ESPOON$_{ERBAC}$: ENFORCING SECURITY POLICIES IN OUTSOURCED ENVIRONMENTS

Muhammad Rizwan Asghar, Mihaela Ion, Giovanni Russello, Bruno Crispo

Note: The final version of this paper has been accepted for publication on Elsevier Computers & Security 2013.

$ESPOON_{ERBAC}$: Enforcing Security Policies in Outsourced Environments

Muhammad Rizwan Asghar[a,c], Mihaela Ion[a,c], Giovanni Russello[b], Bruno Crispo[c]

[a] *CREATE-NET, International Research Center, Trento Italy*
[b] *Department of Computer Science, The University of Auckland, Auckland New Zealand*
[c] *Department of Information Engineering and Computer Science, University of Trento, Trento Italy*

Abstract

Data outsourcing is a growing business model offering services to individuals and enterprises for processing and storing a huge amount of data. It is not only economical but also promises higher availability, scalability, and more effective quality of service than in-house solutions. Despite all its benefits, data outsourcing raises serious security concerns for preserving data confidentiality. There are solutions for preserving confidentiality of data while supporting search on the data stored in outsourced environments. However, such solutions do not support access policies to regulate access to a particular subset of the stored data.

For complex user management, large enterprises employ Role-Based Access Controls (RBAC) models for making access decisions based on the role in which a user is active in. However, RBAC models cannot be deployed in outsourced environments as they rely on trusted infrastructure in order to regulate access to the data. The deployment of RBAC models may reveal private information about sensitive data they aim to protect. In this paper, we aim at filling this gap by proposing $ESPOON_{ERBAC}$ for enforcing RBAC policies in outsourced environments. $ESPOON_{ERBAC}$ enforces RBAC policies in an encrypted manner where a curious service provider may learn a very limited information about RBAC policies. We have implemented $ESPOON_{ERBAC}$ and provided its performance evaluation showing a limited overhead, thus confirming viability of our approach.

Keywords: Encrypted RBAC, Policy Protection, Sensitive Policy Evaluation, Secure Cloud Storage, Confidentiality;

1. Introduction

In recent years, data outsourcing has become a very attractive business model. It offers services to individuals and enterprises for processing and storing a huge amount of data at very low cost. It promises higher availability, scalability, and more effective quality of service than in-house solutions. Many sectors including government and healthcare, initially reluctant to data outsourcing, are now adopting it [26].

Despite all its benefits, data outsourcing raises serious security concerns for preserving data confidentiality. The main problem is that the data stored in outsourced environments are within easy reach of service providers that could gain unauthorised access. There are several solutions for guaranteeing confidentiality of data in outsourced environments. For instance, solutions as those proposed in [14, 20] offer a protected data storage while supporting basic search capabilities performed on the server without revealing information about the stored data. However, such solutions do not support access policies to regulate the access to a particular subset of the stored data.

1.1. Motivation

Solutions for providing access control mechanisms in outsourced environments have mainly focused on encryption techniques that couple access policies with a set of keys, such as the one described in [11]. Only users possessing a key (or a set of hierarchy-derivable keys) are authorised to access the data. The main drawback of these solutions is that security policies are tightly coupled with the security mechanism, thus incurring high processing cost for performing any administrative change for both the users and the policies representing the access rights.

A policy-based solution, such the one described for the Ponder language in [29], is more flexible and easy to manage because it clearly separates the security policies from the enforcement mechanism. However, policy-based access control mechanisms are not designed to operate in outsourced environments. Such solutions can work only when they are deployed and operated within a trusted domain (i.e., the computational environment managed by the organisation owning the data). If these mechanisms are outsourced to an untrusted environment, the access policies that are to be enforced on the server may leak information on the data they are protecting. As an example, let us consider a scenario where a hospital has outsourced its healthcare data management services to a third party service provider. We assume that the service provider is honest-but-curious, similar to the existing literature on data outsourcing (such as [13]), i.e., it is honest to perform the required operations as described in the protocol but curious to learn information about stored or

Email addresses: asghar@create-net.org (Muhammad Rizwan Asghar), ion@create-net.org (Mihaela Ion), g.russello@auckland.ac.nz (Giovanni Russello), crispo@disi.unitn.it (Bruno Crispo)

* The final version of this paper has been accepted for publication in Elsevier Computers & Security 2013 [2].

exchanged data. In other words, the service provider does not preserve data confidentiality. A patient's medical record should be associated with an access policy in order to prevent an unintended access. The data is stored with an access policy. As an example, let us consider the following access policy: *only a Cardiologist may access the data*. From this policy, it is possible to infer important information about the user's medical conditions (even if the actual medical record is encrypted). This policy reveals that a patient could have heart problems. A misbehaving service provider may sell this information to banks that could deny the patient a loan given her health conditions.

Now-a-days, the most widely used security model is Role-Based Access Controls (RBAC) [31] that makes decision based on role in which a user is active in [25]. However, the current variants of RBAC model cannot be deployed in outsourced environments as they assume a trusted infrastructure in order to regulate access on data. In RBAC models, RBAC policies may leak information about the data they aim to protect. Asghar *et al.* [1] propose *ESPOON* that aims at enforcing authorisation policies in outsourced environments. They extend *ESPOON* [1] to support RBAC policies and role hierarchies [3]. However, they consider that the role assignment is performed by the Company RBAC Manager, which is run in the trusted environment.

1.2. Research Contributions

In this paper, we present an RBAC mechanism for outsourced environments where we support full confidentiality of RBAC policies. We named our solution $ESPOON_{ERBAC}$ (Enforcing Security Policies in OutsOurced envirOnmeNts with Encrypted RBAC). One of the main advantages of $ESPOON_{ERBAC}$ is that we maintain the clear separation between RBAC policies and the actual enforcing mechanism without loss of policies confidentiality under the assumption that the service provider is honest-but-curious. Our approach allows enterprises to outsource their RBAC mechanisms as a service with all the benefits associated with this business model without compromising the confidentiality of RBAC policies. Summarising, the research contributions of our approach are threefold. First, the service provider does not learn anything about RBAC policies and the requester's attributes during the policy deployment or evaluation processes. Second, $ESPOON_{ERBAC}$ is capable of handling complex contextual conditions (a part of RBAC policies) involving non-monotonic boolean expressions and range queries. Third, the system entities do not share any encryption keys and even if a user is deleted or revoked, the system is still able to perform its operations without requiring re-encryption of RBAC policies. As a proof-of-concept, we have implemented a prototype of our RBAC mechanism and analysed its performance to quantify the overhead incurred by cryptographic operations used in the proposed scheme.

1.3. Organisation

The rest of this paper is organised as follows: Section 2 reviews the related work. Section 3 provides an overview of RBAC models. Section 4 presents the proposed architecture of $ESPOON_{ERBAC}$. Section 5 and Section 6 focus on solution details and algorithmic details, respectively. Section 7 provides security analysis of $ESPOON_{ERBAC}$. Section 8 analyses the performance overhead of $ESPOON_{ERBAC}$. Finally, Section 9 concludes this paper and gives directions for the future work.

2. Related Work

Work on outsourcing data storage to a third party has been focusing on protecting the data confidentiality within the outsourced environment. Several techniques have been proposed allowing authorised users to perform efficient queries on the encrypted data while not revealing information on the data and the query [33, 7, 15, 10, 18, 8, 35, 5, 28, 32, 14]. However, these techniques do not support the case of users having different access rights over the protected data. Their assumption is that once a user is authorised to perform search operations, there are no restrictions on the queries that can be performed and the data that can be accessed.

The idea of using an access control mechanism in an outsourced environment was initially explored in [12, 13]. In this approach, Vimercati *et al.* provide a selective encryption strategy for enforcing access control policies. The idea is to have a selective encryption technique where each user has a different key capable of decrypting only the resources a user is authorised to access. In their scheme, a public token catalogue expresses key derivation relationships. However, the public catalogue contains tokens in the clear that express the key derivation structure. The tokens could leak information on access control policies and on the protected data. To circumvent the issue of information leakage, in [11] Vimercati *et al.* provide an encryption layer to protect the public token catalogue. This requires each user to obtain the key for accessing a resource by traversing the key derivation structure. The key derivation structure is a graph built (using access key hierarchies [4]) from a classical access matrix. There are several issues related to this scheme. First, the algorithm of building key derivation structure is very time consuming. Any administrative actions to update access rights require the users to obtain new access keys derived from the rebuilt key derivation structure and it consequently requires data re-encryption with new access keys. Therefore, the scheme is not very scalable and may be suitable for a static environment where users and resources do not change very often. Second, the scheme does not support complex policies where contextual information may be used for granting access rights. For instance, only specific time and location information associated with an access request may be legitimate to grant access to a user.

Another possible approach for implementing an access control mechanism is protecting the data with an encryption scheme where the keys can be generated from the user's credentials (expressing attributes associated with that user). Although these approaches are not devised particularly for outsourced environments, it is still possible to use them as access control mechanisms in outsourced settings. For instance, a recent work by Narayan *et al.* [23] employ the variant of Attribute Based Encryption (ABE) proposed in [6] (i.e., Ciphertext Policy ABE,

or CP-ABE in short) to construct an outsourced healthcare system where patients can securely store their Electronic Health Record (EHR). In their solution, each EHR is associated with a secure search index to provide search capabilities while guaranteeing no information leakage. However, one of the problems associated with CP-ABE is that the access structure, representing the security policy associated with the encrypted data, is not protected. Therefore, a curious storage provider might get information on the data by accessing the attributes expressed in the CP-ABE policies. The problem of having the access structure expressed in cleartext affects in general all the ABE constructions [30, 16, 27, 6]. Therefore, this mechanism is not suitable for guaranteeing confidentiality of access control policies in outsourced environments.

Asghar *et al.* [1] propose *ESPOON* that aims at enforcing authorisation policies in outsourced environments. In *ESPOON*, a data owner (or someone on the behalf of data owners) may attach an authorisation policy with the data while storing it on the outsourced server. Any authorised requester may get access to the data if she satisfies the authorisation policy associated with that data. However, *ESPOON* lacks to provide support for RBAC policies. In [3], Asghar *et al.* extended *ESPOON* to support RBAC policies and role hierarchies. However, in [3] the role assignment is performed by the Company RBAC Manager, which is run in the trusted environment. On the other hand, in our current architecture, the role assignment is performed by the service provider running in the outsourced environment. In other words, we have eliminated the need of an additional online-trusted-server i.e., the Company RBAC Manager.

Related to the issue of the confidentiality of the access structure, the hidden credentials scheme presented in [17] allows one to decrypt ciphertexts while the involved parties never reveal their policies and credentials to each other. Data can be encrypted using an access policy containing monotonic boolean expressions which must be satisfied by the receiver to get access to the data. A passive adversary may deduce the policy structure, i.e., the operators (AND, OR, m-of-n threshold encryption) used in the policy but she does not learn what credentials are required to fulfill the access policy unless she possesses them. Bradshaw *et al.* [9] extend the original hidden credentials scheme to limit the partial disclosure of the policy structure and speed up the decryption operations. However, in this scheme, it is not easy to support non-monotonic boolean expressions and range queries in the access policy. Last, hidden credentials schemes assume that the involved parties are online all the time to run the protocol.

