Efficient Processing of Secured XML Metadata

Ling Feng[1] and Willem Jonker[1,2]

[1] Dept. of Computer Science, University of Twente, The Netherlands
{ling,jonker}@cs.utwente.nl
[2] Philips Research, The Netherlands
willem.jonker@philips.com

Abstract. Metadata management is a key issue in intelligent Web-based environments. It plays an important role in a wide spectrum of areas, ranging from semantic explication, information handling, knowledge management, multimedia processing to personalized service delivery. As a result, security issues around metadata management needs to be addressed in order to build trust and confidence to ambient environments. The aim of this paper is to bring together the worlds of security and XML-formatted metadata management in such a way that, on the one hand the requirement on secure metadata management is satisfied, while on the other other hand the efficiency on metadata processing can still be guaranteed. To this end, we develop an effective approach to enable efficient search on encrypted XML metadata. The basic idea is to augment encrypted XML metadata with encodings which characterize the topology and content of every tree-structured XML metadata, and then filter out candidate data for decryption and query execution by examining query conditions against these encodings. We describe a generic framework consisting of three phases, namely, *query preparation, query pre-processing* and *query execution*, to implement the proposed search strategy.

Keywords: Security, metadata, XML, encryption, search

1 Introduction

Ambient intelligence is an important theme in today's industrial and public research [3,9]. A key issue towards ambient intelligence is metadata management. From semantic explication, information handling, knowledge management, multimedia processing to personalized service delivery, metadata plays an important role. With XML [4] becoming the dominant standard for describing and interchanging data between various systems and databases on the Internet, Web-based applications nowadays increasingly rely on XML metadata to convey machine-understandable information and provide interoperability across the Web.

With this sheer volume of metadata flowing throughout ambient environments, the need to protect XML metadata content from being disclosed or tampered with is growing. One prototypical technique for building security and trust

R. Meersman and Z. Tari (Eds.): OTM Workshops 2003, LNCS 2889, pp. 704–717, 2003.

is to distribute and store these metadata in encrypted form [12]. W3C recommends an "XML Encryption Syntax" to allow the encryption of XML data using a combination of symmetric and public keys, where element content is encrypted by means of a symmetric key that in turn is encrypted by means of the public key of the recipient [6,5]. Nevertheless, securing metadata in ciphertext should not hinder its processing for various applications. The deployed metadata security techniques should on the one hand satisfy the security requirements on XML metadata, while at the same time allow efficient manipulation of metadata without loss of confidentiality.

Since search is one of the basic operations that are carried out on metadata, a first step to proceed is to address the issue around effective and efficient searching for information in encrypted XML data. A straightforward approach to search on encrypted XML data is to decrypt the ciphertext first, and then do the search on the clear decrypted XML data. However, this inevitably incurs a lot of unnecessary decryption efforts, leading to a very poor query performance. especially when the searched data is huge, while the search target comes only from a small portion of it. To solve this problem, we ask two questions here: (1) "*Can we discard some non-candidate XML data and decrypt the remaining ones instead of the whole data set?*" (2) "*If so, how can we effectively and efficiently distinguish candidate XML data from non-candidate ones?*"

There is some previous work in different research areas that are related to our work. [13] presents techniques to support *keyword*-based search on encrypted textual string. Recently, [10,11] explore techniques to execute *SQL*-based queries over encrypted relational tables in a database-service provider model, where an algebraic framework is described for query rewriting over encrypted attributed representation. However, compared to the problem to be addressed in this study, the functionalities provided by the above work are still very limited and insufficient in performing complex *XPath*-based XML queries over encrypted semistructured XML data.

In this paper, we propose to augment encrypted XML metadata with *encodings* which characterize the topology and content of every encrypted tree-structured XML data. Here, a hash-based approach is employed to compute hashed XML path information into encoding schemas. The filtering of non-candidate data set can then be performed by examining query conditions, expressed in terms of XPath expressions [7], against the encodings. We outline a generic framework for conducting efficient queries on encrypted XML data, which is comprised of three phases, namely, *query preparation, query pre-processing,* and *query execution.* The *query preparation* phase aims to prepare for efficient query answering by encoding XML DTDs and XML documents before they are encrypted and stored in the database. This phase runs off-line. When a query is issued at runtime, the *query pre-processing* phase filters out impossible query candidates, so that the decryption and query execution that the *query execution* phase undertakes can be more focused on potentially target XML data.

