Securing XML Data in Third-Party Distribution Systems*

Barbara Carminati
University of Insubria at Como
Como, Italy
barbara.carminati@uninsubria.it

Elena Ferrari
University of Insubria at Como
Como, Italy
elena.ferrari@uninsubria.it

Elisa Bertino
CERIAS, Purdue University
Lafayette, USA
bertino@cerias.purdue.edu

ABSTRACT

Web-based third-party architectures for data publishing are today receiving growing attention, due to their scalability and the ability to efficiently manage large numbers of users and great amounts of data. A third-party architecture relies on a distinction between the Owner and the Publisher of information. The Owner is the producer of information, whereas Publisher provides data management services and query processing functions for (a portion of) the Owner's information. In such architecture, there are important security concerns especially if we do not want to make any assumption on the trustworthy of the Publishers. Although approaches have been proposed [4, 5] providing partial solutions to this problem, no comprehensive framework has been so far developed able to support all the most important security properties in the presence of an untrusted Publisher. In this paper, we develop an XML-based solution to such problem, which makes use of non-conventional digital signature techniques and queries over encrypted data.

Categories and Subject Descriptors: [D.4.6] **Security and Protection** Access

General Terms: Security.

Keywords: XML, Third-party architecture, Data outsourcing.

1. INTRODUCTION

Third-party information dissemination represents today an interesting paradigm for data-intensive web-based applications in a large variety of contexts, from grid computing to web services or P2P systems. Relevant applications include large-scale federated Digital Libraries, e-commerce catalogs, e-learning, collaborative applications, content distribution networks. The main idea of third-party architectures is that the information *Owner* outsources all or some portions of its data to one or more *Publishers* that provide specialized data management services and query processing functions. Such an approach is scalable, results in highly efficient query execution, and reduces the management costs of the Owner. Clearly, such architecture has challenging security requirements. Requiring the Publishers to be trusted wrt security properties is not always an appropriate solution in that large web-based systems cannot be easily verified to be trusted and can be easily penetrated. Thus, our goal is to ensure security properties even in the presence of untrusted Publishers.

The main security properties that should be assured are: *confidentiality*, *integrity*, and *authenticity*. Confidentiality has two aspects. The first, that we call *confidentiality wrt the user*, refers to protecting data against unauthorized accesses by users. The second, that we call *confidentiality wrt the Publisher*, deals with protecting data from accesses by Publishers. Integrity requires that data contents are not altered during their transmission.[1] Finally, authenticity assures that a user receiving some data can verify that the data have been generated by the Owner and that the Publisher has not modified their contents. To provide strong guarantees about data contents to users, we need to complement the three basic security properties with an additional property that we refer to as *completeness*. By completeness we mean that the user receiving a portion of data is able to verify whether he/she has received all the information is allowed to see according to the specified access control policies. Although some proposals exist assuring the satisfaction of some of these properties [4, 5], no comprehensive framework exists able to enforce all such properties.

In this paper, we develop such a comprehensive solution, by focusing our attention on data expressed in XML [10]. Our solution relies on the use of encryption and non-conventional signature techniques. Encryption is used to support confidentiality. The basic idea is that the Publisher does not operate on clear text data, but on an encrypted version. Encryption is driven by the specified access control policies: all data portions to which the same policies apply are encrypted by the Owner with the same key. Then, each user is provided by the Owner with all and only the keys corresponding to the portions of data he/she is allowed to access. Clearly, both users and Publishers must be provided with additional information to make them able to submit and answer queries on encrypted data. Authenticity/integrity requirements cannot be ensured by traditional digital signature techniques. The reason is that since a user may be returned only selected portions of a document, depending on his/her queries and the specified policies, it is not enough that the Owner signs each document it sends to the Publisher. Thus, we propose an alternative solution, based on Merkle hash trees [8], to generate document signatures.

Finally, completeness is verified through the so-called *query template*, which consists of the encrypted structure of the original document. We show that, by executing the queries submitted to a Pub-

*The work reported in this paper has been partially supported by the Italian MIUR under the project 'Web-based management and representation of spatial and geographical data'.

CIKM'05, October 31–November 5, 2005, Bremen, Germany.

[1] Here we do not consider integrity wrt the specified access control policies, since third-party architectures are mainly conceived for read accesses.

lisher on the query template, a user is able to verify the completeness of the query answer without accessing information he/she is not allowed to see.

Our work has been inspired by the work by Hacigumus et al. [5, 6], which developed a method for querying encrypted data stored in relational databases. From such work we borrow the method for querying encrypted data, which is based on partitioning the domains of the relation attributes. The same method has been exploited also in the context of XML data in [7]. However, such approaches only consider confidentiality wrt the Publisher, and they do not address the requirement of confidentiality wrt the users, nor authenticity/integrity and completeness. By contrast, in this paper we extend these approaches to data encrypted with different keys, thus providing confidentiality wrt the users. In addition, we consider also all the other security properties. Other related work is by Miklau and Suciu [9], which proposed a method for a controlled sharing of XML data, dealing only with confidentiality. As in our approach, confidentiality is ensured by the use of cryptographic techniques. However, the main difference with our proposal is that they do not rely on a Publisher for managing data, rather data are simply published on the web in an encrypted form and each user can access the authorized portions, using the keys he/she receives from the data Owner. However, we strongly believe that relying on a data Publisher has many benefits, in terms of efficiency and optimization of resource usage. Indeed, simply publishing the data over the web would make them the target of a huge number of attacks, with many users trying to perform queries over them, and thus consuming a huge amount of computational resources. By contrast, the Publisher can be equipped with sophisticated anti-intrusion tools and techniques avoiding queries floods. Additionally, not relying on a Publisher requires each user be equipped with a query engine able to process queries over encrypted contents.

Merkle hash trees are a well-known mechanism used in several computer areas for certified query processing. For instance, they have been exploited for authenticating XML documents by Devanbu et al. in [4]. However their approach has many differences wrt our proposal. First, it addresses only authenticity, whereas we address all security properties. Moreover, our approach to authenticity verification does not have limitations on the structure of XML documents to which it can be applied, whereas the approach by Devanbu et al. does not handle attributes and it assumes that data contents can be only present in leaf nodes. Another important difference is that we can certify the authenticity for each possible kind of XPath queries, whereas the approach by Devanbu et al. only handles queries returning whole sub-trees. The work reported in this paper builds on a previous paper by us [1], where we develop the technique for authenticity verification that we use in the current paper. However, the current paper significantly extends our previous work with techniques for confidentiality and completeness enforcement. Moreover, we define also an architecture and related data structures supporting security properties verification.

