

# Efficient Tree Search in Encrypted Data

R. Brinkman, L. Feng, J. Doumen, P.H. Hartel, and W. Jonker

University of Twente, Enschede, the Netherlands  
{brinkman,ling,doumen,pieter,jonker}@cs.utwente.nl

**Abstract.** Sometimes there is a need to store sensitive data on an untrusted database server. Song, Wagner and Perrig have introduced a way to search for the existence of a word in an encrypted textual document. The search speed is linear in the size of the document. It does not scale well for a large database. We have developed a tree search algorithm based on the linear search algorithm that is suitable for XML databases. It is more efficient since it exploits the structure of XML. We have built prototype implementations for both the linear and the tree search case. Experiments show a major improvement in search time.

## 1 Introduction

Nowadays the need grows to store data securely on an untrusted system. Think, for instance, of a remote database server administered by somebody else. If you want your data to be secret, you have to encrypt it. The problem then arises how to query the database. The most obvious solution is to download the whole database locally and then perform the query. This of course is terribly inefficient. Song, Wagner and Perrig [1] have introduced a protocol to search for a word in an encrypted text. We will summarise this protocol in section 2.

In this paper we propose a new protocol that is more suitable for handling large amounts of semi-structured XML data. This new protocol exploits the XML tree structure. XPath queries can be answered fast and secure.

We have built prototype implementations for both the linear and the tree search protocol (section 3). We use these prototypes to find optimal settings for the parameters used within the protocols and to show the increase in search speed by using the tree structure. We did some experiments (section 4) for which the results can be found in section 5.

## 2 Search Strategy

Before we describe our tree search strategy (section 2.2) we will give a short summary of the original linear search strategy of Song, Wagner and Perrig [1].

### 2.1 Linear Search Strategy for Full Text Documents

Song, Wagner and Perrig [1] describe a protocol to store sensitive data on an untrusted server. A client (Alice) can store data on the untrusted server (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 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 $F$
$S_i$	random number $i$
$T_i = \langle S_i, F_{k_i}(S_i) \rangle$	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 and  $f$  and  $F$  are keyed hash functions:

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

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 ( $n > 0$ ,  $0 < m \leq \frac{n}{2}$ ). Section 5.1 gives guidelines for efficient values of  $n$  and  $m$ . 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  $S_i$ .

**Search** After the encrypted data is stored by Bob in the previous phase Alice can query Bob. 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  $C_p$ :

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

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 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 the seed for  $S$ , 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  $D : key \times \{0, 1\}^n \rightarrow \{0, 1\}^n$  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 store 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 of the plain text. The only information Bob learns from a search query is the location where an encrypted word is stored.

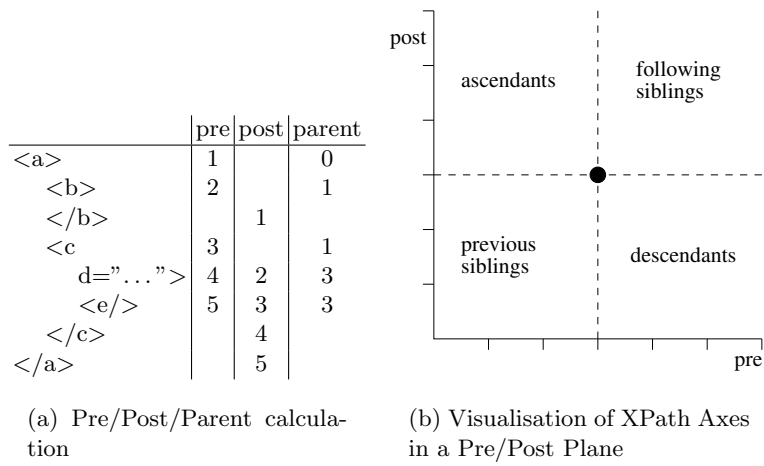
## 2.2 Tree Search Strategy for XML Documents

So far, we considered only text files. Using structured XML data can improve efficiency.

Torsten Grust [2, 3] introduces a way to store XML data in a relational database such that search queries can be handled efficiently. An XML document is translated into a relational table with a predefined structure. Each record consists of the name of the tag or attribute and its corresponding value. The information about the tree structure of the original XML document is captured in the pre, post and parent fields. All fields can be computed in a single pass over the XML document. The pre and post fields are sequence numbers that count the open tags respectively the close tags. The parent value is the pre value of the parent element (see figure 1(a)).