The remainder of the paper is organized as follows. Section 2 outlines the framework for efficient querying over encrypted XML metadata. Section 3 and

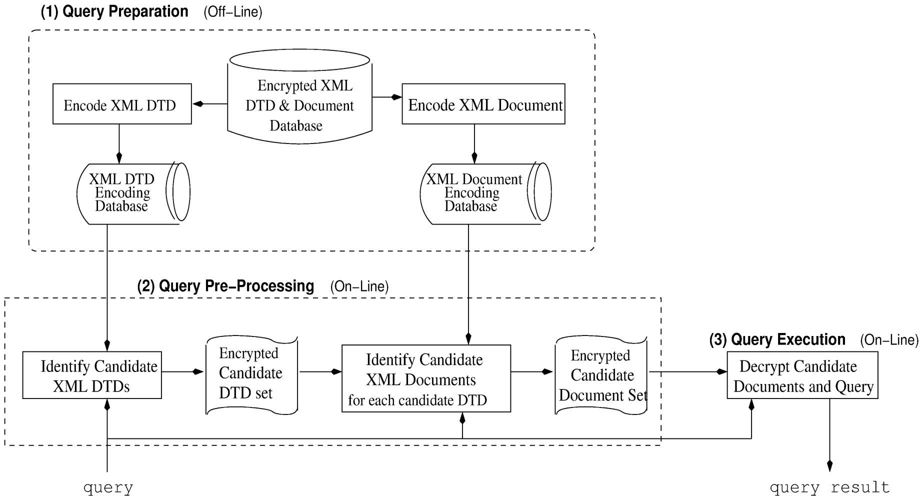

(1) Query Preparation (Off-Line)
Encode XML DTD
Encrypted XML DTD & Document Database
Encode XML Document
XML DTD Encoding Database
XML Document Encoding Database
(2) Query Pre-Processing (On-Line)
Identify Candidate XML DTDs
Encrypted Candidate DTD set
Identify Candidate XML Documents for each candidate DTD
Encrypted Candidate Document Set
(3) Query Execution (On-Line)
Decrypt Candidate Documents and Query
query
query result

Fig. 1. A framework for querying over encrypted XML data

Section 4 describe in detail the query preparation phase and query pre-processing phase, respectively. Section 5 concludes the paper.

2 A Framework for Searching Encrypted XML Metadata

We assume the availability of DTD for each XML document in this study. For security reason, all XML DTDs and XML documents are encrypted and stored in a database. Figure 1 shows a generic framework for conducting efficient queries on encrypted XML data. It is comprised of three phases: *query preparation, query pre-processing* and *query execution.*

- **Phase-1 (query preparation)**
  The aim of this phase is to prepare for efficient querying over encrypted XML data by encoding each XML DTD and associated documents before they are encrypted and stored in the database. Such an encoding is carried out in two steps, "*Encode XML DTD*" and "*Encode XML Document*". The coding results of XML DTDs and documents are stored in two separate databases, called "*XML DTD Encoding Database*" and "*XML Document Encoding Database*". In response to a query at run-time, these encodings can be used to pre-select **potential target documents** without the need to decrypt the whole document set in the database. We call these potential target documents **candidate documents** in the paper. Detailed encoding schemas and their computation will be described in Section 3.
- **Phase-2 (query pre-processing)**
  It is obvious that decrypting all encrypted XML documents to answer a query inevitably incurs an excessive overhead, especially when the target

data constitutes a small portion of the database. In order to make encrypted XML query processing truly practical and computationally tractable, and meanwhile preserve security, for each query, we incorporate a pre-processing stage, whose aim is to filter out impossible candidates so that decryption and query execution can be more focused on potentially target documents. Two steps are conducted in this phase. First, a set of candidate XML DTDs are identified through the step "*Identify Candidate XML DTDs*", which examines query conditions, expressed in terms of XPath expressions [7,8], against DTD encodings in the "*DTD Encoding Database*". Then, corresponding to each selected candidate DTD, the "*Identify Candidate XML Documents*" step further filters out candidate documents based on documents' encodings in the "*Document Encoding Database*". The candidate DTD set and document set returned are subsets of the original encrypted DTD set and document set, respectively. Detailed descriptions of these two steps will be given in Section 4.

– **Phase-3 (query execution)**
The identified candidate DTDs and documents, returned from Phase-2, are decrypted into clear DTDs and documents, on which the query can be executed. Here, conventional XML query engines can be employed in this phase. As querying over non-encrypted XML data has been widely investigated in the literature, we focus our study on Phase-1 and Phase-2 in this study.

3 Query Preparation Phase

In this section, we propose a hash-based strategy to encode encrypted XML data for the query preparation phase. Based on the encodings obtained, the query pre-processing phase, which will be addressed in Section 4, can then effectively filter out query candidates, i.e., potential targets, from among a large set of blind documents in the database. Due to different characteristics and functions of XML DTDs and documents, we encode XML DTDs and XML documents separately using different encoding schemas.

In the following, we first describe the computation method for encoding XML DTDs, followed by the method for encoding XML documents. For ease of explanation, a running example shown in Fig. 2 is used throughout the discussion. A graphical representation of the DOM tree structure of the example DTD, DTD_1, and the example document is outlined in Fig. 2(c).