The remainder of this paper is organized as follows. Next section presents an overview of the proposed framework. Sections 3 and 4 describe authenticity and confidentiality enforcement; Section 5 introduces the XML encoding we propose to represent the needed security information, whereas Section 6 focuses on query processing. Section 7 deals with completeness. Section 8 formally states some properties of the proposed framework, whereas Section 9 concludes the paper.

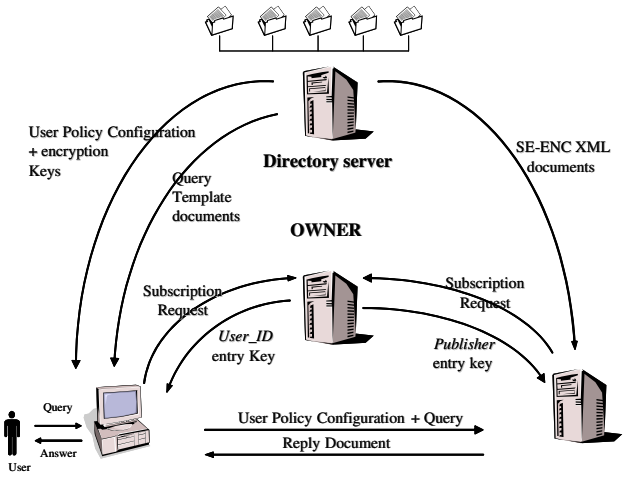

Figure 1: Overall architecture

2. SYSTEM OVERVIEW

In the architecture we propose (see Figure 1) users submit queries to Publishers through a *client*, that the user can download from the Owner site, and which makes the user able to verify the security properties on the received answers. The novelty of our proposal is that we do not make any assumption on the trustworthy of Publishers. In the following we give a general overview of the techniques we have devised.

Because the proposed framework requires additional information to be transmitted by the Owner to both Publishers and users, we store all such information into a directory server (which can also be shared among different Owners belonging to the same domain or to federated ones), to limit the overhead that these operations require. In particular, as depicted in Figure 1, the directory server contains three kinds of entries: the *Publishers* entry, which is shared by all Publishers and contains all encrypted documents that they are entitled to manage, plus additional information they need for the correct functioning of the system; the *Users* entry, shared by all the subscribed users, storing common information; and a distinct *User_ID* entry for each single user, containing needed information to verify the security properties. Thus, after a mandatory registration phase, each user/Publisher receives by the Owner the keys for accessing the corresponding entries in the directory.

We now briefly discuss how the security properties are enforced. Confidentiality wrt final users are expressed by the Owner by means of a set of access control policies, regulating the operations that can be performed on its data [2]. To enforce confidentiality wrt the Publisher, the Owner encrypts its data before delivering them to the Publisher. Following an approach we will explain in Section 4, the Publisher is able to answer user queries, without the need of decrypting the data. Thus, it returns the requesting user an encrypted answer. If the Publisher operates according to the Owner policies, this answer contains all and only the portions of the requested data the user can access. Otherwise, it may contain a superset of the data the requesting user is allowed to see. The key aspect is thus that a user must be able to decrypt all and only the portions of the returned answer he/she is authorized to see. This is obtained by selectively encrypting the documents in the Owner source: the Owner encrypts them in such a way that all the portions that are protected by the same policies, hereafter called *policy configuration*, are encrypted with the same key.[2] These keys are then stored in the *User_ID* entries by the Owner, on the basis of the policies the

[2]Here and in the following we refer to symmetric encryption.

corresponding users satisfy. Additionally, the entry stores the *user policy configuration*, a certificate signed by the Owner maintaining information on the access control policies the user satisfies, which are determined according to the credentials the user submits during a mandatory subscription phase.

To make a Publisher able to correctly answer queries over encrypted data, the Owner provides the Publisher with information on which users can access which portions of the managed documents, according to the access control policies it has specified. Additionally, the Owner supplies the Publisher some information that makes it able to query encrypted data. The basic idea is that the Owner divides the domain of each document node (i.e., attribute and element) into distinguished partitions, to which a unique id is assigned. Then, the Owner provides the Publisher together with the encrypted nodes also the ids of the partitions corresponding to their values. The Publisher is thus able to perform queries directly on the encrypted documents, by exploiting the partitioning ids (see Section 4 for more details). Similarly, users find in the common *Users* entry information on the partitioning techniques adopted by the Owner.

Authenticity and integrity are assured by the use of the *Merkle signature*, a signature generated by the Owner using a bottom-up computation on the whole document, based on Merkle hash trees [8]. Such signature is generated before encrypting a document and it is provided to the Publisher along with the corresponding document. The Publisher will then forward it to a user querying the document to which it refers to. The problem here is that, since the Publisher answer may not contain all the document portions over which the signature has been generated, a user may not be able to validate the signature. To avoid this shortcoming, the Owner gives the Publisher a set of additional hash values, one for each node, which represent the information needed to validate the signature if the corresponding node is not inserted into the Publisher answer. Thus, when a user queries a certain document, the Publisher sends him/her, besides the corresponding Merkle signature, also these additional hash values, referring to the document portions not contained in the query answer. This makes the user able to locally perform the computation of the Merkle signature and comparing it with the one generated by the Owner.

All the additional information needed by the Publisher for confidentiality and authenticity/integrity enforcement is encoded in XML and attached to the encrypted document, forming the so called *security enhanced encryption* (SE-ENC) of the original document. All the SE-ENC documents are stored by the Owner in the Publishers directory entry. Similarly, all information needed by a user to verify the security properties are encoded by the Publisher in XML and attached to the query answer, resulting in what we have called the *reply document*.

