Abstract—The popularity of role-based access control (RBAC) policies within industry has generated considerable interest in the research community. Since XML has become a de facto standard for data representation, most RBAC policies are expressed in XML. Although XML documents can be very large, no succinct implementations for these policies exist. This paper describes a novel implementation (not previously proposed) for schema-less and streamed XML documents to provide authorized users with the results of queries on compressed documents. The designer of the policy does not need to be aware of any implementation details. Results of this research will be essential for industry, which could take advantage of efficient implementations of RBAC policies.

Keywords: XML; XML Compression; Encryption; Role Based Access Control.

I. INTRODUCTION

Access control lists (ACL) define users who have rights to access various resources; in particular ACL for documents specify permissions to read, write or modify a specific document or parts thereof. One of the most popular schemes for defining ACL is known as role-based access control (RBAC) policy, see [1], [2], and [3] that associate access rights with roles.

The eXtensible Markup Language, XML [4], is a World Wide Web Consortium (W3C) endorsed standard where textual markups are used to describe the structure of the data. The adoption of XML has allowed organizations to more easily share data, and some government departments have made portions of their databases accessible to the public in XML format. In recent years there has been significant growth in the use of RBAC policies for managing resources in the form of XML documents or parts thereof. There has been a lot of research on RBAC policies for XML, see [5], and [6], for which parts of XML documents, called views, are defined using XPath expressions, see [7], using one of the following two implementing techniques. In an original, centralized server-based approach, the client uses secure channels to (1) provide a credential and a role name to the server; and (2) upon receiving the proper authorization, receive the materialized view (generated by the server) associated with this role. Since this approach cannot scale up to handle applications involving a large number of users and/or complex access control policies, a newer, decentralized client-based approach, see [8], [9], [10], [11], [12], and [13] provides the clients with a secure access to published documents using encryption. The system generates keyrings associated with roles, and it encrypts views using the public keys, thereby creating a single version of the document, called a multi-encrypted document, which is published and can be downloaded through possibly insecure channels. Visibility only to authorized parties is enforced by the server, and these parties are provided with keyrings consisting of private keys, to partially decrypt the encrypted document and access the view associated with the role. Since the server only has to send keyrings, scalability issues are greatly lessened. Instead of public/private key pair, a single private key (encrypted using public key) can also be used.

However, due to XML verbosity, an encrypted document may be very large, e.g., using a block cipher (such as AES [14]) the input document and the output document are always approximately the
same size. Note that with the latter approach each published document is static, and access control policies only provide read permissions associated with views of documents.

There has been considerable research on compressing XML documents, using XML-conscious compressors, e.g., XMill [15], XGrind [16] and XQueC [17], aware of specific syntactical features of XML, to increase the compression ratio in comparison to standard compression techniques (e.g., GZIP, BZIP, etc.). However, to our knowledge, there has been no research on combining compression with encryption and using this technique for RBAC. Such an approach would provide the benefit of reducing data sizes complemented by enlarged informational entropy that increases security against known plaintext attacks, see [18]. This paper investigates succinct, client-based implementation of RBAC policies for schema-less XML documents. The policy for the XML document \( D \) can specify occurrences of individual nodes of \( D \), entire subtrees of \( D \) or specific text elements in \( D \). The compression process is based on an XML compressor, called XSAQCT, see [19] and [20]. (For details, see Section II). However, the designer of policies does not need to be aware of inner-workings of this compressor, or the encryption tools.

The most important rule behind our approach is that encryption should not harm the compression ratio/speed, what needs to be encrypted, and how can multiple users decrypt and decompress the same data. In general, the biggest issue is how can multiple users decrypt and decompress the same data. In general, the biggest issue is how can multiple users decrypt and decompress the same data. In general, the biggest issue is how can multiple users decrypt and decompress the same data.

Example 1: For example, for \( D \) of the form: \(<a>t1<b<t4</b>t2</b>t5</b>t3</a>\) there are two datasets \( T1 = \{t1,t2,t3\} \) and \( T2 = \{t4,t5\} \). In each of these two datasets, text values are stored in a DFS-order (corresponding to the order imposed by the SAX parser of \( D \)) and each text is \( \backslash 0 \)-terminated. Every dataset is compressed using a text compressor, such as [21]. Now, assume that there is an RBAC policy with two roles \( R1 \) and \( R2 \) such that texts \( t1,t2,t4,t5 \) are visible in the role \( R1 \) and texts \( t3,t4,t5 \) are visible in the role \( R2 \), see Figure 1. Here, the entire \( T1 \) cannot be compressed, because, texts \( t1,t2 \) are accessible in role \( R1 \) and \( t3 \) is accessible in role \( R2 \), and so \( t1,t2 \) have to be encrypted with a different key than \( t3 \) (and it is impossible to find in the compressed \( T1 \) the separation of \( t2 \) and \( t3 \)). Therefore, \( T1 \) needs to be split into two segments, storing respectively \( t1,t2 \) and \( t3 \), and then each of these segments can be compressed and encrypted. On the other hand, \( T2 \) does not need to be split into segments, because both texts \( t4,t5 \) are visible in both roles \( R1,R2 \).

Figure 1: Example of an RBAC policy

Splitting datasets belonging to the identical paths (as described above) can also be useful for other reasons, outside of secure publishing, for example, to improve efficiency of compression of schema-less XML documents.

Contributions: There are several contributions in this paper: (1) Design of a space-efficient and secure, client-based implementation of RBAC policies for schema-less XML documents; (2) Providing space-efficiency by using a modified version of the XML compressor XSAQCT; (3) Providing security by generating a minimum
number of public/private keys necessary, and using public keys to encrypt the compressor’s output; (4) Evaluating queries such that only authorized users can access these views of the XML document that have been made available by the policy; and (5) Applications of the idea of splitting datasets into segments to other areas.

Note that while encryption is expensive, comparatively speaking, validation is not that expensive and we can not know if a document is valid until it has been completely traversed, so we may perform several costly and wasteful encryptions before discovering that the document is invalid. In such a case, combining validation and encryption only serves to slow down the validation, while performing two separate passes likely would not take much longer and would avoid wasted encryptions. Therefore, we first validate the document and then encrypt it.

