The Journal of Systems and Software 84 (2011) 1292–1305

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Developing an efficient query system for encrypted XML documents☆

Tao-Ku Chang[a,*], Gwan-Hwan Hwang[b]

[a] Department of Computer Science and Information Engineering, National Dong Hwa University, Hualien, Taiwan
[b] Department of Computer Science and Information Engineering, National Taiwan Normal University, Taipei, Taiwan

ARTICLE INFO

Article history:
Received 30 July 2010
Received in revised form 28 January 2011
Accepted 6 April 2011
Available online 22 April 2011

Keywords:
XML
XQuery
DSL
Security

ABSTRACT

XQuery is a query and functional programming language that is designed for querying the data in XML documents. This paper addresses how to efficiently query encrypted XML documents using XQuery, with the key point being how to eliminate redundant decryption so as to accelerate the querying process. We propose a processing model that can automatically translate the XQuery statements for encrypted XML documents. The implementation and experimental results demonstrate the practicality of the proposed model.

1. Introduction

XQuery language (Boag et al., 2007) is a technology from W3C that is designed to be broadly applicable across all types of XML data sources. It provides flexible query functionality to extract data from real and virtual documents on the fly or in a database. XQuery uses an XML data model that can represent XML documents, sequences, or atomic elements such as integers or strings. The XQuery specification specifies a processing model that prescribes that an XQuery processor interacts with its environment and what steps it must take in order to evaluate a query (see Fig. 1). An XQuery program includes navigation in XML documents using XPath (Clark and DeRose, 1999), database statements (the so-called FLWOR expressions), construction of new XML elements, operations on XML Schema types, and function calls. The XQuery processor queries and formats data from an XML database that stores XML documents according to the instructions in the XQuery program, and produces the applicable XML document.

The W3C lists more than 50 XQuery implementations (see http://www.w3.org/XML/Query/). For example, Galax is a lightweight and extensible implementation of XQuery 1.0 that closely tracks the definition of XQuery 1.0 as specified by the W3C, which means that it also implements XPath 2.0 (a subset of XQuery 1.0). Qexo is a partial implementation of the XQuery language that exhibits good performance because a query is compiled down to the Java byte codes using the Kawa framework. Sedna, which is based on a native XML store, implements the standard layered architecture using dynamically switching between pull- and push-based execution at run-time. Zorba is an open-source XQuery processor that is designed to be embeddable in a variety of environments, such as other programming languages extended with XML processing capabilities, browsers, database servers, XML message dispatchers, or smart phones. Saxon is a complete and conformable implementation of XSLT 2.0, XQuery 1.0, and XPath 2.0.

XML is useful because it reduces costs by increasing the flexibility of data management in various ways. XML is platform-independent and based on Unicode, which means that it supports all languages and alphabets. XML is becoming a widespread data-encoding format for Web applications and services, which makes it increasingly important to be able to safeguard the accuracy of the information represented in XML documents. For example, we may need to sign and encrypt XML documents in order to ensure nonrepudiation and confidentiality (Schneier, 1995). Based on XML element-wise encryption (Maruyama and Imamura, 2000), the XML-encryption working group of the W3C (XML Encryption, 2001) delivered a recommendation specification for XML encryption (Imamura et al., 2002). The encrypted document specifies a process for encrypting data and representing the result in XML. The encrypted data may be arbitrary data, an XML element, or the content of an XML element. Fig. 2 illustrates the concept of element-wise encryption. Fig. 2(A) represents that Tony Chen's payment information. Chen's credit number is sensitive and it must be kept confidential. The entire "`CreditCard`" element is encrypted and shown in Fig. 2(B). Fig. 2(C) represents that both "`CreditCard`" and

☆ T.K. Chang's and G.H. Hwang's work were supported in part by the ROC National Science Council under grant 98-2221-E-259-011.
* Corresponding author. Fax: +886 3 8634010.
E-mail address: tkchang@mail.ndhu.edu.tw (T.-K. Chang).

0164-1212/$ – see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2011.04.012

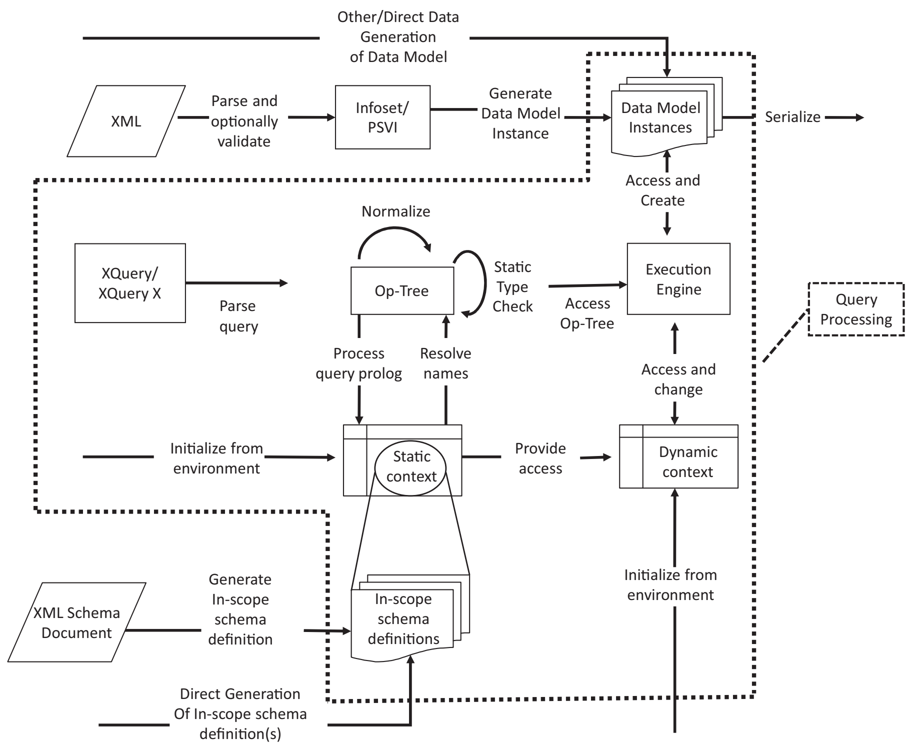

Fig. 1. The query processing model.

"Number" are in the clear, but the character data content of "Number" is encrypted. This enables XML files to be protected because the sensitive data in XML are encrypted by particular keys.