Finally, to make a user able to verify the completeness of a query result, the Owner generates a *query template*, containing the encrypted structure of the original document. The query template has the twofold goal of making a user able to verify the completeness of the received answers, as well as to make easier the task of query submission, in that by inspecting the query template a user can obtain information on the structure of the documents (or portions) he/she is allowed to access. The query template is encrypted by the Owner using the same strategy employed for XML documents. This means that a user can see only the portions of the query template on which he/she can perform the queries, according to the specified access control policies. The query template is digitally signed by the Owner, through a Merkle signature, to prevent alterations. All the query templates are stored by the Owner in the *Users* directory entry.

```
<Investments>
  <Investment  Partner='Partner1'>
    <Inv  Date=''  Amount=''  Type=''> ...</Inv>
    <Inv  Date=''  Amount=''  Type=''> ...</Inv>
  </Investment>
  <Investment  Partner='Partner2'>
    <Inv  Date=''  Amount=''  Type=''> ... </Inv>
  </Investment>
</Investments>
```

Figure 2: An example of XML document

3. AUTHENTICITY ENFORCEMENT

For authenticity enforcement we adopt an alternative way to compute the digest value of an XML document wrt traditional digital signature techniques [1]. The function we use to compute the digest value is the *Merkle hash function*. This function univocally associates an hash value (referred to as *Merkle hash value*) with a whole XML document through a recursive bottom-up computation on its structure. The basic idea is to associate a Merkle hash value with each node n of the XML document, denoted as $MhX(n)$: the Merkle hash value associated with an attribute is obtained by applying an hash function over the concatenation of the attribute value and the attribute name; the Merkle hash value associated with an element is the result of the same hash function computed over the concatenation of the element content, the element tag name, and the Merkle hash values of its children nodes, both attributes and elements. The digest of the XML document is thus the Merkle hash value of the root of the document. Once the digest has been computed, it is signed by the Owner, generating what we call the *Merkle Signature* of the document. The Merkle signature is inserted by the Owner into the corresponding SE-ENC document, by adding a `Sign` subelement to the document root, which contains the signature value. When a user submits a query to the Publisher, the Publisher returns him/her, besides the query result, also the Merkle signatures of the documents on which the query is performed. Moreover, to make the user able to validate the signature, the Publisher sends him/her a set of hash values, referring to the portions of the requested documents not returned in the query answer. This additional information is called *Merkle hash paths*.

EXAMPLE 3.1. *Let us consider the XML document presented in Figure 2, and suppose that the Owner states two access control policies allowing the manager of Partner1, (Partner2, respectively) to access all the subtree rooted at the* `Investment` *element related to Partner1 (Partner2, respectively). These policies allow the manager of Partner1 (Partner2, respectively) also to access the type of the investments done by Partner2 (Partner1, respectively), that is, only the* `Type` *attributes of the* `Inv` *elements. Suppose that a manager belonging to Partner2 requires all the investments of Partner1. Then, the Publisher returns the manager the* `Inv` *elements with only the* `Type` *attributes. We show now which are the additional hash values needed by the manager to validate the signature of the requested document. First of all, the manager needs to compute the Merkle hash value of the* `Inv` *element. Since the query result contains only the* `Type` *attribute of such element, the manager needs the Merkle hash values of* `Type`*'s siblings (i.e.,* `Amount` *and* `Date` *attributes) plus the hash value of the tagname and element content of* `Inv`*. Once the Merkle hash value of* `Inv` *is computed, the manager needs the Merkle hash values of all* `Inv`*'s siblings (i.e., the second* `Inv` *element), to compute the Merkle hash value of the* `Investment` *element. Finally, to compute the Merkle hash value of the root, he/she needs the Merkle hash value of* `Investment`*'s siblings (i.e., the* `Investment` *element referring to Partner1). Thus, the hash values needed by the manager are: MhX(*`Amount`*); MhX(*`Data`*);*

h(`Inv.content`[3]*); h(*`Inv.tagname`*); MhX(*`Inv`*); h(*`Investment.content`*); h(*`Investment.tagname`*); MhX(*`Investment`*); h(*`Investments.content`*); h(*`Investments.tagname`*); MhX(*`Investments`*), where h() is a collision-resistant hash function, used to compute the Merkle hash values.*

More formally, given two nodes v, w such that $v \in Path(w)$[4], the *Merkle hash path* between w and v is the set of hash values necessary to compute the Merkle hash value of v having the Merkle hash value of w. The Merkle hash path between w and v consists of all the Merkle hash values of w's siblings, together with the hash value of tagname and content of w's father node. Indeed, according to Merkle hash function definition, given w, these hash values make possible the computation of the Merkle hash value of w's father node. Thus, given this Merkle hash value to compute the Merkle hash value of v are necessary also the Merkle hash values of all the siblings of the nodes belonging to the path connecting v to w. Thus, for each node n belonging to the query result, the user must be supplied by the Publisher with the Merkle Hash path between n and the root element. Since the Publisher operates on encrypted data, it is not able to compute the Merkle hash values, and, as a consequence, to generate the appropriate Merkle hash paths to be returned to the user submitting the query. For this reason, the Owner gives Publisher some additional information, called *Authenticity information*, which makes the Publisher able to compute the Merkle hash values of all the document nodes, respecting, at the same time, confidentiality requirements. Such information are attached to the SE-ENC document using the strategy we will illustrate in Section 5.

4. CONFIDENTIALITY ENFORCEMENT

To ensure confidentiality, we propose a solution based on encryption techniques. The idea is that the Owner, before outsourcing a document to Publishers, encrypts it on the basis of the specified access control policies. All the portions of an XML document to which the same policy configuration applies are encrypted with the same secret key (we refer to the document encryption driven by the Owner policies as *well-formed encryption*). The appropriate keys are then stored in the Owner directory server, in such a way that each user obtains all and only the keys corresponding to the policies he/she satisfies. Moreover, to limit the number of keys that need to be permanently maintained we adopt an hierarchical key management schema defined in such a way that from the encryption key associated with an access control policy it is possible to derive all and only the encryption keys corresponding to policy configuration containing such a policy. In this way the number of keys that need to be managed is linear in the number of the specified access control policies.

