In contrast to the approach of Hacıgümüş et al, Song, Wagner and Perrig [7] do not need extra meta-data. In their approach the search is done in the encrypted data itself. They use a protocol that uses several encryption steps, which will be explained in this section.

Using the protocols described below, a client (Alice) can store data on the untrusted server (of Bob) and search in it, without revealing the plain text of either the stored data, the query or the query result. The protocol consists of three parts: storage, search and retrieval.

Storage

Before Alice can store information on Bob's server she has to do some calculations. First of all she has to fragment the whole plain text W into several fixed-sized words W_i. Each W_i has a fixed-length n. She also generates encryption keys k' and k'' and a sequence of random numbers S_i using a pseudo random generator. Then she has, or calculates, the following for each block W_i:

W_i	plain-text block
k''	encryption key
$X_i = E_{k''}(W_i) = \langle L_i, R_i \rangle$	encrypted text block
k'	key for f
$k_i = f_{k'}(L_i)$	key for F
S_i	ith random number
$T_i = \langle S_i, F_{k_i}(S_i) \rangle$	tuple used by search
$C_i = X_i \oplus T_i$	value to be stored ($\oplus$ stands for xor)

where E is an encryption function, L_i and R_i are the left and right parts of X_i and f and F are keyed hash functions:

$$E : key \times \{0,1\}^n \rightarrow \{0,1\}^n$$
$$f : key \times \{0,1\}^{n-m} \rightarrow key$$
$$F : key \times \{0,1\}^{n-m} \rightarrow \{0,1\}^m$$

The encrypted word X_i has the same block length as W_i (i.e. n). L_i has length $n-m$ and R_i has length m. The parameters n and m may be chosen freely ($n > 0$, $0 < m \leq \frac{n}{2}$). The value C_i can be sent to Bob for storage. Alice may now forget the values W_i, X_i, L_i, R_i, k_i, T_i and C_i, but should still remember k', k'' and S_i (or the seed to regenerate S_i).

Search

After the encrypted data is stored by Bob in the previous phase, Alice can query Bob's server. Alice provides Bob with an encrypted version of a plain-text word W_j and asks him if and where W_j occurs in the original document. Note that Alice does not have to know the position j. If W_j was a block in the original data then $\langle j, C_j \rangle$ is returned. Alice has or calculates:

k''	encryption key
k'	key for f
W_j	plain-text block to search for
$X_j = E_{k''}(W_j) = \langle L_j, R_j \rangle$	encrypted block
$k_j = f_{k'}(L_j)$	key for F

Then Alice sends the value of X_j and k_j to Bob. Having X_j and k_j Bob is able to compute for each C_p:

$$T_p = C_p \oplus X_j = \langle S_p, S'_p \rangle$$
$$\text{IF } S'_p = F_{k_j}(S_p) \text{ THEN RETURN } \langle p, C_p \rangle$$

If $p = j$ then $S'_p = F_{k_j}(S_p)$, otherwise S'_p is garbage. Note that all locations with a correct T_p value are returned. However there is a small chance that T satisfies $T = \langle S_q, F_{k_j}(S_q) \rangle$ but where $S_q \neq S_p$. Therefore, Alice should check for each answer whether the correct random value is used or not.

Retrieval

Alice can also ask Bob for the cipher text C_p at any position p. Alice, knowing k', k'' and S_i (or the seed to generate it), can recalculate W_p by

p	desired location
$C_p = \langle C_{p,l}, C_{p,r} \rangle$	stored block
S_p	random value
$X_{p,l} = C_{p,l} \oplus S_p$	left part of encrypted block
$k_p = f_{k'}(X_{p,l})$	key for F
$T_p = \langle S_p, F_{k_p}(S_p) \rangle$	check tuple
$X_p = C_p \oplus T_p$	encrypted block
$W_p = D_{k''}(X_p)$	plain text block

where D is the decryption function $D : key \times \{0,1\}^n \to \{0,1\}^n$ such that $D_{k''}(E_{k''}(W_i)) = W_i$.

This is all Alice needs. She can store, find and read the text while Bob cannot read anything of the plain text. The only information Bob gets from Alice is C_i in the storage phase and X_j and k_j in the search phase. Since C_i and X_j are both encrypted with a key only known to Alice and k_j is only used to hash one particular random value, Bob does not learn anything about the plain text. The only information Bob learns from a search query is the location where an encrypted word is stored.