This paper addresses how to query data from these encrypted XML documents with XQuery. The intuitive, trivial way is to first decrypt the whole encrypted XML documents and then use an XQuery program to obtain the desired documents (see Fig. 3). The drawback of this approach is that it is quite inefficient in certain situations because all of the encrypted elements in the queried XML document must be decrypted. According to its operational semantics, XQuery is normally used to obtain a small set of elements from the target XML documents. It is not theoretically necessary to decrypt all the encrypted elements in the target XML document – we only have to decrypt those elements that belong to the result elements of the issued query. It is obvious that a scheme that does not need to decrypt unwanted elements should be more efficient than a scheme that decrypts all the encrypted elements.

The first aim is to eliminate unnecessary decryption. According to the specification of W3C XML encryption (Imamura et al., 2002), the scopes of encryption could be "element", which encrypts a whole element (including the start/end tags), or "content", which encrypts the content of an element (between the start/end tags). Consider the XML document shown in Fig. 4. The "payer" and "cardinfo" elements are encrypted as a whole; that is, their

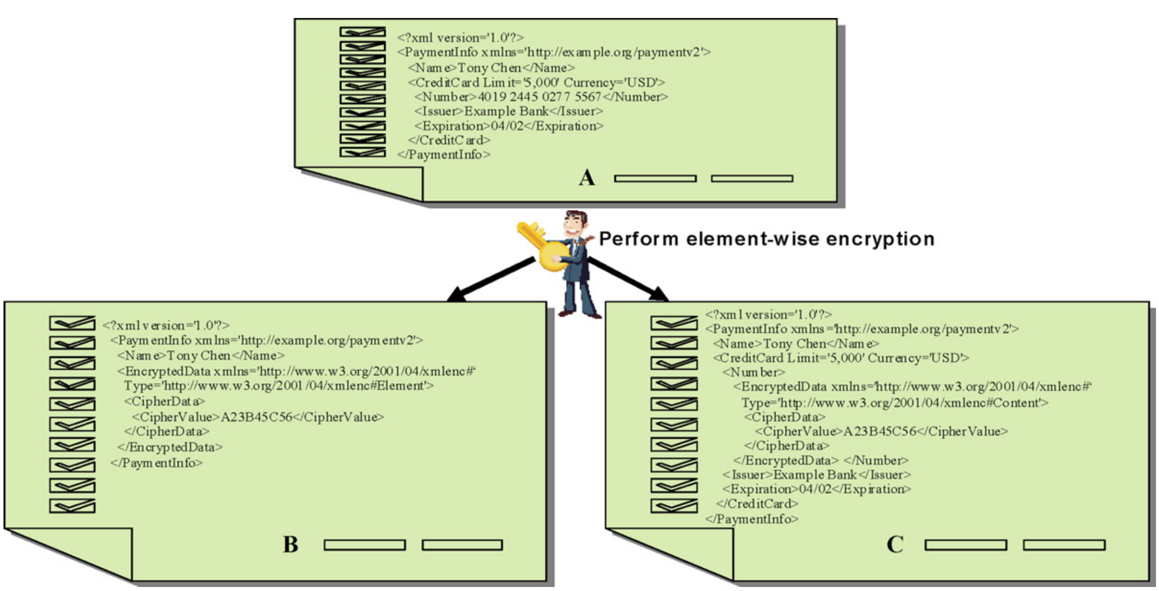

Fig. 2. Example of element-wise encryption.

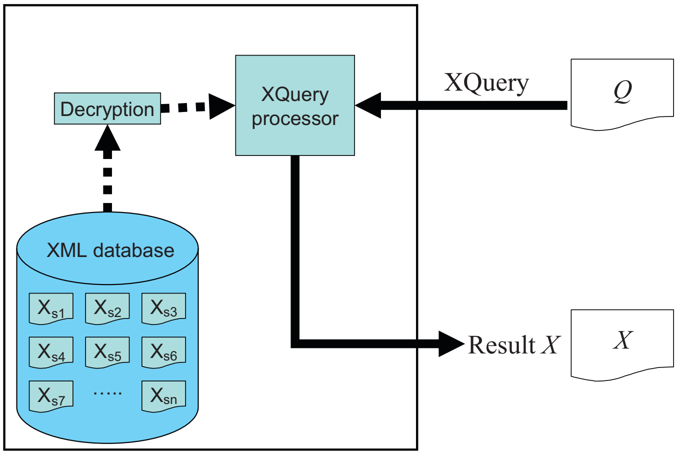

Fig. 3. A trivial way to query encrypted XML documents.

encryption scope is set to “element”. In the encrypted XML document shown in Fig. 5, the “CipherData” element contains the encrypted data of the “payer” and “cardinfo” elements, and is wrapped by the “EncryptedData” element. Note that the tag names of the “payer” and “cardinfo” elements disappear. Fig. 5 indicates that once the encryption scope of an element is set to “element”, its tag name cannot be examined unless we first decrypt the element. The type of encryption scope is helpful to data security because there is no clue about which element is encrypted. Fig. 6 lists an XQuery program that is used to obtain the value of the “cardinfo” element from Fig. 4. It is obvious that we cannot use this program to query the encrypted document shown in Fig. 5; it appears that we have to decrypt the two encrypted elements before performing the query. However, since we only want to query one of them, the other decryption is redundant.

Some researchers have been working on queries over encrypted XML document (Schrefl et al., 2005; Yang et al., 2006; Wang and Lakshmanan, 2006; Lee and Whang, 2006). For example, Schrefl et al. (2005) proposed techniques for processing queries and updates encrypted XML documents stored on the distrustful servers. By performing encryption and decryption only on the client but not on the server, it guarantees that neither the document structure nor the document content is disclosed on the server. Yang et al. (2006) proposed XQEnc an XML encryption technique using vectorization and skeleton compression. This approach stores the schema of the XML document as a compressed skeleton on the client and is very efficient since it only retrieves the necessary data for the client to decrypt. Wang and Lakshmanan (2006) proposed a metadata mechanism including structural and value indices at server side that enables efficient query evaluation. Lee and Whang (2006) proposed the notion of Query-Aware Decryption. This approach disseminates an encrypted XML index along with the encrypted XML data. This index will inform where the query results are located in the encrypted XML data; thus preventing doing unnecessary decryption in other parts of the data.

