Search in Encrypted Data Project

Secure Data Management 2011-2012

Peter Wagenaar s0113565
Tjeerd Boerman s0118400

Table of Contents

1 Introduction

This report describes the prototype system that was the end result of the “Search in Encrypted Data” assignment of the Secure Data Management course. Its design and implementation took place during the fall of 2011. With this document – and the full source code that goes with it – we intend to give the reader an overview of the system, the algorithms at the heart of it and the choices that motivated its design.

For the assignment we consider a scenario of a financial consultant that uses a cloud storage service to store the financial data of his clients. The cloud storage server is considered to be honest but curious, meaning it will learn – and possibly disclose - anything it can from the data that is stored on it, but never forge or tamper with the data. Clients of the consultant need to be able to store and search in their own data on the server and the consultant should have be able to do the same for all data. In order to prevent data leakage, the consultant requires all data to be stored in encrypted form. Our goal is to design and implement a system that enables the consultant and clients to do this.

2 Solution Overview

In this chapter we will give an overview of the provided solution. We will take a look at the use cases, requirements and at the high level design.

2.1 Use cases

In our solution there are two subjects, the consultant and the client. We assume that there is only one consultant (who has administration rights in the system) and that there are multiple clients. The consultant handles the financial data for his/her clients and is also responsible for all key management. After a client made him/herself known to the consultant and the consultant handled the appropriate key setup the client can insert and query his/her own financial data. We support the following use-cases:

Consultant

1. Insert a new client
2. Insert new data for a client
3. Query data for a specific client or for all clients

Client

1. Insert new data for him/herself
2. Query his/her own data

2.2 Requirements

From the use cases and the assignment we distilled the following requirements. The first four requirements are focused on the consultant, the second four are focused on the clients and the final three requirements have to do with encryption.

1. The system should allow the consultant to insert a new client
2. The system should allow the consultant to insert or update the financial data for a specific client
3. The system should allow the consultant to query the data of all his/her clients
4. The system should allow the consultant to query the data of a specific client
5. The system should allow a client to insert or update his/her own financial data
6. The system should allow a client to query his/her own financial data

7. The system should not allow a client to insert or update another client's financial data
8. The system should not allow a client to query another client's financial data
9. All textual data should be stored in an encrypted manner
10. The storage server should not be able to derive information about the content or the result of a query
11. The storage server may derive the structure of the stored data

The last requirement is included to make specific that we're only interested in the textual content of the stored data and the queries, it is not a problem if the server derives the structure of the data.

2.3 Data Model

Any scheme that enables search in encrypted data must define a data model that defines the format of the data in the system. This data model implies what kinds of unencrypted data are supported and in conjunction with the search scheme it defines the types of search that are supported and what the smallest search unit is. An example of a data model is that of a text document. If the search scheme entails that every word in the document is encrypted separately, then the smallest search unit might be a word.

From the requirements it is not immediately clear what kinds of data the consultant and client are dealing with, so we have to make a few assumptions. First, we assume all search queries are textual. Text queries are the bread and butter of any search scheme and they should easily suffice for this scenario. Second, we assume the stakeholders require the data to be structured, in order to be able to categorize or group the data of each client. The attentive reader will know the most prevalent structured and textual data format is XML today and this is the format of choice for our data model.

By opting for XML data as the format of the unencrypted system input and output, we enable the use of search schemes that have been proven to work on XML[1]. Working with XML data in a relational context – such as an SQL database – requires a conversion to take place. The scheme by Brinkman et al. includes a way to store XML data in a relational database and execute queries on it efficiently. Below we describe the conversion algorithm in detail.

Preprocess XML

When an XML file is entered by the client or the consultant, it first undergoes a transformation step that changes the structure of the tree. An XML file consists of a tree of nodes with parent/child relationships and each node can have attributes and text values. The next conversion step does not work well with mixed content – a node that has both nodes and text as its children – or attributes, so we transform text values and attributes into normal nodes. Figure 1 shows the input and output of this algorithm.