3. Overview of RBAC Models

RBAC [31] is an access control model that logically maps well to the job-function specified within an organisation. In the basic RBAC model, a system administrator or a security officer assigns permissions to roles and then roles are assigned to users. A user can make an access request to execute permissions corresponding to a role only if he or she is active in that role. A user can be active in a subset of roles assigned to him/her by making a role activation request. In RBAC, a session keeps mapping of users to roles that are active.

In [31], Sandhu *et al.* extend the basic RBAC model with role hierarchies for structuring roles within an organisation. The concept of role hierarchy introduces the role inheritance. In the role inheritance, a derived role can inherit all permissions from the base role. The role inheritance incurs extra processing overhead as requested permissions might be assigned to the base role of one in which the user might be active.

The RBAC model may activate a role or grant permissions while taking into account the context under which the user makes the access request or the role activation request [21, 19, 34, 24, 22]. The RBAC model captures this context by defining contextual conditions. A contextual condition requires certain attributes about the environment or the user making the request. These attributes are contextual information, which may include access time, access date and location of the user who is making the request. The RBAC model grants the request if the contextual information satisfy the contextual conditions.

4. The $ESPOON_{ERBAC}$ Approach

$ESPOON_{ERBAC}$ aims at providing RBAC mechanism that can be deployed in an outsourced environment. Figure 1 illustrates the proposed architecture that has similar components to the widely accepted architecture for the policy-based management proposed by IETF [36]. In
$ESPOON_{ERBAC}$, an **Admin User** deploys (i) RBAC policies and sends them to the **Administration Point** that stores (ii) RBAC policies[1] in the **Policy Store**. These policies may include permissions assigned to roles, roles assigned to users and the role hierarchy graph that are stored in the Permission Repository, the Role Repository and the Role Hierarchy repository, respectively.

A **Requester** may send (1) the role activation request to the **Policy Enforcement Point** (PEP). This request includes the Requester's identifier and the requested role. The PEP forwards (2) the role activation request to the **Policy Decision Point** (PDP). The PDP retrieves (3) the policy corresponding to the Requester from the Role Repository of the **Policy Store** and fetches (4) the contextual information from the **Policy Information Point** (PIP). The contextual information may include the environmental and Requester's attributes under which the requested role can be activated. For instance, consider a contextual condition where a role doctor can only be activated during the duty hours. For simplicity, we assume that the PIP collects all required attributes and sends all of them together in one go. Moreover, we assume that the PIP is deployed in the trusted environment. However, if attributes forgery is an issue, the PIP can request a trusted authority to sign the attributes before sending them to the PDP. The PDP evaluates role assignment policies against the attributes provided by the PIP checking if the contextual information satisfies contextual conditions and sends to the PEP (5) the role activation response. In case of *permit*,

[1] In the rest of this paper, by term *policies* we mean *RBAC policies*.

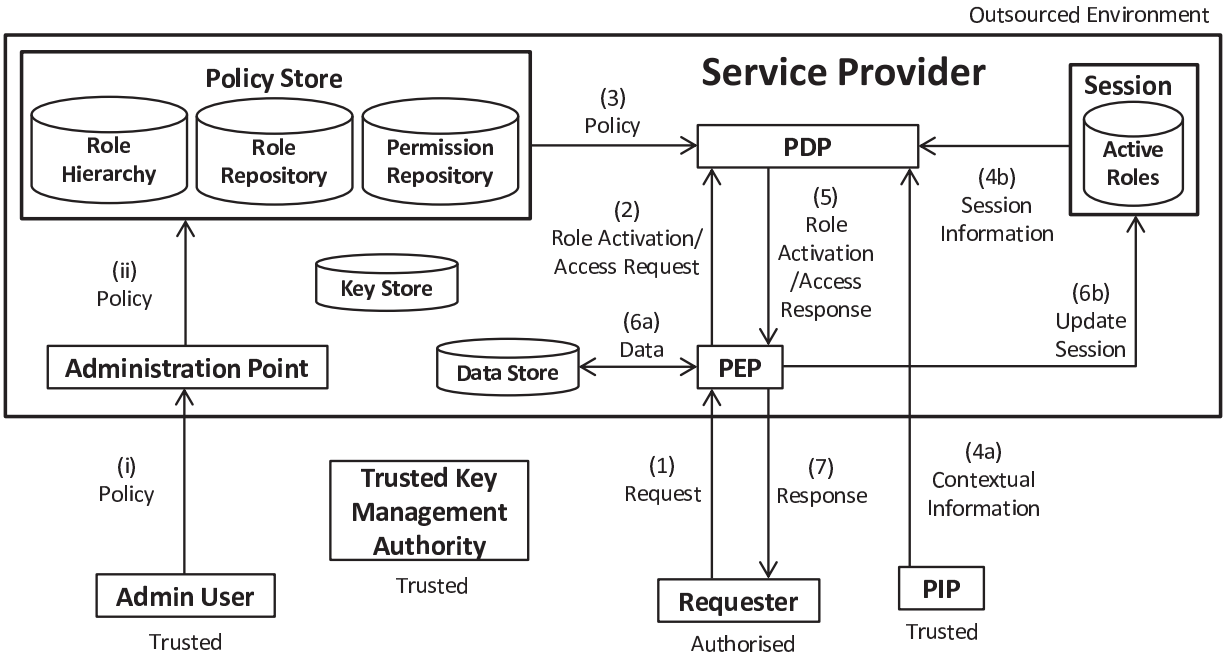


Figure 1: The $ESPOON_{ERBAC}$ architecture for enforcing RBAC policies in outsourced environments

the PEP activates the requested role by updating the **Session** containing the Active Roles repository (6a). Otherwise, in case of *deny*, the requested role is not activated. Optionally, a response can be sent to the Requester (7) with either *success* or *failure*.

After getting active in a role, a Requester can make the access request that is sent to the PEP (1). This request includes the Requester's identifier, the requested data (target) and the action to be performed. The PEP forwards (2) the access request to the PDP. After receiving the access request, the PDP first retrieves from the Session information about the Requester if she is already active in any role (3a). If so, the PDP evaluates if the Requester's (active) role is permitted to execute the requested action on the requested data. For this purpose, the PDP retrieves (3) the permission assignment policy corresponding to the active role from the Permission Repository of the Policy Store and fetches (4) the contextual information from the PIP required for evaluating contextual conditions in the permission assignment policy. For instance, consider the example where a *Cardiologist* can access the cardiology report during the office hours. The PDP evaluates the permission assignment policies against the attributes provided by the PIP checking if the contextual information satisfies any contextual conditions and sends to the PEP (5) the access response. In case of *permit*, the PEP forwards the access action to the **Data Store** (6b). In case if no contextual condition is satisfied, the PDP retrieves the role hierarchy from the Role Hierarchy repository of the Policy Store and then traverses this role hierarchy graph in order to find if any base role, the Requester's role might be derived from, has permission to execute the requested action on the requested data. If so, the PEP forwards the access action to the Data Store (6b). Otherwise, in case of *deny*, the requested action is not forwarded. Optionally, a response can be sent to the Requester (7) with either *success* or *failure*.

The main difference with the standard proposed by IETF is that the $ESPOON_{ERBAC}$ architecture is outsourced in an untrusted environment (see Figure 1). The trusted environment comprises only a minimal IT infrastructure that is the applications used by the Admin Users and Requesters, together with the PIP. This reduces the cost of maintaining an IT infrastructure. Having the reference architecture in the cloud increases its availability and provides a better load balancing compared to a centralised approach. In outsourced environments, $ESPOON_{ERBAC}$ guarantees that the confidentiality of policies is protected not only when they are deployed but also when they are enforced. This offers a more efficient evaluation of policies. For instance, a naive solution would see the encrypted policies stored in the cloud and the PDP deployed in the trusted environment. At each evaluation, the encrypted policies would be sent to the PDP that decrypts the policies for a cleartext evaluation. After that, the policies need to be encrypted and send back to the cloud. The **Service Provider**, where the architecture is outsourced, is honest-but-curious. This means that the provider allows the $ESPOON_{ERBAC}$ components to follow the specified protocols, but it may be curious to find out information about the data and the policies regulating the accesses to the data. As for the data, we assume that data confidentiality is preserved by one of the several techniques available for outsourced environments [14, 28, 32]. However, to the best of our knowledge, no solution exists that addresses the problem of guaranteeing the policy confidentiality while allowing an efficient evaluation mechanism that is clearly separated from the policies. Most of the techniques discussed in the related work section require the security mechanism to be tightly coupled

if ⟨*CONDITION*⟩ then ⟨*USER*⟩ can be active in ⟨$\{R_1, R_2, \ldots, R_n\}$⟩

Figure 2: RBAC Policy: Role assignment

if ⟨*CONDITION*⟩ then ⟨*R*⟩ can execute ⟨$\{(A_1, T_1), (A_2, T_2), \ldots, (A_n, T_n)\}$⟩

Figure 3: RBAC Policy: Permission assignment

with the policies. In the following section, we can show that it is possible to maintain a generic PDP separated from the security policies and able to take access decisions based on the evaluation of encrypted policies. In this way, the policy confidentiality can be guaranteed against a curious provider and the functionality of the access control mechanism is not restricted.

4.1. System Model

Before presenting the detail of the scheme used in $ESPOON_{ERBAC}$, it is necessary to discuss the system model. In this section, we identify the following system entities:

- **Admin User:** This type of user is responsible for the administration of policies stored in the outsourced environment. An Admin User can deploy new policies or update/delete already deployed policies.
- **Requester:** A Requester is a user that requests an access (e.g., read, write or search) over the data residing in the outsourced environment. Before the access is permitted, policies deployed in the outsourced environment are evaluated.
- **Service Provider (SP):** The SP is responsible for managing the outsourced computation environment, where the $ESPOON_{ERBAC}$ components are deployed and to store the data, and policies. It is assumed the SP is honest-but-curious (as [13] does), i.e., it allows the components to follow the protocol to perform the required actions but curious to deduce information about the exchanged and stored policies.
- **Trusted Key Management Authority (TKMA):** The TKMA is fully trusted and responsible for generating and revoking the keys. For each type of authorised users (including an Admin User and a Requester), the TKMA generates two key sets and securely transmits the client key set to the user and the server key set to the Administration Point. The Administration Point inserts the server side key set in the **Key Store**. The TKMA is deployed on the trusted environment. Although requiring a TKMA seems at odds with the need of outsourcing the IT infrastructure, we argue that the TKMA requires less resources and less management effort. Securing the TKMA is much easier since a very limited amount of data needs to be protected and the TKMA can be kept offline most of the time.

It should be clarified that in our settings an Admin User is not interested in protecting the confidentiality of policies from other Admin Users and Requesters. Here, the main goal is to preserve the confidentiality of data and policies from the SP.

