A Processing Model for the Optimal Querying of Encrypted XML Documents in XQuery

Tao-Ku Chang
Graduate Institute of Information and Computer Education, National Taiwan Normal University
Taipei, Taiwan
tkchang@ice.ntnu.edu.tw

Gwan-Hwan Hwang
Department of Computer Science and Information Engineering, National Taiwan Normal University Taipei, Taiwan
ghhwang@csie.ntnu.edu.tw

Abstract

XQuery is a powerful and convenient language that is designed for querying the data in XML documents. In this paper, we address how to optimally query encrypted XML documents using XQuery, with the key point being how to eliminate redundant decryption so as to accelerate the querying. We propose a processing model that can automatically and appropriately translate the XQuery statements for encrypted XML documents. Furthermore, we show that XML schema is significantly associated with queries over XML documents. The implementation and experimental results demonstrate the practicality of the proposed model.

Keyword: XML, XQuery, DSL, Security, Database.

1 Introduction

The XQuery language (Scott et al., 2005) proposed by W3C was designed to be broadly applicable across all types of XML data sources. Its mission is to provide flexible query facilities to extract data from real and virtual documents on the Web. XQuery uses an XML data model that can represent XML documents, sequences, or atomic elements (such as integers or strings). The concept of XQuery is depicted in Figure 1. *Q* represents an XQuery program that 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 engine queries and formats data from an XML database that stores XML documents according to *Q*, with the resultant XML document being *R*.

XML is becoming a widespread data-encoding format for Web applications and services, which makes it important to secure XML documents in various ways. 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 W3C's XML encryption working group (http://www.w3.org/Encryption/2001/Overview.html) 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. Figure 2 illustrates the concept of element-wise encryption. Only one element (“`Number`”) of the original document is encrypted. This enables XML files to protect themselves because the sensitive data in XML are encrypted by particular keys.

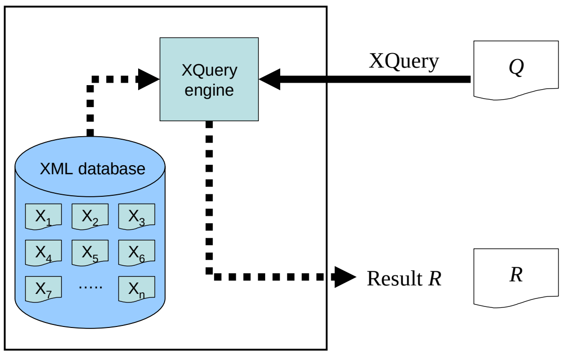
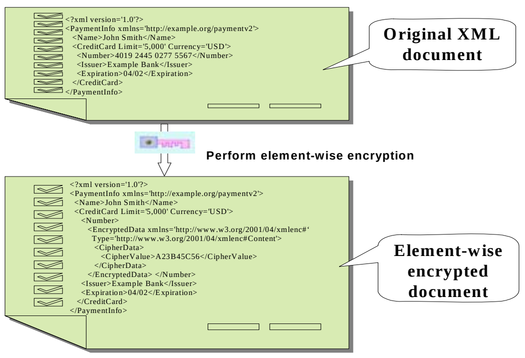