The XPath axes like *descendant*, *ascendant*, *child*, etc can be expressed as simple expressions over the pre, post and parent fields. For instance:

- $v$  is a child of  $v' \iff v.parent = v'.pre$
- $v$  is a descendant of  $v' \iff v'.pre < v.pre \wedge v'.post > v.post$
- $v$  is following  $v' \iff v'.pre < v.pre \wedge v'.post < v.post$



**Fig. 1.** Calculation and Usage of Pre, Post and Parent fields

Some XPath axes can also be drawn in a pre/post plane (see figure 1(b)). Each element can be drawn as a dot in the graph. The solid circle indicates just one of them. Taking the solid circle as starting element, the quadrants indicate where its ascendants, descendants and siblings are located.

Not all updates are efficient. Modification and deletion are no problem, but element insertion causes the need to recalculate the pre, post and parent values for all following elements. The number of recalculations can be reduced by an initial sequence with a larger step (100, 200, 300, ...).

Torsten Grust aims at storing XML data in the clear. To protect the data cryptographically we combine his strategy with the linear search approach of Song, Wagner and Perrig (SWP) [1]. Only some slight modifications to the SWP approach are necessary:

1. The input file is not an unstructured text file but a tree structured XML document. The division of the data into fixed sized blocks does not seem natural. Therefore, we use variable block lengths that depend on the lengths of the tag names, attribute names, attribute values and the text between tags.
2. The sequence number of a block is no longer appropriate to define the location within a document. We use the pre value instead.

The equations of section 2.1 can be rewritten to the equations below. Note that all subscripts have changed. For simplicity we only describe the encryption of tag names. Exactly the same scheme is used for attribute names (prefixed with a @ sign) or the data itself by simply substituting value for tag.

## Storage

$W_{tag}$	plain text block
$k''$	encryption key
$X_{tag} = E_{k''}(W_{tag}) = \langle L_{tag}, R_{tag} \rangle$	encrypted text block
$k'$	key for $f$
$k_{tag} = f_{k'}(L_{tag})$	key for $F$
$S_{pre}$	random number $pre$
$T_{pre,tag} = \langle S_{pre}, F_{k_{tag}}(S_{pre}) \rangle$	tuple used by search
$C_{pre,tag} = X_{tag} \oplus T_{pre,tag}$	value to be stored

Note that the random value  $S_{pre}$  does not depend on the tag name but on the location (expressed in the pre field) because all elements with the same tag name should be stored differently.

**Search** An XPath query like  $/tag_1//tag_2[tag_3 = "value"]$  is encrypted to  $/\langle X_{tag_1}, k_{tag_1} \rangle // \langle X_{tag_2}, k_{tag_2} \rangle [ \langle X_{tag_3}, k_{tag_3} \rangle = \langle X_{value}, k_{value} \rangle ]$  before sending it to the server. The server calculates the result traversing the XPath query from left to right. Each step consists of two or three sub steps:

- Evaluating the XPath axis  $/$ ,  $//$ ,  $[$  and  $]$  using the pre, post and parent fields. It is possible to find all children ( $/$ ) or all descendants ( $//$ ) of elements found in a previous step by just using the pre, post and parent field. See section 3.2 for an example.
- Filtering out the records that do not satisfy  $S'_p = F_{k_{tag}}(S_p)$  in  $T_{p,tag} = C_{p,tag} \oplus X_{tag} = \langle S_p, S'_p \rangle$ .
- Eventually filtering out the records with an incorrect value field.

## Retrieval

$k'$	key for $f$
$k''$	encryption key
$pre$	desired location
$C_{pre,tag} = \langle C_{pre,tag,l}, C_{pre,tag,r} \rangle$	stored block
$S_{pre}$	random value
$X_{tag,l} = C_{pre,tag,l} \oplus S_{pre}$	left part of encrypted block
$k_{tag} = f_{k'}(X_{tag,l})$	key for $F$
$T_{tag} = \langle S_{pre}, F_{k_{tag}}(S_{pre}) \rangle$	check tuple
$X_{tag} = C_{pre,tag} \oplus T_{tag}$	encrypted block
$W_{tag} = D_{k''}(X_{tag})$	plain text block

## 3 Implementation

For each search strategy a prototype has been developed. Each prototype consists of two tools: one for encryption and one for searching. All tools use the standard crypto packages shipped with JDK 1.4.