Generation of the well-formed encryption ensures confidentiality both wrt the users and the Publishers. Each node of the resulting encrypted document is accessible only to authorized users, that is, those users who have been provided with the appropriate keys. Since the Publisher does not have keys, this solution prevents its accesses to the managed data, thus ensuring the confidentiality wrt Publisher. Additionally, the fact that a user submits queries to a Publisher in encrypted form ensures a certain degree of privacy to the user in that the Publisher does not know the details of the submitted queries.

[3]Given an element e, we use the notation $e.tagname$, $e.content$ to denote the tagname and the data content of e, respectively. Given an attribute a, the notation $a.val$ and $a.name$ are used to denote the value and the name of attribute a, respectively.

[4]Given a node w, *Path*(w) denotes the set of nodes connecting w to the root of the corresponding document.

To make the Publisher able to evaluate queries on encrypted documents, we adopt an approach similar to the one proposed in [5, 6] for relational databases. The underlying idea of this approach is the following: given a relation R, the Owner divides the domain of each attribute in R into distinguished partitions, to which it assigns a different id. Then, the Owner sends the Publisher the encrypted tuples, together with the ids of the partitions corresponding to each attribute value in R. According to this approach, the Publisher is able to perform queries directly on the encrypted tuples, by exploiting the partition ids. As an example, consider the relation $Employee(eid, ename, salary)$, and, for simplicity, consider only the $salary$ attribute. Suppose that the domain of $salary$ is in the interval [500k, 5000k], and that an equi-partition with 100k as range is applied on that domain. Thus, each encrypted tuple is complemented with the id of the partition corresponding to the value of the $salary$ attribute for that tuple. By using this id the Publisher is able to perform queries such as: "SELECT * FROM Employee WHERE salary =1000k", which is translated into the query: "SELECT * FROM Employee WHERE salary =XX", where XX is the id of the partition containing the value 1000k. It is interesting to note that this query returns an approximate result, in that it returns all the tuples of the *Employee* relation whose $salary$ attribute belongs to the range [1000K, 1100K). A further query processing has thus to be performed by the client to refine the answer returned by the Publisher.

We adapt such an idea to the XML context. This requires first of all to deal with partition generation. In general, the choice of the most appropriate partitioning technique mainly depends on the attribute domain. Thus, in defining the partitioning techniques for an XML document, we need to consider the data types that it may contain. For numeric data (such as integer, real, etc.), a strategy based on an equi-partitioning of the domain could be appropriate. However, an XML document mainly contain textual information (for instance, the data content of an element). For this reason, it is necessary to devise ad-hoc partitioning techniques for textual data, which are not so important in the relational context. The solution we propose for partitioning textual data requires a first phase during which the Owner preprocesses the textual data contained in an attribute/element and extracts from them a set of keywords.[5] Then, a partition id is associated with each keyword. More precisely, all possible keywords are organized into a dictionary. Therefore, partition ids are associated with groups of dictionary terms (for instance, assuming that the terms in the dictionary are in alphabetic order, we can generate a different id for each group of N terms). In the rest of the paper, we assume that there exists a function $PI()$ that given as input a value val returns the index of the partition to which val belongs to.

5. SECURITY-ENHANCED ENCRYPTION

All information for confidentiality and authenticity enforcement is encoded in XML and attached to the well-formed encryption, forming the SE-ENC document. Generation of SE-ENC documents consists of two main steps: generation of 1) the well-formed encryption, and 2) security information.

Generation of well-formed encryption is done by first marking the nodes of the input document with the policies that apply to them. Then, all the nodes to which the same marking applies are encrypted with the same key. In the literature, there exists different proposals for the encryption of an XML document (see for instance [7, 10]). However, we prefer to adopt a slightly different approach,

[5]Several techniques developed in the Information Retrieval field can be used to this purpose.

to preserve as much as possible the structure of the original XML document in the document encryption. Indeed, since users formulate queries according to the structure of the original document, this choice makes query processing easier. Thus, given an XML document d, the well-formed encryption of d is an XML document d^e, which preserves the elements/attributes relationships of the original document, but which has the names and contents of all the nodes encrypted. More precisely, the resulting document is formally defined as follows.

DEFINITION 5.1. *(*Well-formed encryption of an XML document*). Let $d = (V_d, \bar{v_d}, E_d, \phi_{E_d})$[6] be an XML document. Let $\mathcal{PC}_{\mathcal{PB}}(d)$ be the set of policy configurations which apply to d. Let $Key(pc)$ be the encryption key associated with policy configuration pc, and let $V_d(pc)$ be the set of nodes to which pc applies. The well-formed encryption of d is an XML document $d^e = (V_{d^e}, \bar{v}_{d^e}, E_{d^e}, \phi_{E_{d^e}})$, such that:*

- *d^e preserves the elements/attributes relationships of d;*
- *$\forall \overline{pc} \in \mathcal{PC}_{\mathcal{PB}}(d), \forall v \in V_d(\overline{pc}), \exists v' \in V_{d^e}$ such that:*
  v'.tagname= Enc(v.tagname, $Key(\overline{pc})$), v'.content= $Enc(v.content, Key(\overline{pc}))$, if $v \in V^e_{d^e}$;
  v'.name= Enc(v.name, $Key(\overline{pc})$), v'.val= Enc(v.val, $Key(\overline{pc})$), if $v \in V^a_{d^e}$;
  where $Enc(string, key)$ encrypts a $string$ with the input key.

Once the well-formed encryption has been generated, it undergoes a second phase, during which it is complemented with information for authenticity and confidentiality enforcement. All this information are wrapped into a unique element, called *Security Information* element. The SE-ENC document contains a different Security Information element for each element of the well-formed encryption. Such element is added as an additional child of the corresponding element and contains confidentiality and authenticity information of both the element itself and of all its attributes.

The authenticity information associated with each node n of the original document consists of the hash values needed to compute the Merkle Hash Path to be sent to users (cfr. Section 3). More precisely, these are the hash value of the name of n (i.e., the tagname or the attribute name, depending on whether n is an attribute or an element) and the hash value of the content of n (i.e., the data content or the attribute value, respectively). All these values are contained into a unique element, called `Auth-Info` element, child of the Security Information element corresponding to n.