1 Efficient Tree Search in Encrypted Data, Brinkman et al., Information System Security, 2006

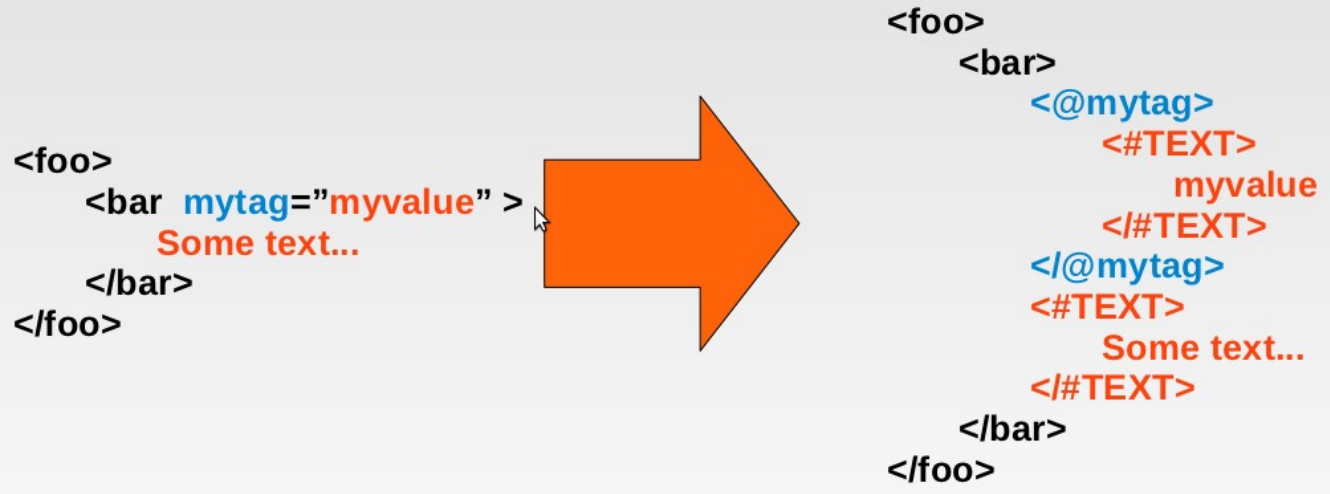

Figure 1: The first step of the XML conversion algorithm.

Input is any valid XML document that needs to be uploaded to the storage server. First any attributes are converted to normal nodes, where the tag name is the attribute name with an at-sign (@) prefixed to it. This uniquely identifies it as an attribute node and it also matches the syntax that is used in the XML query language XPath to search for specific attribute values:

//foo/bar[@mytag=”myvalue”]

Textual data such as the string “Some text...” is encoded in a normal node with tag name **#TEXT** and the textual data as the tag value. The resulting document is not valid XML anymore, but it has the important property that everything is represented as a node and no node contains both a textual value and a node value. After this transformation is executed, the output document is used as input for the next step.

Row Conversion

In order to store XML in a relational database the tree structure needs to be encoded somehow. An efficient method is to encode the location of each node using three integer values called `pre`, `post` and `parent`. A `pre` value denotes the order of nodes when the tree is traversed using the preorder traversal method. Analogously, the post values denotes the order of the nodes for postorder traversal. Finally the `parent` value of a node is the `pre` value of its parent node, where the root of the tree is assigned a `parent` of -1.

Using this method to find the location values of the nodes in our example tree we find the values displayed in Figure 2.

	pre	post	parent
<foo>	0		-1
<bar>	1		0
<@mytag>	2		1
<#TEXT>	3		2
myvalue			
</#TEXT>		0	
</@mytag>		1	
<#TEXT>	4		1
Some text...			
</#TEXT>		2	
</bar>		3	
</foo>		4	

Figure 2: This step encodes the location of each node with three integer values.