### 3.1 Linear Search Prototype

Section 2.1 introduces three functions:  $E$ ,  $f$  and  $F$ .  $E$  should be a block cipher in ECB mode and  $f$  and  $F$  keyed hash functions. For our prototype we chose

DES for all three of them.  $E$  is exactly DES in ECB mode. Since DES works on blocks of 64 bits  $n$  should be a multiple of 64 bits.

$f$  and  $F$  are keyed hash functions with variable sized hash values. Standard hash functions like SHA-1 have a fixed sized hash value. It is possible to use the last (or the first)  $m$  bits of the hash value, but then  $m$  should be less than the size of the hash value (160 bits for SHA-1). To allow a larger value for  $m$  our prototype uses DES in CBC mode. To hash a data block of length  $n - m$  to a hash value of length  $m$  the block is encrypted with the specified key (56 bits DES key) but only the last  $m$  bits are used as hash value. The only restriction for  $m$  is that  $n - m \geq m$  and thus  $n \geq 2m$ . See Menezes et al [4] for a more detailed description of the used hash algorithm.

The search algorithm implements the protocol described in [1] as summarised in section 2.1. The program takes the whole cipher text along with the query as input and produces the  $\langle i, C_i \rangle$  pairs as output.

### 3.2 Tree Search Prototype

Like the linear prototype the tree search prototype is split into two parts: one for encryption and one for searching.

The Encrypt tool uses a SAX parser to read the input XML document. In one pass over the input, the pre, post and parent values can be calculated. When an end tag is encountered all the information to encrypt the element is available. Attributes are handled as tags with a leading @ sign. A new record  $\langle pre, post, parent, C_{pre,tag}, C_{pre,value} \rangle$  is inserted into the relational database, where  $C_{pre,tag}$  and  $C_{pre,value}$  are calculated as in section 2.2. In our prototype we use a MySQL database to store the encrypted document.

In contrast with the linear prototype there are no predefined block sizes  $n$  and  $m$ . Instead of using a fixed sized block,  $n$  is simply set to the length of the tag name.  $m$  is a predefined fraction of  $n$  (for example 0.5).

In order to speed up the search process, indices are added to the MySQL table for the pre, post and parent fields.

The XPath expression is evaluated step by step. Preliminary results are stored in a result table. Each step consists of two or three sub steps:

1. Carry out the path delimiter ( $/$ ,  $//$ ,  $[$  or  $]$ ). For this step only the pre, post and parent fields are needed. For example  $//$  (descendants) is translated into the SQL query:

```
CREATE TABLE new_result
SELECT data.*
FROM data, previous_result
WHERE data.pre > previous_result.pre AND
      data.post < previous_result.post
```

2. Filter out the records in the preliminary result with the wrong tag/attribute names. In this step we use the original linear search method.
3. When the step consists of an equation expression the previous step is repeated but now for the value instead of the name.

## 4 Experimental Data

The two prototypes give us the opportunity to experiment with the parameters used in the protocol and, more importantly, compare the linear search approach with the tree search approach. We are especially interested in the influence the approach and the parameters  $n$  and  $m$  have on the encryption and search speed. We used the XML benchmark<sup>1</sup> [5] to generate three sample XML files of sizes 1 MB, 10 MB and 100 MB. Although the linear approach does not use the structure of these XML files the benchmark is used in both cases to compare the results with the tree search approach.

Also the number of collisions has been measured (see figure 2(a)). Collisions are the false hits that occur because of the collisions in the hash function  $F$ .  $F$  hashes the random value  $S_i$  of size  $n - m$  to a hash value of length  $m$ , where  $n - m \geq m$ . Therefore collisions are unavoidable (collisions are avoidable when  $n - m = m$  and  $F$  is bijective, but bijective functions are not good hash functions).