3.1 Encoding XML DTDs

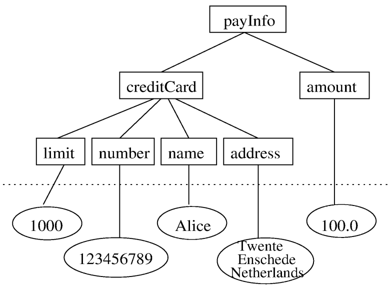
An XML DTD defines the legal building blocks of its conforming XML documents, like what elements, attributes, etc. are permitted in the documents [4]. These components construct a hierarchical tree structure that underlies the contents of the documents, with each path of the tree addressing a certain part of an document.

```
<!DOCTYPE payInfo [
    <!ELEMENT payInfo (creditCard?, amount+)>
    <!ELEMENT creditCard (number, name, address)>
    <!ATTLIST creditCard limit CDATA #IMPLIED>
    <!ELEMENT number (#PCDATA)>
    <!ELEMENT name (#PCDATA)>
    <!ELEMENT address (#PCDATA)>
    <!ELEMENT amount (#PCDATA)>
]>
```

(a) An XML DTD exmple – DTD1

```
<payInfo>
  <creditCard limit=1000>
      <number> 123456789 </number>
      <name> Alice </name>
      <address> Twente 7500 AE, Netherlands </address>
  </creditCard>
  <amount> 100.0 </amount>
</payInfo>
```

(b) An XML document exmple that conforms to DTD1



(c) A graphical representation of the DOM tree structure of DTD1 with the example document

Fig. 2. A running example of an XML document with its DTD

As the query pre-processing phase works on the basis of XPath expressions embedded in a query like XQuery, to prepare for efficient candidate selection, we thus take the strategy to encode each XML DTD in the unit of *path*. The notions of *path* and *path length* are defined as follows.

Definition 1. *A* ***path*** *p is a sequence of nodes $n_1, n_2, \ldots, n_k$, denoted as $p = (n_1/n_2/\ldots/n_k)$, where for any two consecutive nodes, n_i and n_{i+1} $(1 \leq i \leq k-1,\ k \geq 1)$, there exists an edge between them.*

The ***length*** *of path p, denoted as $|p|$, is the total number of edges in the path. That is, $|p = (n_1/n_2/\ldots/n_k)| = k\text{-}1$.* □

Table 1 lists all the paths, which are of various lengths, extracted from the example DTD DTD_1 in Fig. 2. Here, the content nodes under the dotted line are exempt from consideration, since they do not appear in the DTD.

In essence, we use the technique of hashing on each path of an XML DTD to compose DTD encodings. Paths of different lengths will be hashed into different hash tables named $DTDHashTable_0, DTDHashTable_1, DTDHashTable_2, \ldots,$ $DTDHashTable_{max\_pathLen}$, respectively. All paths of length l (where $1 \leq l \leq max\_pathLen$), no matter which DTD it comes from, will share one single hash table $DTDHashTable_l$, with each bucket indicating a set of DTDs, whose paths have been hashed into the bucket. Suppose we have a path p extracted from DTD_1, the hash function $HashFunc\ (p)$ computes its hash value, i.e., bucket address in the hash table $DTDHashTable_{|p|}$. (Detailed computation of hash values will be given shortly.) We mark the corresponding bucket entry with an indicator of DTD_1, signifying the DTD where p locates.

To filter out non-candidate DTDs for a query, we compute the hash values for all XPaths in the query using the same hash function, and then check the corresponding buckets in the DTD hash tables to obtain a subset of DTDs that

Table 1. Paths extracted from the example XML DTD

Path length	**Path**
2	p_1=(payInfo/creditCard/limit)
	p_2=(payInfo/creditCard/number)
	p_3=(payInfo/creditCard/name)
	p_4=(payInfo/creditCard/address)
1	p_5=(payInfo/creditCard)
	p_6=(payInfo/amount)
	p_7=(creditCard/limit)
	p_8=(creditCard/number)
	p_9=(creditCard/name)
	p_{10}=(creditCard/address)
0	p_{11}=(payInfo)
	p_{12}=(creditCard)
	p_{13}=(amount)
	p_{14}=(limit)
	p_{15}=(number)
	p_{16}=(name)
	p_{17}=(address)