Finally, each node in the tree is converted to a row. The result consists of five rows – one for each node – and is displayed in Figure 3. All rows combined contain the information required to fully reconstruct the tree.

The attentive reader might notice that no row has both a tag and a value originating from the input file and how the text values can be 'inlined' in their parent node. Although a valid point for this example tree, our approach has two upsides. One advantage is that it enables mixed content, but at least as relevant is that textual strings are uniformly encoded. If one would decide to extend the search scheme to be able to search for specific words in tag values, then having a uniform format for all strings is convenient, if not required.

pre	post	parent	tag	value
0	4	-1	foo	
1	3	0	bar	
2	1	1	@mytag	
3	0	2	#TEXT	myValue
4	2	1	#TEXT	Some text...

Figure 3: The final results is a set of five rows and these rows can be used to reconstruct the original XML tree.

2.4 System Design

From quick glance at the requirements we can see that there are two separations required within our solution:

1. There has to be a separation between encrypted and plain text domains
2. There has to be a separation between two encrypted domains of different clients

As a solution to the first problem we defined two services; one which operations in the encrypted domain and one which operates in the plaintext domain. Both services are formed by different applications, there is a storage server which handles only encrypted data and encrypted queries and a client application which works with the plaintext data. The storage server is an XMLRPC service which exposes the required functionality using XMLRPC functions that can be called by the client.

The second problem is the separation between data belonging to two different clients. We solved this problem using a public-key scheme in which the public key for a client is used to determine which data it can access and on what data it can perform queries. The consultant has the keys for all his/her clients, meaning that he/she has full access to all the data available. The consultant also manages all the required key setup and distribution. The storage server requires the public key of a client to verify the permissions for the client, meaning that the consultant has to make sure that keys for a new client are inserted in the storage server. As these keys are only the public parts they can be stored in the clear, no encryption is required. Such a public key scheme isn't required for querying but is required to support insertion and update to only a client's own data. The exact way in which the keys are used is described, along with an overview of the interaction between both services, in the following sections.

The prototype was implemented using PHP5 and Sqlite for the client and Python with Sqlite for the storage server.

2.5 Service interactions

The server part of the system functions purely as a data storage and query engine. It stores the public keys for all the clients, stores all the encrypted data and handles insertions/updates and queries. The client is the party that handles the keys, encrypts the data and encrypts the queries. We will first discuss the signing procedure and then elaborate on the insertion/update and query procedures.

Key management and signing

To make sure that client can only access his/her own data we use a signing scheme using the client's aforementioned public/private keys. Every function that a client calls at the server has to be signed using the client's private key. In practise this means that the client signs the function name that it wants to call. This signature, along with a client id and the other parameters required for the call, is given to the server which uses the client id to find the client's public key and then uses that public key to check the signature. Every client id also corresponds to a tree id, which in turn corresponds to a dataset for this client. The server will then only insert/update/query this dataset, meaning that access permissions are enforced.

Insertions/updates

Let us take insertion as an example. If a client wants to insert a tree into the database it first turns the tree into a list of encrypted rows as mentioned in the data model chapter. These rows contain the following information:

1. tree_id
2. pre/post/parent values
3. encrypted tag/encrypted value pairs

Then it signs the name of the function that it wants to call ('insert') using it's private key and calls the function with the parameters (signature, client_id, list_of_encrypted_rows).

Updating a tree goes in exactly the same manner but only takes one row instead of a list of rows. The server then finds the row to update using the given pre value and changes the values to the given new values. Note that due to the encryption scheme in use, we do not support partial tree updates, see chapters 3 and 4.

Queries

Since the data is assumed to be XML structured data, queries are done in the form of XPath queries. If a client gets a query of the form `//foo/bar[mytag="myvalue"]` it first has to restructure and encrypt the query in such a way that the storage server can process it, but will learn nothing about the actual queried nodes; not even their names. Encryption of the query is explained in the chapter on algorithms (see chapter 3). The client then calls the query function at the storage server, using the signature, client_id, tree_id and encrypted query as parameters.