Figure 1: The data flow for querying XML documents

This paper addresses how to query data from these encrypted XML documents in XQuery. The intuitive, trivial method is to first decrypt the encrypted XML documents and then use an XQuery program to obtain the desired documents (see Figure 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 Figure 4. The “`payer`” and “`cardinfo`” elements are encrypted as a whole; that is, their encryption scope is set to “`element`”. In the encrypted XML document shown in Figure 5, the “`CipherData`” element contains the encrypted data of the “`payer`” and “`cardinfo`” elements, and is wrapped by the “`EncryptedData`” element. We see that the tag names of the “`payer`” and “`cardinfo`” elements disappear. Figure 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. Figure 6 lists an XQuery program that is used to obtain the value of the “`cardinfo`” element from Figure 4. It is obvious that we cannot use this program to query the encrypted document shown in Figure 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, one of the decryptions is redundant.



```
<?xml version='1.0'?>
<PaymentInfo xmlns='http://example.org/paymentv2'>
  <Name>John Smith</Name>
  <CreditCard Limit='5,000' Currency='USD'>
    <Number>4019 2445 0277 5567</Number>
    <Issuer>Example Bank</Issuer>
    <Expiration>04/02</Expiration>
  </CreditCard>
</PaymentInfo>
```

```
<?xml version='1.0'?>
<PaymentInfo xmlns='http://example.org/paymentv2'>
  <Name>John Smith</Name>
  <CreditCard Limit='5,000' Currency='USD'>
    <Number>
      <EncryptedData xmlns='http://www.w3.org/2001/04/xmlenc#'
        Type='http://www.w3.org/2001/04/xmlenc#Content'>
        <CipherData>
          <CipherValue>A23B45C56</CipherValue>
        </CipherData>
      </EncryptedData> </Number>
    <Issuer>Example Bank</Issuer>
    <Expiration>04/02</Expiration>
  </CreditCard>
</PaymentInfo>
```

Original XML document
Perform element-wise encryption
Element-wise encrypted document

Figure 2: Example of element-wise encryption

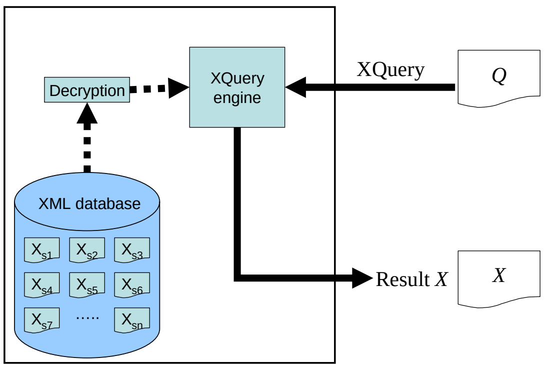


Figure 3: A trivial way to query encrypted XML documents

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 Figure 4, Figure 5, and Figure 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. As noted above, we use XML Schema (Fallside and Walmsley, 2004) that provides a means for defining the structure, content and semantics of XML documents to support it. It is usually used to validate XML documents but plays an important role in the XML queries in this research. We will illustrate it in Section 3. 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.

```
<?xml version='1.0'?>
<transactions>
  <transaction>
     <payer id = ”M123456789”>tony yao</payer>
     <price current="TWD">1350</price>
     <cardinfo>
        <cardtype>g</cardtype>
        <orgination>visa</orgination>
        <owner>tony yao</owner>
        <creditline>200000</creditline>
        <expiredate>12/01/2007</expiredate>
     </cardinfo>
  </transaction>
</transactions>
```