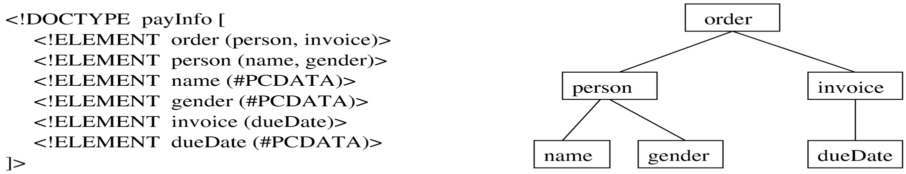
Algorithm 1 Hash function $HashFunc(p)$

```
Input: path $p = (n_1/n_2/\ldots/n_k)$,   a fixed size $s$ for node names,
          hash table size $SizeDTDHashTable_{|p|}$;
Output: hash value of $p$

1  For each node $n_i$ $(1 \le i \le k)$, chop its name uniformly into an $s$-letter string
   $ChopName(n_i,\ s) = x_{n_{i,1}} x_{n_{i,2}} \ldots x_{n_{i,s}}$,
   where $x_{n_{i,1}}, x_{n_{i,2}}, \ldots, x_{n_{i,s}}$ are letters in the name string of node $n$.

2  For each $s$-letter node name $x_{n_{i,1}} x_{n_{i,2}} \ldots x_{n_{i,s}}$, convert it into a decimal integer
   $Base26ValueOf(x_{n_{i,1}} x_{n_{i,2}} \ldots x_{n_{i,s}}) =$
      $offset(x_{n_{i,1}}) * 26^{s-1} + offset(x_{n_{i,2}}) * 26^{s-2} + \ldots + offset(x_{n_{i,s}}) * 26^0 \ = \ V_{n_i}$,
   where $offset(x_{n_{i,j}})$ $(1 \le j \le s)$ returns the position of letter $x_{n_{i,j}}$ among 26 letters.

3  Compute hash value of $p = (n_1/n_2/\ldots/n_k)$
   $HashFunc(n_1/n_2/\ldots/n_k) =$
      $(V_{n_1} * 10^{k-1} + V_{n_2} * 10^{k-2} + \ldots + V_{n_k} * 10^0)\ \ mod\ \ SizeDTDHashTable_{|p|}$.
```

possibly contain the requested paths. These DTDs are candidate DTDs to be considered for the query.

Algorithm 3.1 elaborates the procedures in computing the hash value for path $p = (n_1/n_2/\ \ldots\ /n_k)$. It proceeds in the following three steps.

First, node names in path p which could be of different lengths are uniformly chopped into the same size s, given by users as an input parameter,



```
<!DOCTYPE  payInfo [
   <!ELEMENT  order (person, invoice)>
   <!ELEMENT  person (name, gender)>
   <!ELEMENT  name (#PCDATA)>
   <!ELEMENT  gender (#PCDATA)>
   <!ELEMENT  invoice (dueDate)>
   <!ELEMENT  dueDate (#PCDATA)>
]>
```

(a) Another DTD example – DTD2

(b) A tree structure of DTD2

Fig. 3. Another DTD example with its DOM tree-structure

through the function $ChopName$ (Algorithm 3.1, line 1). For example, let s=4, $ChopName$(*"creditCard"*, 4) = *"cred"*, $ChopName$(*"payInfo"*, 4) = *"payI"*, $ChopName$(*"name"*, 4) = *"name"*.

Second, the chopped node name strings which are of fixed size after Step 1 are further converted into decimal integers via function $Base26ValueOf$ (Algorithm 3.1, line 2). Example 1 explicates how it works when the size of node name string is set to 4.

Example 1. When we let a 4-letter node name $x_1x_2x_3x_4$, which are case insensitive, represent a base-26 integer, we let the letter "a" represent the digit-value 0, the letter "b" represent the digit-value 1, the letter "c" represent the digit-value 2, the letter "d" represent the digit-value 3, and so on, up until the letter "z", which represents the digit-value 25. Given a letter, function "$offset$" returns such a digit-value. The 4-letter node name $x_1x_2x_3x_4$ can thus be converted into a decimal integer using the formula: $Base26ValueOf(x_1x_2x_3x_4) = offset(x_1) * 26^3 + offset(x_2) * 26^2 + offset(x_3) * 26^1 + offset(x_4) * 26^0$.

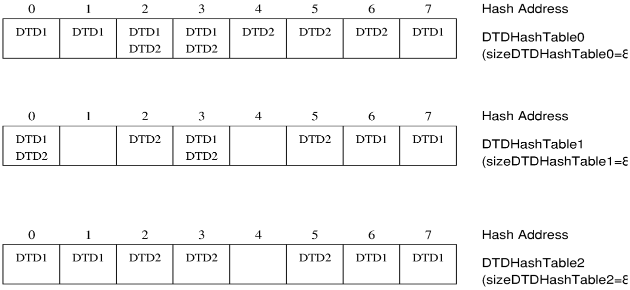
Assume that $x_1x_2x_3x_4$ = *"name"*, since the digit-values of *"n"*, *"a"*, *"m"* and *"e"* are $offset$(*"n"*) = 13, $offset$(*"a"*) = 0, $offset$(*"m"*) = 12, and $offset$(*"e"*) = 4 respectively, we have $Base26ValueOf$(*"name"*) $= 13 * 26^3 + 0 * 26^2 + 12 * 26^1 + 4 * 26^0 = 13 * 17576 + 0 + 312 + 4 = 228802$.