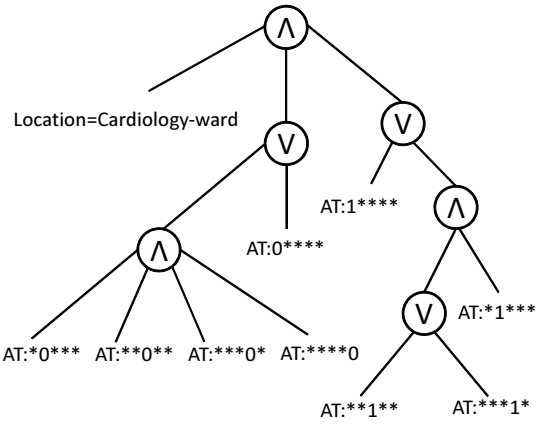


Figure 4: An example of contextual condition illustrating *Location* = *Cardiology-ward* and $AT > 9\#5$ and $AT < 17\#5$

4.2. Representation of RBAC Policies/Requests

In this section, we provide details about how to represent policies and requests used in our approach. An RBAC policy contains a role assignment policy, a permission policy and a role hierarchy graph. In the following, we discuss each of them. Figure 2 illustrates how we represent role assignment policies in $ESPOON_{ERBAC}$. The meaning of role assignment policy is as follows: if contextual condition, *CONDITION*, is *true* then *USER* can be active in any role(s) out of role set $\{R_1, R_2, \ldots, R_n\}$. Figure 3 illustrates how we represent permission assignment policies in $ESPOON_{ERBAC}$. The meaning of permission assignment policy is as follows: if contextual condition, *CONDITION*, is *true* then role *R* can execute any permission(s) out of permission set $\{(A_1, T_1), (A_2, T_2), \ldots, (A_n, T_n)\}$.

The PDP evaluates contextual conditions of both role assignment and permission assignment policies before granting the access. In order to evaluate a contextual condition, the PDP requires contextual information. The contextual information captures the context in which a Requester makes access or role activation requests. The PIP collects and sends required contextual information to the PDP. To represent contextual conditions, we use the tree structure described in [6] for CP-ABE policies. This tree structure allows an Admin User to express contextual conditions as conjunctions and disjunctions of equalities and inequalities. Internal nodes of the tree structure are AND, OR or threshold gates (e.g., 2 of 3) and leaf nodes are values of condition predicates either string or numerical. In the tree structure, a string comparison is represented by a single leaf node. However, the tree structure uses the *bag of bits* representation to support comparisons between numerical values that could express time, date, location, age, or any numerical identifier. For instance, let us consider a contextual condition stating that the Requester location should be *Cardiology-ward* and that the access time should be between 9:00 and 17:00 hrs. Figure 4 illustrates the tree structure representing this contextual condition,

R_1 extends $\langle\{R_i, R_{ii}, \ldots, R_{k_1}\}\rangle$
R_2 extends $\langle\{R_i, R_{ii}, \ldots, R_{k_2}\}\rangle$
.
.
.
R_n extends $\langle\{R_i, R_{ii}, \ldots, R_{k_n}\}\rangle$

Figure 5: RBAC Policy: Role hierarchy

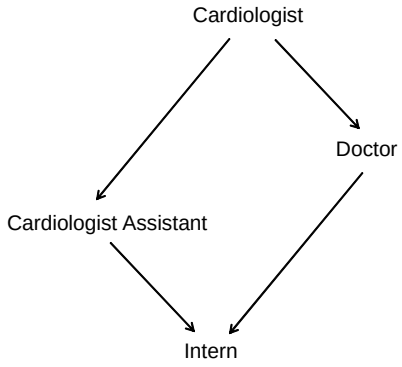


Figure 6: Role hierarchy graph

where access time (AT) is in a 5-bit representation (#5).

A Requester can make a role activation request *ACT* or an access request *REQ*. In $ACT = (i, R)$, a Requester includes her identity i along with role R to be activated. After a Requester is active in R, she can execute permissions assigned to R. For executing any permission, a Requester sends $REQ = (R, A, T)$ that includes R she is active in, action A to be taken over target T. A Requester sends *ACT* or *REQ* requests to the PEP.

The PEP receives and forwards requests *ACT* or *REQ* to the PDP. The PDP fetches policies corresponding to requests from the Policy Store. The PDP may require contextual information in order to evaluate contextual conditions to grant *ACT* or *REQ*. Let us consider *CONDITION* illustrated in Figure 4 requiring location of Requester and access time. We assume the Requester makes the request when she is in *Cardiology-ward* and access time (AT) is 10:00 hrs. The PIP collects and then transforms this contextual information as follows: *Location* = *Cardiology-ward*, $AT : 0****$, $AT : *1***$, $AT : **0**$, $AT : ***1*$, $AT : ****0$, where AT is in a 5-bit representation (same as it is in *CONDITION*). After performing transformation, the PIP sends contextual information to the PDP. The PDP receives contextual information and then evaluates *CONDITION* by first matching attributes in contextual information against leaf-nodes in the *CONDITION* tree and then evaluating internal nodes according to AND and OR gates.

The $ESPOON_{ERBAC}$ architecture supports role inheritance. In role inheritance, a derived role can execute all permissions from its base role. Before denying *REQ*, the PDP may need to check if base role of one in *REQ* can execute requested permissions. In order to find base roles, we store a role hierarchy graph on the SP. In $ESPOON_{ERBAC}$, the PDP traverses in the role hierarchy graph to find base roles. Figure 5 illustrates how we represent a role hierarchy graph. In Figure 5, each line represents a role that may extend a set of roles. All these inheritance rules may form a role hierarchy graph. For instance, consider an example from healthcare domain where a *Cardiologist Assistant* extends *Intern*, a *Doctor* extends *Intern* and finally a *Cardiologist* extends both *Cardiologist Assistant* and *Doctor*. If we combine all these inheritance rules then it can form a graph as shown in Figure 6.

In this representation, leaf-nodes in *CONDITION*, *R*, *A*, *T* of both *ACT* and *REQ*, roles in the role hierarchy graph, and attributes in contextual information are in cleartext. Therefore, such information is easily accessible in the outsourced environment and may leak information about the data that policies protect. In the following, we show how we protect such representation while allowing the PDP to evaluate policies against requests and contextual information.

5. Solution Details

$ESPOON_{ERBAC}$ aims at enforcing policies in outsourced environments. The main idea of our approach is to use an encryption scheme for preserving confidentiality of policies while allowing the PDP to perform the correct evaluation. In $ESPOON_{ERBAC}$, we can notice that the operation performed by the PDP for evaluating policies (against attributes in the request and contextual information) is similar to the search operation executed in a database. In particular, in our case the policy is a query; while, attributes in the request (*ACT* or *REQ*) and contextual information represent the data.

For $ESPOON_{ERBAC}$, as a starting point we consider the multiuser Searchable Data Encryption (SDE) scheme proposed by Dong *et al.* in [14]. The SDE scheme allows an untrusted server to perform searches over encrypted data without revealing to the server information on both the data and elements used in the request. The advantage of this method is that it offers multi-user access without requiring key sharing between users. Each user in the system has a unique set of keys. The data encrypted by one user can be decrypted by any other authorised user. However, the SDE implementation in [14] is only able to perform keyword comparison based on equalities. One of the major extensions of our implementation is that we are able to support the evaluation of contextual conditions containing complex boolean expressions such as non-conjunctive and range queries in multi-user settings.

In general, we distinguish four phases in $ESPOON_{ERBAC}$ for managing life cycle of policies in outsourced environments. These phases include *initialisation*, ***policy deployment***, ***policy evaluation*** and *user revocation*. In the following, we provide details of each phase.

5.1. Initialisation Phase

In $ESPOON_{ERBAC}$, each user (including an Admin User and a Requester) obtains a client side key from the TKMA while the SP (as a proxy server) receives a server side key set corresponding to the user. The client side key set serves as a private key for a user. The SP stores all key sets in the Key Store. The Key

Store is accessible to the Administration Point, the PEP and the PDP.

5.2. Policy Deployment Phase

For deploying (or updating existing) policies, an Admin User performs a first round of encryption using her client side key set. An Admin User encrypts elements of policies. In role assignment policies, an Admin User encrypts all roles assigned to a user. In permission assignment policies, an Admin User encrypts both action and target parts of each permission and also encrypts the role to which these permissions are assigned. As we know that a tree represents condition conditions of both role assignment and permission assignment policies (as shown in Figure 4), an Admin User encrypts each leaf node of the tree while non-leaf (internal) nodes representing AND, OR or threshold gates are in cleartext. In a role hierarchy graph (as shown in Figure 6), an Admin User encrypts each of its node representing a role. After completing the first round of encryption on policies, an Admin User sends client encrypted policies to the Administration Point on the SP. These client encrypted policies are protected but cannot be enforced as these are not in common format. To convert client encrypted policies to common format, the Administration Point performs a second round of encryption using server side key set corresponding to the Admin User. The second round of encryption serves as a proxy re-encryption. In the second round of encryption, the Administration Point encrypts all elements that are encrypted in the first round of encryption. Finally, the Administration Point stores server encrypted policies in the Policy Store.

5.3. Policy Evaluation Phase

A Requester can make a role activation request *ACT*. Before sending *ACT* to the SP, a Requester generates a client trapdoor of the role in *ACT*. A Requester generates client trapdoor using her client side key set. The trapdoor representation does not leak information on elements of requests. Similarly, a Requester can make an access request *REQ* after getting active in a role. A Requester generates a client trapdoor for each element in *REQ* including the role, the action and the target. A Requester sends requests containing client generated trapdoors to the PEP on the SP. The PEP performs another round of trapdoor generation for converting all trapdoors into a common format. After performing a second round of trapdoor generation on the server side, the PEP forwards server generated trapdoors to the PDP. The PDP fetches policies from the Policy Store and then performs encrypted matching of trapdoors in request against encrypted elements in policies. The encrypted matching in outsourced environments does not leak information about elements of requests or policies.

The PDP may require contextual information in order to evaluate the contextual conditions of policies. The PIP collects contextual information and generates client trapdoors for elements of contextual information using her client side key set. The PIP sends client generated trapdoors of contextual information to the PDP. The PDP performs another round of trapdoor generation using server side key set corresponding to the PIP. Finally, the PDP evaluates the contextual condition by matching trapdoors of contextual information against encrypted leaf nodes of the tree representing the contextual condition (as shown in Figure 4). After evaluating leaf nodes, the PDP evaluates non-leaf nodes of the tree based on AND, OR and threshold gates. The PDP grants the access request if (the root node of) the tree evaluates to *true*.

The PDP may need to find base roles corresponding to the role in *REQ* considering the fact that a derived role has all permissions from its base role. In order to find base role, the PDP fetches the role hierarchy graph from the Policy Store. The PDP matches trapdoor of role in *REQ* against server encrypted roles in the role hierarchy graph. While deploying the role hierarchy graph, we store also server generated trapdoor of the role along with each server encrypted of role because the PDP needs a trapdoor of each base role so that it can match this trapdoor against roles in the Permission Repository. After traversing in the role hierarchy graph, the PDP extracts server generated trapdoors of all base roles of one that matches with trapdoor of role in *REQ*. The PDP verifies if any base role has requested permissions. If so, the PDP grants the request.

5.4. User Revocation Phase

In $ESPOON_{ERBAC}$, users do not share any keys and a compromised user can be revoked without requiring re-encryption of policies or re-distribution of keys. For revoking a compromised user, the Administration Point removes the server side key set (corresponding to the user) from the Key Store.

Algorithm 1 Init

Input: A security parameter 1^k.
Output: The public parameters *param* and the master secret key *msk*.