### 4.1 Experiments with the Linear Search Prototype

For the linear prototype both  $n$  and  $m$  may be chosen freely. Tests are carried out  $\forall n \in \{8, 16, 24, 32, 40, 48, 56, 64\}$  where these values are the number of bytes and not bits. Because we use DES in ECB mode for the encryption function  $E$ , we only use multiples of 8 bytes.  $m$  should be less than or equal to  $\frac{n}{2}$  so  $m \in \{1, 2, \dots, \frac{n}{2}\}$  (also in bytes). Measurement results of the 100 MB case are plotted in figure 2(b). Tests with data inputs of 1 MB and 10 MB showed that the number of collisions, the search and the encryption times are proportional to the data size. In our technical report [6] more experimental data is provided. All tests were carried out on a Pentium IV 2.4 MHz with 512 MB memory.

For the search query a word guaranteed to be in at least one location was chosen. The search engine does not stop when one occurrence is found; all the text is scanned for each query.

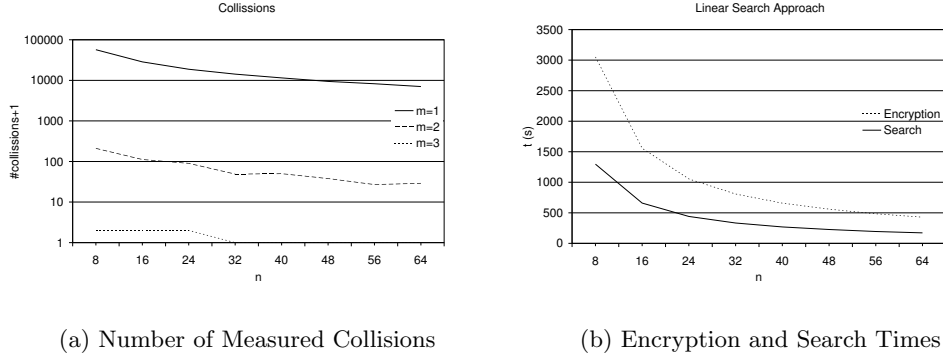
### 4.2 Experiments with the Tree Search Prototype

For the tree search prototype the only configurable parameters are  $m$  and the data size. The block length  $n$  depends on the tag names and values. Encryption tests are carried out on the same XML documents as in the linear prototype. In this case  $m$  is relative to  $n$ ;  $m \in \{0.1, 0.2, 0.3, 0.4, 0.5\}$ . The encryption times for the 1 MB, 10 MB and the 100 MB files were 21.5, 188 and 1195 s and did not depend on  $m$ .

Search tests were carried out with a fixed  $m = 0.5$  because  $m$  does not seem to have much influence. Some queries are shown in table 1. Also the number of elements in the result is shown for each query. All three files have approximately the same tree depth but have different branch factors (average number of sub children per element).

---

<sup>1</sup> <http://www.xml-benchmark.org>



**Fig. 2.** Measurement Results of Linear Search Prototype for the 100 MB Case

**Table 1.** Search Times Calculated for Search Queries with Different Depth and Branch Factor

t (ms)	t (ms)	t (ms)	query	count	count	count
1 MB	10 MB	100 MB		1 MB	10 MB	100 MB
1281	1506	1285	/site	1	1	1
1266	1380	1321	/site/regions	1	1	1
1358	1435	1342	/site/regions/asia	1	1	1
1409	1687	2464	/site/regions/asia/item	20	200	2000
1518	2030	4135	/site/regions/asia/item/description	20	200	2000
1376	1591	2442	/site/regions/africa/item/description	5	55	550
1448	2777	9059	/site/regions/europe/item/description	60	600	6000
1455	2098	4577	/site/regions/australia/item/description	22	220	2200
1654	3226	13672	/site/regions/namerica/item/description	100	1000	10000
1336	1817	3028	/site/regions/samerica/item/description	10	100	1000
1398	2382	18530	//*	21048	206130	2048180
3639	21775	191899	//item	217	2175	21750

## 5 Analysis of the Results

First we will analyse the results of the individual experiments in the first two subsections. In subsection 5.3 we will compare the linear search approach with the tree search approach.

### 5.1 Results from the Linear Search Approach

From the linear search prototype we can conclude the following:

- As expected the larger the dataset the larger the encryption and search times. Encryption and search times grow linear in the size of the dataset. Therefore the protocol does not scale well and can only be used for reasonable small databases.