In a similar way, we have $Base26ValueOf$(*"cred"*) $= 2 * 26^3 + 17 * 26^2 + 4 * 26^1 + 3 * 26^0 = 2 * 17576 + 17 * 676 + 104 + 3 = 35152 + 11492 + 104 + 3 = 46751$. □

A general calculation of $Base26ValueOf$ is: $Base26ValueOf(x_1x_2 \ldots x_s) = offset(x_1) * 26^{s-1} + offset(x_2) * 26^{s-2} + \ldots + offset(x_s) * 26^0$.

Finally, hash function $HashFunc$ derives the hash value of $p = (n_1/n_2/\ldots/n_k)$ based on the value V_{n_i} returning from function $Base26ValueOf$ on each node n_i (Algorithm 3.1, line 3). $HashFunc(n_1/n_2/\ldots/n_k) = (V_{n_1} * 10^{k-1} + V_{n_2} * 10^{k-2} + \ldots + V_{n_k} * 10^0)\ mod\ SizeDTDHashTable_{|p|}$

In order to provide a more complete overview on the hash-based encoding method, we introduce another DTD example DTD_2 as shown in Fig. 3. Using the same hash function, Fig. 4 illustrates the hash results for all the paths from DTD_1 and DTD_2.



0	1	2	3	4	5	6	7	Hash Address
DTD1	DTD1	DTD1 DTD2	DTD1 DTD2	DTD2	DTD2	DTD2	DTD1	DTDHashTable0 (sizeDTDHashTable0=8

0	1	2	3	4	5	6	7	Hash Address
DTD1 DTD2		DTD2	DTD1 DTD2		DTD2	DTD1	DTD1	DTDHashTable1 (sizeDTDHashTable1=8

0	1	2	3	4	5	6	7	Hash Address
DTD1	DTD1	DTD2	DTD2		DTD2	DTD1	DTD1	DTDHashTable2 (sizeDTDHashTable2=8

Fig. 4. Encodings of the example DTD_1 and DTD_2 ($DTDHashTable_0$, $DTDHashTable_1$ and $DTDHashTable_2$)

Table 2. Pairs of element/attribute with content/value in the example XML document, together with their hash and mapped values

(**Element/Attribute** c_{name},, **Content/Value** c_{val})	$HashFunc(c_{name})$	$MapFunc(c_{val})$
c_1 = (limit, 1000)	0	1
c_2 = (number, 123456789)	1	10
c_3 = (name, "*Alice*")	0	0
c_4 = (address, "*Twente, Enschede, Netherlands*")	2	25
c_5 = (amount, 100.0)	1	7

3.2 Encoding XML Documents

XML documents that conform to one XML DTD possess a similar structure, but with possibly different element contents and/or attribute values to distinguish different documents. For instance, one conforming document of the example DTD shown in Fig. 2 has a *limit* attribute of value 1000, represented as *limit*=1000 for simplicity. Its elements *number*, *name*, *address* and *amount* have contents 123456789, "*Alice*", "*Twente, Enschede, Netherlands*" and 100.0, respectively.

After encoding XML DTDs, i.e., all possible paths with each containing a sequence of nodes corresponding to elements or attributes, the second task of the query preparation phase is to encode their conforming documents, i.e., all pairs of element and element content (*element, element content*), attribute and attribute value (*attribute, attribute value*). Due to the different nature of contents, encoding documents is conducted in a different way from encoding DTDs, with the result stored in the "*Document Encoding Database*". In the following, we describe the method of encoding a pair, $c = (c_{name},\ c_{val})$ (where c_{name} denotes the element/attribute, and c_{val} denotes the corresponding element content/attribute value), into a hash table named $DOCHashTable$.

We adopt the *separate chaining* strategy to resolve hashing collision for $DOCHashTable$. That is, we place all pairs that collide at a single hash address on a linked list starting at that address. The hash address of each pair is calculated via function $HashFunc(p)$ (Algorithm 3.1), using a different hash