1: Generate primes p and q of size 1^k such that $q \mid p-1$
2: Create a generator g such that $\mathbb{G}$ is the unique order q subgroup of $\mathbb{Z}_p^*$
3: Choose a random $x \in \mathbb{Z}_q^*$
4: $h \leftarrow g^x$
5: Choose a collision-resistant hash function H
6: Choose a pseudorandom function f
7: Choose a random key s for f
8: $param \leftarrow (\mathbb{G}, g, q, h, H, f)$
9: $msk \leftarrow (x, s)$
10: **return** $(param, msk)$

6. Algorithmic Details

In this section, we provide details of algorithms used in each phase for managing life cycle of policies. All these algorithms constitute the proposed schema.

6.1. Initialisation Phase

In this phase, the system is initialised and then the TKMA generates required keying material for entities in $ESPOON_{ERBAC}$. During the system initlisation, the TKMA takes a security parameter k and outputs the public parameters *params* and the master key set *msk* by running **Init** illustrated in Algorithm 1. The detail of **Init** is as follows: the TKMA generates two prime numbers p and q of size k such that q divides $p-1$ (Line 1). Then, it creates a cyclic group $\mathbb{G}$ with a

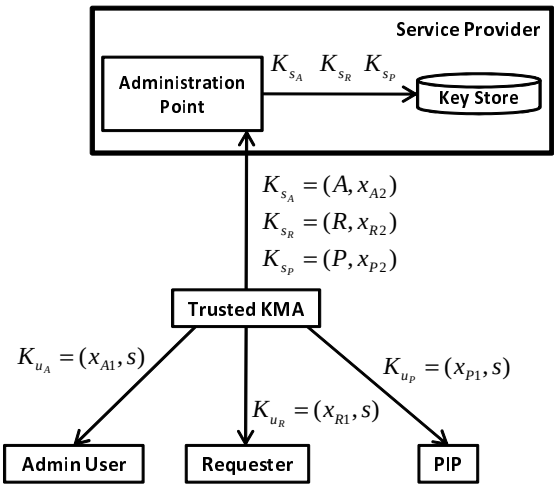


Figure 7: Key distribution

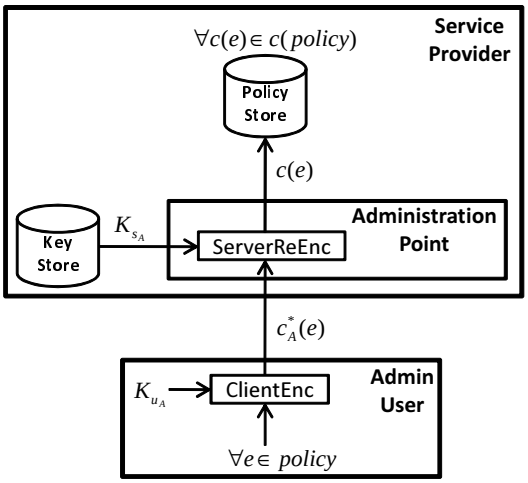


Figure 8: Policy deployment phase

generator g such that $\mathbb{G}$ is the unique order q subgroup of $\mathbb{Z}_p^*$ (Line 2). Next, it randomly chooses $x \in \mathbb{Z}_q^*$ (Line 3) and compute h as g^x (Line 4). Next, it chooses a collision-resistant hash function H (Line 5), a pseudorandom function f (Line 6) and a random key s for f (Line 7). Finally, it publicises the public parameters $params = (\mathbb{G}, g, q, h, H, f)$ (Line 8) and keeps securely the master secret key $msk = (x, s)$ (Line 9).

Algorithm 2 KeyGen

Input: The master secret key msk, the user identity i and the public parameters $params$.
Output: The client side key set K_{u_i} and server side key set K_{s_i}.

1: Choose a random $x_{i1} \in \mathbb{Z}_q^*$
2: $x_{i2} \leftarrow x - x_{i1}$
3: $K_{u_i} \leftarrow (x_{i1}, s)$
4: $K_{s_i} \leftarrow (i, x_{i2})$
5: **return** (K_{u_i}, K_{s_i})

For each user (including an Admin User and a Requester), the TKMA generates the keying material. For generating the keying material, the TKMA takes the master secret key msk, the user identity i and the public parameters $params$ and outputs two key sets: the client side key set K_{u_i} and the server side key set K_{s_i} by running **KeyGen** illustrated in Algorithm 2. In **KeyGen**, TKMA randomly chooses $x_{i1} \in \mathbb{Z}_q^*$ (Line 1) and computes $x_{i2} = x - x_{i1}$ (Line 2). It creates the client side key set $K_{u_i} = (x_{i1}, s)$ (Line 3) and the server side key set $K_{s_i} = (i, x_{i2})$ (Line 4).

After running Algorithm 2, the TKMA sends the client side key set K_{u_i} and the server side key set K_{s_i} to user i and the Administration Point on the SP, respectively. The client side key set K_{u_i} serves as a private key for user i. The Administration Point of the SP inserts K_{s_i} in the Key Store by updating it as follows: $KS = KS \cup K_{s_i}$. The Key Store is initialised as: $KS \leftarrow \phi$. Figure 7 illustrates key distribution where Admin User A, Requester R and PIP P receive K_{u_A}, K_{u_R} and K_{u_P}, respectively. The TKMA sends the corresponding server side key sets K_{s_A}, K_{s_R} and K_{s_P} to the Administration Point on the SP. The Administration Point inserts server side key sets into the Key Store. Please note that only the Administration Point, the PDP and the PEP are authorised to access the Key Store.

Algorithm 3 ClientEnc

Input: Element e, the client side key set K_{u_i} corresponding to Admin User i and the public parameters $params$.
Output: The client encrypted element $c_i^*(e)$.

1: Choose a random $r_e \in \mathbb{Z}_q^*$
2: $\sigma_e \leftarrow f_s(e)$
3: $\hat{c}_1 \leftarrow g^{r_e+\sigma_e}$
4: $\hat{c}_2 \leftarrow \hat{c}_1^{x_{i1}}$
5: $\hat{c}_3 \leftarrow H(h^{r_e})$
6: $c_i^*(e) \leftarrow (\hat{c}_1, \hat{c}_2, \hat{c}_3)$
7: **return** $c_i^*(e)$

Algorithm 4 ServerReEnc

Input: The client encrypted element $c_i^*(e)$ and the server side key set K_{s_i} corresponding to Admin User i.
Output: The server encrypted element $c(e)$.

1: $c_1 \leftarrow (\hat{c}_1)^{x_{i2}}.\hat{c}_2 = \hat{c}_1^{x_{i1}+x_{i2}} = (g^{r_e+\sigma_e})^x = h^{r_e+\sigma_e}$
2: $c_2 = \hat{c}_3 = H(h^{r_e})$
3: $c(e) = (c_1, c_2)$
4: **return** $c(e)$

6.2. Policy Deployment Phase

In the policy deployment phase, an Admin User defines and deploys policies. In general, a policy can be deployed after performing two rounds of encryptions. An Admin User performs a first round of encryption while the Administration Point on the SP performs a second round of encryption. For performing a first round of encryption, an Admin User runs **ClientEnc** illustrated in Algorithm 3. **ClientEnc** takes as input (policy) element e, the client side key set K_{u_i} corresponding to Admin User i and the public parameters $params$ and outputs the client encrypted element $c_i^*(e)$. In **ClientEnc**, an Admin User randomly chooses $r_e \in \mathbb{Z}_q^*$ (Line 1), computes σ_e as $f_s(e)$ (Line 2), and then computes $\hat{c}_1$, $\hat{c}_2$ and $\hat{c}_3$ as $g^{r_e+\sigma_e}$ (Line 3), $\hat{c}_1^{x_{i1}}$ (Line 4) and $H(h^{r_e})$ (Line 5), respectively. $\hat{c}_1$, $\hat{c}_2$ and $\hat{c}_3$ constitute $c_i^*(e)$ (Line 6). An Admin User transmits to the Administration Point the client encrypted elements of a policy as shown in Figure 8.

The Administration Point retrieves the server side key set corresponding to the Admin User and performs a second round of encryption by running **ServerReEnc** illustrated in Algorithm 4. **ServerReEnc** takes as input the client encrypted element $c_i^*(e)$ and the server side key set K_{s_i} corresponding to Admin User i and outputs the server encrypted element $c(e)$. The Administration Point calculates c_1 and c_2 as $(\hat{c}_1)^{x_{i2}}.\hat{c}_2 = \hat{c}_1^{x_{i1}+x_{i2}} = (g^{r_e+\sigma_e})^x = h^{r_e+\sigma_e}$ (Line 1) and $\hat{c}_3 = H(h^{r_e})$ (Line 2), respectively. Both c_1 and c_2 form $c(e)$ (Line 3). The Administration Point stores the server encrypted policies in the Policy Store as shown in Figure 8.

In the following, we describe how to deploy different (parts of) policies including role assignment, permission assignment, contextual conditions and role hierarchy graph. For the deployment of each (part of) policy, we follow general strategy as already described in this section and also illustrated in Figure 8.

Algorithm 5 RoleAssignment:ClientSide

Input: List of roles L to be assigned to Requester j, the client side key set K_{u_i} corresponding to Admin User i and the public parameters *params*.
Output: The client encrypted role assignment list L_{C_i}.

1: $L_{C_i} \leftarrow \phi$
2: **for** each role r in list L **do**
3: $\quad c_i^*(r) \leftarrow$ call **ClientEnc** (r, K_{u_i}, *params*) {see Algorithm 3}
4: $\quad L_{C_i} \leftarrow L_{C_i} \cup c_i^*(r)$
5: **end for**
6: **return** (j, L_{C_i})

Algorithm 6 RoleAssignment:ServerSide

Input: The client encrypted role assignment list L_{C_i} for Requester j and identity i of Admin User.
Output: The server encrypted role assignment list L_S.

1: $K_{s_i} \leftarrow KS[i]$ {retrieve the server side key corresponding to Admin User i}
2: $L_S \leftarrow \phi$
3: **for** each client encrypted role $c_i^*(r)$ in list L_{C_i} **do**
4: $\quad c(r) \leftarrow$ call **ServerReEnc** ($c_i^*(r)$, K_{s_i}) {see Algorithm 4}
5: $\quad L_S \leftarrow L_S \cup c(r)$
6: **end for**
7: **return** (j, L_S)

Deployment of Role Assignment Policies: In order to assign roles to a Requester, an Admin User can deploy role assignment policies. For this purpose, an Admin User runs **RoleAssignment:ClientSide** illustrated in Algorithm 5. This algorithm takes as input a list of roles L to be assigned to Requester j, the client side key set K_{u_i} corresponding to Admin User i and the public parameters *params* and outputs the client encrypted role assignment list L_{C_i}. First, it creates and then initialises new list L_{C_i} (Line 1). For each role in L (Line 2), it generates client encrypted role by calling **ClientEnc** illustrated in Algorithm 3 (Line 3) and then it updates L_{C_i} by adding client encrypted role (Line 4). An Admin User sends the client encrypted role assignment list to the Administration Point. During the second round of encryption, the Administration Point runs **RoleAssignment:ServerSide** illustrated in Algorithm 6. This algorithm takes as input the client encrypted role assignment list L_{C_i} for Requester j and identity i of Admin User and ouputs the server encrypted role assignment list L_S. While running **RoleAssignment:ServerSide**, the Administration Point first retrieves the server side key K_{s_i} corresponding to Admin User i (Line 1). It creates and initialises new list L_S (Line 2). For each role in L_{C_i} (Line 3), it generates server encrypted role by calling **ServerReEnc** illustrated in Algorithm 4 (Line 4) and updates L_S by adding the server encrypted role (Line 5).