Figure 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>
                  mrs79DfdL+ODXzur3DZXBDJx2EwRgz+MRP3Nv9T20J2L
                  ltPYthkSAG0zVoCt+GZhSdcf4T9xLp78tOxRN/PgmGo2
                  hLSO/3OtqTNukDooxPmA7sADaWiZOe6rbrNdFY5QgjBA
                  Z8TlnQ3SSBiSM11rygoDei4LTJEROcN6Lq5lL/c=
          <CipherValue>
        <CipherData>
     </EncryptedData>
     <price current="TWD">1350</price>
     <EncryptedData
      Type='http://www.w3.org/2001/04/xmlenc#Element'
      xmlns='http://www.w3.org/2001/04/xmlenc#'>
        <CipherData>
          <CipherValue>
                  h3IkkoyhsUL0uuC7MtSyw/xMfWlcKb144rH5EAQQ8vrj
                  rs3B1RwmIDF9lYBChHkfghk3eW4Jb6fQrnemykms7ZIA
                  y7dHpxL2lC7sJ0rX1UlDjzNoRHKVZo80IZzQ9yP/+mBl
                  br6C/mD5vE9aa2FEEAlFvdGxPeW62fKCD3ZM15kotIRw
                  yf5O+Ja1UJgLN2Juu5AQ3qkpScJBeocSeF207rveeCYP
                  yd+Nh/GrDFzjCndBOB1YV7RXXyUvaDu2PZ55OTwNufUQ
                  ggpvxpDZUZ7fSOkjzHrDN88ZwULKIf6aLBt1M=
          <CipherValue>
        <CipherData>
     </EncryptedData>
  </transaction>
</transactions>
```

Figure 5: An encrypted XML document

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

Figure 6: An XQuery to extract “`cardinfo`” from an XML file

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). 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, we previously proposed a security language that allows a programmer to specify the security detail of XML documents: the document security language (DSL) (Hwang and Chang, 2001, 2003, 2004, 2005). The DSL can be used to define how to perform encryption and decryption, and the embedding and verification of signatures. It offers a security mechanism that integrates *element-wise encryption* and *temporal-based element-wise digital signatures*. Also, 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.

Figure 7 illustrates the relationship between XML, DSL, and the DSL securing tool. Figure 7A shows the process of encrypting and embedding digital signatures. The details of the encryption process and the digital signature itself are stored in a DSL document in D_{P}, D_{T}, and D_{Sig}: D_{P} is the security pattern definition that specifies the combination of security algorithms and encryption and decryption keys, D_{T} is the transformation description definition that specifies the actual data transformation of element-wise encryption, and D_{Sig} specifies how to embed digital signatures in the resulting XML document. The target XML document that is ready to be encrypted and signed is X. The DSL securing tool reads, parses, and analyzes D_{P}, D_{T}, D_{Sig}, and X, and then generates X_{s} and $D_{\mathrm{P'}}$. X_{s} is still an XML document, but some of its elements contain ciphertexts that are translated by the DSL securing tool according to the encryption details recorded in D_{P} and D_{T}. In addition to the encrypted elements, X_{s} also contains signatures that are embedded by the DSL securing tool. Each signature signs a portion of the data in X. It should be noted that D_{P} and $D_{\mathrm{P'}}$ may contain different information: D_{P} holds information describing how to encrypt X, whereas $D_{\mathrm{P'}}$ should include details of how to decrypt X_{s}. In addition, we have developed a DSL editor with a graphical user-friendly interface to make it easier for users to generate DSL documents (Hwang and Chang, 2005).

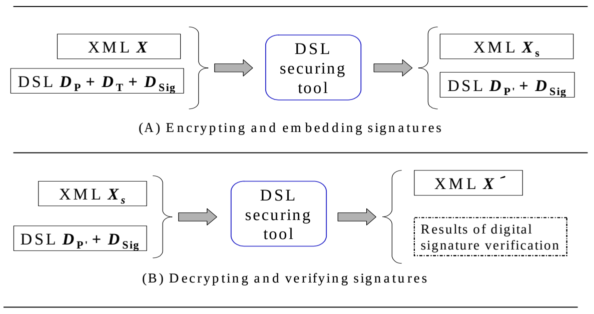

XML X
DSL D_P + D_T + D_Sig
DSL securing tool
XML X_s
DSL D_P' + D_Sig
(A) Encrypting and embedding signatures
XML X_s
DSL D_P' + D_Sig
DSL securing tool
XML X´
Results of digital signature verification
(B) Decrypting and verifying signatures

Figure 7: The operational model for securing XML documents

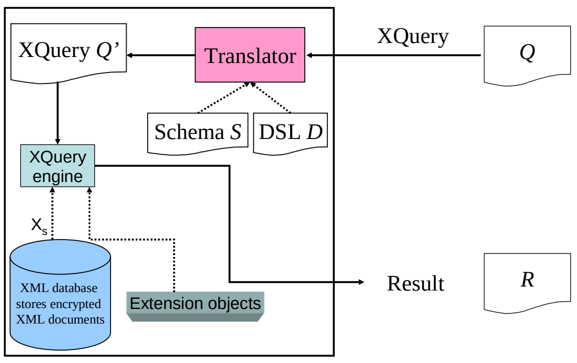


Figure 8: The processing model for querying encrypted XML documents

Figure 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 engine, 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 elements 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 may not be available; however, D is generally sufficient to generate an efficient XQuery Q'. 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 Statements for Querying Encrypted XML Documents

Now we present our design of an algorithm that is used to transform the XQuery statements; that is, the design of the translator shown in Figure 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/IEC 9075-2, 2003). The formal grammar for a FLWOR expression in XQuery is defined in (Boag et al., 2005) 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.

In this paper, we focus on FLWOR expressions to implement the transformation algorithm, which is listed in Figure 9. In the following we use four examples to demonstrate this algorithm.