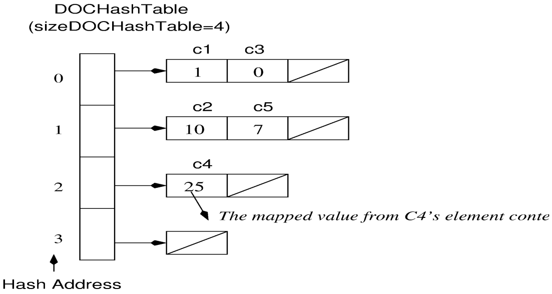


Fig. 5. Encodings of the example document (*DOCHashTable*)

table size, which is $SizeDOCHashTable$ rather than $SizeDTdHashTable_{|p|}$. In this case, path p always contains only one node, which is $p = (c_{name})$ and $|p| = 0$. For example, let s=4, and the size of hash table $DOCHashTable$ equal to 4 (i.e., $SizeDOCHashTable = 4$). We have $ChopName$(*"limit"*) = *"limi"*. $Base26ValueOf$(*"limi"*) $= 11 * 26^3 + 8 * 26^2 + 12 * 26 + 8 = 199064$, $HashFunc(limit) = 199064 * 10^0 \; mod \; 4 = 0$.

After the derivation of bucket address in the hash table $DOCHashTable$ from c_{name}, the entry to be put into the corresponding bucket is computed based on c_{val}, using the technique developed in [10]. The basic idea is to first divide the domain of node c_{name} into a set of *complete* and *disjoint* partitions. That is, these partitions taken together cover the whole domain; and any two partitions do not overlap. Each partition is assigned a *unique* integer identifier. The value c_{val} of element/attribute node c_{name} is then mapped to an integer, corresponding to the partition where it falls [10]. For example, we can partition the domain of attribute *limit* into [0, 500], (500, 1000], (1000, ∞] of identifier 0, 1, 2, respectively. The *limit* value 1000 is thus mapped to integer 1, and stored in the first bucket of $DOCHashTable$, since $HashFunc(limit) = 0$. The hash values for other pairs in the example document are calculated in the same way, which are shown in Table 2.

Note that the partition of a domain can be done based on the *semantics* of data and relevant applications. For instance, we can categorize the domain of element *name* according to the alphabetical order. The domain of element *address* can be partitioned according to *province* or *country* where located. For simplicity, in the current study, we enforce *order preserving* constraint on such a mapping "$MapFunc : domain(c_{name}) \rightarrow Integer$", which means that for any two values c_{val_1} and c_{val_2} in the domain of c_{name}, if $(c_{val_1} \leq c_{val_2})$, then $MapFunc(c_{val_1}) \leq MapFunc(c_{val_2})$.

Assume the mapping functions for *number*, *name*, *address* and *amount* return identifiers, as indicated in Table 2. Figure 5 plots the resulting encoding, i.e., $DOCHashTable$, for the example XML document given in Fig. 2.

4 Query Pre-processing Phase

The aim of the query pre-processing phase is to identify candidate DTDs and documents by checking the query against the encodings of DTDs and documents, obtained after the query preparation phase. In this section, we first provide a brief description of XPath expressions used in query representation. We then discuss a method to match such XPath expressions to paths as described in Section 3 in order to facilitate candidate DTD and document selection. A two-step procedure is finally illustrated to identify candidate DTDs, followed by candidate documents for each selected candidate DTD.

4.1 XPath Expressions

The XPath language is a W3C proposed standard for addressing parts of an XML document [7]. It treats XML documents as a tree of nodes corresponding to elements/attributes, and offers an expressive way to specify and locate nodes within this tree.

XPath expressions state structural patterns that can be matched to paths, consisting of a sequence of nodes in the XML data tree [2,1]. Such paths can be either absolute paths from the root of the data tree, or relative one starting with some known context nodes. The hierarchical relationships between the nodes are specified in XPath expressions using parent-child operator ("/") and ancestor-descendant operator ("//"). For example, the XPath expression "*/payInfo/creditCard/@limit*" addresses *limit* attribute of *creditCard* which is a child element of the *payInfo* root element in the document. The *name* element in the relative path expression "*//creditCard/name*" is a child relative to its parent *creditCard* element. The expression "*/payInfo//name*" addresses *name* descendant element of the *payInfo* root element.

XPath also allows the use of a wildcard operator ("*" or "@*"), which can match any element or attribute node with respect to the context node in the document data tree. In addition, predicates, enclosed in square brackets ("[ ]"), can also be applied to further refine the selected set of nodes in XPath expressions. For example, "*/payInfo/creditCard*[*@limit* $<$ 1000]*/name*" selects the *name* elements of the XML document if the attribute *limit* of *creditCard* has a value less than 1000.

Operators like ("|") and ("*and*") can also be applied to select constituent nodes of paths [7]. For instance, "*/payInfo/*(*creditCard*|*cash*)*/name*" expression selects every *name* element that has a parent that is either a *creditCard* or a *cash* element, that in turn is a child of a root element *payInfo*. On the contrary, "*/payInfo/creditCard*[*@limit* and *@dueDate*]" indicates all the *creditCard* children of the root element *payInfo* that must have both a *limit* attribute and a *dueDate* attribute.