Algorithm 7 PermissionAssignment:ClientSide

Input: List of permissions L to be assigned to role r, the client side key set K_{u_i} corresponding to Admin User i and the public parameters *params*.
Output: The client encrypted permission assignment list L_{C_i} assigned to the client generated role $c_i^*(r)$.

1: $c_i^*(r) \leftarrow$ call **ClientEnc** (r, K_{u_i}, *params*)
2: $L_{C_i} \leftarrow \phi$
3: **for** each permission (*action*, *target*) in L **do**
4: $\quad c_i^*(action) \leftarrow$ call **ClientEnc** (*action*, K_{u_i}, *params*)
5: $\quad c_i^*(target) \leftarrow$ call **ClientEnc** (*target*, K_{u_i}, *params*)
6: $\quad L_{C_i} \leftarrow L_{C_i} \cup (c_i^*(action), c_i^*(target))$
7: **end for**
8: **return** ($c_i^*(r)$, L_{C_i})

Algorithm 8 PermissionAssignment:ServerSide

Input: The client encrypted permission assignment list L_{C_i} for client generated role $c_i^*(r)$ and identity i of Admin User.
Output: The server encrypted permission assignment list L_S and the server generated role $c(r)$.

1: $K_{s_i} \leftarrow KS[i]$ {retrieve the server side key corresponding to Admin User i}
2: $c(r) \leftarrow$ call **ServerReEnc** ($c_i^*(r)$, K_{s_i})
3: $L_S \leftarrow \phi$
4: **for** each client encrypted permission ($c_i^*(action), c_i^*(target)$) in list L_{C_i} **do**
5: $\quad c(action) \leftarrow$ call **ServerReEnc** ($c_i^*(action)$, K_{s_i})
6: $\quad c(target) \leftarrow$ call **ServerReEnc** ($c_i^*(target)$, K_{s_i})
7: $\quad L_S \leftarrow L_S \cup (c(action), c(target))$
8: **end for**
9: **return** ($c(r)$, L_S)

Deployment of Permission Assignment Policies: An Admin User can assign permissions to a role. In order to deploy policies regarding permissions assignment to roles, an Admin User runs Algorithm 7. This algorithm takes as input a list of permissions L to be assigned to role r, the client side key set K_{u_i} corresponding to Admin User i and the public parameters *params* and outputs the client encrypted permission assignment list L_{C_i} assigned to client generated role $c_i^*(r)$. First, it generates client encrypted role $c_i^*(r)$ by calling **ClientEnc** illustrated in Algorithm 3 (Line 1). Next, it creates and initialises new list L_{C_i} (Line 2). For each permission in L (Line 3), it generates the client encrypted action $c_i^*(action)$ (Line 4) and the client encrypted target $c_i^*(target)$ (Line 5) and updates L_{C_i} by adding the client encrypted permission (Line 6). An Admin User sends the client encrypted permission list along with the client encrypted role to the Administration Point. The Administration Point runs another round of encryption by running Algorithm 8. This algorithm takes as input the client encrypted permission assignment list L_{C_i} for client generated role $c_i^*(r)$ and identity i of Admin User and outputs the server encrypted permission assignment list L_S and the server generated role $c(r)$. First, it retrieves from the Key Store the server side key set K_{s_i} corresponding to Admin User i (Line 1). Next, it generates the server encrypted role by calling **ServerReEnc** illustrated in Algorithm 4 (Line 2). Then, it creates and initialises new list L_S (Line 3). For each client encrypted role in L_{C_i} (Line 4), it generates the server encrypted action (Line 5) and the server encrypted target (Line 6) and updates L_S by adding the server encryption permission

(Line 7).

Algorithm 9 ContextualConditionDeployment:ClientSide

Input: The contextual condition T, the client side key set K_{u_i} corresponding to Admin User i and the public parameters *params*.
Output: The client encrypted contextual condition T_{C_i}.

1: $T_{C_i} \leftarrow T$
2: **for** each leaf node e in T_{C_i} **do**
3: $c_i^*(e) \leftarrow$ call **ClientEnc** (r, K_{u_i}, *params*)
4: replace e of T_{C_i} with $c_i^*(e)$
5: **end for**
6: **return** T_{C_i}

Algorithm 10 ContextualConditionDeployment:ServerSide

Input: The client encrypted contextual condition T_{C_i} and identity of Admin User i.
Output: The server encrypted contextual condition T_S

1: $K_{s_i} \leftarrow KS[i]$ {retrieve the server side key corresponding to Admin User i}
2: $T_S \leftarrow T_{C_i}$
3: **for** each client encrypted leaf node $c_i^*(e)$ in T_S **do**
4: $c(e) \leftarrow$ call **ServerReEnc** ($c_i^*(e)$, K_{s_i})
5: replace $c_i^*(e)$ of T_S with $c(e)$
6: **end for**
7: **return** T_S

Deployment of Contextual Conditions: The contextual condition (part of role assignment and permission assignment policies) can be deployed in two steps. In the first step, an Admin User performs a first round of encryption by running Algorithm 9. This algorithm takes as input the contextual condition T, the client side key set K_{u_i} corresponding to Admin User i and the public parameters *params* and outputs the client encrypted contextual condition T_{C_i}. First, it copies T to T_{C_i} (Line 1). For each leaf node in T_{C_i} (Line 2), it generates the client encrypted element by calling **ClientEnc** illustrated in Algorithm 3 (Line 3) and then updates T_{C_i} by replacing element e with the client encrypted element $c_i^*(e)$ (Line 4). An Admin User sends the client encrypted contextual condition to the Administration Point. In the second step, the Administration Point performs another round of encryption by running Algorithm 10. This algorithm takes as input the client encrypted contextual condition T_{C_i} and identity of Admin User i and outputs the server encrypted contextual condition T_S. First, it retrieves from the Key Store the server side key K_{s_i} corresponding to Admin User i (Line 1). Next, it copies T_{C_i} to T_S (Line 2). For each each client encrypted leaf node in T_S (Line 3), it generates the server encrypted element by calling **ServerReEnc** illustrated in Algorithm 4 (Line 4). Then, it replaces the client encrypted element $c_i^*(e)$ of T_S with the server encrypted element $c(e)$ (Line 5).

Algorithm 11 RoleHierarchyDeployment:ClientSide

Input: The role hierarchy graph G, the client side key set K_{u_i} corresponding to Admin User i and the public parameters *params*.
Output: The client generated role hierarchy graph G_{C_i}.

1: $G_{C_i} \leftarrow G$
2: **for** each node r in G_{C_i} **do**
3: $c_i^*(r) \leftarrow$ call **ClientEnc** (r, K_{u_i}, *params*)
4: $td_i^*(r) \leftarrow$ call **ClientTD** (r, K_{u_i}, *params*) {see Algorithm 13}
5: replace r of G_{C_i} with $(c_i^*(r), td_i^*(r))$
6: **end for**
7: **return** G_{C_i}

Algorithm 12 RoleHierarchyDeployment:ServerSide

Input: The client generated role hierarchy graph G_{C_i} and identity of Admin User i.
Output: The server generated role hierarchy graph G_S

1: $K_{s_i} \leftarrow KS[i]$ {retrieve the server side key corresponding to Admin User i}
2: $G_S \leftarrow G_{C_i}$
3: **for** each client generated node $(c_i^*(r), td_i^*(r))$ in G_S **do**
4: $c(r) \leftarrow$ call **ServerReEnc** ($c_i^*(r)$, K_{s_i})
5: $td(r) \leftarrow$ call **ServerTD** ($td_i^*(r)$, K_{s_i}) {see Algorithm 14}
6: replace $(c_i^*(r), td_i^*(r))$ of G_S with $(c(r), td(r))$
7: **end for**
8: **return** G_S

Deployment of Role Hierarchy Graph: We know that a derived role inherits all permissions from its base role. In case if requested permissions are not assigned to the Requester's role, the PDP may need to traverse in the role hierarchy graph to find base roles corresponding to the Requester's role and then PDP verifies if any base role can fulfil requested permissions. For this purpose, the PDP needs a trapdoor of each base role so that it can match this trapdoor against roles in the Permission Repository. Therefore, a role hierarchy graph stores a role trapdoor along with each encrypted role. The deployment of role hierarchy graph takes place in two steps. In the first step, an Admin User runs Algorithm 11. This algorithm takes as input the role hierarchy graph G, the client side key set K_{u_i} corresponding to Admin User i and the public parameters *params* and outputs the client generated role hierarchy graph G_{C_i}. First, it copies G to G_{C_i} (Line 1). For each node r in G_{C_i} (Line 2), it generates the client encrypted role by calling **ClientEnc** illustrated in Algorithm 3 (Line 3) and the client trapdoor by calling **ClientTD** (Line 4) illustrated in Algorithm 13 that is explained later in this section. Next, it replaces r of G_{C_i} with the client encrypted role and the client generated trapdoor (Line 5). An Admin User sends the client generated role hierarchy graph to the Administration Point. In the second step, the Administration Point runs Algorithm 12. This algorithm takes as input the client generated role hierarchy graph G_{C_i} and identity of Admin User i and outputs the server generated role hierarchy graph G_S. First, it retrieves from the Key Store the server side key K_{s_i} corresponding to Admin User i (Line 1). Next, it copies G_{C_i} to G_S (Line 2). For each client generated node (Line 3), it generates the server encrypted role by calling **ServerReEnc** illustrated in Algorithm 4 (Line 4) and the server trapdoor by calling **ServerTD** (Line 5) illustrated in Algorithm 14 that is explained later in this section and then updates G_S by replacing the client generated node with the server generated node (Line 6).

Algorithm 13 ClientTD

Input: Element e, the client side key set K_{u_i} corresponding to user i and the public parameters *params*.
Output: The client generated trapdoor $td_i^*(e)$.

1: Choose a random $r_e \in \mathbb{Z}_q^*$
2: $\sigma_e \leftarrow f_s(e)$
3: $t_1 \leftarrow g^{-r_e} g^{\sigma_e}$
4: $t_2 \leftarrow h^{r_e} g^{-x_{i1} r_e} g^{x_{i1} \sigma_e} = g^{x_{i2} r_e} g^{x_{i1} \sigma_e}$
5: $td_i^*(e) \leftarrow (t_1, t_2)$
6: **return** $td_i^*(e)$

Algorithm 14 ServerTD

Input: The client generated trapdoor $td_i^*(e)$ and the server side key set K_{s_i} corresponding to user i.
Output: The server generated trapdoor $td(e)$.

