Appears in *Secure Data Management: Second VLDB Workshop—SDM 2005*, *Lecture Notes in Computer Science* **3674 (2005) 33–46. Springer-Verlag.**

Experiments With Queries Over Encrypted Data Using Secret Sharing

Richard Brinkman[1,3], Berry Schoenmakers[2,3], Jeroen Doumen[1], and Willem Jonker[1,3]

[1] University of Twente, The Netherlands,
{brinkman,doumen,jonker}@cs.utwente.nl
[2] Technical University of Eindhoven, The Netherlands,
berry@win.tue.nl
[3] Philips Research, The Netherlands

Abstract. To avoid insider attacks one cannot rely on access control to protect a database scheme. Encrypting the database is a better option. This paper describes a working prototype of an encrypted database system that allows remote querying over the encrypted data. Experiments with the system show the practical impact of our encoding scheme on storage space and CPU time. Two algorithms, each with two different matching rules, are compared to each other.

1 Introduction

Enterprises often rely on access control to protect their assets. However, a study of the Computer Science Institute and the FBI [1] shows that most successful attacks are conducted by insiders. A possible solution is to replace access control with database encryption where the user keeps the encryption key secret. This shift opens up a new research area of query evaluation over encrypted data.

In this paper we present an extension of our encrypted database system of [2]. We summarise this scheme in section 3. See [2] for further information on the background of searching in encrypted databases. In this paper we present an augmented version of the database system. The former solution lacks the ability to search in the data itself, it only allows searching the XML tags. In the new solution the textual data of an XML document is represented as a *trie* [3], enabling searching tags as well as data. In section 4 we show how to represent the text as a *trie*.

To investigate the practical impact of our database scheme, we have built an implementation. In section 5 we describe some of the implementation issues. In section 6 we use the implementation together with a test database to do several experiments in order to measure the storage space and the influence of the search algorithm and the configuration settings on the CPU time.

2 Related Work

Traditionally, databases are protected against malicious use by means of an access control mechanism. However, the database management system itself is

trusted. When the data is outsourced the database system cannot be trusted anymore to keep the query and the answer secret. Private Information Retrieval [4] aims at letting a user query the database system without leaking to the database which data was queried. The idea behind PIR is to replicate the data among several non-communicating servers. A client can hide his query by asking all servers for a part of the data in such a way that no server will learn the whole query by itself. Chor et al [4] prove that PIR with a single server can only be done by sending all data to the client for each query. In practice database replication is not preferable.

PIR aims at hiding the query from the database leaving the data in the clear. Song, Wagner and Perrig [5] suggest a different technique that supports encrypting the data itself. An encrypted keyword can be found in an encrypted text without the server learning either the keyword or the plaintext. We adapted this work to exploit the tree structure in XML documents in [6].

3 Overview Of Our Approach

In our database scheme a plaintext XML document is transformed into an encrypted database by following the steps below. See figure 1 for the encoding of a concrete example.

1. Define a function $map : node \rightarrow \mathbb{F}_{p^e}$, which maps the tag names of the nodes to values of the finite field $\mathbb{F}_{p^e}$, where p^e is a prime power (p prime and e a positive integer) which is larger than the total number of different tag names (figure 1(b)).
2. Transform the tree of tag names (figure 1(a)) into a tree of polynomials (figure 1(d)) of the same structure where each *node* is transformed to $f(node)$ where function $f : node \rightarrow \mathbb{F}_{p^e}[x]/(x^{p-1} - 1)$ 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 (figure 1(e)) and a server tree (figure 1(f)). Both trees have the same structure as the original one. The polynomials in the client tree are generated by a pseudorandom 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 pseudorandom generator it suffices to store the seed on the client. The client tree can be discarded. When necessary, it can be regenerated using the pseudorandom generator and the seed value.

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.