Supported queries combinations are the following:

Name	Example	Explanation
Search for named children	`/foo`	Search for all nodes named foo that are child of the current node(s)
Search for named descendants	`//foo`	Search for all nodes named foo that are descendants of the current node(s)
Search for attribute/value pairs	`/foo[@mytag="myvalue"]`	Search for all nodes named foo that are child of the current node(s) and have attribute foobar with value myvalue

At the server side, the query is evaluated using the method described in the paper by T. Grust et. al[2]. Once a list of candidate nodes is formulated using the pre, post and parent values the encrypted nodes are matched to the encrypted query content using the algorithm described in the chapter on algorithms (see chapter 3). The result for a query is a list of matching nodes. According to the XPath specification, the result of an XPath query should be the matching nodes, but the children of the result nodes should remain available. Therefore, after the list of result nodes has been computed we rebuild the list and add the subtree for each result node. The final result of the query will be a list of subtrees where the root of each subtree is a node that matched the query. Returning subtrees instead of single nodes also has the advantage that the client can immediately see the context in which a node resides and may easily refine the query to search for more specific sets of information.

Example query

On the next page we can see an example XML tree. On this tree we want to perform the following query:

```
//foo/bar[@mytag="myvalue"]
```

We begin the algorithm with as a current node a root node that is the (virtual) parent of “`<foo>`”. We take the first part of the query ("`//foo`") and parse this query. The "`//`" tells us that we need all descendants of the current node, meaning that we fetch the entire tree, because all nodes are a descendant of our (virtual) root node. We then filter out all the nodes that don't match by looking for the node name "`foo`". This leaves us with two nodes namely the two “`<foo>`” nodes. The next part of the query ("`/bar`") tells us to get all the children of our current nodes and filter out all nodes that aren't called bar. Since we have only one current node with a child called "`bar`" the second “`<foo>`” node is left out and our new list of current nodes consists of the two "`<bar>`" nodes. Since attributes are converted to tag nodes and text nodes we can search for attributes in the same manner as we search for named children.

The actual algorithms used for getting the children and descendants are described in [2]. Also, in our implementation we don't use basic node name matching since everything needs to be done under encryption; see the algorithms chapter for more details on node matching.

2 Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps, T. Grust et. al, 2003

```
<foo>
  <bar>
    <@mytag>
      <#TEXT>myvalue</#TEXT>
    </@mytag>
    <#TEXT>Lorem ipsum</#TEXT>
  </bar>
  <bar>
    <@mytag>
      <#TEXT>mydifferentvalue</#TEXT>
    </@mytag>
    <#TEXT>dolor sit</#TEXT>
    <foo>
      <#TEXT>amet</#TEXT>
    </foo>
  </bar>
</foo>
```

Figure 4: Example XML document

3 Search Algorithms

Encoding the XML input results in a set of rows, where each row codes for one XML node. The information in each row that is relevant to the encryption and matching algorithm is the `pre`, the `tag` and the `value`. Both `tag` and `value` are encrypted by the client using the scheme in Figure 5, where the input is the plaintext value. The resulting cipher texts are substituted into the rows before they are uploaded to the untrusted storage server.