Confidentiality information associated with a node consists of *policy information* and *query-processing information*. Policy information gives the Publisher information on which access control policies apply to each node of the original document, and is encoded into a string of hexadecimal values. With each node n in the well-formed encryption d^e we associate a binary string of length equal to the cardinality of the set of access control policies which applies to the corresponding clear-text document d, where, starting from the left side, the value of the i-th bit is: 1, if the i-th policy [7] applies to n; 0, otherwise. Then, we translate each 4-bits block of the resulting binary string into the corresponding hexadecimal representation. This information is then stored as an additional attribute of the Security Information element. Finally, to make policy configurations meaningful to Publishers it is necessary to insert an additional element into the SE-ENC document. This element, called `Policy`, contains the identifiers of the policies which apply to d. These identifiers help the Publisher to match a user policy configuration with the policy information in the SE-ENC document.

Query-processing information associated with a node consists of the partition ids corresponding to it. All partition ids are contained into a unique element, called `Query-Info` element, child of the Security Information element. We are now ready to formally introduce the Security Information element.

DEFINITION 5.2. *(*Security Information element*). Let $d^e = (V_{d^e}, \bar{v}_{d^e}, E_{d^e}, \phi_{E_{d^e}})$ be the well-formed encryption of an XML document d. Let $v' \in V^e_{d^e}$ be the encrypted element corresponding to $v \in V^e_d$. The Security Information element associated with v' is an XML element s such that:*

- *s.tagname=*`Sec-Info`*;*
- *s contains two subelements:* `Node-Info` *and* `Attributes`*, where:*
  - `Node-Info` *has two attributes:* `Name`*, which contains the encrypted name of the node;* `PC`*, which contains the policy information associated with v.* `Node-Info` *has two subelements:* `Auth-Info` *and* `Query-Info`*.* `Auth-Info` *has two subelements:* `H-Name`*, storing $h(v.tagname)$ ($v.name$, respectively); and* `H-Content`*, storing $h(v.tagname)$ ($v.value$, respectively).* `Query-Info` *has as many* `Id` *subelements as the number of partition ids associated with $v.content$.*
  - `Attributes` *has many* `Node-Info` *subelements as the number of attributes in v, with the same structure described above.*

When all the `Sec-Info` elements have been added to a well-formed encryption, the final SE-ENC document is obtained by adding the `Policy` element previously illustrated, and the `Sign` element described in Section 3. Figure 3 reports an example of the `Sec-Info` element associated with the `Investment` element of Partner2 (see Figure 2), computed by considering the access control policies presented in Example 3.1.[8]

```
<Sec-Info>
  <Node-Info  Name='Enc(Investment,Key(PC(Investiment)))'
  PC='b'>
    <Auth-Info>
      <H-Name> h(Investment.tagname) < /H-Name>
      <H-Content> h(Investment.content) < /H-Content>
    < / Auth-Info >
    < Query-Info > ....</ Query-Info >
  </ Node-Info >
  <Attributes >
  <Node-Info  Name='Enc(Partner,Key(PC(Partner)))' PC='b'>
    <Auth-Info  Name='Enc(Partner,Key(PC(Partner)))'>
      <H-Name> h(Partner.tagname) < /H-Name>
      <H-Content> h(Partner2.content) < /H-Content>
    </ Auth-Info >
    <Query-Info >
      <Id  Value='PI(Partner2)'/>
    </Query-Info >
  </ Node-Info >
</Sec-Info >
```

Figure 3: An example of `Sec-Info` element

6. QUERY PROCESSING

In this section, we explain how the user can formulate queries to the Publisher. We assume that users submit queries by means of XPath expressions. XPath allows one to traverse the graph structure of an XML document and to select specific portions on the document according to some properties, such as the type of the elements, or specified content-based conditions. In this paper, we consider conditions specified by means of equality or comparison

[6] We exploit a graph-based representation of an XML document. More precisely, we define an XML document as a tuple $d = (V_d, \bar{v_d}, E_d, \phi_{E_d})$, where: $V_d = V^e_d \cup V^a_d$ is a set of nodes representing elements (V^e_d), and attributes (V^a_d); $\bar{v}_d$ is a node representing the document element (called *document root*); E_d is the set of edges representing element-subelement, element-attribute relationships, or links between elements; ϕ_{E_d} is the edge labelling function.

[7] The order is given by the policy identifier values.

[8] Given a node n, we denote with $Key(PC(n))$ the encryption key associated with the policy configuration applied on n.

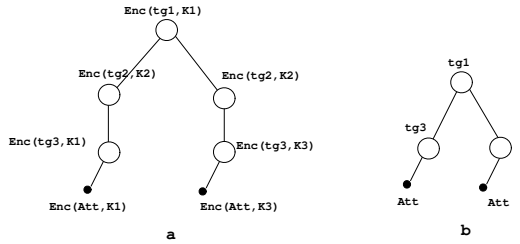

Figure 4: An example of view of a query template

operators on data content. Moreover, among the functions supported by XPath, we consider the *contains()* function, which allows the specification of conditions on textual data. In general, an XPath expression consists of a *location path*, that allows one to select a set of nodes from the target documents. A location path consists of one or more *location steps*, separated among each other by a slash. A location step consists of: an *axis*, specifying the tree relationships between the nodes selected by the location step and the current node (e.g., ancestor, ancestor-or-self, attribute, child, descendant, descendant-or-self); a *node test*, used to identify a node within an axis, by specifying a node type or the node name (e.g., text(), node()); and zero or more *predicates*, placed inside square brackets, used to further refine the set of nodes selected by the location step (e.g., [@Type='IT']). In the following, given a location step ls we use the dot notation to identify its components (i.e., $ls.axis$, $ls.nodetest$, $ls.p$).

To query an XML document through XPath, it is thus necessary to know the corresponding schema. For this reason, the user retrieves from the *Users* entry the *query template* of the interested document, which consists of the encrypted structure of the corresponding document. This operation is required only the first time the user inquiries a document. To make a user able to submit queries on encrypted documents, the query template contains further information. One of this information is the `Policy` element and `PC` attributes contained in the SE-ENC document, that allow the client to correctly encrypt the queries to be submitted to the Publisher.[9]

We explain now how the user can exploit the query template for formulating an XPath expression on encrypted documents. First of all, it is important to point out that a user can access only selected portions of the query templates, that is, only the nodes for which he/she has the appropriate decryption keys. Thus, as a first step, the client extracts the authorized view from the query template. To decrypt a node, the client has to know which key has to be used. This information can be derived from the `PC` attribute contained in the query template. The view of the query template is built by a function, called $View()$, which takes as input the policy configuration of a user u and the query template, and turns the set of decrypted nodes, into a well-formed XML document. This resulting view is, then, displayed to the user, making him/her able to formulate XPath queries on it. However, before the user XPath queries can be submitted to the Publisher they have to be properly transformed and encrypted. The following example clarifies the discussion.