This paper is organized as follows: Section II recalls basic concepts of XSAQCT, and Section III describes document-based RBAC policies. Section IV overviews the implementation, details of which are provided in Section V. Finally, Section VI describes conclusions and future work, and Section VII provides description of various functions used in the algorithms.

II. XSAQCT

This section defines paths, similar paths, and only briefly describes XML compressor, called XSAQCT, used in the implementation, for more details see [19].

Definition 1: A path in the XML document tree is of the form \( /x_1/x_2[i_2]/\ldots/x_n[i_k] \), where \( n, k > 0 \), for \( 1 \leq i \leq k \), \( i_k > 0 \), \( x_1 \) is the root of \( D \), and \( x_i \) is tag name (the node \( x_1 \) has an implicit index 1). A path may also end at text expression of the form \( \text{text}(i) \) or \( \text{text()} \). Two paths in the tree \( D \) ending at element nodes are called similar if they have the same length, and the tag names at equal positions are identical. Paths ending at texts are similar if after disregarding the text expression(s) they are similar.

If the last element of a path is a tag name \( x \), then this path represents a single node \( x \). For \( q = p/\ast \), the path \( q \) represents a subtree rooted at the node specified by \( p \). For \( q = p/\text{text()} \), the path \( q \) represents a concatenation of text children of the node \( x \) specified by \( p \), and for \( q = p/\text{text}(i) \), \( q \) represents the \( i \)-th text child of \( x \).

The similarity relation is an equivalence relation and the equivalence class of a path \( p \) is called an s-path for \( p \), see [22]. For example, for the document \( D \) from Figure 2 an s-path \( q \) for the path \( p=/>a/b[3]/e[1]/> \) consists of \( p \) and \( /a/b[3]/e[2]/ \), or using a simplified notation, an s-path is denoted by the label (in upper case) of the last element of this path, i.e., using \( E \) for the s-path \( q \) defined above.

The underlying philosophy of XSAQCT is based on the following observation. An XML document \( D \) can be represented as a tree, often composed of many similarly named and structured subtrees. Thus, it can be transformed to a more compressible form, namely an annotated tree, in which all similar paths are merged into s-paths. Figure 3 shows the annotated tree \( T_{A,D} \) for \( D \).

![Figure 2: XML document \( D \), \( t_1 \) – \( t_{19} \) are text data.](image)

![Figure 3: The annotated tree \( T_{A,D} \).](image)
Each node of \( T_{A,D} \) is labeled with a sequence of integers, called an annotation list, representing the number of occurrences of this node’s children, and is needed to restore the original XML document. For example, the annotation [0, 1, 0] for the node \( D \) indicates that the first and third occurrence of \( b \), \( b_1 \) and \( b_3 \), have no children labeled \( d \), and that the second occurrence, \( b_2 \), of \( b \) has one child labeled \( d \). In this paper, we assume the so-called full mixed-content property: non-leaf elements have \( N \) text elements, often corresponding to the whitespace/structure data (or a null-byte if the XML is unstructured), where \( N \) is equal to the number of children each XML element has, plus one. We also assume that there are no cycles, such as consecutive siblings of the form \( x \rightarrow y \rightarrow x \), or contradicting subtree orderings such as \( x \rightarrow y \), \( y \rightarrow x \) for different parents of the same similar path, for more details see [20].

The idea behind using an annotated tree to represent a document is that non-zero annotations of the node \( X \) represent consecutive occurrences of \( X \). For the node \( X[2, 0, 4, 0, 3] \) and its child \( Y[1, 2, 0, 1, 0, 2, 0, 0, 1] \) there are nine such occurrences of \( Xs \). For a child \( Y \) of \( X \), annotations in items represent occurrences of \( Ys \) as children of respective \( Xs \). Specifically, for the first two occurrences of \( X \), i.e., the annotation 2, the first two annotations in \( Y \) represent occurrences of \( Ys \) as children of these two \( Xs \), here the second \( X \) has two children \( Y \), while the first \( X \) has only one such child. Similarly, the third, fourth, fifth and sixth occurrences of \( X \) are represented by the annotation 4, and they have four occurrences of \( Ys \) as children of these \( Xs \), i.e., the fourth \( X \) has one child \( Y \), the third and fifth \( X \) have no children \( Y \), and the sixth \( X \) has two children \( Y \).

The XML compression process involves creating for each s-path of an XML document text containers, which store a delimited list of character data for all paths in the s-path. Character data is a general term for all the characters not defined in the syntax of XML, mainly text elements and structure (whitespace) data. This philosophy has shown to reduce compression ratios because similar paths often have similar data, thus improve compression efficiency. For example, in Figure [3] the text container \( T4 \) for \( C[1, 1, 1] \) stores three text elements \( t_{14}, t_{15}, \) and \( t_{19} \), because the three following paths are similar: \(/a/b[1]/c[1]/, /a/b[2]/c[1]/, /a/b[3]/c[1]/\). Each text container is compressed using a back-end compressor; a general-purpose compressor (such as GZIP). Query evaluation is lazy because typically only a single container has to be decompressed.

### III. ROLE-BASED ACCESS CONTROL POLICIES

The RBAC policy can provide separate permissions for the following elements of an XML document: a single node \( n \), a subtree rooted at \( n \), or some/all text values for \( n \).

**Definition 2:** An atomic path is a path as defined in the Definition 1. A valid path is of the form \( p_1 + p_2 + \ldots + p_k \), where \( k \geq 1 \) representing a union of all values specified by the atomic paths \( p_1, p_2, \ldots, p_n \).

A role is specified by an identifier, e.g., Admin. Let \( P_D \) denote the set of all valid non-empty paths in an XML document \( D \).

**Definition 3:** For a finite set \( \Psi \) of roles and an XML document \( D \), the document-level RBAC (DRBAC) policy is a mapping \( \pi_D : \Psi \rightarrow P_D \), typically shown as a tuple \([(R_1 : p_1), (R_2 : p_2), \ldots, (R_n : p_n)] \), where each \( p_i \in P_D \).

The designer of the DRBAC policy may decide to leave some parts of documents accessible (or inaccessible) to all users, i.e., not to encrypt them or to encrypt them, but not provide the keys needed for decryption of these nodes to any user. In this paper, we assume that all parts of the XML document not covered by \( \pi_D(\Psi) \) are accessible to all users. The set \( \pi_D(R) \) combined with all elements of \( D \) that are accessible to all users, is called a view for the role \( R \).