- The larger  $n$  is the shorter the encryption and search times gets (figure 2(b)). This can be explained by looking at the number of blocks. The larger  $n$  is the fewer blocks there are. For each block a fixed number of steps is taken. Most of these steps do not depend on the length of the blocks. Therefore less time is needed for the whole database.
- Searching is faster than encryption, because fewer operations have to be calculated for each block.
- The larger  $n$  is the fewer collisions occur (figure 2(a)). This can also be explained by the fewer blocks.
- For a fixed value of  $n$  the encryption and search times hardly depend on the value of  $m$ .
- Collisions can be avoided by choosing a sufficiently large value of  $m$ . The largest value for  $m = \frac{n}{2}$  which is also the most optimal one. But also for  $m > 2$  the number of collisions is negligible.

## 5.2 Results from the Tree Search Approach

From the tree search prototype we can conclude that:

- The encryption time is linear in the size of the input.
- The search time depends both on the structure of the XML document and the search query. The search time is of order  $O(p)$  where  $p$  is the number of elements to be read. For queries without `//` this comes down to  $O(bd)$  where  $b$  is the branch factor (the average number of sub elements) and  $d$  is the depth in the tree where the answer is found.

## 5.3 Benefits of using Tree Structure

From the experiments with the linear search method we know that the encryption time depends on the block size. Therefore, to make a fair comparison between the linear text search and the tree search, we have to take into account the block size of the tree search method. We analysed the XML documents and found the data shown in table 2.

Comparison of the encryption speed in the tree search case (with an average block size of around 18) with the linear case, shows that the tree encryption is slightly faster than in the linear case. The reason for this is that there is no need to encrypt the close tag.

**Table 2.** Block Sizes

data size	avg tag length	standard deviation	avg text size	standard deviation	avg all blocks	standard deviation
1 MB	9.8	3.4	28.0	70	18.2	48
10 MB	9.8	3.4	28.6	70	18.4	48
100 MB	9.8	3.4	28.9	70	18.6	49

The major benefit of using the tree structure is the increase in search speed. Only a small part of the whole tree has to be searched. Because the search time totally depends on the data and the query, a straight comparison between the linear and the tree case is impossible. However, linear search is of order  $O(n) = O(b^d)$ , whereas tree searching is of order  $O(bd)$ .

## 6 Conclusions

We have implemented a prototype for the theory described in [1]. We showed that the search complexity is linear in the size of the text. We have defined a new protocol for semi-structured XML data that exploits the tree structure. Experiments with the implementations of both protocols showed that the encryption speed remains linear in the size of the input, but that a major improvement in the search speed can be achieved.

## References

1. 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. <http://citeseer.nj.nec.com/song00practical.html>.
2. Torsten Grust. Accelerating xpath location steps. In *Proceedings of the 21st ACM International Conference on Management of Data (SIGMOD 2002)*, pages 109–120. ACM Press, Madison, Wisconsin, USA, June 2002. <http://www.informatik.uni-konstanz.de/~grust/files/xpath-accel.pdf>.
3. Torsten Grust, Maurice van Keulen, and Jens Teubner. Staircase join: Teach a relational dbms to watch its (axis) steps. In *Proceedings of the 29th Int'l Conference on Very Large Databases (VLDB 2003)*, Berlin, Germany, Sep 2003. <http://citeseer.nj.nec.com/593676.html>.
4. Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, October 1996. <http://www.cacr.math.uwaterloo.ca/hac/>.
5. A. Schmidt, F. Waas, M. Kersten, D. Florescu, I. Manolescu, M. Carey, and R. Busse. The xml benchmark project. Technical Report INS-R0103, CWI, April 2001. <http://citeseer.ist.psu.edu/schmidt01xml.html>.
6. R. Brinkman, L. Feng, S. Etalle, P. H. Hartel, and W. Jonker. Experimenting with linear search in encrypted data. Technical report TR-CTIT-03-43, Centre for Telematics and Information Technology, Univ. of Twente, The Netherlands, Sep 2003. <http://www.ub.utwente.nl/webdocs/ctit/1/000000d9.pdf>.