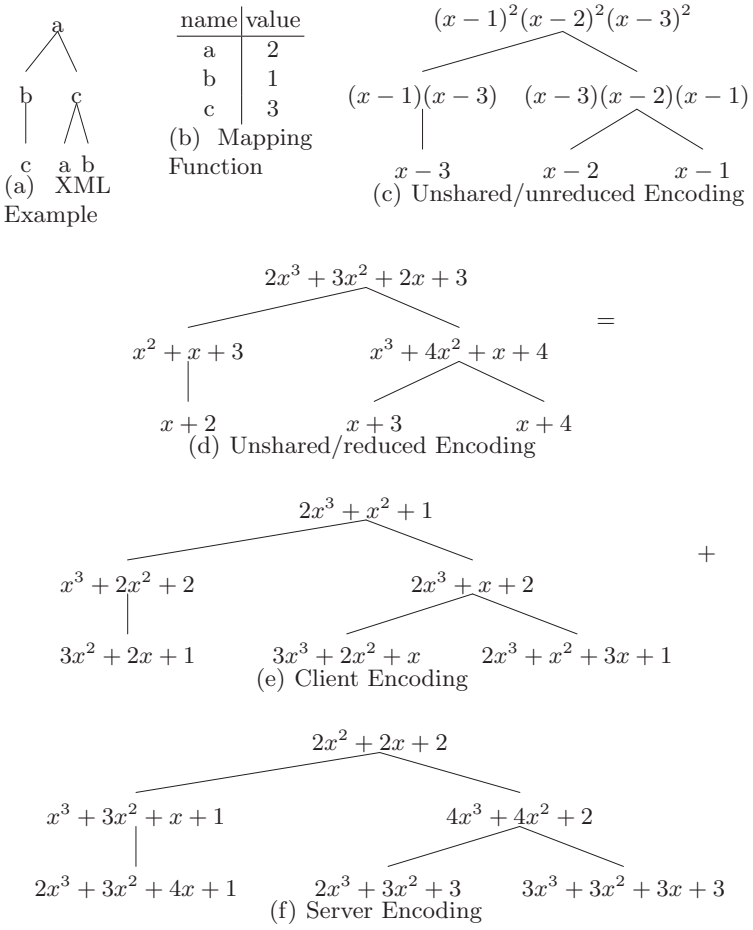

Fig. 1. The mapping function (1(b)) maps each name of an input document (1(a)) to an integer. The XML document is first encoded to a tree of polynomials (1(c)) before it is reduced to the finite field $\mathbb{F}_5[x]/(x^4-1)$ (1(d)) and split into a client (1(e)) and a server (1(f)) part.

4 *Trie* Enhancement

The approach sketched in section 3 is only efficient when p^e is small. This is no problem for tag names that are chosen from a fixed sized set (described in a DTD), but cannot be used for the data because the number of different data nodes is unbounded. And since each polynomial takes $(p^e - 1)\log_2 p^e$ bits of storage space, it is important to keep p^e as small as possible.

In this paper we propose a representation of XML documents allowing for efficient searching in data nodes. Basically, all data nodes are transformed to their *trie* representation [3].

A data string in the original XML document is translated to a path of nodes where each node is chosen from a small set. Assume this set contains $a, b, \ldots, z$. With this set we can translate the tree shown in figure 2(a) to an equivalent *trie* 2(b) or an uncompressed *trie* 2(c). An uncompressed *trie* stores exactly the same information as the original data string, whereas the compressed *trie* loses the order and cardinality of the words. If this is a problem an encryption of the data string may be added to the node. In this example we first split a string into words, represented by paths, and then each path is split into several characters. Other ways of splitting the string into nodes are possible.

On average removing duplicate words from a text reduces the size by 50%. Reducing a text into a compressed *trie* reduces the size by 75-80%. However each node is converted into a polynomial of size $(p^e - 1)\log_2 p^e$ bits. In case $p = 29$ a polynomial costs 17 bytes. Due to the *trie* compression the 'encryption' of a single letter will cost approximately $3\frac{1}{2} - 4\frac{1}{2}$ bytes.

Having translated the original XML tree into a (compressed) *trie*, the same strategy of [2] can be used to encode the document. Like the document, also the queries should be pre-tuned to the new scheme. A query like

```
/name[contains(text(), "Joan")]
```

is first translated to

```
/name[//J/o/a/n]
```

before it is translated to

$$/map(\texttt{name})[//map(\texttt{J})/map(\texttt{o})/map(\texttt{a})/map(\texttt{n})].$$

Simple regular expressions like `.` and `.*` can be mapped to their *trie*-equivalents `*` and `//`.

5 Implementation

In the previous sections we described our theory of searching in encrypted data by using secret sharing and a special kind of encoding/encryption. To demonstrate that searching in encrypted data is not only possible in theory, but also in practice, we have built a prototype implementing the encoding and search strategy described in section 3.