EXAMPLE 6.1. *Consider the query template in Figure 4a, where, for simplicity, we do not report policy and query-processing information. Suppose that the view of the query template for a user u is the one presented in Figure 4b. Moreover, suppose that u is interested only in those nodes* `tg3` *whose attribute* `Att` *is equal to 'IT'. Thus, according to the view in Figure 4b, u formulates the following XPath query: /tg1/tg3[@Att='IT']. Obviously, this path cannot be directly evaluated on the corresponding SE-ENC document, because it is not encrypted and refers to a partial view.*

[9]We postpone the details of the query template generation in the next section, where we will explain the role played by this document in the completeness verification.

ALGORITHM 1. *The Client Query Generator*

INPUT:
1. An XPath expression exp given in input by a user u
2. The query template qt of the document d to which exp applies
3. $View_u(qt)$, that is, the view of the query template qt generated, according to the policies satisfied by u
OUTPUT:
The set of XPath expressions EXP to be submitted to the Publisher

1. Let ls, exp_{new}, and exp_s be initialized to be empty
2. EXP=Transform(ls,exp,exp_{new},exp_s)
*3.***Return** EXP
Function **Transform**(ls_{prec},exp,exp_{new},exp_s)
1. Let ls be the location step following ls_{prec} in exp
2. **If** *ls is empty* **then**:
a. Insert exp_{new} into exp_s
b. **Return** *exp_s*
3. Let $Nodes$ be the set of nodes of $View_u(qt)$ identified by $ls.nodetest$
4. **For** *each $n \in Nodes$:*
a. Let $path$ be the absolute path connecting the root of qt to $Enc(n, Key(PC(n)))$
b. Let $prec.node$ be the node specified in $ls_{prec}.nodetest$ belonging to $path$
c. Let Ls_{miss} be the missing location steps between $Enc(prec.node, Key(PC(pre.node)))$ and $Enc(n, Key(PC(n)))$ in $path$
d. Insert the Ls_{miss} location steps into exp_{new}
e. Let ls_{new} be an empty location step
f. $ls_{new}.axis$=$ls.axis$
g. $ls_{new}.node$=$Enc(n, Key(PC(n)))$
h. $ls_{new}.p$ = [/`Sec-Info`*//*`Node-Info`*[@Name=$Enc($ $ls.p.node, Key(PC(ls.p.node))$]/`Query-Info/id` *[@Value $ls.p.\oplus PI(ls.p.value)$]]*
i. Insert ls_{new} into exp_{new}
l. exp_s= $exp_s \bigcup$ Transform(ls,exp,exp_{new},exp_s)
EndFor

Figure 5: The Client Query Generator Algorithm

There are three main transformations to which each location step of a user XPath expression must undergo before being submitted to the Publisher. Since the user XPath expression has been generated on a partial view of the query template, it is first necessary its completion, by inserting all the missing location steps. For instance, considering again Example 6.1, between the first and the second location step of the user query it is necessary to insert the location step referring to element 'tg2'. Moreover, since the SE-ENC document is encrypted, it is necessary to encrypt the tagnames specified in the node test of the location steps with the proper keys, so that they can be evaluated by the Publisher. Finally, the third transformation is the transformation of the location step predicates, by computing the ids of the partitions, using the information obtained during the subscription phase, and by adapting the resulting predicates to the SE-ENC structure.

An Algorithm doing all the above mentioned operations is presented in Figure 5. For simplicity, the algorithm considers only XPath queries whose predicates contain a unique condition; however, it can be easily extended to consider more complex predicates. Due to the nature of well-formed encryption, where nodes with the same name could be encrypted with different keys, the encryption of a location step does not always return a unique value. For instance, the second location step of the XPath query in Example 6.1 (i.e., '/tg3[@Att='IT']') must be transformed in two dif-

ferent location steps: '/Enc(tg3,K_1)[...]' and '/Enc(tg3,K_3)[...]' (cfr. Figure 4). Let us see how Algorithm 1 works. It receives as input an XPath expression exp submitted by a user u, the query template qt on which the query is submitted, and the view of qt generated by the client according to the policies satisfied by u. To generate the set of XPath queries to be submitted to the Publisher, the algorithm exploits a recursive function, called $Transform()$, which recursively applies the same transformations to each location step of exp. The $Transform()$ function first verifies whether all the location steps in exp have been processed. If this the case, step $2.a$ returns the resulting set of XPath expressions. Otherwise, the $Transform()$ function computes the set of nodes, called $Nodes$, whose tagname is specified by the node test of the current location step. Then, in step 4 the algorithm iteratively considers each node in $Nodes$. For each of these nodes the $Transform()$ function generates a different location step by applying the abovementioned transformations. After the insertion of this new location step into the XPath expression (step $4.i$), the $Transform()$ function recursively calls itself, to consider the next location step of exp. Let us see how the algorithm applies the needed transformations to each location step. At first, the algorithm completes the input XPath expression by inserting the missing location steps (steps $4.a - d$). To verify whether the path between the considered location step and the previous one in the input XPath expression must be completed, the algorithm first computes the absolute path connecting the node identified by the current location step to the root of the query template (step $4.a$). Then, in step $4.c$ it determines the nodes that are missing in the input XPath expression wrt the absolute path. Note that, since the query template contains encrypted nodes, the nodes are first encrypted with the corresponding key. The second transformation is performed by step $4.g$, and encrypts the tagname of the current node. The encrypted name is inserted as node test in an empty location step. Finally, the last phase is the translation of the predicates in the location step. In general, a predicate p of a location step specifies a node ($p.node$) to which a comparison operator $\oplus$ is applied (e.g., $<$,$<=$,$>$,$>=$, $contains()$), matching it with the contained value ($p.value$). Obviously, the Publisher can not evaluate $\oplus$ directly on the $p.node$ of the SE-ENC document, since it contains only encrypted data. By contrast, the predicate has to be adapted to the SE-ENC document. This implies that the condition has to be evaluated directly on the `Value` attribute of the `Id` subelements, that is, the attribute containing the partition id. More precisely, the new XPath expression must be applied to the `Id` subelements of the `Query-Info` element contained into the `Node-Info` element corresponding to $p.node$ in the SE-ENC document. Thus, the predicate that replaces p in the new location step is the following: [/`Sec- Info`//`Node-Info`[@Name=$Enc(ls.p.node, Key(PC(ls.p.node))$]/`Query-Info` /`id`[@Value $\oplus PI(ls.p.value)$]], where $PI()$ is the function returning the partition id corresponding to the input value (step $4.h$).