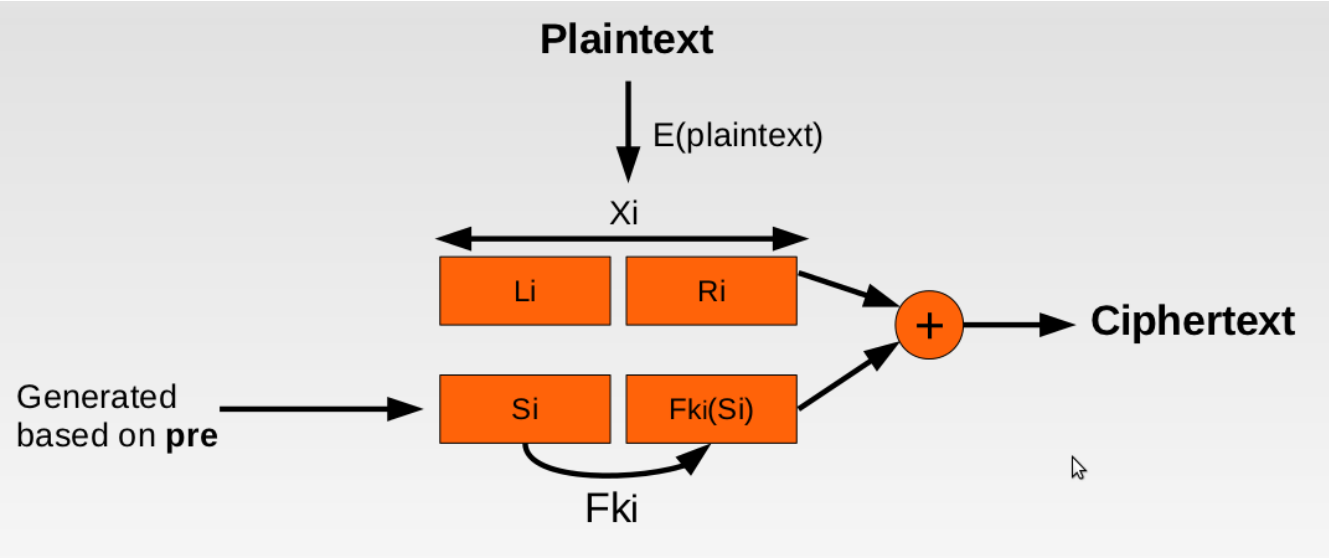

Figure 5: An overview of the searchable encryption algorithm.

Recall that each client possesses the following keys:

- an encryption key (`k''`),
- a hash key (`k'`),
- an RSA private key.

Plaintext Encryption

We individually discuss each step in the encryption process. First, the plaintext value is encrypted through an operation `E(plaintext, key)`. In our prototype, `E` is AES-128 encryption in CBC mode. Each client possesses a personal encryption key that is used to encrypt his/her XML information. The resulting cipher text `Xi` has a length which is a multiple of 16 bytes. It is split into two parts: `Li` and `Ri`. The width of `Li` and `Ri` must be a constant factor; our prototype splits `Xi` into two parts of equal width:

```
<Li, Ri> = Xi = E(plaintext, k'')
```

The encryption of this plaintext is never reversed by the server for the search process, so this encryption step ensures the privacy of the data when it is uploaded to the cloud storage server.

Searchable Text Generation

The next step is to generate a string that can used to conduct the hidden query. First, a semi-random value `Si` is generated by the client, based on the pre value of the row. How this works exactly can be looked up in the source code, the important thing is that it is the same width as `Li` and that the value can be reproduced by the client later.

Next, the client's hash key `k'` is used to generate a row-specific hash-key `ki` from the most significant half of the encrypted plaintext: `Li`. This is achieved by doing AES-128 encryption of `Li` in CBC mode and taking the most significant 128 bits of the result.

```
ki = E(Li, k')
```

This key `ki` is used to derive a value from `Si` using – you guessed it – AES-128 encryption in CBC mode, which is denoted in the figure by operation `Fki`. The resulting value has the same width as `Ri`.

Final cipher text

The encrypted plaintext `Xi` and the search-able string `<Si, Fki(Si)>` are XOR-ed to form the final cipher text. The corresponding plaintext value is substituted with the cipher text and empty tags and values are filled with randomly generated information in order to make them indistinguishable from filled values.

pre	post	parent	tag	value
1	8	0	offshore	Ä□□Ke□ø/¾¿�É¾KV
2	3	1	name	h�@-7a����□s¬�:n
3	1	2	@country	».1Þ×È□MÝ«�+=H'
4	0	3	#TEXT	caymans
5	2	2	#TEXT	Pyramid Investment Holding B.V.