However, the protocol has two drawbacks:

- The splitting of the plain text into fixed-sized words is not natural, especially not for human languages.
- The search time complexity is linear in the length of the whole data. It does not scale up to large databases.

Both drawbacks are solved by Brinkman et al. [8]. They use XML as a data format and exploit its tree structure to get a logarithmic search complexity.

Waters et al. [9] use a similar technique, which is based on [7], to secure audit logs. Audit logs contain detailed and probably sensitive information about past execution. It should therefore be encrypted. Only when there is a need to find something in the encrypted audit log, a trusted party can generate a trapdoor for a specific keyword. Boneh et al. [10] use a different trapdoor strategy to achieve the same goal.

13.2.3 Using Secret Sharing

A third solution to our problem uses secret sharing [11, 12]. In this context, sharing a secret does not mean that several parties know the same secret. In cryptography secret sharing means that a secret is split over several parties such that no single party can retrieve the secret. The parties have to collaborate in order to retrieve the secret.

Secret sharing can be very simple. To share, for instance, the secret value 5 over 3 parties a possible split can be 12, 4 and 26. To find the value back all three parties should collaborate and sum their values modulo 37 ($5 \equiv 12 + 4 + 26 \pmod{37}$).

The database scheme described in this section uses the idea of secret sharing to accomplish the task of storing data such that you need both the server

and the client to collaborate in order to retrieve the data. Further requirements are:

- The server should not benefit from the collaboration. Its knowledge about the data should not increase (much) during the collaboration.
- The data split should be unbalanced, meaning that the server share is heavier (in terms of storage space) than the client share.

Encoding

A plain text XML document is being transformed into an encrypted database by following the steps below. See Fig. 13.6 for the encoding of a concrete example.

1. Define a function $map : node \rightarrow \mathbb{F}_p$, which maps the tag names of the nodes to values of the finite field $\mathbb{F}_p$, where p is a prime that is larger than the total number of different tag names (Fig. 6(b)).
2. Transform the tree of tag names (Fig. 6(a)) into a tree of polynomials (Fig. 6(d)) of the same structure where each *node* is transformed to $f(node)$ where function $f : node \rightarrow \mathbb{F}_p[x]/(x^{p-1}-1)$ is defined recursively:

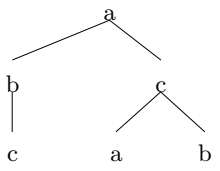
$$f(node) = \begin{cases} x - map(node) & \text{if } node \text{ is a leaf node} \\ (x - map(node)) \prod_{d \in child(node)} f(d) & \text{otherwise} \end{cases}$$

   Here $child(node)$ returns all children of a *node*.
3. Split the resulting tree into a client (Fig. 6(e)) and a server tree (Fig. 6(f)). Both trees have the same structure as the original one. The polynomials of the client tree are generated by a pseudo-random generator. The polynomials of the server tree are chosen such that the sum of a client node and the corresponding server node equals the original polynomial.
4. Since the client tree is generated by a pseudo-random generator it suffices to store the seed on the client. The client tree can be discarded. When necessary, it can be regenerated using the pseudo-random generator and the seed value.

Retrieval

It is simple to check whether a node n is stored somewhere in a subtree by evaluating the polynomials of both the server and the client at $map(n)$. If the sum of these evaluations equals zero, this means that n can be found somewhere in the subtree n. To find out whether n is the root node of this subtree, you have to divide the unshared polynomial by the product of all its direct children. The result will be a monomial $(x - t)$ where t is the mapped value of the node.

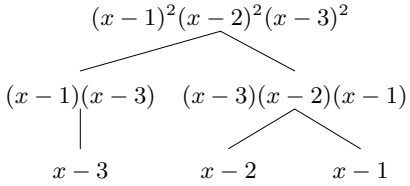
In a real query evaluation you start at the XML root node and walk downwards until you encounter a dead branch. Whether you choose to traverse