4.2 Mapping XPath Expressions to Paths

Considering that DTD encodings are computed against *paths* as defined in Definition 1 in Section 3, for efficient encoding-based query candidate pre-selection, we first need to match an *XPath expression* e, which is used to locate parts of a data tree, to a set of *path*s through the following three steps.

Step 1. Decompose XPath expression e into several ones at the point of "//" operator.

Since paths to be encoded by the offline query preparation phase have only parent-child relationships ("/") between two consecutive nodes (as shown in Table 1), we break an XPath expression from the points where the "//" operator locates, into several ones where each node, except for the first one, is prefixed only by "/". The resulting XPath expressions thus contain no ancestor-descendant relationships ("//") between every two consecutive nodes.

For ease of explanation, we signify the XPath expressions derived after Step 1 using a prime symbol like e'. They form the input of Step 2.

Step 2. Simplify predicate constraints in each XPath expression e' to only hierarchical relationships.

As DTD encoding relieves value constraints on path nodes, and focuses only on their hierarchical relationships, to facilitate candidate DTD filtering based on path encodings, we relax value constraints on nodes like "[$amount > 100$]" and "[@$limit = 1000$]", specified in XPath predicate expressions, and keep only their inherent parent-child or element-attribute relationships.

Let e'' denote an XPath expression returned after Step 2.

Step 3. Eliminate logical "|" and "*and*" operators in each XPath expression e'' by rewriting the expression into several ones logically connected with "$\wedge$" or "$\vee$".

To match the notion of path in Definition 1, every XPath expression after Step 2 which contains the logical operators "|" and "*and*" is substituted by a set of shorter XPath expressions, which are logically connected via "$\wedge$" or "$\vee$".

After undergoing the above three steps, an original XPath expression is transformed into a set of **simple XPath expressions**, which contain no ancestor-descendant relationships between two consecutive nodes, no value constraints on nodes, and no logical operators ("|") and ("*and*"). Each such kind of simple XPath expressions corresponds to a path defined in Definition 1.

Example 2. From an original XPath expression "$/payInfo[amount > 100]//name$", we can derive two simple XPath expressions "$/payInfo/amount$" $\wedge$ "$//name$".

An XPath expression with a predicate constraint and operator ("|") like "$/payInfo[amount > 100]/(creditCard|cash)/name$" leads to three simple XPath expressions which are: "$/payInfo/amount$" $\wedge$ ("$/payInfo/creditCard/name$" $\vee$ "$/payInfo/cash/name$"). □

4.3 Identification of Candidate DTDs and Documents

On the basis of simple XPath expressions generated from XPath expressions embedded in a query, we can now define the concepts of candidate DTDs and documents for the given query.

An XML DTD is called a **candidate DTD** for a query, if for every simple XPath expression derived from the query, there *possibly* exists a path p in the DTD, that matches this simple XPath expression.

In a similar fashion, we define that an XML document is a **candidate document** for a query, if and only if: 1) its DTD is a candidate DTD[1]; and 2) it *possibly* satisfies all predicate constraints on the nodes inside all the XPath expressions embedded in the query.

The pre-selection of potential query targets starts with the identification of candidate DTDs, followed by the identification of candidate documents under each candidate DTD that has been identified.

I. Identify Candidate DTDs by Hashing Paths

Given a query, to check out which encrypted DTDs are candidate DTDs for each simple XPath expression generated from the query, we match it to a path p, and compute the hash value for p using the same hash function $HashFunc(p)$ (Algorithm 3.1) while encoding the DTDs. According to the hash value (i.e., bucket address) returned, we consult with the corresponding bucket in the hash table $DTDHashTable_{|p|}$, which gives all the DTDs that may possibly contain path p. The rationale for this is straightforward: *if path p is present in the DTD, it will be hashed to the bucket in* $DTDHashTable_{|p|}$*, leaving a mark for this DTD in the bucket entry.*

Example 3. Suppose a query consists of only one simple XPath expression, corresponding to the path $p = (payInfo/creditCard/dueDate)$. Referring to the DTD encoding schema illustrated in Fig. 4, where $s = 4$ and
$SizeDTDHashTable_2 = 8$, its hash value is computed as follows:
Step 1: $ChopName(\text{“}payInfo\text{”},\ 4) = \text{“}payI\text{”}$, $ChopName(\text{“}creditCard\text{”},\ 4) = \text{“}cred\text{”}$, $ChopName(\text{“}dueDate\text{”},\ 4) = \text{“}dueD\text{”}$.
Step 2: $Base26ValueOf(\text{“}payI\text{”}) = 264272$, $Base26ValueOf(\text{“}cred\text{”}) = 46751$, $Base26ValueOf(\text{“}dueD\text{”}) = 66355$.
Step 3: $HashFunc(payInfo/creditCard/dueDate) = (Base26ValueOf(\text{“}PayI\text{”}) *10^2 + (Base26ValueOf(\text{“}cred\text{”}) * 10^1 + Base26ValueOf(\text{“}dueD\text{”}) * 10^0)\ mod\ SizeDTDHashTable_2 = (264272{*}100{+}46751{*}10{+}66355)\ mod\ 8\ =\ 1$