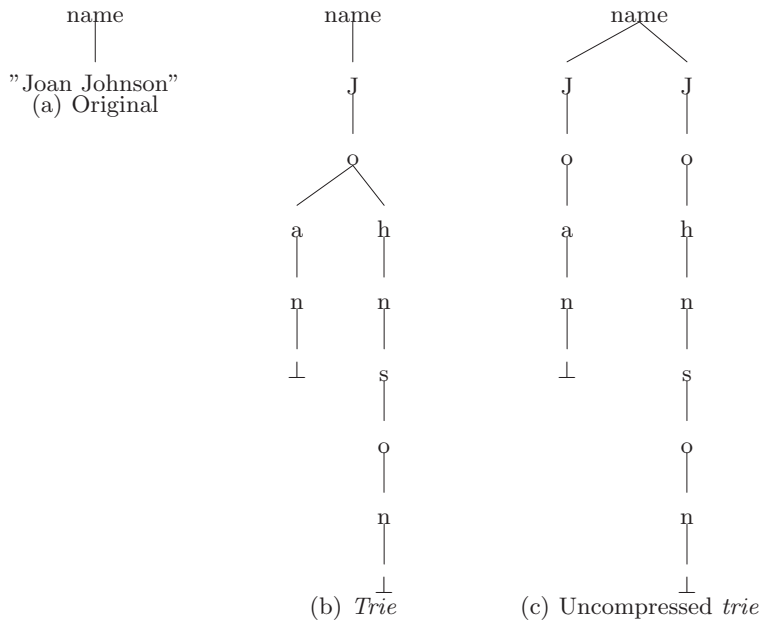

Fig. 2. Transformation of an XML document tree into either a compressed or an uncompressed *trie*

The implementation is written in Java and set up using a client/server model. Figure 3 shows the architecture. We will elaborate on each component in the following sections.

The server stores all the polynomials in a database. The database is not protected and can be considered publicly readable. However, the client encodes the original plaintext XML document into encoded polynomials by using the `MySQLEncode` class. The encoder needs a private seed and a private map file which will be re-used by the query engines. The map file is just a text file which stores the mapping between tag names and corresponding values from $\mathbb{F}_{p^e}$.

The prototype consists of two different query engines: `SimpleQuery` and `AdvancedQuery`. Both engines share the same filtering technique. The filter is distributed over the client and the server. The filter classes perform basic operations like function evaluation and tree reconstruction.

5.1 MySQLEncode

Since the server should not learn the information it is storing, it is the client's responsibility to fill the database.

The `MySQLEncode` class acts on three files which are provided on the command-line:

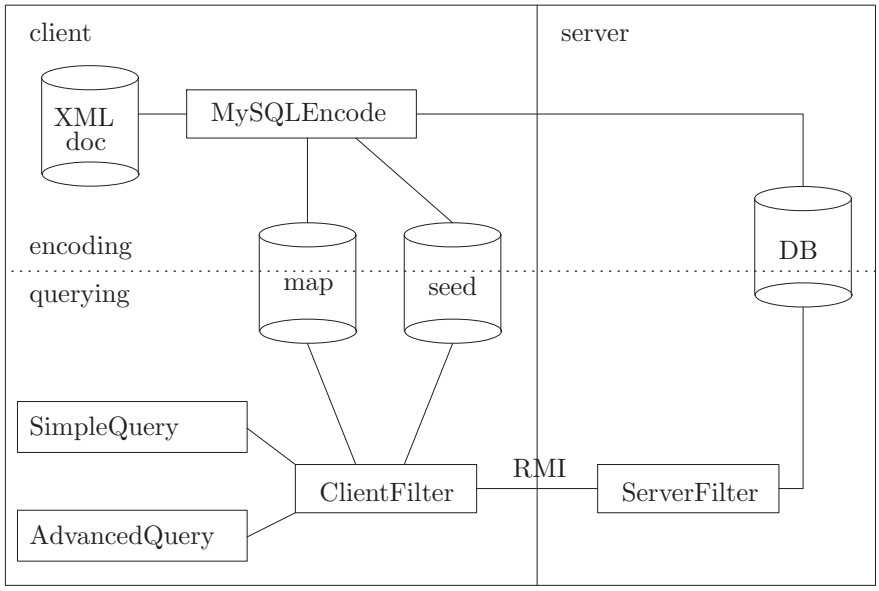

Fig. 3. Client/Server Architecture

1. A map file
2. A seed file
3. The original XML document

The map file is a property file where each line is of the form $name = value$, where *name* is one of the tag-names as specified by the DTD or XML schema and $value \in \mathbb{F}_{p^e}$ is the value it is mapped to.

The seed file acts as the encryption key and should therefore be kept secure. Without the seed file it is impossible to regenerate the client tree, and without the client tree the data on the server is meaningless.