To improve the efficiency of decryption of encrypted XML documents in the query process, we should avoid performing unnecessary decryption. For the example shown in Figs. 4–6, it is obvious that some additional information is necessary to eliminate the redundant decryption because the encryption may break the structure of the XML document. Sometimes the structure information should be referred to during the query. In this paper, we present the type of information required to eliminate redundant decryption and propose a processing model to automatically translate an XQuery program written by users to another one that can accurately locate the target elements that should be decrypted. The presented translation algorithm is optimal in terms of the computation required for decryption.

The remainder of this paper is organized as follows: Section 2 presents the proposed processing model, Section 3 presents an algorithm for the transformation of XQuery statements for querying encrypted XML documents, Section 4 presents our implementation and experimental results, and Section 5 concludes the paper.

2. The processing model for querying encrypted XML documents

Optimally querying the encrypted XML documents in XQuery requires information about security. Note that an optimal query is defined as that requiring minimal decryption for encrypted elements in the target XML documents. Generally speaking, the encryption and signature standards proposed by W3C offer a complete definition of the format for the encrypted XML document (Imamura et al., 2002; Bartel et al., 2008). However, the language is not sufficiently powerful for the programmer to specify how to encrypt and sign his or her XML documents. To overcome this limitation, a security language called document security language (DSL) that allows a programmer to specify the security detail of XML documents was proposed (Hwang and Chang, 2001, 2004). Fig. 7 illustrates the relationship between XML, DSL, and the DSL securing tool. The DSL can be used to define how to perform encryption and decryption, and the embedding and verification of signatures. It

```
<?xml version='1.0'?>
<transactions>
  <transaction>
    <payer id = “M123456789”>Jessie Chang</payer>
    <price current="TWD">1688</price>
    <cardinfo>
      <cardno>1234-5678-8765-4321</cardno>
      <cardtype>g</cardtype>
      <issuer>visa</issuer>
      <owner>Jessie Chang</owner>
      <creditline>200000</creditline>
      <expiredate>12/01/2007</expiredate>
    </cardinfo>
  </transaction>
</transactions>
```

Fig. 4. An XML document.

```
<?xml version='1.0'?>
<transactions>
  <transaction>
    <EncryptedData Type='http://www.w3.org/2001/04/xmlenc#Element'
     xmlns='http://www.w3.org/2001/04/xmlenc#'>
      <CipherData>
        <CipherValue>
          NiuuCTf1c/vlFmdL0+jAk7P2C7XaMvEFXh3/YRh3o/8YcFXOQLvOifWNs0Idditl
          x1zXVnAF2T23TdHTMbPhZDpeLIj3o0/kvaWx5wh0l7lF4mfOGbcGIxygb1bGfIYi
          NqnrwPz7g7TTW08/XPMUxaNw 2NB/rHBYZnjF0uN/ivs=
        <CipherValue>
      <CipherData>
    </EncryptedData>
    <price current="TWD">1688</price>
    <EncryptedData Type='http://www.w3.org/2001/04/xmlenc#Element'
     xmlns='http://www.w3.org/2001/04/xmlenc#'>
      <CipherData>
        <CipherValue>
          Ua245VGhYlAHFmwfoFmzcb9oVGgR7AoN+fD3I9LHHI5QinunkPssKEns8Ss79MYX
          sOLR1GE335F6TcFC5BjM1YEFnhfsmHUylWK/WGrPn0PHLYRKgGliuWdPuFJRlqwT
          uRg1E6Hi8cG6g+QGdcV9ndIfHRkjN10nj8XGrC+J6Z8WUP1zsmpSH+fiP98z0Kx4
          abFf40dl2byY7jyGyLX2/Mi15Nt3izsZ9scurWHTYpLoV7GRNbENLV3ONw5t498d
          mcSCPPaHNba+Z9UrPd1fMiHjBsJyYrzfYvEI0j69fW2PLP845YyEGxjAkjrV+sb1
          BhtgN+KTaUnRhE7lExKA1rHxZiAomCSd0+JxVK7BblFQ7Zu519DOfUikoQC5NaLJ
          sk3IMhqtbzhcyJgJHoiee0ldyc62HER7fYrKD/zBfKbM3r4C8k0aVAs6hgsdTfIy
          TngoEAGionLHGxkWergsw1xrrQuwgw9x7e5zt5F9Mu/S5UXu77hUxaxL2VBQE71q
          H+MMOHBsolevOlfodJ+Rbk5LVCUryxtjEfTJdPWayY42crcb2Biq2WSgPdjug0b9
          m1lhZWqx2b2PoamykWIaBL6Jqa9LOLFTGE8CiyXCjdlD19ynmDHChW6MOu+YJwJb
          HMrzC3eQtnrQ8n+ZU/jKYCcZO1K6wCzYAxwB9y/n+NU=
        <CipherValue>
      <CipherData>
    </EncryptedData>
  </transaction>
</transactions>
```

Fig. 5. An encrypted XML document.

offers a security mechanism that integrates *element-wise encryption* and *temporal-based element-wise digital signatures*.

Because the syntax of the "EncryptedData" element in the XML encryption standard prevents its extension to handle attribute encryption, the DSL supports a type of element-wise encryption that is more general: the scope of encryption (or encryption granularity) can be a whole element, some of the attributes of an element, or the content of an element; where an attribute has two possible types of encryption: (1) to only encrypt its value and (2) to encrypt both its name and value (Chang and Hwang, 2003). The encrypted document produced by the *DSL securing tool* can be made compatible with the XML encryption and digital signature standard in cases where attribute encryption is not applied. In addition, we have developed a DSL editor with a graphical user-friendly interface to make it easier for users to generate DSL documents (DSL Editor, 2011).

Fig. 8 depicts the processing model we propose for the efficient querying of encrypted XML documents. Q is the original XQuery program. Note that Q is written to query data from the original XML document (i.e., the unencrypted document). D is a DSL document. The encrypted XML document X_S is encrypted according to D and is stored in the XML storage. Before Q is sent to the XQuery processor, the translator parses it and translates it into Q'. Q' is also an XQuery program, but some expressions in it are translated according to D and the XML Schema S (Fallside and Walmsley, 2004). In cases where the result document R contains some encrypted ele-

```
<transactions>
   {
     for $b in doc("example.xml")/transactions/transaction/cardinfo
     where $b/cardno = "1234-5678-8765-4321"
     return
       $b/cardno
   }
</transactions>
```

Fig. 6. An XQuery to extract "cardinfo" from an XML file.