A policy can also be defined by abstracting away the XML document and specifying paths, which are checked for validity once the policy is instantiated to an XML document. Let \( P \) denote the set of non-empty paths, and \( \pi \) be a policy defined as \( \pi : \Psi \rightarrow P \), where \( \Psi \) is defined as in Definition 3. Such a policy \( \pi \) can be instantiated
to $\pi_D$ for various XML documents $D$, as long as paths used in $\pi$ are valid in $D$. Several examples of DRBAC policies are provided below.

**Example 2:** The policy $\pi_1$ is defined as follows: $\{(R_1 : /a/ + /a/b[2]/c[1]/), (R_2 : /a/b[1]/*)\}$

The policy $\pi_2$ is defined as follows: $\{(R_1 : /a/b[1]/text(1)), (R_2 : /a/b[1]/text(1) + a/b[2]/text(2)), (R_3 : a/b[1]/text(2) + a/b[2]/text(1))\}$

The policy $\pi_3$ is defined as follows: $\{(R_1 : /a/text(2) + /a/b[2]/text() + a/b[3]/text(4)), (R_2 : a/b[2]/text(1) + /a/b[2]/text(2) + a/b[3]/text())\}$

### A. Querying

In this version of the paper, the authorized user $U$ can only execute the so-called basic queries based on paths from the RBAC policy. Such a query evaluates to a compressed and encrypted result, which has to be first decrypted using keys available to $U$, and then decompressed.

For Example 2 for the policy $\pi_1$ $\{(R_1 : /a/ + /a/b[2]/c[1]/), (R_2 : /a/b[1]/*)\}$ the only allowable queries are:

1) for the user in role R1 and query Q1: $/a/ + /a/b[2]/c[1]/, /a, /a/b[2]/c[1]/$

2) for the user in role R2 and query Q2: $/a/b[1]/*$

Future work will support for any type of a query.

### IV. OVERVIEW OF IMPLEMENTATION

This section provides an overview of the implementation, more details are provided in Section V. Below, $\pi$ is a DRBAC policy, $D$ is an XML document and $\pi_D$ is the instantiated policy.

#### A. System architecture

The system architecture is shown in Figure 4. There are two kinds of servers, Secure Server and Un-secure Server. The secure server stores two modules:

1) $PPM$, a pre-processing module, which based on the RBAC policy $\pi$ creates an symbolic annotated tree $SA(\pi)$. The information in this tree serves as a prototype of the structure of an annotated tree. In particular, while the $SA(\pi)$ stores annotated nodes, these nodes may be different when the XML document is known. In addition, $SA(\pi)$ does not include any specific text data, as such data are not available at this stage; instead it saves the information that will be used to split incoming text data into segments (when the document is streamed), which will be individually compressed. In addition, $SA(\pi)$ stores the information as to which nodes and text segments are to be visible in specific roles, and this information is later used to encrypt these nodes and segments; or not encrypt segments leaving the nodes visible to everybody.

2) $PM$, a processing module, that will be activated after $PPM$ completes its actions and when $D$ is being streamed and processed by the SAX parser. $PM$ does not consult the policy $\pi$; instead it uses the information from $SA(\pi)$ to create a multi-encrypted tree $MA(D, \pi)$.

The un-secure server stores the tree $MA(D, \pi)$ that may be downloaded by any user, e.g., using an HTTP server.
B. Policy Assumptions

Since \( \pi \) is created with no knowledge of an XML document tree and the annotated tree will be multi-encrypted, i.e., various parts of the annotated tree will be compressed (using XSAQCT) and encrypted, we make the following assumptions:

- The role granularity is the node’s label, rather than occurrences of this label, i.e., for a given node \( n \) that is not a root of any document \( D \) used to instantiate \( \pi \), giving access in role \( R \) (in \( \pi \)) to \( n \) by specifying a path \( p \) leading to \( n \) implies that for any other path \( q \) similar to \( p \), the role \( R \) has access to the last element of \( q \);
- \( \pi \) using a path \( p/** \) provides access to all subtrees rooted at the last element of any path similar to \( p \);

It should be noted that the above assumptions do not limit the usefulness of a typical DRBAC policy, because the text granularity is more important than node granularity (typically, the information provided in the XML document appears in the text values). For the XML document \( D \) in Figure 2, instantiated policies from Example 2 give access respectively to the following nodes of \( D \):

1) For \( \pi_{1,D} \) and users in role \( R1: a \) and \( c_1, c_2, c_3 \); for users in role \( R2: \) all subtrees rooted at all nodes with the tag \( b \), i.e., \( b_1, b_2, b_3, c_1, c_2, c_3, d, e_1, e_2 \);
2) For \( \pi_{2,D} \), users have access to the following text values: (a) in role \( R1: t_5 \), in role \( R2: \) \( t_5 \) and \( t_8 \), and in role \( R3: t_6 \).
3) For \( \pi_{3,D} \), users have access to the following text values: (a) in role \( R1: t_2, t_7-t_9 \) and \( t_{10} \), in role \( R2: t_7-t_8 \) and \( t_10-t_{13} \).

C. Implementation Design Rules

Since each unit of data that will be first compressed and then encrypted with a single key, incoming text data from the XML document that is being streamed need to be analyzed to create units (called segments), each of which will be compressed but will require no more than one key for encryption. For example, for the document \( D \) from Figure 2 text data for any node such as \( b \) may have to be split into segments. On the other hand, segments should be large enough so that their compression will result in the decrease of their sizes. Therefore, the implementation is guided by the following rules:

1) When the input document is streamed for each role \( R \) in \( \pi \) two keyrings are created; a keyring \( KP(R) \) consisting of private keys and a keyring \( KU(R) \) consisting of public keys;
2) Each keyring stores a minimal number of keys necessary to enforce the RBAC policy;
3) A user in a role \( R \) in \( \pi \) can access precisely the view for \( R \);
4) A multi-encrypted tree stores text in segments, which are compressed. In creating segments, every effort is made to create as large segments as possible so that it is reasonable to expect that the compression rate will be high;
5) Each node and each segment is encrypted with a single public key to create \( MA(D, \pi) \), however nodes and segments, which are accessible to all users are not encrypted;
6) The user \( U \) who wishes to play role \( R \) has to provide an authorization request to \( PM \) and, if approved, she receives the private keyring \( KP(R) \). \( U \) downloads the multi-encrypted tree \( MA(D, \pi) \) from the un-secure server, and then can proceed to process queries. The result of a query is returned in a compressed and encrypted form, and it is decrypted with the private key in \( U \)’s possession and finally decompressed.