1: $td(e) \leftarrow t_1^{x_{i2}}.t_2 = g^{x\sigma_e}$
2: **return** $td(e)$

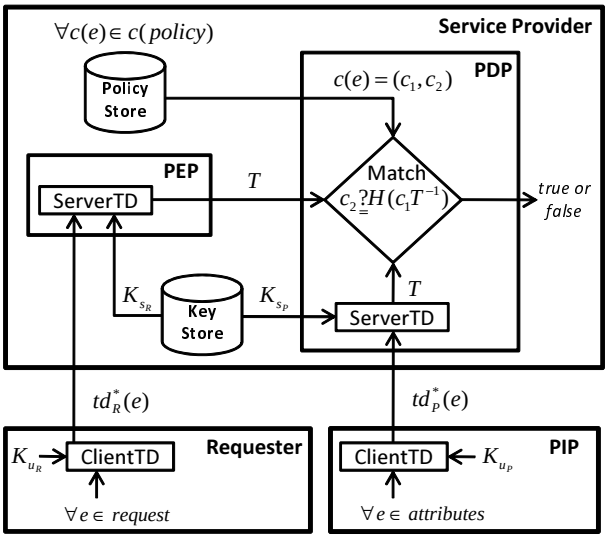


Figure 9: Policy evaluation phase

6.3. Policy Evaluation Phase

The policy evaluation phase is executed when a Requester makes a request either *ACT* or *REQ*. In this phase, a Requester sends client generated trapdoors (using Algorithm 13) of a request to the PEP. The PEP converts client generated trapdoors into server generated trapdoors (using Algorithm 14) and sends them to the PDP. The PDP matches server encrypted trapdoors of the request with server encrypted elements of the policy (using Algorithm 15). Optionally, the PDP may require contextual information in order to evaluate contextual conditions. The PIP sends client generated trapdoors of contextual information to the PDP. The PDP converts client generated trapdoors into server generated trapdoors and then evaluates contextual conditions based on contextual information. Finally, the PDP returns either *true* or *false* as shown in Figure 9. In the following, we describe how we generate trapdoors and perform the match.

For calculating client generated trapdoors of a request (or contextual information), a Requester (or the PIP) runs **ClientTD** illustrated in Algorithm 13. **ClientTD** takes as input each element e of the request, the client side key set K_{u_i} corresponding to user i and the public parameters *params* and outputs the client generated trapdoor $td_i^*(e)$. First, it choose randomly $r_e \in \mathbb{Z}_q^*$ (Line 1). Next, it calculates σ_e as $f_s(e)$ (Line 2). Then it calculates t_1 and t_2 as $g^{-r_e}g^{\sigma_e}$ (Line 3) and $h^{r_e}g^{-x_{i1}r_e}g^{x_{i1}\sigma_e} = g^{x_{i2}r_e}g^{x_{i1}\sigma_e}$ (Line 4), respectively. Both t_1 and t_2 form $td_i^*(e)$ (Line 5). A Requester sends client generated trapdoors of the request to the PEP. The PEP receives client generated trapdoors and runs **ServerTD** illustrated in Algorithm 14 for calculating server generated trapdoors. **ServerTD** takes as input the client generated trapdoor $td_i^*(e)$ and the server side key set K_{s_i} corresponding to user i and outputs the server generated trapdoor $td(e)$. It calculates $td(e)$ as $t_1^{x_{i2}}.t_2 = g^{x\sigma_e}$ (Line 1).

Algorithm 15 Match

Input: The server encrypted element $c(e) = (c_1, c_2)$ and the server generated trapdoor $td(e) = T$.
Output: *true* or *false*

1: **if** $c_2 \stackrel{?}{=} H(c_1.T^{-1})$ **then**
2: **return** *true*
3: **else**
4: **return** *false*
5: **end if**

In order to match a server encrypted element of a policy with a server generated trapdoor of a request, the PDP runs **Match** illustrated in Algorithm 15. **Match** takes as input the server encrypted element $c(e) = (c_1, c_2)$ and the server generated trapdoor $td(e) = T$ and returns either *true* or *false*. It checks the condition $c_2 \stackrel{?}{=} H(c_1.T^{-1})$ (Line 1). If the condition holds, it returns *true* (Line 2) indicating that the match is successful. Otherwise, it returns *false* (Line 4).

In the following, we describe how to evaluate (parts of) policies including role assignment, permission assignment, contextual conditions and role hierarchy graph. For the evaluation of each (part of) policy, we follow general strategy as already described in this section and also illustrated in Figure 9.

Algorithm 16 SearchRole

Input: The client generated trapdoor of role $td_i^*(r)$ and the server encrypted role assignment list (or list of active roles in session) L_S for Requester i
Output: *true* or *false*

1: $K_{s_i} \leftarrow KS[i]$ {retrieve the server side key corresponding to Requester i}
2: $td(r) \leftarrow$ call **ServerTD** $(td_i^*(r), K_{s_i})$
3: **for** each server encrypted role $c(r)$ in L_S **do**
4: $match \leftarrow$ call **Match** $(c(r), td(r))$ {see Algorithm 15}
5: **if** $match \stackrel{?}{=} true$ **then**
6: **return** *true*
7: **end if**
8: **end for**
9: **return** *false*

Searching a Role: A Requester can make a role activation request *ACT* and sends it to the SP. In order to grant *ACT*, the SP runs **SearchRole** illustrated in Algorithm 16. This algorithm takes as input the client generated trapdoor of role $td_i^*(r)$ and the server encrypted role assignment list L_S for Requester i. First, it retrieves from the Key Store the server side key K_{s_i} corresponding to Requester i (Line 1). Next, it calculates the server generated trapdoor $td(r)$ by calling Algorithm 14 (Line 2). For each server encrypted role $c(r)$ in L_S (Line 3), it performs matching against $td(r)$ by calling Algorithm 15 (Line 4). If any match is successful (Line 5), it returns *true* (Line 6), meaning that *ACT* is granted. Otherwise, it returns *false* (Line 9).

After *ACT* is granted, the PEP updates Session by adding in the Active Roles repository the server generated trapdoor of role. Once a Requester is active in a role, she can make an access request *REQ*. Before granting *REQ*, the SP checks if the Requester is already in the role in *REQ*. For this purpose, the SP runs Algorithm 16, where L_S shows a list of active roles in the session. Furthermore, the PDP also runs Algorithm 16 for searching the role in *REQ* in the Permission Repository with a

slight modification of ignoring the server trapdoor generation (in Line 2) as it is already generated when the role of *REQ* is searched in the session.

Algorithm 17 SearchPermission

Input: The client generated trapdoor of permission ($td_i^*(action)$, $td_i^*(target)$ and the server encrypted permission assignment list L_S for Requester i
Output: *true* or *false*

1: $K_{s_i} \leftarrow KS[i]$ {retrieve the server side key corresponding to Requester i}
2: $td(action) \leftarrow$ call **ServerTD** ($td_i^*(action)$, K_{s_i})
3: $td(target) \leftarrow$ call **ServerTD** ($td_i^*(target)$, K_{s_i})
4: **for** each server encrypted permission ($c(action), c(target)$) in L_S **do**
5: $match_{action} \leftarrow$ call **Match** ($c(action)$, $td(action)$)
6: $match_{target} \leftarrow$ call **Match** ($c(target)$, $td(target)$)
7: **if** $match_{action} \stackrel{?}{=} true$ and $match_{target} \stackrel{?}{=} true$ **then**
8: **return** *true*
9: **end if**
10: **end for**
11: **return** *false*

Searching a Permission: A Requester can send *REQ* for executing certain permissions. The PEP on the SP checks if the Requester is active in the role indicated in *REQ* and then the searches that role in the Permission Repository by running Algorithm 16. After a role is matched in the Permission Repository, the PEP searches the permission in *REQ* by running Algorithm 17. This algorithm takes as input the client generated trapdoor of permission ($td_i^*(action)$, $td_i^*(target)$ and the server encrypted permission assignment list L_S for Requester i and returns either *true* or *false*. First, it retrieves from the Key Store from the Key Store the server side key K_{s_i} corresponding to Requester i (Line 1). Next, it calculates server generated trapdoors of both action (Line 2) and target (Line 3) by calling Algorithm 14. For each server encrypted permission ($c(action), c(target)$) in L_S (Line 4), it matches the server encrypted action with the server generated action (Line 5) and the server encrypted target with the server generated taret (Line 6), respectively, by calling Algorithm 15. If both matches are successful (Line 7) for any permission ($c(action), c(target)$) in L_S, it returns *true* (Line 8). Otherwise, it returns *false* (Line 11).

Algorithm 18 ContextualConditionRequest

Input: List of attributes contextual attributes L, the client side key set K_{u_i} corresponding to Requester i and the public parameters *params*.
Output: The client generated list of trapdoors of contextual attributes L_{C_i}.

1: $L_{C_i} \leftarrow \phi$
2: **for** each attribute e in L **do**
3: $td_i^*(e) \leftarrow$ call **ClientTD** (r, K_{u_i}, *params*)
4: $L_{C_i} \leftarrow L_{C_i} \cup td_i^*(e)$
5: **end for**
6: **return** T_{C_i}

Generating Contextual Attributes: The PIP runs **ContextualAttributesRequest** illustrated in Algorithm 18 to calculate client generated trapdoors of contextual information. **ContextualAttributesRequest** takes as input a list of contextual attributes L, the client side key set K_{u_i} corresponding to Requester i and the public parameters *params* and outputs the client generated list of trapdoors of contextual attributes L_{C_i}. First, it creates and initialises new list L_{C_i} (Line 1). For each attribute e in L (Line 2), it calculates the client generated trapdoor $td_i^*(e)$ by calling Algorithm 13 (Line 3) and adds $td_i^*(e)$ in L_{C_i} (Line 4).

Algorithm 19 EvaluateTree

Input: Node n and tree T.
Output: *true* or *false*.

1: **if** $n.decision \neq null$ **then**
2: **return** $n.decision$
3: **end if**
4: **for** each child c of n in tree T **do**
5: call **EvaluateTree** (c, T) {recursive call}
6: **end for**
7: $t \leftarrow 0$
8: $m \leftarrow 0$
9: **for** each child c of n in tree T **do**
10: $t \leftarrow t + 1$
11: **if** $c.decision \stackrel{?}{=} true$ **then**
12: $m \leftarrow m + 1$
13: **end if**
14: **end for**
15: **if** ($n.gate \stackrel{?}{=} AND$ and $m \stackrel{?}{=} t$) or ($n.gate \stackrel{?}{=} OR$ and $m \geq 1$) **then**
16: $n.decision \leftarrow true$
17: **else**
18: $n.decision \leftarrow false$
19: **end if**
20: **return** $n.decision$

Evaluating Contextual Conditions: For evaluating any contextual condition, the PDP runs **ContextualConditionEvaluation** illustrated in Algorithm 20. This algorithm takes as input the client generated list of trapdoors of contextual attributes L_{C_i}, the server encrypted contextual condition T_S and identity of Requester i and returns either *true* or *false*. First, it retrieves from the Key Store the server side key K_{s_i} corresponding to Requester i (Line 1). Next, it creates and initialises a new list L_S (Line 2). For each client generated trapdoor $td_i^*(e)$ in L_{C_i} (Line 3), it calculates the server generated trapdoor $td(e)$ by calling Algorithm 14 (Line 4) and adds $td(e)$ in L_S (Line 5). Next, it copies T_S to *TREE* (Line 7) and adds decision field to each node in *TREE* (Line 8). For each node n in *TREE* (Line 9), it initialises $n.decision$ as *null* (Line 10). For each leaf node n in *TREE* (Line 12), it checks if any server generated trapdoor $td(e)$ in L_S (Line 13) matches with it by calling Algorithm 15 (Line 14). Next, it evaluates non-leaf nodes of *TREE* by running Algorithm 19 (Line 20). Finally, it returns either *true* or *false* depending upon the evaluation of *TREE* (Line 21).