The original XML document is parsed by a SAX parser[4]. This means that there is no need for a big client machine with lots of memory. This fits nicely into our philosophy of small clients (cell phones, for example) and big servers. The parser linearly reads the document and constructs the tree on the fly. It only needs memory proportional to the depth of the tree. The tree structure is stored by adding *pre*, *post* and *parent* values to each polynomial. The *pre* and *post* fields are sequence number that count the open tags respectively close tags. The *parent* fields refers to the *pre* value of its parent. This is a common way to store a tree structure into a flat relational table [2,7]. In our prototype we use MySQL[5] as the database back-end. In order to speed up the search process the *pre*, *post* and *parent* fields are indexed by a B-tree.

[4] `www.saxproject.org/`
[5] `www.mysql.com`

5.2 The Filter Implementation

Each different query engine (see section 5.3) will use the same set of basic operations. These operations are offered by `ServerFilter` and `ClientFilter`. Both classes implement a common interface `Filter` but are adapted to work on the server site respectively the client site. The two objects communicate with each other using Java's Remote Method Invocation (RMI). The operations consist of functions to query the tree structure as well as to evaluate the polynomials. `ServerFilter` will evaluate the polynomials stored in the database for the given values. `ClientFilter` first regenerates the client polynomial by using the pseudorandom generator with the secret seed and the *pre* location of the polynomial. After the evaluation of its generated polynomial it will add the result to the retrieved value from the server. Only when the sum equals zero, the location is returned to the invoking query engine, otherwise the next candidate node is generated/retrieved, evaluated and added together.

With the evaluation method only the *containment* of a node in a subtree is tested. To be sure that the node is *equal* to the root of the subtree there is an option to check the first factor of a node. To retrieve the factor $(x - t)$ in $f(x) = (x - t)\prod c(x)_{c \in children(f)}$ it is necessary to reconstruct the node's polynomial and all its child polynomials. Because the equality test is expensive it should only be invoked when absolutely necessary.

The operator `nextNode()` acts as a pipeline. The thin client only needs to have one node in memory at a time. The big server will do the buffering of the intermediate results.

5.3 Query Engines

Since it was not a priori clear which search strategy is the best, we have decided to implement two query engines, called `SimpleQuery` and `AdvancedQuery`, each using a different search strategy, as explained below.

SimpleQuery The most simple search strategy parses the XPath[6] query into steps where each step consists of a direction (child (`/`) or descendant (`//`)) and a tag name. Two special tag names exist: `..` matches the parent and `*` matches every child.

In this example we make use of the containment test only. In section 6 we will also use the equality test. There we will compare the two tests to see whether one is preferable to the other. We will sketch the algorithm by using an XML document generated by the XMark benchmark [8] and the example query `/site/*/person//city`. See appendix A for the DTD. This query is parsed into the following steps:

`/site` The first slash instructs the search engine to locate the root node (i.e. the only node without a parent (parent=0)). Since the parent field is indexed

[6] `www.w3.org/TR/xpath`

this is done in constant time. After the root node has been located both the stored polynomial on the server and the generated polynomial on the client are being evaluated at $map(site)$. Only when the sum equals zero the next steps are carried out.

`/*` At this point the preliminary result set (implemented as a `Queue` on the server) will consist of only a single element. This step will change the result set into all children of the root node (i.e. regions, categories, catgraph, people, open_auctions and closed_auctions). The `*` reduces the workload because no additional filtering is needed.

`/person` All children of the 6 nodes in the result set are being examined in this step. Evaluation at $map(person)$ is done for all the polynomials found. Only those nodes for which the sum of the server and client evaluations equals zero remain in the result set.

`//city` This step is quite expensive in terms of execution time. The result of the previous step is already quite large and this step even increases the number of possible nodes that have to be checked. All the descendants of the person-nodes (i.e. name, emailaddress, phone, address, homepage, creditcard, profile, watches, street, city, country, province, zipcode, interest, education, gender, business, age, watch, category, open_auction and description) have to be checked against $map(city)$.

AdvancedQuery In contrast to the `SimpleQuery` the `AdvancedQuery` takes the tree as the starting point and parses it from root to leaf nodes. At each step the whole remaining query is taken into account. We take advantage of the fact that nodes have knowledge of all descendants. This way it is possible to identify dead branches early in the search process at the cost of more evaluations for each node.

For easy comparison we use the same query and the same test (containment) as before.

`/site/*/person//city` The `AdvancedQuery` engine always starts at the root node. This node is checked against $map(site)$, $map(person)$ and $map(city)$. Only when all three sums are zero the next steps are carried out. Note that we can only check for the existence of a node. The structure of the query cannot be taken into account since the nodes don't store the structure of the subtree.