Figure 6: The rows before the encryption step.

After the algorithm completes, the input rows in Figure 6 will look something like the rows in Figure 7.

pre	post	parent	tag	value
1	8	0	â��sKÕÙ=;l□�ªlÔi	åä□y_¬\Ôá�ÇÇ□6çò
2	3	1	©□a(�□¯¹á£â°Y!□	Ë□½ Ç□ý□±×I□#
3	1	2	@hÌfú*#ÑMCó□□]Î□	ªf□□ÀÍ□uò□Rb]>»q
4	0	3	□çÔ4□W□□þ□G,,□p.	/¡ª□□6KF0bN°^@
5	2	2	'ãY¦□-û[Cô¤5ç□	7□□¤9ö #7□f¥{]êåçøçºñ§ª□äÊ □}¨Ü`P

Figure 7: The rows after the encryption step. They will be uploaded to the server in this form.

Query generation

When a query is entered in the client, it will process each term in the query, in order to hide it from the server. If the entered query is:

```
//foo/bar
```

then the client will process the “foo” and “bar” to generate the tuple `<Xi, ki>` from each term in the same manner as it creates the encrypted values for the xml document. The final query that is sent to the storage server becomes:

```
//0/1
```

and it includes an array that contains the tuples for “foo” and “bar” respectively.

```
tuples[0] = <Xi, ki> for “foo”
tuples[1] = <Xi, ki> for “bar”
```

Note that the generation of `<Xi, ki>` for the query terms does not require information about the structure of the tree.

Search Algorithm

Upon receiving a query, the storage server attempts to find rows that match the XPath query using the methods described in the System Design chapter. The tuples come into play when a row must be checked for a match. This checking is displayed schematically in Figure 8.

First, the cipher text is extracted from the row's tag or value column and XOR-ed with the `Xi` component of the tuple. This leaves the search-able part of the query, where the left part is `Si` and the right part may or may not be `Fki(Si)`. Since `ki` is supplied as the other part of the tuple, this relation can be checked by the storage server. If the calculated `Fki(Si)` matches the value `Y` then the storage server assumes it is a match.