Algorithm 20 ContextualConditionEvaluation

Input: The client generated list of trapdoors of contextual attributes L_{C_i}, the server encrypted contextual condition T_S and identity of Requester i.
Output: *true* or *false*

1: $K_{s_i} \leftarrow KS[i]$ {retrieve the server side key corresponding to Requester i}
2: $L_S \leftarrow \phi$
3: **for** each client generated trapdoor $td_i^*(e)$ in L_{C_i} **do**
4: $td(e) \leftarrow$ call **ServerTD** ($td_i^*(e)$, K_{s_i})
5: $L_S \leftarrow L_S \cup td_i^*(e)$
6: **end for**
7: $TREE \leftarrow T_S$
8: Add decision field to each node in *TREE*
9: **for** each node n in *TREE* **do**
10: $n.decision \leftarrow null$
11: **end for**
12: **for** each leaf node n in *TREE* **do**
13: **for** each server generated trapdoor $td(e)$ in L_S **do**
14: $n.decision \leftarrow$ call **Match** ($n.c(e)$, $td(e)$)
15: **if** $n.decision \stackrel{?}{=} true$ **then**
16: **return** *break*;
17: **end if**
18: **end for**
19: **end for**
20: call **EvaluateTree** ($TREE.root$, $TREE$) {see Algorithm 19}
21: **return** $TREE.root.decision$

EvaluateTree evaluates a tree containing AND and OR gates. It takes as input root node n and tree T and returns either *true* or *false*. First, it checks if the decision for n is already made (Line 1). If so, it returns the decision (Line 2). For each child c of n in tree T (Line 4), it recursively calls **EvaluateTree** (Line 5). Next, it creates and initialises t (Line 7) and m (Line 8) indicating total children of n and a count of matched children, respectively. For each child c of n in tree T (Line 9), it counts total children (Line 10) and matched children by checking made decisions (Line 12). Next, it checks if non-leaf node is AND and all children are matched or non-leaf node is OR and at least one child is matched (Line 15). If so, it is set as *true* (Line 16) and *false* (Line 18) otherwise.

Algorithm 21 SearchRoleHierarchyGraph

Input: The server generated trapdoor of role $td(r)$ and the server generated role hierarchy graph G_S
Output: *true* or *false*

1: **for** each server encrypted role $c(r)$ in G_S **do**
2: $match \leftarrow$ call **Match** $(c(r), td(r))$
3: **if** $match \stackrel{?}{=} true$ **then**
4: **return** *true*
5: **end if**
6: **end for**
7: **return** *false*

Searching Roles in Role Hierarchy Graph: The PDP may need to search base roles of one in *REQ* since a derived role inherits all permissions from its base role. The PDP runs **SearchRoleHierarchyGraph** illustrated in Algorithm 21 to find base roles from the encrypted role hierarchy graph. This algorithm takes as input the server generated trapdoor of role $td(r)$ and the server generated role hierarchy graph G_S and returns *true* if any base role is found and *false* otherwise. For each server encrypted role $c(r)$ in G_S (Line 1), it checks if $td(r)$ matches with any $c(r)$ by calling Algorithm 15 (Line 2). If any match is found (Line 3), it returns *true* (Line 4). Otherwise, it returns *false* (Line 7).

Algorithm 22 UserRevocation

Input: The user identity i.
Output: *true* or *false*.

1: **if** $exits(KS[i]) \stackrel{?}{=} false$ **then**
2: **return** *false*
3: **end if**
4: $K_{s_i} \leftarrow KS[i]$
5: $KS \leftarrow KS \backslash K_{s_i}$
6: **return** *true*

6.4. Revocation Phase

In this phase, the PEP can remove a compromised user from the system. In order to remove a user, the PEP runs **UserRevocation** illustrated in Algorithm 22. This algorithm takes as input the user identity i and returns either *true* (indicating that the user has been removed successfully) or false (indicating that the user does not exist in the system). First, it checks if the given user exists by checking the Key Store. If no, it returns *false* (Line 2). Otherwise, it retrieves from the Key Store the server side key set K_{s_i} corresponding to user i (Line 4), removes K_{s_i} from the Key Store (Line 5) and returns *true* (Line 6).

7. Security Analysis

In this section, we analyse the security of the policy deployment phase that includes Role Assignment (RA) encryption (Algorithms 5 and 6), Permission Assignment (PA) encryption (Algorithms 7 and 8), Contextual Condition (CC) encryption (Algorithms 9 and 10), and Role Hierarchy (RH) encryption (Algorithms 11 and 12). We then analyse the security of the policy evaluation phase that include Search Role (SR) (Algorithms 13 and 16), Search Permission (Algorithms 13 and 17), Contextual Condition Evaluation (Algorithms 18 and 20) and Search Role Hierarchy (Algorithms 13, 14 and 21).

We first define some basic concepts on which we build our security proofs.

7.1. Preliminaries

In general, a scheme is considered secure if no adversary can break the scheme with probability significantly greater than random guessing. The adversary's advantage in breaking the scheme should be a negligible function of the security parameter.

Definition 1 (Negligible Function)**.** *A function f is negligible if for each polynomial $p()$ there exists N such that for all integers $n > N$ it holds that $f(n) < \frac{1}{p(n)}$.*

We consider a realistic adversary that is computationally bounded and show that our scheme is secure against such an adversary. We model the adversary as a randomised algorithm that runs in polynomial time and show that the success probability of any such adversary is negligible. An algorithm that is randomised and runs in polynomial time is called a Probabilistic Polynomial Time (PPT) algorithm.

Our scheme relies on the existence of a pseudorandom function f. Intuitively, the output a pseudorandom function cannot be distinguished by a realistic adversary from that of a truly random function. Formally, a pseudorandom function is defined as:

Definition 2 (Pseudorandom Function)**.** *A function $f : \{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^*$ is pseudorandom if for all PPT adversaries $\mathcal{A}$, there exists a negligible function negl such that:*

$$|Pr[\mathcal{A}^{f_k(\cdot)} = 1] - Pr[\mathcal{A}^{F(\cdot)} = 1]| < negl(n)$$

where $k \rightarrow \{0,1\}^n$ is chosen uniformly randomly and F is a function chosen uniformly randomly from the set of function mapping n-bit strings to n-bit strings.

Our proof relies on the assumption that the Decisional Diffie-Hellman (DDH) is hard in a group $\mathbb{G}$, i.e., it is hard for an adversary to distinguish between group elements $g^{\alpha\beta}$ and g^{γ} given g^{α} and g^{β}.

Definition 3 (DDH Assumption)**.** *The DDH problem is hard regarding a group $\mathbb{G}$ if for all PPT adversaries $\mathcal{A}$, there exists a negligible function negl such that $|Pr[\mathcal{A}(\mathbb{G}, q, g, g^{\alpha}, g^{\beta}, g^{\alpha\beta}) = 1] - Pr[\mathcal{A}(\mathbb{G}, q, g, g^{\alpha}, g^{\beta}, g^{\gamma}) = 1]| < negl(k)$ where $\mathbb{G}$ is a cyclic group of order q ($|q| = k$) and g is a generator of $\mathbb{G}$, and $\alpha, \beta, \gamma \in \mathbb{Z}_q$ are uniformly randomly chosen.*

Encryption algorithms in policy deployment phase are based on **ClientEnc** and **ServerReEnc** functions that is equivalent to encrypting a single keyword in the SDE scheme [14]. Dong *et al.* [14] show that the single keyword encryption scheme is indistinguishable under chosen plaintext attack (*IND-CPA*). A cryptosystem is considered IND-CPA secure if no PPT adversary, given an encryption of a message randomly chosen from two plaintext messages chosen by the adversary, can identify the message choice with non-negligible probability. Dong *et al.* [14] prove the following theorem about the single Keyword Encryption (KE) scheme:

Theorem 1. *If the DDH problem is hard relative to $\mathbb{G}$, then the single keyword encryption scheme KE is IND-CPA secure against the server S, i.e., for all PPT adversaries $\mathcal{A}$ there exists a negligible function negl such that:*

$$Succ^{\mathcal{A}}_{KE,S}(k) = Pr\left[b' = b \;\middle|\; \begin{array}{c} (param, msk) \leftarrow Init(1^k) \\ (K_u, K_s) \leftarrow KeyGen(msk, U) \\ w_0, w_1 \leftarrow \mathcal{A}^{ClientEnc(K_u,\cdot)}(K_s) \\ b \overset{R}{\leftarrow} \{0,1\} \\ c^*_i(w_b) = ClientEnc(x_{i1}, w_b) \\ b' \leftarrow \mathcal{A}^{ClientEnc(K_u,\cdot)}(K_s, c^*_i(w_b)) \end{array} \right] < \tfrac{1}{2} + negl(k) \tag{1}$$

Proof. See Theorem 1 in [14].

7.2. Security of Encryption Algorithms in the Policy Deployment Phase

Using the fact that the *KE* scheme is IND-CPA secure, we show that the four encryption schemes: RA, PA, CC and RH are also IND-CPA against the server. We give the proof details for the Roles Assignment encryption scheme *RA*. We will show that the following theorem holds:

Theorem 2. *If the single keyword encryption KE scheme is IND-CPA secure against the server, then the RA encryption scheme RA is also IND-CPA, i.e., for all PPT adversaries $\mathcal{A}$, there exists a negligible function negl such that $Succ^{\mathcal{A}}_{RA,S}(k) < \frac{1}{2} + negl(k)$.*

Proof. We prove the theorem by showing that breaking the *RA* encryption reduces to breaking the *KE* encryption. We define the following game in which the adversary $\mathcal{A}$ challenges the game with two lists of roles L_0 and L_1 having the same number of roles t. We construct the following vector containing the encryption of roles from both lists: $\vec{C}^{(i)} = C(r^1_0), \ldots, C(r^i_0), C(r^{i+1}_1), \ldots, C(r^t_1)$. The success probability of the adversary in distinguishing the encryption of the two lists of roles is defined as:

$$Succ_{\mathcal{A}}(k) = \frac{1}{2} Pr[A(\vec{C}^0) = 0] + \frac{1}{2} Pr[A(\vec{C}^t) = 1] \tag{2}$$

In the following, we show that breaking the *RA* scheme reduces to breaking the *KE* game. In the *KE* game from [14], the adversary challenges the game with two keywords w_0 and w_1 and tries to distinguish between their encryptions. Let us consider a PPT adversary $\mathcal{A}'$ who attempts to challenge the single keyword encryption scheme *KE* using the corresponding *RA* adversary $\mathcal{A}$ as a sub-routine The game is the following:

- $\mathcal{A}'$ is given the parameters $(\mathbb{G}, q, g, h, H, f)$ as input and for each user i is given (i, x_{i2}).
- $\mathcal{A}'$ passes these parameters to $\mathcal{A}$.
- $\mathcal{A}$ generates two lists of roles L_0 and L_1 having the same number of roles t and gives them to $\mathcal{A}'$.
- $\mathcal{A}'$ chooses $i \overset{r}{\leftarrow} [1, t]$. It then uses r^i_0, r^i_1 to challenge the single keyword encryption *KE* game. The adversary gets back c^i_b as the result, where c^i_b is the encryption of either r^i_0 or r^i_1. $\mathcal{A}'$ uses this result to construct a hybrid vector $(c^1_0, \ldots, c^{i-1}_0, c^i_b, c^{i+1}_1, \ldots, c^t_1)$ and sends it to $\mathcal{A}$.
- $\mathcal{A}'$ outputs b', the bit output by $\mathcal{A}$.