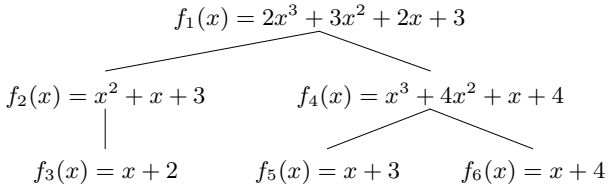
(a) XML example

name	value
a	2
b	1
c	3

(b) Mapping function

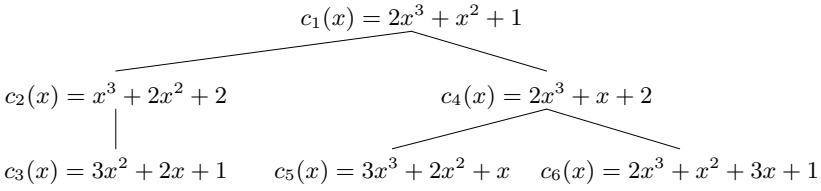


(c) Unshared, unreduced encoding



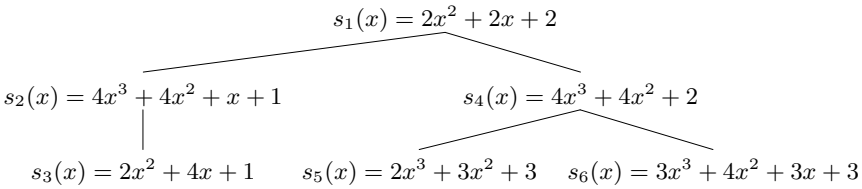
=

(d) Unshared, reduced encoding



+

(e) Client encoding



(f) Server encoding

Fig. 13.6. The mapping function (b) maps each name of an input document (a) to an integer. The XML document is first encoded to a tree of polynomials (c) before it is reduced to the finite field $\mathbb{F}_5[x]/(x^4 - 1)$ (d) and split into a client (e) and a server (f) part.

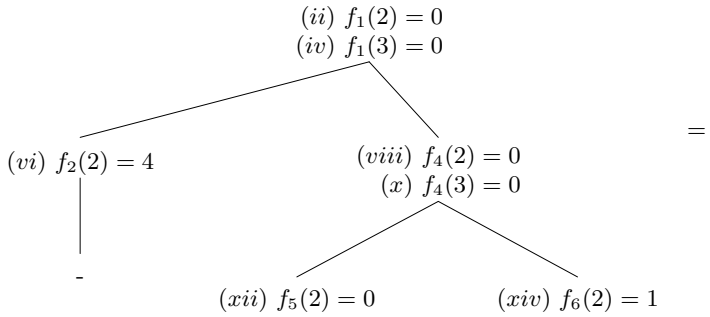
the tree depth- or breadth-first, the strategy remains the same: try to find dead branches as early as you can. Fortunately, each node contains information about all the subnodes. Therefore, it is almost always the case that you find dead branches (where the unshared evaluation return a nonzero value) before reaching the leaves.

To illustrate the search process we will follow the execution run with the example query `//c/a`. This XPath query should be read as: start at the root node, go one or more steps down to all `c` nodes that have an `a` node as child. The roman numbers in Fig. 13.7 correspond to the following sequence of operations:

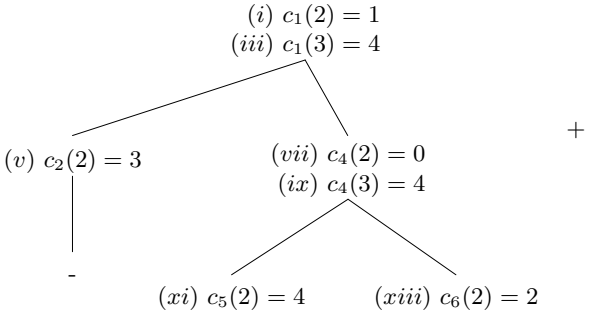
(*i*) We start the evaluation process at the root nodes of the server and the client. In parallel, they can substitute the values in the root polynomials. Both $s_1(map(\texttt{c})) = s_1(3)$ and $s_1(map(\texttt{a})) = s_1(2)$ should be evaluated, but it does not matter in which order (analogously for $c_1(\cdot)$). To mislead the server we choose to evaluate first the `a` nodes and then the `c` node, although the query suggests otherwise.
(*ii*) Each time the server has substituted a value for x in one of its polynomials, it sends the result to the client, which can add the server result to its own. In this example $f_1(2) = c_1(2) + s_1(2) = 1 + 4 = 0$, which means that either the original root node was `a` or the root node has a descendant `a`.
(*iii*) The next task is to check that the root node is or contains `c`.
(*iv*) $f_1(3) = 0$. Now we know that the root node contains both `a` and `c`, a prerequisite of our query. Thus, we proceed one step down in the tree.
(*v*) The left child is checked for `a`.
(*vi*) This time $f_2(2) = 4 \neq 0$. Thus the left subtree does not contain an `a` node. Apparently this is a dead branch. It is not even necessary to check for a `c` node; the query `//c/a` can never hold in this branch. We can stop evaluating it and backtrack to the right subtree.
(*vii*) In the right subtree we start checking for a `c` node.
(*viii*) Since $f_4(2) = 0$, the right subtree seems promising.
(*ix*) Therefore we also check for an `a` node.
(*x*) The right tree still seems promising so we walk one level down.
(*xi*) Since the client knows the structure of the tree (if not, he can ask the server for it), he knows that we have reached a leaf node. Therefore, it is unnecessary to check for a `c` node.
(*xii*) Since this is a leaf node and $f_5(2) = 0$ we now know for sure that node 5 *is* an `a` node.
(*xiii*) The rightmost leaf node is also checked for an `a` node.
(*xiv*) But it is not.