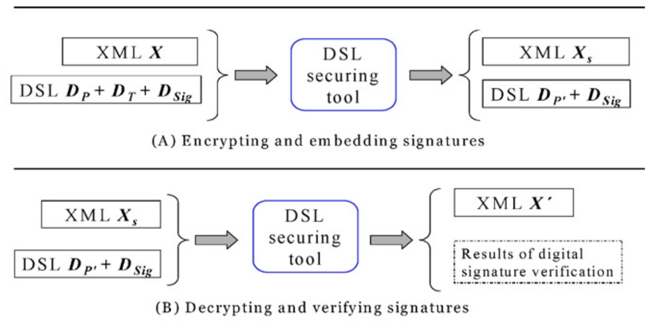

Fig. 7. The operational model for securing XML documents.

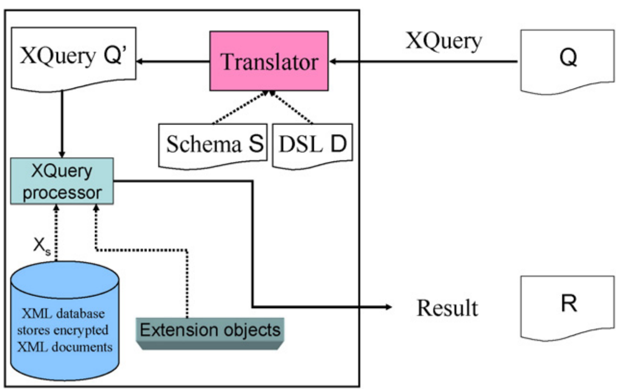

Fig. 8. The processing model for querying encrypted XML documents.

ments in X_s or the query needs to consult some encrypted element in X_s, Q' contains codes to invoke decryption functions that are the extension objects. Note that the XML Schema S plays an import role; in certain circumstances the information contained in S can be used to generate a more efficient query compared with a transformation obtained by only consulting D. The translation from Q to Q' is detailed in Section 3.

3. The transformation algorithm of XQuery for querying encrypted XML documents

Now we present our design of an algorithm that is used to transform the XQuery program; that is, the design of the translator shown in Fig. 8. We begin by considering the syntax of the XQuery statement. Each XQuery program contains one or more query expressions. The FLWOR expression is the most powerful of the XQuery expressions and is, in many ways, similar to the `SELECT-FROM-WHERE` statement used in SQL (ISO, 1999). The formal grammar for a FLWOR expression in XQuery is defined in (Boag et al., 2007) as follows:

```
FLWORExpr ::= (ForClause | LetClause)
              WhereClause? OrderByClause? return ExprSingle
```

The above BNF[1] form of the FLWOR expression is quite protean, being capable of generating a large number of possible query instances. The `ExprSingle` term following the "`return`" keyword can itself be replaced by another FLWOR expression, so that FLWOR expressions can be strung together ad infinitum. The replacement of an `ExprSingle` term by any other expression type is what makes XQuery composable and gives it its rich, expressive power. There are many expression types in XQuery, each of which can be plugged into the grammar wherever a more generic `ExprSingle` expression is called for.

3.1. The transformation algorithm

In this paper, we focus on FLWOR expressions to implement the transformation algorithm, which is listed in Fig. 9. The input includes a FLWOR expression, a DSL document, and an XML Schema. The output is a translated FLWOR expression. `Step 1` defines some variables: "*T_set*" represents the set of path templates in the DSL file, "*I_set*" represents the set of paths in "`ForClause`" and "`WhereClause`", and "*R_set*" represents the set of paths referred to in `ExprSingle`. Note that if `ExprSingle` is a FLWOR expression, we do not add the paths referred to in the FLWOR expression to "*R_set*". We present the situation in which `ExprSingle` is a FLWOR expression in the third example. In `Step 2`, we first compute the intersections of "*I_set*" and "*T_set*" and of "*R_set*" and "*T_set*". The intersection of "*I_set*" and "*T_set*" is not the empty set when the queried elements according to "`ForClause`" and "`WhereClause`" contain encrypted elements. Similarly, the intersection of "*I_set*" and "*R_set*" is not empty when the return elements contain encrypted elements. According the previous computation, the original XQuery expression will be appropriately translated. Some functions and procedures are designed for this algorithm. `Intersection` function returns the true Boolean value that indicates there is intersection of two sets of the paths. `Xpath_Transformation` procedure is to translate the set of paths in `ForClause` when they contain encrypted elements. `Decryption_Scope` procedure returns the scope information that will be used by `Decryption` function. `Decryption` function performs the decryption. In the appendix, we present the algorithm of these two procedures. We design a `Schema` class that provides one static method "`getIndex`" to consult information from an XML Schema.

In the following we use four examples to demonstrate this algorithm.

The first example demonstrates an XQuery program that queries some of the encrypted elements from the target XML document. Fig. 10A lists a FLWOR expression that performs a simple search that returns the "`cardinfo`" element from the document `example.xml` (see Fig. 4) where the value of "`/transactions/transaction/price`" is "`1688`". The XML document shown in Fig. 5 is that encrypted according to the DSL document shown in Fig. 11. In this example there are two path templates in the DSL document (see Fig. 11), and we have *T_set* = {"`/transactions/transaction/payer`," "`/transactions/transaction/cardinfo`"}, `ForClause` = "`for $b in doc("example.xml")/transactions/transaction`", `WhereClause?` = "`where $b/price=1688`", *I_set* = {"`/transactions/transaction`," "`/transactions/transaction/price`"}, `ExprSingle` = "`$b/cardinfo`", and *R_set* = {"`/transactions/transaction/cardinfo`"}. The intersection of "*I_set*" and "*T_set*" is the empty set, whereas that of *R_set* and *T_set* is not the empty set. According to the algorithm listed in Fig. 9, the translator then generates the transformed FLWOR expression. The "`ForClause`" and "`WhereClause?`" statements are changed to "`for $b_1 in doc("example.xml")/transactions/transaction`" and "`where $b_1/price=1688`", respectively. A "`LetClause`" statement ("`let $b=decryption($b_1, "child:EncryptedData[2]")`") is added after the "`ForClause`" and "`WhereClause?`" statements. Note that "`LetClause`" invokes a decryption function to decrypt the `$b_1` variable since it contains