EXAMPLE 6.2. *Let us consider the user XPath expression presented in Example 6.1: '/tg1/tg3[@Att='IT']'. The algorithm starts to consider the first location step, i.e., '/tg1'. According to Figure 4b, only the root element is referred by the node test of this location step. This implies that the for cycle is iterated only once. Moreover, the location step has no missing paths and no predicates. Thus, the only transformation performed in the cycle is the encryption, obtaining thus exp_{new}='Enc(tg1, K_1)'. By contrast, the node test of the second location step, i.e., 'tg3[@Att='IT']', indicates two different nodes, which are separately transformed by different iteration of the for cycle. Let us consider the first iteration, for the first 'tg3' node. At first the Transform() function computes the missing path, that is, Enc(tg2,K_2), which is added to exp_{new}. Then, it encrypts the node with the appropriate key, i.e., Enc(tg3,K_1). Finally, it transforms the '[@Att='IT']' predicate in '*[/`Sec-Info`//`Node-Info`[@Name=Enc(Att, K_1)]/`Query-Info`/`id`[@Value='PI(IT)']]'. The last step of the for cycle recursively calls the Transform function. However, since the new call of the Transform function does not find further location steps to be transformed, it adds the new expression (i.e., '*[/`Sec-Info`//`Node-Info`[@Name=Enc(Att, K_1)]/`Query-Info`/`id`[@Value='PI(IT)']]') into exp_{new} and it ends.*

The second iteration of the cycle is similar to the first, with the difference that the encryption key used is K_3 instead of K_1. When the Algorithm stops set $EXPs$ is equal to {Enc(tg1,K_1)/Enc(tg2,K_2)/Enc(tg3,K_1)[/`Sec-Info Node-Info`[@Name=Enc(Att,K_1)]/`Query-Info`/`id`[@Value='PI(IT)']]; Enc(tg1,K_1)/Enc(tg2,K_2)/Enc(tg3,K_3)[/`Sec-Info`//`Node-Info`[@Name= Enc(Att,K_3)]/`Query-Info`/`id`[@Value='PI(IT)']].

7. COMPLETENESS ENFORCEMENT

In this section we show how the client can verify the completeness of the query answer by using the query template. The query template of d is generated by the Owner, by applying a simple XSLT transformation [10] on the corresponding SE-ENC document. This transformation prunes from the SE-ENC document the encrypted data contents and authenticity information, which are not necessary for completeness verification. To prevent alterations of the query template, the Owner signs it with a Merkle signature, which is stored into a `Sign` element. Once the client receives a query template, it is able to verify the completeness of the queries submitted on XML documents conforming to the template. Completeness verification can be done for all XPath queries whose conditions are based on $=$, $<$, $<=$, $>$, $>=$ operators or the $contains()$ function.

The node-set returned by evaluating a query on the query template could be a superset of the nodes the user is entitled to see, according to the Owner access control policies. Thus, in order to verify the completeness, the client must also consider the access control policies specified on the document. For this reason, the query template contains also policy information (i.e., the `Policy` element and `PC` attributes).

EXAMPLE 7.1. *Consider the query template associated with the XML document in Figure 2 and the access control policies presented in Example 3.1. Suppose that a Partner1 manager submits a query asking for all the* `Inv` *elements associated with Partner1. Suppose, moreover, that an untrusted Publisher sends the manager only the first* `Inv` *element. The completeness verification process executed by the manager first verifies the authenticity and integrity of the query template. Then, it considers the query submitted to the Publisher, that is, the XPath expressions returned by Algorithm 1. More precisely, the user XPath expression '/Investments/Investment[Partner='Partner1']/*' is transformed into: Enc(Investments, $Key(PC(Investiments))$)/Enc(Investment,$Key(PC(Investment))$) [[/`Sec-Info`//`Node-Info`[@Name=Enc(Partner,$Key(PC(Partner))$]/`Query-Info`/`id`[@Value = $PI(Partner2)$]]]. The evaluation of this query on the query template returns two elements with tagname Enc(Inv, $Key(PC(Inv))$). Moreover, for each of them three attributes are returned, that is, 'Enc(Date, $Key(PC(Data))$)', 'Enc(Amount, $Key(PC(Amount))$)', and 'Enc(Type,$Key(PC(Type))$)'. The client must then prune from these nodes, those for which the user has no authorization. To do that, the client verifies the policy configuration of each of these nodes by checking the* `PC` *attribute stored into the corresponding* `Node-Info` *element. Thus, considering the value of the* `Policy` *element, the nodes for which the Partner1 manager has an authorization are those whose policy configuration has the 1-st bit set equal to 1.*[10] *Thus, all the possible values are 0001, 0011, to which characters 'a' and 'c' correspond. Thus, the nodes that the Partner1 manager is authorized to access are all the nodes returned by the client evaluation on the query template. Therefore, he/she verifies that in the answer received by the Publisher an element is omitted.*

[10] We suppose that the ids of access control policies applied on document in Figure 2 are 15 and 16, respectively.

8. FORMAL RESULTS

In this section, we state the correctness of the proposed solution, proofs can be found in [3]. In particular, we show how the proposed framework is able to enforce the considered security properties. Before presenting the formal results, we introduce the *reply document*, that is, the XML document generated by the Publisher and containing the query answer plus additional information needed for authenticity verification.