V. DETAILS OF IMPLEMENTATION

This section provides a detailed description of the creation of the symbolic annotated tree and multi-encrypted annotated tree.

A. Symbolic Annotated Tree

This section uses the following two standard definitions: A subsequence of a sequence \( s = s_1, \ldots, s_m \) consists of arbitrary elements of \( s \), and for \( i, j, 1 \leq i \leq j \leq n \), a substring \( \langle i, j \rangle \) of \( s \) is a subsequence of \( s \) consisting of consecutive
elements of \( s \) starting at position \( i \) and ending at position \( j \). By \( |s| \) we denote a length of the sequence \( s \).

A symbolic annotated tree \( SA(\pi) \) is an unordered tree resembling an XSAQCT’s annotated tree, whose nodes are annotated and additionally labeled by zero, one or more roles. There may be a special node * indicating a subtree rooted at this node. The reason that \( SA(\pi) \) is unordered is that neither a single atomic path in \( \pi \) nor the + operator determine the order of children.

A pessimistic approach is used by PPM to create a symbolic annotated tree \( SA(\pi) \) because it has access only to the policy \( \pi \) and has no knowledge of any XML document \( D \) that will arrive later. For example, for the atomic path /\textit{a/b}[3] in a policy, PPM will assume that there will be three occurrences of \( b \) and will create an annotated node \( B[3] \) (we use the upper-case letters for the annotated tree). The module \( PM \) will use the information from the incoming XML document \( D \) to create an annotated tree \( A(\pi,D) \), using properties of annotated trees, as described below.

Consider a node \( X[\alpha_1, \ldots, \alpha_n] \) and its child \( Y[\beta_1, \ldots, \beta_m] \). For \( j, 1 \leq j \leq n \), let \( S_X(j) = \sum_{i=1}^{j} \alpha_i \) be the sum of the first \( j \) annotations of the node \( X[\alpha_1, \ldots, \alpha_n] \), \( S_X(0) = 0 \), and \( S_X \) be the sum of all annotations of the node \( X \). For \( \alpha_j \neq 0 \) let \( I(j) \) be a substring of the sequence \( 1, \ldots, m \), called an item and defined as follows: \( I(j) = <S_X(j-1) + 1, S_X(j)> \). Thus, if \( \alpha_1 \neq 0 \) then \( I(1) = <1, \alpha_1> \).

Every node in the tree \( SA(\pi) \) has to satisfy the following conditions:

1) The root of \( SA(\pi) \) must have an annotation \( [1] \)
2) For any node \( X[\alpha_1, \ldots, \alpha_n] \) there exists \( j, 1 \leq j \leq n \) s.t. \( \alpha_j \neq 0 \)
3) If \( \alpha_1, \ldots, \alpha_i \) is a subsequence consisting of all non-zero annotations in the node \( X[\alpha_1, \ldots, \alpha_n] \) and the node \( Y[\beta_1, \ldots, \beta_m] \) is a child of \( X \) then items \( I(i_1), \ldots, I(i_r) \) form a disjoint partition of the sequence \( 1, \ldots, m \), consisting of \( |I(i_1)| \) integers, followed by \( |I(i_2)| \) integers, ..., followed by \( |I(i_r)| \) integers, see Figure 5.

Thus, the sequence \( \beta_1, \ldots, \beta_m \), which annotates the child of the node \( X \) is split into a sequence of substrings \( <1, I(i_1)> \), \( <I(i_2)+1, I(i_3)> \ldots <I(i_r-1)+1, I(i_r)> \), which is the same as the sequence of substrings \( <S_X(i_1)+1, S_X(i_2)+1, S_X(i_3)+1 \ldots S_X(i_r)+1, S_X(i_r)> \).

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure5.png}
\caption{Items for non-zero annotations.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure6.png}
\caption{Example of items.}
\end{figure}

Example 3: For the node \( X[2,0,4,0,3] \) and its child \( Y[1,2,0,1,0,2,0,0,1] \) (with items underlined), we have: \( I(1) = <1,2> \), \( I(3) = <3,6> \) and \( I(5) = <7,9> \), see Figure 6.

Corollary.

1) For \( \alpha_j \neq 0 \), \( |I(j)| = S_X(j) - (S_X(j-1) + 1) + 1 = \alpha_j \), i.e., each item \( I(j) \) has \( \alpha_j \) elements.
2) For the node \( X[\alpha_1, \ldots, \alpha_n] \) and its child \( Y[\beta_1, \ldots, \beta_m] \) we have: \( m = \sum_{j=1,\alpha_j\neq0}^{n} |I(j)| = \sum_{j=1,\alpha_j\neq0}^{n} \alpha_j = \sum_{j=1}^{n} \alpha_j = S_X(n) \), i.e., the sum of all annotations of \( X \) is equal to the number of annotations of \( Y \).

\section{Text Processing}

\( SA(\pi) \) stores the information about the segments that will be allocated for texts in the tree \( MA(\pi,D) \), using symbolic text containers. A symbolic text container stores a list of symbolic segments; each symbolic segment has three associated data: sub-container number \( N \), text number \( M \), and a list \( R \) of 0,1 or more roles. Here, \( M \) is of
one of the following forms: (1) integer value \(i\); (2) range \(i..j\); (3) values greater or equal than an integer value \(i\); or (4) any positive integer value \(*\). In our examples, the complete information about all symbolic segments will be represented by the table, with columns representing sub-container numbers, rows representing text numbers (with conventions described above, e.g., using \(3\ast\)), and storing 0, 1 or more roles. However, the actual implementation may use a more efficient solution, e.g., multilist structures.

The reason for introducing text sub-containers is that when the XML document \(D\) will be streamed, first its text data will be arriving for the first sub-container, then for the second sub-container, and so on. Therefore, segments will be created during the creation of the annotated tree, based on symbolic segments for the symbolic annotated tree and the streamed document.