[1] See Fischer and LeBlanc (1991) for more information about the BNF representation. In this paper, all the nonterminal symbols are underscored.

```
Algorithm: Transform a FLWOR expression for querying encrypted XML documents
Input:
  Let F[2] be a FLWOR expression of the form:
      FLWORExpr ::= (ForClause | LetClause)

                    WhereClause?  OrderByClause?  return ExprSingle
  Let D be a DSL file
  Let S be an XML Schema
Output:
  N = A FLWOR expression
Begin_of_Algorithm
{
 • Step 1:
    Let T_set represents the set of the path templates in the DSL file
    Let IF_set represents the set of the paths in ForClause
    Let IW_set represents the set of the paths in WhereClause?
    Let I_set = (IF_set ∪ IW_set)
    Let R_set represents the set of paths referred in ExprSingle. Note that
            if the ExprSingle is a FLWOR expression, we do not add the paths
            referred in the FLWOR expression to R_set
    BoundVariable_set = The bound variables in ForClause
    TargetXML_set = The file names of target XML documents in doc function
    ForClause_String = The string of ForClause in F
    WhereClause_String = The string of WhereClause? in F
    ReturnClause_String = The string of "return" + ExprSingle in F
    N = Null string

 • Step 2:
    P_set = XPath_Transformation (IF_set,T_Set,S);
    Scope_Array = Decryption_Scope (IF_Set,IW_set,R_set,T_set);
    1.Find all bound variables and store them into BoundVariable_set;
    2.Rename all variables in BoundVariable_set by adding "_1" and
      store them to BoundVariable_set_1;
    3.Transform the XQuery expressions.
      i = 1 to sizeof (BoundVariable_set);
      3A.1 Reconstruct For, where, Let, and return clauses:
        ForClause :
          for BoundVariable_set_1(i) in doc(TargetXml_set(i) + P_set(i);
        LetClause :
          let BoundVariable_set(i) =
              decryption(BoundVariable_set_1(i),Scope_Array(i);
      3B.1 Reconstruct For, Where, Let, and return clauses:
        ForClause :
          ForClause_String.replace(BoundVariable_set(i),
                                   BoundVariable_set_1(i));
        WhereClause :
          WhereClause_String.replace(BoundVariable_set(i),
                                     BoundVariable_set_1(i));
        LetClause :
          let BoundVariable_set(i)=
              decryption(BoundVariable_set_1(i),Scope_Array(i);
      3A.2 Reconstruct return clause:
        ReturnClause :
           return if + "(count(+BoundVariable_set(i))>0)"
      3A.3 Reconstruct the XQuery expressions:
        N = ForClause + LetClause + ReturnClasue +
             " and WhereClause_string then ReturnCluase_string else ()";
    if Intersection(I_set,T_set)=∅[3] and Intersection(R_set,T_set)=∅
    {
     No change for XQuery expression;
     Return F;
    }
    if [Intersection(I_set,T_set)≠∅ and Intersection(R_set,T_set)=∅] or
       [Intersection(I_set,T_set)≠∅ and Intersection(R_set,T_set)≠∅]
       {perform 3A.1}
    if [Intersection(I_set,T_set)=∅ and Intersection(R_set,T_set)≠∅]
       {perform 3B.1}
    Perform 3A.2
    Perform 3A.3
    Return N;
}
End_of_Algorithm
```

Fig. 9. Transformation algorithm.

```
<transactions>
  {
  for $b in doc("example.xml")/transactions/transaction
  where $b/price=1688
  return
      $b/cardinfo
  }
</transactions>
```
(A) An input FLWOR expression

```
<transactions>
  {
  for $b_1 in doc("example.xml")/transactions/transaction
  where $b_1/price=1688
  let $b = decryption($b_1, "child:EncryptedData[2]")
  return
     if (count($b) >0
     then
       {
         $b/cardinfo
       }
     else ()
  }
</transactions>
```
(B) An output FLWOR expression

Fig. 10. An XQuery to extract "`cardinfo`" from an encrypted XML file.

the encrypted elements that the original XQuery statement wants to query. Finally, we change `ExprSingle` to "`if (count($b) >0 then {$b/cardinfo} else ()`". The output FLWOR expression is listed in Fig. 10B.

Fig. 12A shows our second XQuery program, whose "`ForClause`", "`WhereClause?`", and `ExprSingle` expressions contain XPaths that point to encrypted elements. The program performs a search that returns the "`cardno`" element from the document `example.xml` (see Fig. 4), where the value of "`/transactions/transaction/cardinfo/cardno`" is "`1234-5678-8765-4321`". In this example, we have *T_set* = {"`/transactions/transaction/payer`," "`/transactions/transaction/cardinfo`"}, *I_set* = {"`/transactions/transaction/cardinfo`," "`/transactions/transaction/cardinfo/cardno`"}, and *R_set* = {"`/transactions/transaction/cardinfo/cardno`"}. The intersections of *I_set* and *T_set* and of *R_set* and *T_set* are not the empty set. According to the algorithm listed in Fig. 9, "`ForClause`" is changed to "`for $b_1 in doc("example.xml")/transactions/transaction/EncryptedData[2]`". A "`LetClause`" statement ("`let $b = decryption($b_1,"all")`") is added after the "`ForClause`" statement. "`LetClause`" invokes a decryption function to decrypt the `$b_1` variable which represents the elements pointed at by the XPath `/transactions/transaction/EncryptedData[2]`. Finally, `ExprSingle` is modified by adding "`if (count($b) >0 and $b/cardno = "1234-5678-8765-4321" then $b/cardno else ()`". The output FLWOR expression is listed in Fig. 12B.

Fig. 13A is the third example, which is a more complicated XQuery program. The `ExprSingle` statement contains an FLWOR expression. The "`WhereClause?`" statement in the outer FLWOR expression contains encrypted elements. The FLWOR expressions `ExprSingle` and "`ForClause`" also contain encrypted elements. The transformation process occurs from outside to inside. We first transform the outer FLWOR expression: we have *T_set*={"`/transactions/transaction/payer`," "`/transactions/transaction/cardinfo`"} and *I_set*={"`/transactions/transaction`," "`/transactions/transaction/payer`"}. The inner FLWOR expression "`for $a in $b/cardinfo return $a`" will not be changed when transforming the outer FLWOR expression: thus we have *R_set*={"`/transactions/transaction/price`"}. After invoking the intersection function, the intersection of "*I_set*" and "*T_set*" is not the empty set whereas that of *R_set* and *T_set* is the empty set. The "`ForClause`" statement is changed to "`for`

```
<?xml version="1.0" ?>
<dsl:security_document
   xmlns:dsl="http://www.xml-dsl.com/2002/dsl" version="1.0">
   :
   :
  <dsl:template match="/transactions/transaction/payer">
    <dsl:value-of-encrypted-node scope="element" pattern="pattern1"/>
  </dsl:template>
  <dsl:template match="/transactions/transaction/cardinfo">
    <dsl:value-of-encrypted-node scope="element" pattern="pattern2"/>
  </dsl:template>
</dsl:security_document >
```