Due to its hash value 1, we can be sure that the example DTD_2 does not contain that path, since the entry at address 1 in $DTDHashTable_2$ only signifies DTD_1. As a result, only DTD_1 will be returned as the candidate DTD, DTD_2 and its associated documents can thus be discarded from the further search. □

II. Identify Candidate Documents by Hashing Element/Attribute and Content/Value Pairs

After pre-selecting the candidate DTD set for the given query, we are now in the position to filter out candidate documents underneath each candidate DTD. At this stage, various value constraints in the form of $[c_{name}\ \theta\ c_{val}]$ (where c_{name} denotes the name of an element/attribute node, θ is one of the operators in $\{=, \neq, <, \leq, >, \geq\}$, and c_{val} denotes the element content/attribute value) on

[1] Recall that we assume the availability of DTD for each document in this study.

path nodes are taken into consideration. Clearly, a candidate document must not violate any of the value constraints specified within the XPath expressions in the query. We perform such kind of examination based on the document encodings (i.e., $DOCHashTable$).

Taking the constraint $[c_{name}\ \theta\ c_{val}]$ for example, we first hash the node name c_{name} (i.e., a path containing only one node) into $DOCHashTable$ via hash function $HashFunc(c_{name})$. Meanwhile, we also calculate the range identifier of c_{val} using the order preserving function $MapFunc(c_{val})$. Finally, we compare each entry value v linked to the bucket address $HashFunc(c_{name})$ in $DOCHashTable$: if $\exists v\ (v\ \theta\ MapFunc(c_{val}))$, then the constraint $[c_{name}\ \theta\ c_{val}]$ possibly holds.

Example 4. Assume a query embeds an XPath expression "$/payInfo/creditCard\ [@limit\ >\ 2000]/name$", which enforces a constraint $[@limit > 2000]$ on $creditCard$ element. Referring to the document encoding schema in Fig. 5, where $s = 4$ and $SizeDOCHashTable = 4$. We have $HashFunc(limit) = 0$ and $MapFunc(2000) = 2$. Since all the entries at address 0 in $DOCHashTable$ are either 1 or 0, which is not greater than 2 ($= MapFunc(2000)$), therefore, the example document is not a candidate document for this query, and can thus be discarded. □

5 Conclusion

In this paper, we employ the hash technique to compute encodings associated with each encrypted XML metadata to allow effective pre-filtering of non-candidate data for a given query, expressed in terms of XPath expressions. We are currently conducting experiments to investigate the performance of the proposed strategy.

References

1. M. Altinel and M. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proc. the 26th Intl. Conf. on Very Large Data Bases*, pages 53–64, Cairo, Egypt, September 2000.
2. C. Chan, P. Felber, M. Carofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. In *Proc. the Intl. Conf. on Data Engineering*, California, USA, February 2002.
3. European Commission. Scenarios for ambient intelligence in 2010. http://www.cordis.lu/ ist/istag.htm, 2001.
4. World Wide Web Consortium. Extensible markup language (XML) 1.0. http://www.w3.org/ TR/REC-xml, October 2000.
5. World Wide Web Consortium. XML encryption requirements. http://www.w3.org/ TR/xml-encryption-req, March 2002.
6. World Wide Web Consortium. XML encryption syntax and processing. http://www.w3.org/ TR/xmlenc-core/, August 2002.

7. World Wide Web Consortium. XML path language (XPath) 2.0. http://www.w3.org/ TR/xpath20/, november 2002.
8. World Wide Web Consortium. XQuery 1.0: an XML query language. http://www.w3.org/ TR/xquery/, november 2002.
9. E. Dijkstra, W. Jonker, and H. van Gageldonk. Data and content management (chapter). In The New Everyday – Views on Ambient Intelligence, E. Aarts and S. Marzano (eds.), Koninklijke Philips Electronics N.V., ISBN 90-6450-502-0, 2003.
10. H. Hacigümüş, B. Lyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *Proc. the ACM SIGMOD Intl. Conf. on Management of Data*, pages 216–227, Wisconsin, USA, June 2002.
11. H. Hacigümüş, B. Lyer, and S. Mehrotra. Providing database as a service. In *Proc. Intl. Conf. on Data Engineering*, 2002.
12. W. Jonker. XML and secure data managament in an ambient world. *Computer Systems Science & Engineering (to appear)*, 2003.
13. D. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proc. the IEEE Symposium on Security and Privacy*, 2000.