`/*/person//city` The engine proceeds by consuming the `/site` part of the query and traversing the tree one step down to find the root's children. This unfiltered set of nodes are regions, categories, catgraph, people, open_auctions and closed_auctions. After filtering only the people, open_auctions and closed_auctions remain; all the other nodes do not contain person or city nodes. Thus we may skip these branches.

`/person//city` In this step the `/*` has been removed. This means we traversed the tree one step downwards. The children of people, open_auctions and closed_auctions are person, open_auction and closed_auction. Because open_auction and closed_auction contain person and city nodes they remain

in the result set even after filtering. The implementation does not check if the node *is* a person but if it *contains* it. This is done because we chose to use the containment test instead of the equality test. In section 6 we investigate whether this was a good choice or not.

`//city` From the person, open_auction and closed_auction nodes we interactively walk downwards in the tree evaluating the polynomials at $map(city)$ until this results in a non-zero sum. The result set now contains all nodes having a city inside. If we had chosen the equality test only the city nodes would have been in the result set.

6 Experiments

The prototype is an ideal instrument to perform experiments with. With the experiments described in this section we would like to find out what the practical impact of our encrypted database scheme is. We investigated the storage space overhead (section 6.1), the influence of the different search engine algorithms (section 6.2) and the difference between the equality and containment tests (section 6.3). All experiments act on an auction database synthesized by the XMark benchmark [8]. The DTD (see appendix A) contains 77 elements. We chose $p = 83$ and $e = 1$ throughout this section.

6.1 Encoding

Encoding an XML document as polynomials requires extra storage space. This is due to the fact that each polynomial not only stores the information of its own node but also of all its descendants. Figure 4 plots the encoded database size against the input XML size. Approximately 17% of the output size is caused by the pre, post and parent values (not plotted in the figure). The remainder is thus approximately 1.5 times the size of the input. To speed up the search process we added indices to the pre, post and parent fields using B-trees. The size of these indices is added on top of the output size. As expected both the storage space and the encoding time are strictly linear in the input size.

6.2 Query Engines

One of the main reasons for building the prototype was because it was not a priori clear what the most efficient query engine algorithm is. Is it best to evaluate a polynomial at as many points as possible at each node to find an early dead branch or should you evaluate at a single point at a time? To answer this question we performed two tests: one with the simplest of all queries at increasing length and one with more advanced queries containing `//` and `*`.

The first test is the worst case scenario for the advanced query engine. The queries in table 1 are chosen in such a way that there is no gain for the advanced algorithm. For instance it is a waste of effort to check whether a europe node

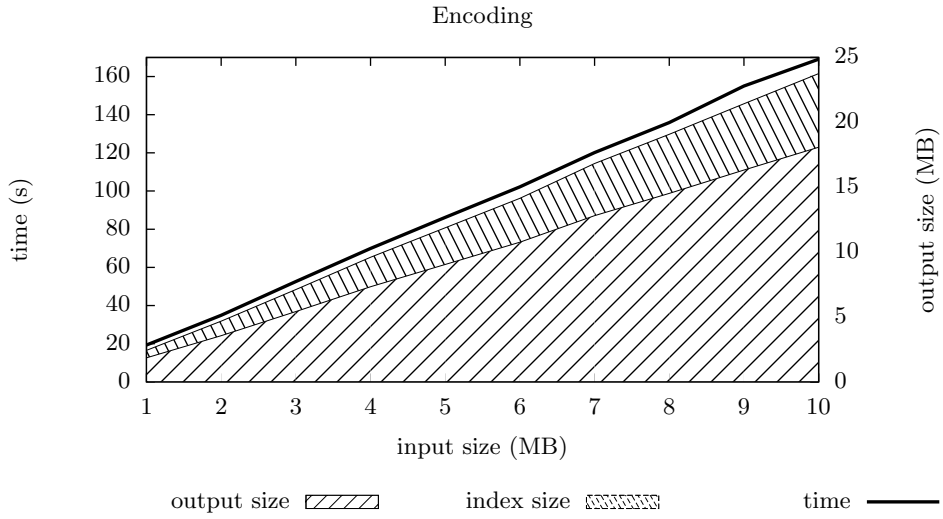

Fig. 4. Encoding

contains an item, description, parlist, listitem, text and keyword node, because the DTD (see appendix A) dictates it to be always the case.

As can be seen in figure 5, where the number of evaluations is plotted against the queries of increasing length shown in table 1, the two search algorithms are comparable. They differ by at most a constant factor.

The second test with queries containing `//` and `*` was performed in conjunction with the strictness test. The test result are given in the next section.