Fig. 11. A DSL document.

```
<transactions>
   {
   for $b in doc("example.xml")/transactions/transaction/cardinfo
   where $b/cardno = "1234-5678-8765-4321"
   return
     $b/cardno
   }
</transactions>
```

(A) An input FLWOR expression

```
<transactions>
   {
   for $b_1 in doc("example.xml")/transactions/transaction/EncryptedData[2]
   let $b = decryption($b_1, "all")
   return
     if (count($b) >0 and $b/cardno = "1234-5678-8765-4321"
     then  $b/cardno
     else ()
   }
</transactions>
```

(B) An output FLWOR expression

Fig. 12. An XQuery to extract "`cardinfo`" from an encrypted XML file.

```
<transactions>
 {
   for $b in doc("example.xml")/transactions/transaction
   where $b/payer="Jessie Chang"
   return
      <transaction>
        {
          $b/price
          for $a in $b/cardinfo
          return $a
        }
      </transaction>
 }
</transactions>
```

(A) An input FLWOR expression

```
<transactions>
 {
   for $b_1 in doc("example.xml")/transactins/transaction
   let $b =  decryption($b_1, "child:EncryptedData[1]")
   return
      if (count($b)>0 and $b/payer="Jessie Chang")
      then
      {
        <transaction>
          {
            $b/price
            for $a_1 in $b/EncryptedData[2]
            $a = decryption($a_1,"all")
            if count($a)>0
            then
              return $a
            else()
          }
        </transaction>
      }
      else()
 }
</transactions>
```

(B) An output FLWOR expression

Fig. 13. An XQuery to extract "`cardinfo`" from an encrypted XML file.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="transaction">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="payer"/>
                <xs:element ref="price"/>
                <xs:element ref="cardinfo"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="transactions">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="transaction"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    :
    :
</xs:schema>
```

Fig. 14. An XML Schema.

`$b_1 in doc("example.xml")/transactions/transaction`". A "LetClause" statement ("`let $b=decryption($b_1, "child:EncryptedData[1]")`") is added after "ForClause", which invokes a decryption function to decrypt the `$b_1` variable. Finally, we transform the ExprSingle into the following statements: "

```
if (count($b) >0 and $b/payer = "Jessie Chang"
then
{
  <transaction>
    {
      $b/price
      for $a in $b/cardinfo return $a
    }
  </transaction>
}
else ()
```

".

After transforming the outer FLWOR expression, we should proceed to transform the inner FLWOR expression "`for $a in $b/cardinfo return $a`" to "`for $a_1 in $b/EncryptedData[2] $a=decryption($b_1,"all") if count($a)>0 then return $a else()`" according to the algorithm listed in Fig. 9. The output XQuery program is listed in Fig. 13B.

3.2. *XML Schema for efficient querying of encrypted XML documents*

XML Schema is the successor of DTD (Document Type Definition) that expresses shared vocabularies and provides a guide for characterizing the structure, content, and semantics of an XML document. Furthermore, XML Schema offers (1) identification of parent–child relationships, which improves the performance in solving XML queries for applications that require detection of these and other ancestor–descendant relationships; and (2) XML query validation, by exploiting the XML query language syntax to translate relative paths into absolute paths. In the proposed processing model the translator must investigate the DSL document in order to determine which elements were encrypted. For higher efficiency, XML Schema can be used to further speed up the decryption.

Fig. 14 shows an example of the partial XML Schema document. An XML document that refers to it is with start tag named "`transactions`". "`transactions`" contains only one child element, "`transaction`", which contains three elements. Note that the child elements of "`transaction`", "`payer`", "`price`", and "`cardinfo`", are surrounded by the `<sequence>` indicator. This means that the child elements must appear in the same order as they are declared.

In the following, we demonstrate that the XML Schema can be used to optimize the query. Fig. 15 is an encrypted version of the XML document shown in Fig. 4. Note that all child nodes of the "`transaction`" element are encrypted as a whole. If the user wants to obtain the value of the "`cardinfo`" element, s/he must write a "ForClause" statement such as "`$b in doc("example.xml")/transactions/transaction/EncryptedData`" in an XQuery program. However, there are three elements with tags named "`EncryptedData`". These elements will be decrypted to check their tag names to identify which is the "`cardinfo`" element. We can use XML Schema to avoid the redundant decryption. The translator looks it up by calling `Schema.getIndex` method to determine that the "`cardinfo`" element is the third child element of the "`transaction`" element. Thus, the "ForClause" statement can be changed to "`doc("example.xml")/transactions/transaction/EncryptedData[3]`", where the "`[3]`" means that only the third "`EncryptedData`" element needs to be decrypted.

4. Implementation and experimental results

We employ Saxon as the XQuery processor for executing XQuery programs. Saxon directly compiles XQuery programs into an operator tree and performs advanced transformations and optimization on the operator tree. According to the processing model shown in Fig. 8, we implement a translator that enables XQuery programs written by users to query data from encrypted XML documents according to the algorithm listed in Fig. 9. We also implement extension objects to perform the decryption processes.

We have conducted experiments to evaluate the performance of querying data from encrypted XML documents. All of the experiments were performed on a PC with a 2.4-GHz Pentium 4 processor, 1024 MB of RAM, the MS Windows 2000 operating system, and Java Development Kit 1.4 (Sun Microsystems, 2011). The original XML document had 101 elements: a tree with one root node and its 100 child element nodes, in which each child node was associated with a text node which in turn comprised either 100 or 500 bytes. Table 1 lists the times required to decrypt the whole encrypted XML document and then to query target elements. The processing time increases dramatically with the number of