```
Algorithm: Transform a FLWOR expression for querying
encrypted XML documents
Input:
  Let F is a FLWOR expression of the form:
      FLWORExpr ::= (ForClause | LetClause)
      WhereClause? OrderByClause? return ExprSingle
  Let D is a DSL file
  Let S is 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:
    if Intersection(I_set,T_set)=∅[2] and
       Intersection(R_set,T_set)=∅
    {
     N = F
    }

    if Intersection(I_set,T_set)≠∅ and
       Intersection(R_set,T_set)=∅
    {
      P_set = XPath_Transformation (IF_set,T_Set,S);
      Scope_Array =
      Decryption_Scope (IF_Set,IW_set,R_set,T_set);

      For i = 1 to (the number of bound variables in
         BoundVariable_set)
      {
         BoundVariable_set_1(i) =
         BoundVariable_set(i) +"_1";
      }
      N = "for "
      For i = 1 to (the number of bound variables in
         BoundVariable_set)
      {
         N = BoundVariable_set_1(i) +" in
             doc("+TargetXml_set(i)+")"+P_set(i)+"\n";
      }
      For i = 1 to (the number of bound variables in
         BoundVariable_set)
      {
         N = N +"let " + BoundVariable_set(i)+
         "=decryption("+ BoundVariable_set_1(i)+",\""+
                         Scope_Array(i)+"\")"+"\n";
      }
      N = N + "return" + "\n";
      N = N + "if"
      For i = 1 to (the number of bound variables in
         BoundVariable_set)
      {
        N = N +" "+"(count("+BoundVariable_set(i)+")>0)"
        if BoundVariable_set(i) ≠ null
        {
           N = N + " and"
        }
      }
      N = N + " and "+WhereClause_string+"\n"+"then "+
           ReturnCluase_string+"\n"+"else ()"+"\n";
     }

    if Intersection(I_set,T_set)=∅ and
       Intersection(R_set,T_set)≠∅
    {
      Scope_Array =
      Decryption_Scope(IF_set,IW_set,R_set,T_set);
      For i = 1 to (the number of bound variables in
         BoundVariable_set)
      {
         BoundVariable_set_1(i) =
         BoundVariable_set(i) +"_1";
      }
      For i = 1 to (the number of bound variables in
         BoundVariable_set)
      {
        New_forClause =
         ForClause_String.replace(BoundVariable_set(i),
                                  BoundVariable_set_1(i))
      }
      For i = 1 to (the number of bound variables in
         BoundVariable_set)
      {
         New_whereClause =
         WhereClause_String.replace(BoundVariable_set(i),
               BoundVariable_set_1(i))
      }
      N = N + New_forClause +"\n"
      N = N + New_whereClause +"\n"
      For i = 1 to (the number of bound variables in
         BoundVariable_set)
      {
         N = N +"let " + BoundVariable_set(i) +
         "=decryption("+ BoundVariable_set_1(i)+",\""+
                         Scope_Array(i)+"\")"+"\n";
      }
      N = N +"retrun"+"\n";