**Example 4:** Figures 7, 8 and 9 shows all steps in the process of creating the symbolic tree, respectively for the policy \(\pi_1\), \(\pi_2\) and \(\pi_3\) from Example 2. In each part of these figures, from the top to bottom, the left-hand side shows an atomic path under consideration, and the right-hand side shows the resulting symbolic annotated tree. Role names appearing in circles next to the annotated node \(N\) indicate that \(N\) is accessible to these roles. Lack of roles indicates that \(N\) will be accessible to everybody. In Figure 7, the \(*\) nodes in the resulting tree (bottom right) indicate that any possible children of the node \(B[2]\) other than \(C[0,1]\) will be accessible in roles \(R1\) and \(R2\), and any possible children of the node \(C[0,1]\) will be accessible in role \(R2\). The tables representing symbolic segments in Figures 8 and 9 indicate when the current segment (in the annotated tree) that stores incoming text data should be sent for compression and the next segment should be created. For example, in Figure 9 there will be three separate segments for \(b[2]\), respectively for the first and the second text data, accessible to

---

**Figure 7:** Process of creating a Symbolic Annotated Tree \(SA(\pi_1)\) for the policy \(\pi_1\)

**Figure 8:** Process of creating a Symbolic Annotated Tree \(SA(\pi_2)\) for the policy \(\pi_2\)

**Figure 9:** Process of creating a Symbolic Annotated Tree \(SA(\pi_3)\) for the policy \(\pi_3\)
Figure 9: Process of creating a Symbolic Annotated Tree $SA(\pi_3)$ for the policy $\pi_3$, (container $T_1$ is shown at the top).

roles $R_1$ and $R_2$ and any following text data accessible to the role $R_1$.

1) Algorithm: Now, we will explain the basic ideas behind Algorithm 10 which creates the symbolic annotated tree $SA(\pi)$. We assume that the functions described in Section VII-A are available. There are three nested loops:

The outermost loop processes each path $P$ associated with the role $R$ in the policy $\pi$:

The inner loop processes each atomic path $Q : /x_1[i_1]/x_2[i_2]/\ldots/x_n[i_k]/T$ in $P$, where $T$ is empty or it is of one of the following forms: $\ast$, $text()$ or $text(K)$;

The innermost loop processes each element $x_j[i_j]$ - (here, $x_j$ is called a label, and $i_j$ is called an index and is denoted by $I$) - and it executes all main steps of the algorithm.
The algorithm uses the following global variables: the tree $SA(\pi)$ and the current node $N$ in this tree and three integer variables; $C$, which represents the position of the current annotation in the annotation list $A$ for $N$; $S$, which represents the sum of all annotations in $A$, from the first position to $C - 1$; and $J$, which represent the value of the index from the previous step of the loop. The algorithm traverses in sync the path $Q$ and the tree $SA(\pi)$, performing some basic error checking (and calling the procedure $ERROR$) and updating the tree if necessary. Correctness of the algorithm follows from the fact that each item starts from the position $S_x(C - 1) + 1$.

![Figure 11: Tracing of creation of a Symbolic Annotated Tree $SA(\pi_1)$](image)

**Example 5**: Figure 11 shows the trace the innermost loop of the Algorithm 10 and shows on the right hand-side the components of the path $a[1]/b[2]/c[3]/d[1]/e[1]/f[2]/g[1]$ and values of variables $C, S$ and $J$ used to create a symbolic annotated tree, and this tree is shown at the left-hand side. For each non-zero annotation $A$ in any node $X$ and its child $Y$, lines from $A$ to annotations in $Y$ show the item for $A$; e.g., for the 2nd annotation for the node $C[1,3,4]$ (equal to 3), annotations at positions 2, 3 and 4 in $D$. Thin arrow indicate the current annotation.

### C. Multi-encrypted Tree

The module $PM$ uses a symbolic annotated tree and a streamed XML document $D$ to create a multi-encrypted tree $MA(D, \pi)$ by applying the following process:

1) While $D$ is being streamed, pessimistic assumptions made in creating $SA(D, \pi)$ are verified and corrected. In particular, the information about symbolic segments is used to create segments. Each container is divided into one or more segments (based on symbolic segments and the information from $D$). Not all symbolic segments will actually be used to create segments, and additional segments representing text containers not specified by the policy may be created;

2) Each segment is compressed with a back-end compressor;

3) Key pairs are generated based on roles attached to nodes and segments. All nodes and segments that are not available to everybody are encrypted and for each list $R$ of roles in the policy $\pi$, $PM$ creates two keyrings: $KU(R)$ consists of public keys for the role $R$ and $KP(R)$ consists of private keys for the role $R$. Note that a key pair for a list $R$ s.t. $R$ consists of more than one role, corresponds to keys for the intersection of roles, i.e., elements accessible to more than one role; e.g., for $R = R1, R2$ a new key pair $(ku, kp)$ will be used for elements accessible in $R1$ and $R2$;

4) Keys are assigned integer key numbers corresponding to keys; e.g., for keys $k_1, ..., k_n$, key numbers are $1, ..., n$.

5) Segments and nodes that will be encrypted are labeled by key numbers;

6) Each query may require more than one key to perform decryption, e.g., the result of the query may include the intersection of more than one role and thus will be split into sub-queries such that each sub-query will require a single key. The authorized user will receive a mapping from the list of sub-queries associated with any basic query for this user so that they can use appropriate private keys in her possession.
Thus, for any role $R$ and a basic query $Q$ taken from the XPath from the role $R$ in the policy $\pi$, $PM$ creates a list of sub-queries $Q_1, \ldots, Q_m$ and a mapping $M(Q)$ from the list of private keys from $KP(R)$ to the list of key numbers;

7) There is one additional master key pair $(MU, MP)$. For any basic query and the associated list of sub-queries, this list and the mapping $M(Q)$ are encrypted with the public master key $MU$, forming $E(Q)$. The annotated tree is first compressed and then encrypted with the public master key $MU$, hiding the structure to unauthorized users, and forming the final version of the multi-encrypted tree $MA(D, \pi)$;

8) The user authorized to be in role $R$ will receive $KP(R)$ and $MP$, and then she will use $MP$ to decrypt $MA(D, \pi)$ and $E(Q)$, and $KP(R)$ to decrypt answers to queries.