```xml
<?xml version='1.0'?>
<transactions>
  <transaction>
    <EncryptedData Type='http://www.w3.org/2001/04/xmlenc#Element'
     xmlns='http://www.w3.org/2001/04/xmlenc#'>
      <CipherData>
        <CipherValue>
          mrs79DfdL+ODXzur3DZXBDJx2EwRgz+MRP3Nv9T20J2LltPYthkSAG0zVoCt+
          GZhSdcf4T9xLp78tOxRN/PgmGo2hLSO/3OtqTNukDooxPmA7sADaWiZOe6rbr
          NdFY5QgjBAZ8TlnQ3SSBiSM11rygoDei4LTJEROcN6Lq5lL/c=
        <CipherValue>
      <CipherData>
    </EncryptedData>
    <EncryptedData Type='http://www.w3.org/2001/04/xmlenc#Element'
     xmlns='http://www.w3.org/2001/04/xmlenc#'>
      <CipherData>
        <CipherValue>
          CyT4UQrOQ1vijcGM8nbKsB1ckUTpBoNH1USfvHTiwhZjN/2+bAyEoqzU07IbY
          XTCKzslnymXivI7waPYZ76V97W2/JqYxRpvkBcml4MSulhbekSW+S//jRSjxP
          uk0FW1POaj7gF9lyWEN+F0VpNvqMLceZAVWB7TKTVRx8LGU5l0w=
        <CipherValue>
      <CipherData>
    </EncryptedData>
    <EncryptedData Type='http://www.w3.org/2001/04/xmlenc#Element'
     xmlns='http://www.w3.org/2001/04/xmlenc#'>
      <CipherData>
        <CipherValue>
          h3IkkoyhsUL0uuC7MtSyw/xMfWlcKb144rH5EAQQ8vrjrs3B1RwmIDF9lYBCh
          Hkfghk3eW4Jb6fQrnemykms7ZIAy7dHpxL2lC7sJ0rX1UlDjzNoRHKVZo80IZ
          zQ9yP/+mBlbr6C/mD5vE9aa2FEEAlFvdGxPeW62fKCD3ZM15kotIRwyf5O+Ja
          1UJgLN2Juu5AQ3qkpScJBeocSeF207rveeCYPyd+Nh/GrDFzjCndBOB1YV7RX
          XyUvaDu2PZ55OtwNufUQggpvxpDZUZ7fSOkjzHrDN88ZwULKIf6aLBt1M=
        <CipherValue>
      <CipherData>
    </EncryptedData>
  </transaction>
</transactions>
```


Fig. 15. An encrypted XML document.

encrypted elements because all encrypted elements need to be decrypted first. For comparison, Table 2 lists the times required to query encrypted documents using the XQuery statements generated by the algorithm listed in Fig. 9. The algorithm ensures that only target elements are decrypted regardless of the number of encrypted elements. It is obvious that eliminating redundant decryption dramatically enhances the performance of the query process: increasing the number of encrypted elements in the target element has little effect on the time required to perform the query, which demonstrates the effectiveness of the processing model proposed in the paper.

We have conducted another experiment for querying large XML documents. We use XMark (Schmidt et al., 2002) benchmark data set and then generate encrypted XML documents. The experiment is on a PC with 2.66 GHz CPU and 2.99 GB main memory, and the environment is on MS Windows XP operating system with Java Development Kit 1.6. In all XML documents, three child elements "`name`", "`emailaddress`", and "`phone`" under the person element are encrypted. In Table 3, first column represents the size of XML document, second column represents the elapsed time to query target element "`name`" by decrypting all encrypted elements and the

Table 1
The time required to obtain encrypted data by decrypting the whole XML document.

Total elements in XML file	Number of queried elements which are encrypted	Number of elements that are decrypted	Number of encrypted elements	Average time (in seconds)	
				100 bytes[a]	500 bytes[a]
101	10	10	10	1.8984	3.7687
101	10	20	20	3.1155	6.7626
101	10	30	30	4.3640	9.8033
101	10	40	40	5.2296	12.7827
101	10	50	50	6.5156	15.7282
101	10	60	60	7.3671	18.6812
101	10	70	70	8.6720	21.4690
101	10	80	80	9.9843	24.8675
101	10	90	90	11.2171	27.3998
101	10	100	100	12.1735	29.9295

[a] Number of bytes to be encrypted in an element.

Table 2
The time required to query encrypted documents using the XQuery statements generated by our algorithm (1).

Total elements in XML file	Number of queried elements which are encrypted	Number of elements that should be decrypted	Number of encrypted elements	Average time (in seconds)	
				100 bytes[a]	500 bytes[a]
101	10	10	10	1.8937	3.7672
101	10	10	20	1.8968	3.7735
101	10	10	30	1.8921	3.7781
101	10	10	40	1.8984	3.7702
101	10	10	50	1.8077	3.7626
101	10	10	60	1.9157	3.7657
101	10	10	70	1.8469	3.7656
101	10	10	80	1.8531	3.7765
101	10	10	90	1.1987	3.7891
101	10	10	100	1.8938	3.7828

[a] Number of bytes to be encrypted in an element.

Table 3
The time required to query encrypted documents using the XQuery statements generated by our algorithm (2).

XML size	Decrypt all elements	Decrypt elements that should be decrypted
1M	0.8565	0.3875
2M	2.6312	1.1329
3M	5.4250	2.2296
4M	8.7249	3.7970
5M	13.5203	5.7812
6M	19.6329	8.4577
7M	26.4096	11.5860
8M	33.8857	14.7423
9M	41.9031	18.0564
10M	53.7561	21.3938

third column represents the elapsed time to decrypt target element "`name`" only.

5. Conclusion

This paper has focused on the development of a processing model for efficiently querying encrypted XML documents using XQuery. This model requires certain documents for efficient querying, including a DSL that specifies how to encrypt the XML documents and the XML Schema of the original XML documents. This model allows for the efficient querying of encrypted XML documents, in terms of the computation required for decryption during the query process. The experimental results presented here demonstrate that XQuery programs that are transformed according to the DSL and XML Schema exhibit good performance.

Appendix.

Procedure XPath_Transformation

```
Procedure XPath_Transformation(IF_set,T_set,S)
Input:
  IF_set = A set of paths
  T_set = The set of path templates in the DSL file
  S = An XML Schema
Output:
  P_set = A set of paths
Begin
{
  For i = 1 to (the number of paths in IF_set)
  {
    if (IF_set(i) ⊆ T_set) {
        Pt0 = A string in IF_set(i) from right to left until character is  "/"
        Pt1  =  Delete Pt0 in IF_set(i) from right
        index = 0
        If S is available {
           index = Schema.getIndex (IF_set(i), S)
        }
        if index >=1 {
            P = Pt1 + "EncryptedData[index.toString()"}
        else{
            P = Pt1 + "EncryptedData"
        }
     }
     else {P=IF_set(i)}
     Write P to P_set
  }
}
End
```

Procedure Decryption_Scope