      N = N + "if"
      For i = 1 to (the number of bound variables in
        BoundVariable_set)
      {
        N = N +" "+"(count("+BoundVariable_set(i)+")>0)"
        if BoundVariable_set(i) ≠ null
        {
           N = N + " and"
        }
      }
      N = N + " and "+WhereClause_string+"\n"+"then "+
          ReturnCluase_string+"\n"+"else ()"+"\n";
    }

    if Intersection(I_set,T_set)≠∅ and
```

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

[2] The symbol ∅ indicates the empty set.

```
       Intersection(R_set,T_set)≠∅
    {
     P_set = XPath_Transformation(IF_set, T_set, S)
     Scope_Array =
     Decryption_Scope(IF_set, IW_set, R_set, T_set);

     N ="for "
     For i = 1 to (the number of bound variables in
        BoundVariable_set)
     {
        N = BoundVariable_set_1(i) +" in
             doc("+TargetXml_set(i)+")"+P_set(i)+"\n";
     }
     For i = 1 to (the number of bound variables in
        BoundVariable_set)
        N = N +"let " + BoundVariable_set(i)+
        "=decryption("+ BoundVariable_set_1(i)+",\""+
                       scope_Array(i)+"\")"+"\n";
     }

     N = N + "return"+ "\n";
     N = N + "if"
     For i = 1 to (the number of bound variables in
       BoundVariable_set)
     {
       N = N +" "+"(count("+BoundVariable_set(i)+")>0)"
       if BoundVariable_set(i) ≠ null
       {
          N = N + " and"
       }
     }
     N = N + " and "+WhereClause_string+"\n"+"then "+
     ReturnCluase_string+"\n"+"else ()"+"\n";
    }
}
End_of_Algorithm

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 = check-schema (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(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 = check-schema (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 = check-schema (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
```

Figure 9: Transformation algorithm

The first example demonstrates an XQuery program that queries some of the encrypted elements from the target XML document.

Figure **10**A lists a FLWOR expression that performs a simple search that returns the "`cardinfo`" element from the document `example.xml` (see Figure 4) where the value of "`/transactions/transaction/price`" is "`1350`". The XML document shown in Figure 5 is that encrypted according to the DSL document shown in Figure 11. The input includes a FLWOR expression, a DSL document, and an XML Schema. `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 "<u>`ForClause`</u>" and "<u>`WhereClause`</u>", and "*R_set*" represents the set of paths referred to in <u>`ExprSingle`</u>. Note that if <u>`ExprSingle`</u> 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 <u>`ExprSingle`</u> 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 "<u>`ForClause`</u>" and "<u>`WhereClause`</u>" contain encrypted elements. Similarly, the intersection of "*I_set*" and "*R_set*" is not empty when the return elements contain encrypted elements. In this example there are two path templates in the DSL document (see Figure 11), and we have *T_set* = {"`/transactions/transaction/payer`," "`/transactions/transaction/cardinfo`"}, <u>`ForClause`</u> = "`for $b in doc("example.xml")/transactions/transaction`", <u>`WhereClause?`</u> = "`where $b/price=1350`", *I_set* = {"`/transactions/transaction`," "`/transactions/transaction/cardinfo/price`"}, <u>`ExprSingle`</u> = "`$b/cardinfo`", and *R_set* = {"`/transactions/transaction/cardinfo`"}. 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.

According to the algorithm listed in Figure 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=1350`", 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 `$b_1` variable since it contains the encrypted elements that the original XQuery statement wants to query. Finally, we change `ExprSingle` to "`if (count($b) >0 then {$b/cardinfo} else ()`". The output FLWOR expression is listed in Figure 10B.