1	`/site`
2	`/site/regions`
3	`/site/regions/europe`
4	`/site/regions/europe/item`
5	`/site/regions/europe/item/description`
6	`/site/regions/europe/item/description/parlist`
7	`/site/regions/europe/item/description/parlist/listitem`
8	`/site/regions/europe/item/description/parlist/listitem/text`
9	`/site/regions/europe/item/description/parlist/listitem/text/keyword`

Table 1. Queries with increasing length. The numbers correspond to figure 5.

6.3 Strictness

Another aspect that is hard to predict is the difference between the equality test and the containment test. On the one hand, it can be argued that, since

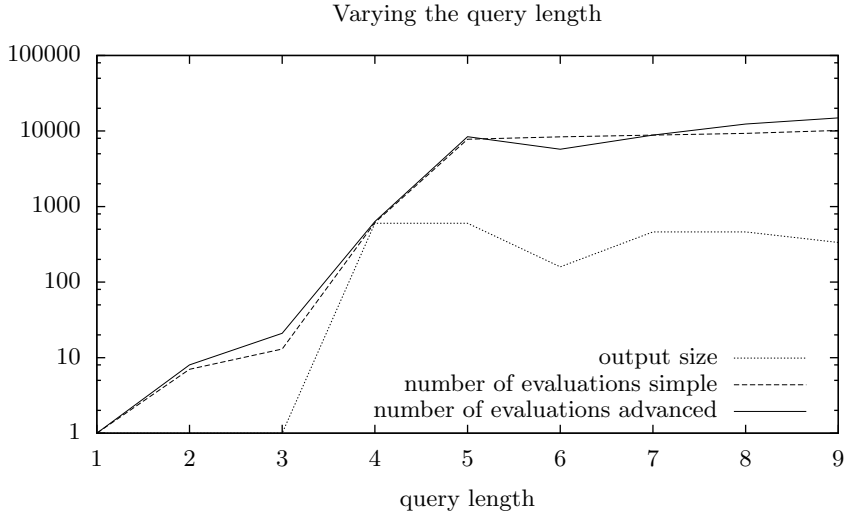

Fig. 5. Several queries with increasing query length. The query numbers refer to the queries summed up in table 1

the reconstruction of the first factor of a polynomial is computationally more expensive than a simple function evaluation, it is preferable to use the containment test. On the other hand, the reduced accuracy causes more nodes to be examined. Therefore we used our prototype to compare the two tests using both search algorithms.

For each query in table 2 four experiments were performed. Each algorithm (simple and advanced) was run twice: once with the equality test (strict checking) and once with the containment test (non-strict checking). The results are plotted in figure 6. For all queries the advanced algorithm outperforms the simple algorithm. Furthermore, it can be noticed that sometimes the strict checking pays off and sometimes it does not. In general, the equality test may cause a slight overhead or a major improvement.

Of course it is unfair to compare the equality test, which always gives the exact answer, with the containment test without considering the accuracy. Figure 7 shows the accuracy of the containment test. It plots the percentage of the nodes in the containment test's result that also pass the equality test. Notice that the accuracy drops for each `//` in the query. For absolute queries which do not contain `//`, the accuracy of the containment test reaches 100%.

7 Conclusions and Future Work

In our previous paper [2] we introduced a new search strategy over encrypted data. All XML nodes are encoded as polynomials. Each polynomial contains

1 `/site//europe/item`
2 `/site//europe//item`
3 `/site/*/person//city`
4 `/*/*/open_auction/bidder/date`
5 `//bidder/date`

Table 2. Queries for the strictness checks. The numbers correspond to figure 6.