```
Procedure Decryption_Scope(IF_set,IW_set,R_set,T_set,S)
Input:
  IF_set = The set of the path in ForClause
  IW_set = The set of the path in WhereClause?
  R_set = The set of paths referred in ExprSingle. Note that if the ExprSingle
          is a FLWOR expression, we do not add the paths referred in the FLWOR
          expression to R_set
  T_set = The set of path templates in the DSL file
  S = An XML Schema
Output:
  Scope_Array = String Array
Begin
{
  for i = 1 to (the number of paths in IF_set)
  {
    scope = null string
    if  (IF_set(i) ⊆ T_set) {
        scope = "all"
        Write scope to scope_Array
        Continue for loop
    }
    if  (IW_set ⊆ T_set) and ((IW_set ∩ IF_set(i) ≠∅){
        If S is available {
            index = Schema.getIndex(IW_set, S)
        }
        if (index >=1){
            scope = scope + "child:EncryptedData[index.toString()]"
        }
        else{
            scope = scope + "child:EncryptedData"
        }
    }
    if  (R_set ⊆ T_set) and ((R_set ∩ IF_set(i) ≠∅) {
        If S is available {
            index = Schema.getIndex(R_set, S)
        }
        if (scope <> null){
             scope = scope + ";"
        }
        if (index >=1){
            scope = "child:EncryptedData[index.toString()]"
        }
        else{
            scope = "child:EncryptedData"
        }
    }
    Write scope to Scope_Array
  }
}
End
```

References

Bartel, M., Boyer, J., Fox, B., LaMacchia, B., Simon, Ed, 2008. XML-Signature Syntax and Processing W3C Recommendation 10 June 2008, http://www.w3.org/TR/xmldsig-core/.

Boag, S., Chamberlin, D., Fernández, M.F., Florescu, D., Robie, J., Siméon, J., 2007. XQuery 1.0: An XML Query Language W3C Recommendation 23 January 2007, http://www.w3.org/TR/xquery/.

Chang, T.-K., Hwang, G.-H., 2003. Towards attribute encryption and a generalized encryption model for XML. In: International Conference on Internet Computing 2003 , 23–26 June, Las Vegas, Nevada, USA, pp. 455–461.

Clark, J., DeRose, S., 1999. XML Path Language (XPath) Version 1.0. W3C Recommendation, 16 November 1999, http://www.w3.org/TR/1999/REC-xpath-19991116.xml.

DSL Editor, http://www.xml-dsl.com/DSL_editor_detail.htm.

Fallside, D.C., Walmsley, P., 2004. XML Schema Part 0: Primer, W3C Recommendation, 28 October 2004. http://www.w3.org/TR/xmlschema-0/.

Fischer, C.N., LeBlanc Jr., R.J., 1991. Crafting A Compiler with C. The Benjamin/Cummings Publishing Company, Inc.

Galax. Available from: http://www.galaxquery.org.

Hwang, G.-H., Chang, T.-K., 2001. Document security language (DSL) and an efficient automatic securing tool for XML documents. In: International Conference on Internet Computing 2001 , 24–28 June, Las Vegas, Nevada, USA, pp. 393–399.

Hwang, G.-H., Chang, T.-K., 2004. An operational model and language support for securing XML documents. Computers & Security 23 (September (6)), 498–529.

Imamura, T., Dillaway, B., Simon, Ed., 2002. XML Encryption Syntax and Processing. W3C Recommendation 10 December 2002, http://www.w3.org/TR/xmlenc-core/.

International Organization for Standardization, 1999. Information Technology-Database Language-SQL-Part 1: Framework (SQL/Framework), ISO/IEC 9075-1: 1999 and Information Technology-Database Language-SQL-Part 2: Foundation (SQL/Foundation), ISO/IEC 9075-2, http://www.iso.org.

The Kawa language framework. Available from: http://www.gnu.org/software/kawa/.

Lee, G.-G., Whang, K.-Y., 2006. Secure query processing against encrypted XML data using Query-Aware decryption. Information Sciences, 1928–1947.

Maruyama, H., Imamura, T., 2000. Element-wise XML Encryption. http://www.alphaworks.ibm.com/tech/xmlsecuritysuite.

Qexo. The GNU Kawa implementation of XQuery. Available from: http://www.gnu.org/software/qexo/.

Saxon: The XSLT and XQuery Processor. http://saxon.sourceforge.net/.

Schmidt, A., Waas, F., Kersten, M., Carey, M.J., Manolescu, I., Busse, R., 2002. XMark: a benchmark for XML data management. In: International Conference on Very Large Data Bases (VLDB), pp. 974–985, http://www.xml-benchmark.org/.

Schneier, B., 1995. Applied Cryptography: Protocols, Algorithms, and Source Code in C, 2nd ed. John Wiley & Sons, New York.

Schrefl, M., Grun, K., Dorn, J., 2005. SemCrypt – Ensuring Privacy of Electronic Documents Through Semantic-Based Encrypted Query Processing. In: 21st International Conference on Data Engineering Workshops.

Sedna: Native XML Database System. Available from: http://modis.ispras.ru/sedna/index.html.

Sun Microsystems. The Source for Java(TM) Technology, http://java.sun.com.

Wang, H., Lakshmanan, L.V.S., 2006. Efficient secure query evaluation over encrypted XML databases. In: 32nd International Conference on Very Large Data Bases.

XML Encryption WG, 2001. http://www.w3.org/Encryption/2001/Overview.html.

Yang, Y., Ng, W., Lau, H.L., Cheng, J., 2006. An Efficient Approach to Support Querying Secure Outsourced XML Information, CAiSE 2006, LNCS 4001, pp. 157–171.

Zorba: The XQuery Processor. http://www.zorba-xquery.com/.

Tao-Ku Chang is an assistant professor in the Department of Computer and Information Science at National Dong Hwa University. He received the Ph.D. degree while in the Graduate Institute of Information and Computer Education at National Taiwan Normal University, Taipei, Taiwan, in 1998. His research interests include SOA, XML related issues, Internet security, and object-oriented technologies.

Gwan-Hwan Hwang is a professor in the Department of Computer Science and Information Engineering at National Taiwan Normal University, Taiwan. He received the B.S. and M.S. degrees while in the Department of Computer Science and Information Engineering at National Chiao Tung University, in 1991 and 1993, respectively, and the Ph.D. degree while in the Department of Computer Science at National Tsing Hua University, Hsinchu, Taiwan, in 1998. His research interests include Internet security, concurrent software testing, groupware, service-oriented architecture, and cloud computing.