```
<transactions>
   {
   for $b in doc("example.xml")/transactions/transaction
   where $b/price=1350
   return
       $b/cardinfo
   }
</transactions>
```

(A) An input FLWOR expression

```
<transactions>
   {
   for $b_1 in doc("example.xml")/transactions/transaction
   where $b_1/price=1350
   let $b = decryption($b_1, "child:EncryptedData[2]")
   return
      if (count($b) >0
      then
        {
          $b/cardinfo
        }
      else ()
   }
</transactions>
```

(B) An output FLWOR expression

Figure 10: An XQuery to extract "`cardinfo`" from an encrypted XML file

```
<?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 >
```

Figure 11: A DSL document

Figure 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 Figure 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 Figure 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 Figure 12B.

```
<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

Figure 12: An XQuery to extract "`cardinfo`" from an encrypted XML file

Figure 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 $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 = "tony yao"
 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 Figure 9. The output XQuery program is listed in Figure 13B.

```
<transactions>
 {
   for $b in doc("example.xml")/transactions/transaction
   where $b/payer="tony yao"
   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="tony yao")
       then
       {
          <transaction>
             {
                $b/price
                for $a_1 in $b/EncryptedData[2]
                $a = decryption($b_1,"all")
                if count($a)>0
                then
                   return $a
                else()
             }
          </transaction>
       }
       else()
 }
</transactions>
```

(B) An output FLWOR expression

Figure 13: An XQuery to extract “`cardinfo`” from an encrypted XML file

It is essential to use the DSL in the proposed processing model because the translator must investigate the DSL document to determine which elements were encrypted. Although it is not compulsory to use XML Schema, it can be used to further reduce the times required for decryption. XML Schema is a DTD successor that expresses shared vocabularies and provides a guide for characterizing the structure, content, and semantics of an XML document. Furthermore, XML Schema offers (1) XML query validation, by exploiting the XML query language syntax to translate relative paths into absolute paths; and (2) 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.

In the following, we demonstrate that the XML Schema can be used to optimize the query. Figure 14 is an encrypted version of the XML document shown in Figure 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. Figure 15 lists the XML Schema of the XML document shown in Figure 4. The translator looks it up 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.

```
<?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+MRP3Nv9T20J2LltPY
                  thkSAG0zVoCt+GZhSdcf4T9xLp78tOxRN/PgmGo2hLSO/3Ot
                  qTNukDooxPmA7sADaWiZOe6rbrNdFY5QgjBAZ8TlnQ3SSBiS
                  M11rygoDei4LTJEROcN6Lq5lL/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+
                  bAyEoqzU07IbYXTCKzslnymXivI7waPYZ76V97W2/JqYxRpv
                  kBcml4MSulhbekSW+S//jRSjxPuk0FW1POaj7gF9lyWEN+F0
                  VpNvqMLceZAVWB7TKTVRx8LGU5l0w=
          <CipherValue>
        <CipherData>
     </EncryptedData>
     <EncryptedData
      Type='http://www.w3.org/2001/04/xmlenc#Element'
      xmlns='http://www.w3.org/2001/04/xmlenc#'>
        <CipherData>
          <CipherValue>
                  h3IkkoyhsUL0uuC7MtSyw/xMfWlcKb144rH5EAQQ8vrjrs3B
                  1RwmIDF9lYBChHkfghk3eW4Jb6fQrnemykms7ZIAy7dHpxL2
                  lC7sJ0rX1UlDjzNoRHKVZo80IZzQ9yP/+mBlbr6C/mD5vE9a
                  a2FEEAlFvdGxPeW62fKCD3ZM15kotIRwyf5O+Ja1UJgLN2Ju
                  u5AQ3qkpScJBeocSeF207rveeCYPyd+Nh/GrDFzjCndBOB1Y
                  V7RXXyUvaDu2PZ55OTwNufUQggpvxpDZUZ7fSOkjzHrDN88Z
                  wULKIf6aLBt1M=
          <CipherValue>
        <CipherData>
     </EncryptedData>
  </transaction>
</transactions>
```