Example 6: Figures [12] [13] and [14] show multi-encrypted trees for the document $D$ and respectively the instantiated policy $\pi_1$, $\pi_2$, and $\pi_3$ from Example 2. Text containers that do not show segments represent un-encrypted text containers from Figure 3. Segments are labeled by numbers representing keys. In this example, the following key pairs are generated for each policy:

For the policy $\pi_1$, key 1 is for the list of roles $R_1, R_2$ and key 2 is for the role $R_2$.

For the policy $\pi_2$, key 1 is for the list of roles $R_1, R_2$, key 2 is for the role $R_2$ and key 3 is for the role $R_3$.

For the policy $\pi_3$, key 1 is for the role $R_1$, key 2 is for the role $R_2$ and key 3 is for the roles $R_1, R_2$.

Segments and nodes not labeled by key numbers are not encrypted and represent parts of the tree, which are accessible to everybody; otherwise they are encrypted using the key corresponding to the key number. Note the each time the minimal number of keys is generated, e.g., for the policy $\pi_1$ there will be only two keys.

1) Algorithms: This section provides the algorithms to create the multi-encrypted tree $MA(D, \pi)$ for a non-cyclic XML document $D$. Parts of this algorithm are based on the algorithm, which maps a labeled tree $D$ into an annotated tree, see [22]. However, since the entire text data in a container cannot be compressed and have to be analyzed to split containers into segments, the original algorithm has been modified, using the
information about text data stored in the symbolic annotated tree. Nodes in the symbolic annotated tree are used for the following two purposes: (1) Error checking, e.g., signalling an error if in the symbolic annotated tree there is a child $B[4]$ of the root but in $D$ there are only three children of the root labeled by $b$; and (2) Assigning keys, based on roles that label nodes in the symbolic annotated tree.

In this paper, a simplified version of the algorithm is shown, with error checking omitted. The symbolic annotated tree $SA(\pi)$ can be used to create a mapping $T(\pi)$ from the set $Q(\pi)$ of s-paths to the lists of roles, and s-paths of the form $s$-path/text to text containers that store symbolic segments (nodes and text containers to be visible to all users are omitted). For example, for the tree $SA(\pi_3)$ from Figure 9 the set $Q(\pi_3)$ = \{a/text,a/b/text\}, and $T(\pi_3)(a/text) = T1$, $T(\pi_3)(a/b/text) = T2$, where $T2$ denotes the symbolic text container for the node $B[3]$. For the full version, with error checking, a synchronous traversal of the symbolic annotated tree $SA(\pi)$ and the DFS-traversal of the incoming XML document $D$ can be used (recall that SAX-parsing results in the DFS-traversal).

The following notations will be used in Algorithm 15:

- $\Path$ denotes the s-path for the path $p$
- $x(n)$ denotes a node $x$ labeled by $n$ in $D$
- $X(n)[A]$ denotes an annotated node $X$ labeled by $n$ with the annotation $A$
- for the path $p$ in $D$ and a node $x$ in $D$, $p/x$ denotes the path $p$ extended by $/x$
- $p\downarrow$ denotes the path $p$ except its last element
- $\mathcal{P}$ denotes the set of four-tuples: (s-path $[p]$, graph $G([p])$, the current node $X0([p])$ in $G([p])$, integer key number $K([p])$ representing an integer value assigned to a key).

The graph and current node may be empty, and the key number is equal to 0 if the node represented by $[p]$ is not to be encrypted.

There are two steps in the process of creating a multi-encrypted tree, respectively implemented by Algorithm 15 and Algorithm 17. Algorithm 15 uses functions defined in Section VII-B and it

**Algorithm 15**

```
1: procedure TRAVERSE
Require: $p0 = /root of D$, empty $G([p0])$, $X0([p0])$, $K([p0]) = 0$
2: while true do
3:   if root is current node and no nodes then
4:     close all current segments; return;
5:   end if
6:   if moving down to text then
7:     perform text processing;
8:     go up; continue;
9:   end if
10:  if moving down to node $x$ then
11:    $p1 \leftarrow p0/x$;
12:    if insert$P(p1)$ then
13:      $X1 \leftarrow$ insert$G(p0, label of x, Ann(p0))$;
14:      if $X0([p0]) \neq \emptyset$ then
15:        addArc$(X0([p0]), X1, p0)$;
16:      end if
17:    end if
18:    else if $p1$ is already defined
19:      $X1 \leftarrow$ member$G([p0], label of x)$;
20:      add1 to $X1$’s last annotation;
21:      if roleKind$(p1) = 0$ then
22:        $k \leftarrow$ getKey$(getRoles(p1))$;
23:        $PsetKey(X1, k)$;
24:      end if
25:      update$G(p1)$;
26:      if $X0([p0]) \neq \emptyset$
27:        & $X1 \neq X0([p0])$ then
28:          addArc$(X0([p0]), X1, [p0])$;
29:        else
30:          $X0([p0]) \leftarrow X1$;
31:        end if
32:      end if
33:      $X0([p0]) \leftarrow X1$;
34:      if $p0 \leftarrow p1$;
35:    else
36:      $X0([p0]) \leftarrow X1$;
37:      $p0 \leftarrow p1$;
38:    end if
39:  end if
40: end while
41: end procedure
```

Figure 15: Algorithm which maps an XML document $D$ to a set $\mathcal{P}$ and associated graphs with keys attached. Text datasets are split into segments and compressed. For functions used here Section VII-B.
performs a depth-first search traversal of the input document, moving down and up, which corresponds to SAX-parsing of the streamed document. These actions would be triggered by entering the beginning of the element, i.e., \(< x\) and the end of the element, i.e., \(< /x\). When this algorithm completes its actions, i.e., when parsing of \(D\) is performed, every s-path has an associated graph of annotated nodes, and every data container is split into segments, each of which is individually compressed and encrypted. The Algorithm [15] is initialized by setting the node \(x_0\) to be the root of \(D\) and \(p_0\) to be the path \(/x_0\). Then it performs a loop, maintaining the current path \(p_0\) in \(D\) and the set \(\mathcal{P}\), and moving down and up until it reaches the root node and there are no more un-visited children of the root. The following invariants are maintained: (1) The current node \(X(\lceil p_0 \rceil)\) is not null; (2) If the algorithm moves down to the node \(x\), such that the path \(p/x\) already exists, then there exists a unique annotated node \(X\) in the graph \(G([p_0])\) which has the same label as the label of \(x\). Once the algorithm complete their actions, the set \(\mathcal{P}\) stores the set of s-paths for \(D\). However, nodes are not encrypted since they have to be used in the second step that uses data created by the first step to perform a topological sort of graphs (for the description as to why this is needed, see [22]), then encrypt nodes and finally create a multi-encrypted tree.