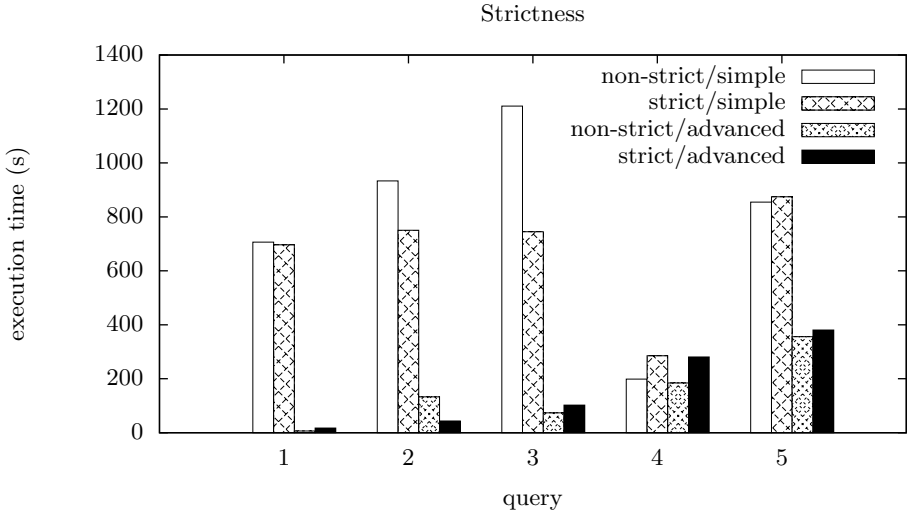

Fig. 6. Equality test versus containment test

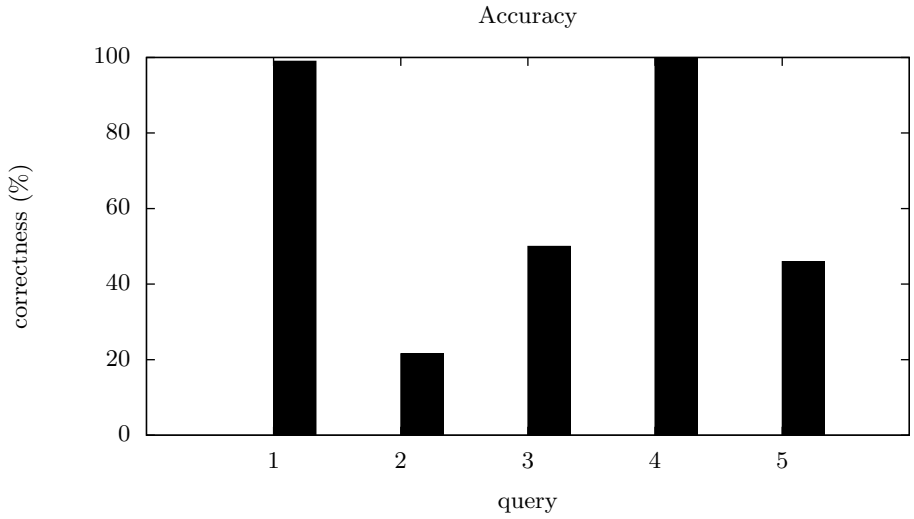

Fig. 7. Accuracy of the containment test as defined by the quotient $\frac{E}{C}$, where E is the size of the result set using the equality test and C is the size of the result set using the containment test.

knowledge of its own node as well as all its decendants. Due to a smart reduction the storage overhead is reduced to 50% as measured by our prototype (using $p = 83$ and $e = 1$). The encoding time is linear in the size of the input.

The prototype can choose between two different search algorithms. The simple algorithm reads a query from left to right carrying out a single evaluation at each node. The more advanced algorithm uses a look-ahead strategy where the whole remaining query is taken into account. Experiments show that the advanced algorithm outperforms the simple algorithm in the majority of cases. Only for the most simple queries it is slightly slower.

The search algorithms can use two comparison tests: the equality test and the containment test. The containment test is just a cheap evaluation whereas the equality test is more expensive because a node's own polynomial should be divided by all its child polynomials. The cost of a single equality test depends on the number of children, whereas the costs of a containment test is always constant. All the child nodes should be retrieved from the server and added to the pseudorandomly generated client polynomials. The accuracy of the containment test is reasonable but it does not result in a major improvement in the running time. On the contrary, it is often better to use the equality test to reduce the number of nodes to check, especially for the simple algorithm.

Using a *trie* to represent data content enables querying of the data inside the XML tags. The *trie*-representation is not yet part of the current prototype but we expect a major improvement especially in the advanced algorithm. Queries over the data are more precise than those over the tag labels and thus the number

of nodes to be examined is being reduced. Since knowledge of the data is present at high level nodes, the query engine can find the path to the answer almost immediately.

References

1. Computer Science Institute. CSI/FBI computer crime and security survey. `http://i.cmpnet.com/gocsi/db_area/pdfs/fbi/FBI2004.pdf`.
2. 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. `http://www.ub.utwente.nl/webdocs/ctit/1/00000106.pdf`.
3. Edward Fredkin, Bolt Beranek, and Newman. Trie memory. *Communications of the ACM*, 3(9):490–499, September 1960.
4. B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *FOCS*, pages 41–50, 1995.
5. 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`.
6. 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. `http://www.ub.utwente.nl/webdocs/ctit/1/000000f3.pdf`.
7. 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`.
8. A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. The XML Benchmark Project. Technical Report INS-R0103, CWI, Amsterdam, The Netherlands, April 2001. `http://monetdb.cwi.nl/xml/index.html`.