Until now, we have two possible matches:

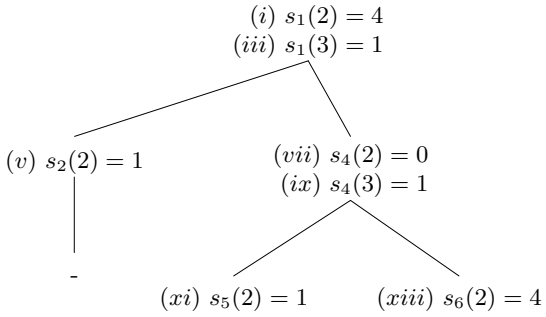
1. node 1 matches `c` and node 4 matches `a`
2. node 4 matches `c` and node 5 matches `a`



(a) Unshared evaluation



(b) Client evaluation



(c) Server evaluation

Fig. 13.7. Evaluation process of the query `//c/a` using the same mapping function and data encoding as in Fig. 13.6. The roman numbers indicate the sequence of operations.

It is sufficient to check the exact value of node 4 only. If this node *is* a `c` node then solution 1 holds, if this node *is* an `a` node solution 2 holds. If it is neither then there are no matches. The exact value of a node n can be found in two different ways:

- Ask the server for the polynomial $s_n(x)$ and the polynomials of all its children (let us name them $s_n^{(1)}(x), \ldots, s_n^{(k)}(x)$). In the mean time calculate $c_n(x)$ and its children $c_n^{(1)}(x), \ldots, c_n^{(k)}(x)$. The exact value can be calculated by dividing $f_n(x)$ by $\prod_{i=1}^{k} f_n^{(i)}(x)$. The result will be a monomial $x - t$ where t is the node's value.
- If $f_n(a) = 0$ for some value a and for all children i of n, $f_i(a) \neq 0$ then you know that node n *is* a. Note that for recursive document type definitions (such as our example) there is no guarantee that this method works.

13.3 Solutions Compared

Having seen three different ways to query encrypted data, one may ask which one is the best. This is not easy to answer, since each has its own advantages and disadvantages. It depends on the requirements which one is the most appropriate.

13.3.1 Index-Based Approach

Advantages

The index-based solutions uses a relational database as back-end. Since relational databases have been around for quite some time, there exists a huge theoretical background including all kinds of indexing mechanisms and even its own relational algebra. Hacıgümüş takes advantage of this to create an efficient solution, pushing as much of the workload to the server.

Disadvantages

This efficiency comes at a price, though. The storage cost doubles compared to the plain text case. Apart from the encrypted data the hash values for each searchable field are also stored. These hashes are almost as big as the original values.

Another disadvantage is the fact that the server can link records together without the cooperation of the client. Values that are equal in the plain text domain are also equal in the encrypted domain. Although the opposite does not hold, the server still learns which records are not the same. Therefore, it can estimate the number of different values or it can join tables fairly accurately.

A more practical disadvantage is that the user should choose the hash map in such a way that the intervals do not get too big or too small. The hash map strongly depends on the distribution of the plain text values. When the distribution changes drastically, the hash map should also be redesigned.