Text Processing Now, let us explain text processing, line 8 of the Algorithm [15] and how symbolic segments are used to create segments. Recall that a symbolic segment \(SS\) is a triple (sub-container number \(SS(C)\), text number \(SS(T)\), non-empty list of roles \(SS(R)\)), and let us call a \(trigger(R)\) a pair \((C,T)\) for which there exists a symbolic segment \(SS\) with \(SS(C), SS(T), SS(R)\). Segments will always store compressed text values (i.e., these values are sent to the segment through the text, back-end compressor), and they are optionally labeled by key numbers and encrypted by the key corresponding to the key number. Text values for a node \(X\) are split into segments using the following procedure. A list \(L\) of segments for \(X\) will be created, and until processing text values for \(X\) has been completed, one segment will be current. Two integer values are maintained during text processing for \(X\): the sub-container number \(C\) corresponding to \(X[C]\) and the text number \(T\) indicating the number of the currently processed text value within the current sub-container. Initially, \(L\) is empty, \(C = 1\), \(T = 1\), a segment \(S\) is created and becomes current. If there is a \(trigger(R)\) for the pair \((C,T)\) then a key \(k\) for the list \(R\), with the key number \(n(k)\), is fetched, and generated if necessary (when \(S\) will be closed, i.e., cease to be current, it will be appended to the list \(L\). In addition, \(S\) will be encrypted with \(k\) and labeled with \(k(n)\); otherwise \(S\) will not be labeled and encrypted. When the first text value for \(X\) is encountered, it is sent to \(S\). Until processing text values for \(X\) has been completed, the following actions are executed. When a new text value \(t\) for \(X\) is encountered, values for \(C\) and \(T\) are updated. If there is a \(trigger(R1)\) for the new pair \((C,T)\) and \(R1\) is different from the \(R\) used in \(trigger(R)\) then the current segment will be closed (appended to the list and encrypted, as describes above), a new segment \(S\) will be created, a key \(k\) for the list \(R1\), with the key number \(n(k)\), is fetched, and generated if necessary. Finally, the text \(t\) is sent to \(S\).

**Example 7:** To explain the above procedure, consider the symbolic annotated tree from Figure [9]. Here, the symbolic segment \(SS1\) for the node \(A[1]\) indicates that there is a \(trigger(R1)\) for \((1,2)\). Therefore, the text \(t1\) is stored in a single, un-encrypted segment, the text \(t2\) is stored in a single segment encrypted with the key number 1, and texts \(t3\) and \(t4\) are both stored in a single, un-encrypted segment (see Figure [14]).

![Figure 16: Symbolic annotated tree for D from Example [1]](image-url)
**Example 8:** Table I shows the trace of the execution of Algorithm 15 for the document $D$ from Example 1. The symbolic annotated tree for the RBAC policy from this example is shown in Figure 16. The current annotated node $X_0(p)$ (if any) is underlined. If the tuple $(q, G(q))$ has not changed, then it is not shown again. The action of going down to the node $x$ is shown as $\downarrow x$ and the action of going up to the node $x$ is shown as $\uparrow x$. Each $q \in P$ will represent a node of the annotated tree, and the annotated nodes from the graph $G(q)$ will represent children of these nodes.

Algorithm 17 inputs the output of the Algorithms 15 and produces the multi-encrypted annotated tree.

### D. CLIENT-BASED AND SERVER-BASED PROCESSING

Until this point, we considered only the client-based version. However, in some cases, the segment to be encrypted may contain a very short text, and therefore its compression will not be efficient; see Example 7. Our future work will consider an alternative, i.e., a hybrid version, for which in some cases the user can decrypt the view they are entitled to access by using a key in their possession, but in other cases, they have to request the view from the server. Thus, DRBAC policies may be implemented in two ways: (1) the client-based access, in which the user receives the multi-encrypted annotated tree through an insecure channel and then will decrypt views using one of their keys; or (2) the hybrid-based access, in which the user receives some views through a secure channel. There is a trade-off between these two versions: limited compression ratio with the single communication with the server (the client-based version) vs. high compression ratio (the hybrid-based version) with the additional communication with the server. The above solution is feasible because while texts are incoming to be processed by the SAX parser, their length is known before they are completely read. Therefore, in the hybrid version, the length of the coming string can be used as a predictor as to how efficient text compression will be.

For the hybrid version, there are two kinds of segments, called *units*. The server-side units store short strings which will not be efficiently compressed; such units are encrypted with a system-defined key. Therefore, server-side units resemble materialized views, except they are always very short and consequently do not affect the efficiency. The client-side units are used as described in previous sections of this paper. At this point, it is not clear whether the hybrid version would be better than the client-side version; only extensive testing can provide the definite answer to this

<table>
<thead>
<tr>
<th>Move</th>
<th>$P$</th>
<th>$G([p])$</th>
<th>$K([p])$</th>
<th>Segments</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>/a</td>
<td>$A[1]$</td>
<td>$A[1]:{t_1}$</td>
<td></td>
</tr>
<tr>
<td>$t_2$</td>
<td>/b</td>
<td>$A[1]:B[1]$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$t_3$</td>
<td>$B[1]:{t_4}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$t_4$</td>
<td>$A[1]:{t_2}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$t_6$</td>
<td>$A[1]:{t_2}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$t_7$</td>
<td>$A[1]:{t_2}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$t_8$</td>
<td>$k1$</td>
<td>$A[1]:{t_1,t_2}, {t_3}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$t_9$</td>
<td>$A[1]:{t_1,t_2}, {t_3}$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table I: Trace of the execution of Algorithm 15. The current segment is underlined (it is *open* and not compressed). After $\downarrow t_3$, the segment $\{t_1,t_2\}$ is compressed and encrypted with the key $k_1$. Once the algorithm completes, all open segments (here, $\{t_4,t_5\}$ and $\{t_3\}$) are compressed and encrypted with new keys.
question.

VI. CONCLUSIONS AND FUTURE WORK

This paper presents a novel idea of using compression combined with encryption to provide a succinct and secure implementation of DRBAC policies for schema-less XML documents. No comparison with other similar systems designed for succinct, compression-focused, role-based encryption has been provided, since no such systems exist.