DEFINITION 8.1. *(*Reply document*) Let $g = (V_g, \bar{v}_g, E_g, \phi_{E_g})$ be the SE-ENC version of an XML document d, let u be a user, and q be a query on d submitted by u to a Publisher. Let $View(q, u) = (V_q, \bar{v_q}, E_q, \phi_{E_q})$ be the XML document answer to q, according to the policy configuration of u. The* reply document of query q with respect to u *is an XML document $r = (V_r, \bar{v}_r, E_r, \phi_{E_r})$ such that:*

- $V_r^e = V_q^e \bigcup V_{ATT}^e \bigcup Sign$, *where:*
  -V_{ATT}^e contains a node, called `AttributeElement`, *for each attribute $a \in V_q^a$. This node represents an element whose data content is the value of a. The name of a is stored into an additional attribute of* `AttributeElement`, *called* `AttrName`. *The node is a direct child of the node in V_r^e corresponding to the element in $View(u, q)$ to which a belongs to;*
  - $Sign$ is an element, direct child of $\bar{v}_r$ containing the content of `Sign` *in g;*
- *each node $e \in V_r^e$ contains an attribute, called* `MhPath`, *containing the Merkle hash path between e and its father.*

The following theorem states the correct enforcement of confidentiality requirements.

THEOREM 8.1. *Let O be an Owner, P be a publisher managing a portion PS of the Owner source. Let $\mathcal{PB}$ be the policy base of O and let u be a user subscribed to O. Let d be a document belonging to P, and let $V_d(u)$ be the portion of d that u is allowed to see according to the policies in $\mathcal{PB}$. Let q be a query on d submitted by u to P and let r be the corresponding reply document. Then,* 1) *P is not able to read information in PS.* 2) *there does not exist a node n in d such that $n \notin V_d(u)$ and u is able to access n by processing r.*

As far as authentication is concerned, the correctness is based on the fact that the Merkle hash paths sent by the Publisher are sufficient for the user to authenticate all the elements he/she is allowed to see in the reply document. We need thus to first state the notion of authenticable element.

DEFINITION 8.2. *(*Authenticable element*) Let $d = (V_d, \overline{v}_d, E_d, \phi_{E_d})$ be an XML document, let $g = (V_g, \bar{v}_g, E_g, \phi_{E_g})$ be the SE-ENC version of d, and $r = (V_r, \bar{v}_r, E_r, \phi_{E_r})$ be the reply document corresponding to a query submitted on d by a user u. Let V_T be the set of terminal nodes of r. For each $v \in V_r^e$, v is a* authenticable by s, *iff there exists $v_t \in V_T$, with $v \in Path(v_t)$, such that it is possible, through a recursive bottom-up computation, to compute the Merkle hash value of $\overline{v}_d$ using only the values in $\{w.$`MhPath`$|w \in Path(v_t)\}$.*

Note that authenticability is required only for the nodes of the reply document that represent elements. Indeed, attribute nodes in the reply document (i.e., `MhPath` attributes) are inserted only to store values needed to check the authenticity and completeness of the answer.

THEOREM 8.2. *Let P be a Publisher, let d be a document, and let d^e be the SE-ENC version of d managed by P. Let r be the reply document corresponding to a query submitted on d by a user u. Each element node belonging to r is authenticable by u.*

Finally, completeness enforcement is ensured by the following theorem.

THEOREM 8.3. *Let P be a Publisher, O be an Owner and $\mathcal{PB}$ its policy base. Let q be a query submitted by a user u to P on a document d. Let qt be the query template associated with d. Let r be the reply document returned by P to u, and let $V_d(u)$ be the portion of d that u is allowed to see according to the policies in $\mathcal{PB}$. By using the information in qt and r, u can verify that he/she receives all the portions of $V_d(u)$ answering query q.*

9. CONCLUSIONS

In this paper we have provided a comprehensive framework able to ensure security properties in the context of a third-party architecture. Our approach also includes a suite of strategies for minimizing the overhead due to updates to the policy base or the document source [3]. The strategies are based on incrementally maintaining the document encryption and the related data structures, upon each update operation, without rebuilding them from scratch each time an update occurs. The work reported in this paper can be extended along several directions. First, we would like to complement our framework with privacy enforcement. An implementation of the proposed system is currently underway. Up to now we have completed the modules for authenticity and completeness verification. We plan to develop also the modules for confidentiality enforcement to test the system performance and to assess the overhead due to update management.

10. REFERENCES

[1] E.Bertino, B.Carminati, E.Ferrari, B. Thuraisingham, A. Gupta. Selective and Authentic Third-Party Distribution of XML Documents. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 16(10):1263–1278, 2004.

[2] E. Bertino and E. Ferrari. Secure and Selective Dissemination of XML Documents. *ACM Transactions on Information and System Security (TISSEC)*, 5(3):290–331, 2002.

[3] B. Carminati, E. Ferrari, E. Bertino. Securing XML Data in Third-Party Distribution Systems. Technical Report, University of Insubria at Como. Available at http://scienze-como.uninsubria.it/carminati/SE-ENC.pdf

[4] P. Devanbu, M. Gertz, A. Kwong, C. Martel, G. Nuckolls, and S.G. Stubblebine. Flexible Authentication of XML documents. In *Proc. of the 8th ACM Conference on Computer and Communications Security*, ACM Press, 2001.

[5] H. Hacigumus, B. Iyer, C. Li, and S. Mehrotra. Executing SQL over Encrypted Data in the Database Service Provider Model. *In Proceedings of the SIGMOD Conference*, 2002.

[6] H. Hacigumus, B. Iyer, and S. Mehrotra, "Providing database as a service, " *In Proceedings of ICDE Conference*, 2002.

[7] R. Jammalamadaka and S. Mehrotra. Querying Encrypted XML Documents . UCI Technical report TR-DB-04-03, 2003.

[8] R.C. Merkle A Certified Digital Signature. In *Advances in Cryptology-Crypto '89*, 1989.

[9] G. Miklau and D. Suciu. Controlling Access to Published Data Using Cryptography, In Proc. of the *29th VLDB Conference*, Berlin, Germany, 2003.

[10] World Wide Web Consortium. Available at `http://www.w3.org`