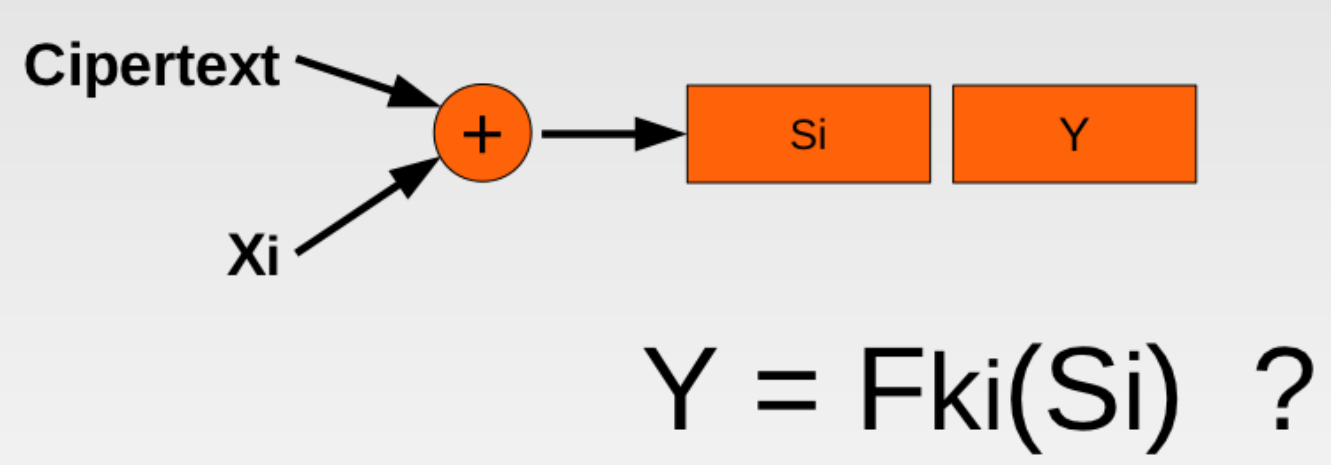

Figure 8: A schematic overview of the row matching process that is executed by the storage server.

Result Decryption

When a client receives a query result, it will be in the form of encrypted rows. The client can decrypt each cipher text by following these steps:

1. generate `Si` from the `pre` value
2. XOR `Si` with the cipher text to learn `Li`
3. generate `ki` from `Li`
4. generate `Fki(Si)`
5. XOR `Fki(Si)` with the cipher text to learn `Ri`
6. Combine `Li` and `Ri` to find `Xi`
7. Decrypt `Xi` with `k''` to find the original plaintext

Because it is possible that collisions occur in F, the client should filter the results to double-check that they match the posed XPath query.

4 Tradeoffs and Choices

Here we will discuss some of the tradeoffs and choices appropriate to our system.

4.1 Search scheme

The implemented search scheme has some advantages and some disadvantages. The main advantage of the scheme is that it's quite easy to implement, it does not require any extra mathematics beside the symmetric crypto and signing used. The main disadvantage remains speed; implementing the required node matching algorithms directly in the database can be done but isn't trivial. This means that the node filtering is done after the rows have been fetched from the database, providing a significant overhead as some rows will not match and be discarded.

Another downside of our approach is that it is (currently) not possible to search for wildcards or for words within text; we can only search for nodes and attributes. Our scheme can, however, be extended to support searching for words within the text by storing each word as a separate row in the database, but this means even more overhead.

Even though we implemented only a basic subset of the XPath query language our system can easily be extended to support a richer subset since all the filtering algorithms stay the same; all that is required is to extend the actual XPath query parsing.

A final remark about the search scheme that we use is that, in very rare cases, it could happen that the algorithm says that a row matches a query, even though it does not. Our proof of concept implementation does not check whether the final results of a query actually match the XPath query, because the chances of a collision are very low. In a production environment, however, this check should be implemented.

4.2 Scalability

As already mentioned, the biggest problem with the solution is speed. The difference in speed is already noticeable when comparing an XML document of 30 lines versus a document of 300 lines; the system will probably become unusable in a real-life situation where the documents could grow a lot further. The reason for the non-scalability is the node filtering algorithm which has to do an AES encryption for every possible row, slowing down the query processing considerably compared to usual database row retrieval.

Another problem arises when inserting nodes. Because the encryption of a row is (amongst other factors) based on the pre value, the resulting cipher text changes when the pre value for a node changes. So, if a tree is updated such that the structure of the tree changes all rows with higher pre values in the document have to be reëncrypted and reinserted in the database. This problem can be partly mitigated by using larger intervals between pre values, but can not be solved completely. For our proof of concept implementation we decided to leave out partial-tree updates completely, a client can only update a single row (thus not changing the structure of the tree) or reinsert the entire tree. Overall this choice is not often a problem since the financial data situation sees most updates in single rows, for instance when changing the balance of an account.

4.3 Security

A final remark has to be made about the security of our implementation. Even though clients can only insert data in their own tree the system is not secure against replay attacks. Consider the example in which client A inserts a tree and directly afterwards inserts another tree. If client B can capture all the traffic between A and the server, he can replay the insertion procedure for the first tree and overwrite the change to the second tree. Mitigating this flaw can be done in a lot of different ways, one could for instance use nonces in every communication between client and server (causing a bit more overhead) or one could wrap the entire communication in an SSL-enabled session.