In our future work, we will remove various limitations described in this paper, e.g., apply our technique to arbitrary (cyclic or not) XML documents and extend the set of allowable queries. In addition, we will investigate generalizations of the approach presented, to develop the schema-based, parameterized, secure and compressed publishing techniques.

To test the security of our approach, we will determine whether or not it is possible to exploit the knowledge of similar paths, especially if there is a segment filled with fixed-sized strings. Finally, we will complete the implementation of the described technique and test its effectiveness using a specialized corpus of XML documents.

VII. APPENDIX

A. Part 1.

1) Functions operating on trees:
   \texttt{label getLabel(Tree W, Node n);} returns the label of the node \( n \);
   \texttt{node getRoot(Tree W);} returns the root of \( W \), or \texttt{null} if the tree is empty
   \texttt{node createTree(Tree W, label lab);} creates a new tree \( W \), with the root labeled by \( lab \) and annotation \( 1 \), and returns the root;
   \texttt{node getChild(Tree W, node N, label lab);} returns a child of the node \( N \) in \( W \) labeled by \( lab \), or null if there is no such child;
   \texttt{node createChild(Tree W, node N, label lab, int annot);} creates a new child of \( N \) labeled by \( lab \) and with \( annot \) annotations, each of which is equal to 0;
   
2) Functions operating on annotations:
   \texttt{int getAnnot(node N, int pos);} returns the annotation of the node \( N \) at position \( pos \);
   \texttt{setAnnot(node N, int pos, int v);} sets the annotation of the node \( N \) at position \( pos \) to \( v \);
   \texttt{update(node N, int ann, int new, int more);} which first appends more annotations (each equal to 0) to the annotation list of each child of the node \( N \), and then sets the annotation at position \( ann \) of \( N \) to \( new \);
   
3) Functions operating on roles, subtrees and keys:
   \texttt{addRole(node N, role R),} which adds the role \( R \) to the list of roles that labels \( N \), or performs no actions if \( R \) is already in this list;
   \texttt{addStarRole(node N, role R),} which adds the role \( R \) to the list of roles that labels \( N \), or performs no actions if \( R \) is already in this list. In addition, it adds the special node * labeled by \( R \) as a child of \( N \), and traverses the subtree labeled at \( N \), adding the role \( R \) to the list of roles that labels any node in this subtree and creating the special node * labeled by \( R \) as a child;
   
4) Functions operating on texts:
   \texttt{processText(node N, role R, text T, int pos),} which, based on the text \( T \), creates or updates symbolic segments within the sub-container number \( pos \) of the node \( N \) and labels them with the role \( R \).

B. Part 2.

1) Functions operating on the set \( P \):
   \texttt{bool insertP(path p) returns false if \([p] \in P \); otherwise it inserts \([p] \) into \( P \), sets \( G([p]) \) to be an empty graph, \( GX0([p]) \) to null, \( K([p]) \) to 0, and returns true;}
   \texttt{AnnotatedNode insertG(path p, label n, annotation A);} inserts a new node \( X(n)[A] \) in \( G([p]) \) and returns it;
   \texttt{node memberG(path p, label n) returns the unique node \( X \) in \( G([p]) \) such that \( X \) is labeled with \( n \);}.
   \texttt{addArc(node X1, node X2, path p) adds a new arc connecting \( X1 \) and \( X2 \) in \( G([p]) \);}.
   \texttt{annotation Ann(path p) if the path \( p \) is of the form \( \backslash root \) or the sum \( S \) of all...}
annotations of of the node $X0([p\downarrow])$ is equal to 1 then return 1, otherwise return the annotation consisting of $S$ occurrences of "0", followed by "1".

$updateG(path\ P)$ for each node $X$ in $G([p\downarrow])$ add 0 after the last annotation of $X$

2) Functions operating on roles and keys:

- $int\ roleKind(path\ p)$; returns 0 if $p$ is not valid in the symbolic annotated tree, 1 if $p$ is valid and the node pointed to by $p$ is labeled with roles, and 2 if $p$ is valid and $p$ points to the text value.

- $RoleList\ getRoles(path\ p)$; assumes that $p$ points to the node and returns the list of roles associated with this node;

- $PsetKeyN(node\ X, int k)$; assign to $X$ an integer key number $k$

C. Part 3.

To generate keys, the following approach is used. Let $RR$ denote a single role, or a list of different roles (from the policy). During the DFS-traversal, the system maintains a mapping $V$ between every $RR$ and a key pair ($V$ is initially empty). When an element node or a segment in the tree labeled by $RR$ is encountered and if $RR$ is not empty, the mapping $V$ is consulted, and if $V(\text{RR})$ is undefined, then a new key pair ($pk, pr$) is generated. Then for $V(\text{RR}) = (pk, pr)$ this node/segment is encrypted with the key $pk$, and if $RR$ consists of a single role $R$, the key $pr$ is added to the keying $KP(R)$ and the key $pk$ is added to the keying $KU(R)$; otherwise if $RR$ consists of roles $R_1, \ldots, R_k$ then for each role $R_i$, $1 \leq i \leq k$, the key $pr$ is added to $KP(R_i)$ and the key $pk$ is added to the keying $KU(R)$.

To assign integer numbers to keys, the mapping $I$ is maintained (initially $I$ is empty), and a key $counter\ kc$ (initially set to 1). Whenever a new key $k$ is generated, $I(kc)$ is set to $k$, and the value of $kc$ is incremented by one.

$sortG(path\ p)$ performs a topological sort of $G([p\downarrow])$;

In addition to functions listed before, the following functions are assumed to be available:

1) Functions operating on keys and implementing the mapping $V$:

- $key\ GetKey(roleList\ R)$; if there are no keys associated with $R$ then generate a new key pair ($pk, pr$). Then add $pk$ and $pr$ as described above, and finally return $pk$.

2) Functions encrypting:

- $encrypt(node\ X, int\ key\ k)$; if $k \neq 0$ then encrypt $X$ with the public key with the key number $k$

- $int\ SgetKeyN(roleList\ RR)$; return a key number for the list of roles $RR$, 0 if $RR$ is empty, and generate a new key pair and the corresponding key number of necessary;

REFERENCES