A Appendix: XMark's Auction DTD

```
<!ELEMENT site            (regions, categories, catgraph, people, open_auctions, closed_auctions)>
<!ELEMENT categories      (category+)>
<!ELEMENT category        (name, description)>
<!ELEMENT name            (#PCDATA)>
<!ELEMENT description     (text | parlist)>
<!ELEMENT text            (#PCDATA | bold | keyword | emph)*>
<!ELEMENT bold            (#PCDATA | bold | keyword | emph)*>
<!ELEMENT keyword         (#PCDATA | bold | keyword | emph)*>
<!ELEMENT emph            (#PCDATA | bold | keyword | emph)*>
<!ELEMENT parlist         (listitem)*>
<!ELEMENT listitem        (text | parlist)*>
<!ELEMENT catgraph        (edge*)>
<!ELEMENT edge            EMPTY>
<!ELEMENT regions         (africa, asia, australia, europe, namerica, samerica)>
<!ELEMENT africa          (item*)>
<!ELEMENT asia            (item*)>
<!ELEMENT australia       (item*)>
<!ELEMENT namerica        (item*)>
<!ELEMENT samerica        (item*)>
<!ELEMENT europe          (item*)>
<!ELEMENT item            (location, quantity, name, payment, description, shipping, incategory+, mailbox)>
<!ELEMENT location        (#PCDATA)>
<!ELEMENT quantity        (#PCDATA)>
<!ELEMENT payment         (#PCDATA)>
<!ELEMENT shipping        (#PCDATA)>
<!ELEMENT reserve         (#PCDATA)>
<!ELEMENT incategory      EMPTY>
<!ELEMENT mailbox         (mail*)>
<!ELEMENT mail            (from, to, date, text)>
<!ELEMENT from            (#PCDATA)>
<!ELEMENT to              (#PCDATA)>
<!ELEMENT date            (#PCDATA)>
<!ELEMENT itemref         EMPTY>
<!ELEMENT personref       EMPTY>
<!ELEMENT people          (person*)>
<!ELEMENT person          (name, emailaddress, phone?, address?, homepage?, creditcard?, profile?, watches?)>
<!ELEMENT emailaddress    (#PCDATA)>
<!ELEMENT phone           (#PCDATA)>
<!ELEMENT address         (street, city, country, province?, zipcode)>
<!ELEMENT street          (#PCDATA)>
<!ELEMENT city            (#PCDATA)>
<!ELEMENT province        (#PCDATA)>
<!ELEMENT zipcode         (#PCDATA)>
<!ELEMENT country         (#PCDATA)>
<!ELEMENT homepage        (#PCDATA)>
<!ELEMENT creditcard      (#PCDATA)>
<!ELEMENT profile         (interest*, education?, gender?, business, age?)>
<!ELEMENT interest        EMPTY>
<!ELEMENT education       (#PCDATA)>
<!ELEMENT income          (#PCDATA)>
<!ELEMENT gender          (#PCDATA)>
<!ELEMENT business        (#PCDATA)>
<!ELEMENT age             (#PCDATA)>
<!ELEMENT watches         (watch*)>
<!ELEMENT watch           EMPTY>
<!ELEMENT open_auctions   (open_auction*)>
<!ELEMENT open_auction    (initial, reserve?, bidder*, current, privacy?, itemref, seller, annotation, quantity, type, interval)>
<!ELEMENT privacy         (#PCDATA)>
<!ELEMENT initial         (#PCDATA)>
<!ELEMENT bidder          (date, time, personref, increase)>
<!ELEMENT seller          EMPTY>
<!ELEMENT current         (#PCDATA)>
<!ELEMENT increase        (#PCDATA)>
<!ELEMENT type            (#PCDATA)>
<!ELEMENT interval        (start, end)>
<!ELEMENT start           (#PCDATA)>
<!ELEMENT end             (#PCDATA)>
<!ELEMENT time            (#PCDATA)>
<!ELEMENT status          (#PCDATA)>
<!ELEMENT amount          (#PCDATA)>
<!ELEMENT closed_auctions (closed_auction*)>
<!ELEMENT closed_auction  (seller, buyer, itemref, price, date, quantity, type, annotation?)>
<!ELEMENT buyer           EMPTY>
<!ELEMENT price           (#PCDATA)>
<!ELEMENT annotation      (author, description?, happiness)>
<!ELEMENT author          EMPTY>
<!ELEMENT happiness       (#PCDATA)>
```