13.3.2 Search *in* the Encrypted Data

Advantages

The encryption method of Song et al. does not need a larger storage space than in the plain text case.

When a word occurs multiple times, the encryptions are different, which makes frequency analysis impossible.

Almost the whole workload is done at the server site. Only the encryption of the keyword and a single hash operation are performed at the client. This fact makes this strategy especially useful for lightweight devices like mobile phones.

Disadvantages

Song's strategy may be efficient when you only look at storage space, but it is not when looking at computation time. For each query all the data are searched linearly. Thus this strategy does not scale well. Brinkman et al. [8] reduce the computation time from linear to logarithmic by using more structured (trees) data input. Unfortunately, this also increases the communication from constant to logarithmic time. They also drop the requirement for fixed-sized keywords, which is another disadvantage of the original scheme.

13.3.3 Using Secret Sharing

Advantages

The main advantage of the secret sharing strategy is its security. Since all the data stored on the server are randomly generated, it is just worthless garbage for an attacker. Even the same nodes are encrypted differently.

Another advantage is the efficient storage. Although knowledge about the whole subtree is stored at each node, the storage remains similar in size to the plain text.

Disadvantages

A disadvantage, though, is the communication costs. Each node that is being traversed costs a round-trip communication (with very little data) between the client and the server. Also the workload on the client is similar to the workload on the server.

References

1. H. Hacıgümüş, Balakrishna R. Iyer, Chen Li, and Sharad Mehrotra. Executing SQL over encrypted data in the database service provider model. In *SIGMOD Conference*, 2002.
2. H. Hacıgümüş, B. Iyer, and S. Mehrotra. Efficient execution of aggregation queries over encrypted relational databases. In *Proc. of the 9th International Conference on Database Systems for Advanced Applications*, Jeju Island, Korea, March 2004.
3. H. Hacıgümüş, B. Iyer, C. Li, and S. Mehrotra. SSQL: Secure SQL in an insecure environment. *VLDB journal*, 2006.
4. Hakan Hacıgümüş, Bala Iyer, and Sharad Mehrotra. Efficient execution of aggregation queries over encrypted relational databases. In YoonJoon Lee, Jianzhong Li, Kyu-Young Whang, and Doheon Lee, editors, *Database Systems for Advanced Applications: 9th International Conference, DASFAA 2004*, volume LNCS 2973, pages 125–136, Jeju Island, Korea, March 2003. Springer Verlag.
5. Josep Domingo-Ferrer and Jordi Herrera-Joancomartí. A privacy homomorphism allowing field operations on encrypted data. *Jornades de Metemàtica Discreta i Algorísmica*, march 1998.
6. E. Damiani, S. De Capitani di Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati. Balancing confidentiality and efficiency in untrusted relational DBMSs. In *Proc. of the 10th ACM Conference on Computer and Communications Security*, pages 93–102, Washington, DC, USA, October 2003. ACM Press New York, NY, USA.
7. Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *IEEE Symposium on Security and Privacy*, pages 44–55, 2000.
8. R. Brinkman, L. Feng, J. M. Doumen, P. H. Hartel, and W. Jonker. Efficient tree search in encrypted data. *Information Systems Security Journal*, 13(3):14–21, July 2004.
9. B. Waters, D. Balfanz, G. Durfee, , and D. K. Smetters. Building an encrypted and searchable audit log. In *Network and Distributed Security Symposium (NDSS) '04*, San Diego, California, 2004.
10. D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *Proceedings of Eurocrypt*, pages 506–522, 2004.
11. R. Brinkman, J. M. Doumen, P. H. Hartel, and W. Jonker. Using secret sharing for searching in encrypted data. In W. Jonker and M. Petković, editors, *Secure Data Management VLDB 2004 workshop*, volume LNCS 3178, pages 18–27, Toronto, Canada, August 2004. Springer-Verlag, Berlin.
12. R. Brinkman, B. Schoenmakers, J. M. Doumen, and W. Jonker. Experiments with queries over encrypted data using secret sharing. In W. Jonker and M. Petković, editors, *Secure Data Management VLDB 2005 workshop*, volume LNCS 3674, pages 33–46, Trondheim, Norway, Sep 2005. Springer-Verlag, Berlin.