Figure 14: An encrypted XML document

```
<?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>
```

Figure 15: An XML Schema

4 Implementation and Experimental Results

Many implementations of the XQuery engine exist. For example, Galax (http://www.galaxquery.org) is a lightweight and extensible implementation of XQuery 1.0. Since it closely tracks the definition of XQuery 1.0 as specified by the W3C, it also implements XPath 2.0, which is a subset of XQuery 1.0. Qexo (http://www.gnu.org/software/qexo/) is a partial implementation of the XQuery language that exhibits a good performance because a query is compiled down to the Java byte codes. Saxon (http://www.saxonica.com/) is a complete and conformable implementation of XSLT 2.0, XQuery 1.0, and XPath 2.0. We employ Saxon as the XQuery engine for executing XQuery programs. According to the processing model shown in Figure 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 Figure 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). 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 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 Figure 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.

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*	500 bytes*
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

*Number of bytes to be encrypted in an element

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*	500 bytes*
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

*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

5 Conclusion

In this paper we have presented a processing model for efficiently querying encrypted XML documents using XQuery. This model requires some documents for optimal querying, including a DSL that specifies how to encrypted the XML documents and the XML Schema of the original XML documents. We can use this model to optimally query the encrypted XML documents, in terms of the computation required for decryption during the query process. Moreover, the experimental results presented here demonstrate that XQuery programs that are transformed according DSL and XML Schema exhibit good performance.

References

Boag Scott, Chamberlin Don, Fernández Mary F., Florescu Daniela, Robie Jonathan and Siméon Jérôme (2005), "*XQuery 1.0: An XML Query Language W3C Working Draft.*" http://www.w3.org/TR/xquery/

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

Schneier Bruce (1995), "*Applied Cryptography: Protocols, Algorithms, and Source Code in C*," 2nd Edition, published by John Wiley & Sons.

Maruyama Hiroshi and Imamura Takeshi (2000), "*Element-wise XML Encryption.*" http://www.alphaworks.ibm.com/tech/xmlsecuritysuite

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

Imamura Takeshi, Dillaway Blair, and Simon Ed (2002), "*XML Encryption Syntax and Processing. W3C Recommendation 10 December 2002.*" http://www.w3.org/TR/xmlenc-core/

Hwang Gwan-Hwan and Chang Tao-Ku (2001), "*Document Security Language (DSL) and an Efficient Automatic Securing Tool for XML Documents*," International Conference on Internet Computing 2001, 24-28 June, Las Vegas, Nevada, USA, pp: 393-399

Hwang Gwan-Hwan and Chang Tao-Ku (2004). "*An operational model and language support for securing XML documents*," Computers & Security, Volume 23, Issue 6, September 2004, pp: 498-529.

Chang Tao-Ku and Hwang Gwan-Hwan (2003), "*Towards Attribute Encryption and a Generalized Encryption Model for XML*," International Conference on Internet Computing 2003, 23-26 June, Las Vegas, Nevada, USA, pp: 455-461.

Hwang Gwan-Hwan and Chang Tao-Ku (2005) "*DSL Editior,*" http://www.xml-dsl.com/DSL_editor_detail.htm

Fallside David C. and Walmsley Priscilla (2004), "*XML Schema Part 0: Primer*," W3C Recommendation, 28 October 2004. http://www.w3.org/TR/xmlschema-0/

International Organization for Standardization, Information Technology- Database Language-SQL-Part 2: Framework (SQL/Framework), ISO/IEC 9075-2: 2003 and Information Technology- Database Language-SQL-Part 2: Foundation (SQL/Foundation), ISO/IEC 9075-2: 2003, http://www.iso.org.

Fischer Charles N. and LeBlanc Richard J Jr. (1991). "Crafting A Compiler with C". The Benjamin/Cummings Publishing Company, Inc.

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

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

Saxon. Available from: http://www.saxonica.com/

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