$\mathcal{A}$ is required to distinguish $\vec{C}^{(i)}$ and $\vec{C}^{(i-1)}$ and the probability of $\mathcal{A}$'s success in distinguishing correctly is:

$$Succ^i_{\mathcal{A}}(k) = \frac{1}{2} Pr[A(\vec{C}^{(i)}) = 0] + \frac{1}{2} Pr[A(\vec{C}^{(i-1)}) = 1] \tag{3}$$

Because i is randomly chosen, it holds that:

$$\begin{aligned} Succ_{\mathcal{A}'}(k) &= \textstyle\sum_{i=1}^{t} Succ^i_{\mathcal{A}}(t) \cdot \frac{1}{t} \\ &= \tfrac{1}{2t} Pr[A(\vec{C}^0) = 0] + \textstyle\sum_{i=1}^{t-1} (Pr[A(\vec{C}^i) = 0] \\ &\quad + Pr[A(\vec{C}^i) = 1]) + \tfrac{1}{2} Pr[A(\vec{C}^t) = 1] \\ &= \tfrac{1}{t}(\tfrac{1}{2} Pr[A(\vec{C}^0) = 0] + \tfrac{1}{2} Pr[A(\vec{C}^t) = 1]) + \tfrac{t-1}{2t} \\ &= \tfrac{1}{t} Succ_{\mathcal{A}}(k) + \tfrac{t-1}{2t} \end{aligned} \tag{4}$$

Because the success probability of $\mathcal{A}'$ to break the single keyword encryption scheme is $Succ_{\mathcal{A}'}(k) < \frac{1}{2} + negl(k)$, it follows that $Succ_{\mathcal{A}}(k) < \frac{1}{2} + negl(k)$.

The proof for the other encryption schemes is similar and for lack of space we do not show all the details.

7.3. Security of Algorithms in the Policy Evaluation Phase

We now analyse the security of SR, Search Permission, Contextual Condition Evaluation and Search Role Hierarchy. These algorithms require the SP to take some client input (i.e., trapdoors computed using Algorithm 13), process it (i.e., re-encrypt it using Algorithm 14), and test whether it matches some information stored on the server. Though a single operation has been proved secure, we are interested in what these algorithms leak to the SP. We follow the concept of non-adaptive indistinguishability security introduced for encrypted databases by [10] and adapted by [14] in a multi-user setting. We show that given two non-adaptively generated histories with the same length and outcome, no PPT adversary can distinguish the histories based on what it can observe from the interaction. A history contains all the interactions between clients and the SP. Non-adaptive history means that the adversary cannot choose sequences of client inputs based on previous inputs and matching outcomes.

In the following, we show the details for the SR scheme. In this scheme, a history is defined as follows:

Definition 4 (SR History)**.** *An SR history $\mathcal{H}_i$ is an interaction between a SP and all clients that connect to it, over i role activation requests. $\mathcal{H}_i = (L_s^{u_1}, \ldots, L_s^{u_i}, r_1^{u_1}, \ldots, r_i^{u_i})$, where u_i represents an identifier of the client making the requests, $L_s^{u_i}$ represents the lists of roles for client u_i, and $r_i^{u_i}$ represents the request made by the client.*

We formalise the information leaked to a SP as a *trace*. We define two kinds of traces: the trace of a single request and the trace of a history. The trace of a request leaks to the SP which role in L_s^i matches the request and can be formally defined as: $tr(r) = \{td *_i (role), L_s^i, idx\}$, where idx is the index of the matched role, if any, in L_s^i.

We define the role matching pattern $\mathcal{P}$ over a history $\mathcal{H}_i$ to be a set of binary matrices (one for each client) with columns corresponding to encrypted roles in the list of the client, and rows corresponding to requests. $\mathcal{P}[j,k] = 1$ if request j matched the k's role and $\mathcal{P}[j,k] = 0$ otherwise.

The trace of a history includes the encrypted role assignment lists of all clients $L_s^{u_i}$ stored by the SP and which can change as new roles are added and clients leave of join the system, the trace of each request, and the role matching pattern $\mathcal{P}_i$ for each client.

During an interaction, the adversary cannot see directly the plaintext of the request, instead it sees the ciphertext. The view of a request is defined as:

Definition 5 (View of a Request)**.** *We define the view of a request $q_1^{u_1}$ under a key set K_{ui} as: $V_{K_{ui}}(q^{u_i}) = tr(q^{u_i})$*

Definition 6 (View of a History)**.** *We define the view of a history with i interactions $\mathcal{H}_i$ as $V_{K_u}(H_i) = (L_s^{u_1}, \ldots, L_s^{u_i}, V_{K_{ui}}(q_1^{u_i}), \ldots, V_{K_{ui}}(q_i^{u_i}))$.*

The security definition is based on the idea that the scheme is secure if nothing is leaked to the adversary beyond what the adversary can learn from traces.

We define the following game in which an adversary $\mathcal{A}$ generates two histories $\mathcal{H}_{i0}$ and $\mathcal{H}_{i1}$ with the same trace over i requests. Then the adversary is challenged to distinguish the views of the two histories. If the adversary succeeds with negligible probability, the scheme is secure.

Definition 7 (Non-adaptive indistinguishability against a curious SP)**.** *The SR scheme is secure in the sense of non-adaptive indistinguishability against a curious SP if for all $i \in \mathbb{N}$ and for all PPT adversaries $\mathcal{A}$ there exists a negligible function negl such that:*

$$Pr\left[b' = b \;\middle|\; \begin{array}{l} (params, msk) \leftarrow Init(1^k) \\ (K_u, K_s) \leftarrow KeyGen(msk, U) \\ \mathcal{H}_{io}, \mathcal{H}_{i1} \leftarrow \mathcal{A}(K_s) \\ b \xleftarrow{R} \{0,1\} \\ b' \leftarrow \mathcal{A}(K_s, V_{K_u}(\mathcal{H}_{ib})) \end{array} \right] < \frac{1}{2} + negl(k) \tag{5}$$

where U is a set of user IDs, K_u is the user side key sets, K_s are the server side key sets, $\mathcal{H}_{i1}$ and $\mathcal{H}_{i0}$ are two histories over i requests such that $Tr(\mathcal{H}_{i0}) = Tr(\mathcal{H}_{i1})$.

Theorem 3. *If the DDH problem in hard relative to $\mathbb{G}$, then the SR scheme is a non-adaptive indistinguishable secure scheme. The success probability of a PPT adversary $\mathcal{A}$ in breaking the SR scheme is defined as:*

$$\begin{aligned} Succ^{\mathcal{A}}(k) = & \tfrac{1}{2} Pr[\mathcal{A}(RA(\vec{L}_0), TD(\vec{r}_0)) = 0] + \\ & \tfrac{1}{2} Pr[\mathcal{A}(RA(\vec{L}_1), TD(\vec{r}_1)) = 1] \\ < & \tfrac{1}{2} + negl(k) \end{aligned} \tag{6}$$

*where $RA(\vec{L}_i)$ is the role encryption of the vector of lists of H_i, and $TD(\vec{r}_i)$ is the **ClientTD** of the roles in the requests of H_i.*

Proof. We consider an adversary $\mathcal{A}'$ that challenges the RE IND-CPA game using $\mathcal{A}$ as a sub-routine. $\mathcal{A}'$ does the following:

- $\mathcal{A}'$ receives public parameters *params* and the server side (i, x_{i2}) keys.
- To generate a view of a history $\mathcal{H}_i = (L_1^{u_1}, \ldots, L_i^{u_i}, q_1^{u_1}, \ldots, q_i^{u_i})$. $\mathcal{A}'$ performs the following steps:
  - For each role assignment list $L_j^{u_j}$, run Algorithm 5 to encrypt it as $RA(L_j^{u_j})$.
  - For each Search Role request $q_j^{u_j}$, run *ClientTD* to generate the trapdoor $TD(r)$ for the role.
- $\mathcal{A}$ outputs $\mathcal{H}_{i0}, \mathcal{H}_{i1}$. $\mathcal{A}'$ encrypts $\mathcal{H}_{i1}$ by itself and challenges the RE IND-CPA game with $\vec{L}_0$ and $\vec{L}_1$, the vectors of all roles lists in the two histories. It gets the result $RA(\vec{L}_b)$ where $b \xleftarrow{R} \{0,1\}$ and forms a view of a history $(RA(\vec{L}_b), TD(\vec{r_1}))$. It sends the view to $\mathcal{A}$.
- $\mathcal{A}$ tries to determine which vector was encrypted and outputs $b' \in \{0,1\}$.
- $\mathcal{A}'$ outputs b'.

Because the *RA* scheme is IND-CPA, it follows that:

$$\begin{aligned} \tfrac{1}{2} + negl(k) > & Succ_{RA}^{\mathcal{A}'}(k) \\ = & \tfrac{1}{2} Pr[\mathcal{A}((RA(\vec{L}_0), TD(\vec{r}_1))) = 0] + \\ & \tfrac{1}{2} Pr[\mathcal{A}((RA(\vec{L}_1), TD(\vec{r}_1))) = 1] \end{aligned} \tag{7}$$

Now let us consider another adversary $\mathcal{A}''$ who wants to distinguish the pseudorandom function f using $\mathcal{A}$ as a sub-routine. The adversary does the following:

- It generates $(\mathbb{G}, q, g, h, H)$ as public parameters, and sends them to $\mathcal{A}$ along with f. For each user i, it chooses randomly x_{i1}, x_{i2} such that $x_{i1} + x_{i2} = x$. It sends all (i, x_{i2}) to $\mathcal{A}$ and keeps all (i, x_{i1}, x_{i2}).
- $\mathcal{A}$ outputs $\mathcal{H}_{i0}, \mathcal{H}_{i1}$. $\mathcal{A}''$ encrypts all the roles lists in $\mathcal{H}_{i0}$ as $RA(\vec{L}_0)$. It chooses $b \xleftarrow{R} \{0,1\}$ and asks the oracle to encrypt all roles in $\mathcal{H}_{ib}$. It combines the results to form a view $(RA(\vec{L}_0), TD(\vec{r}_b))$ and returns it to $\mathcal{A}$.

- $\mathcal{A}$ outputs b'. $\mathcal{A}''$ outputs 1 if $b' = b$ and 0 otherwise.

There are two cases to consider: Case 1: the oracle in $\mathcal{A}''$s game is the pseudorandom function f, then:

$$\begin{aligned} Pr[\mathcal{A}''^{f_s(.)}(1^k) = 1] = \\ \frac{1}{2}Pr[\mathcal{A}(RA(\vec{L}_0), TD(\vec{r}_0)) = 0] + \\ \frac{1}{2}Pr[\mathcal{A}(RA(\vec{L}_0), TD(\vec{r}_1)) = 1] \end{aligned} \tag{8}$$

Case 2: the oracle in $\mathcal{A}''$s game is a random function F, then for each distinct role r, σ_r is completely random to $\mathcal{A}$. Moreover, we know the traces are identical, so $RA(\vec{L}_b)$ and $TD(\vec{r}_b)$ are completely random to $\mathcal{A}$. In this case:

$$Pr[\mathcal{A}''^{f_s(.)}(1^k) = 1] = \frac{1}{2} \tag{9}$$

Because f is a pseudorandom function, by definition it holds that:

$$\begin{aligned} |Pr[\mathcal{A}''^{f_s(.)}(1^k) = 1] - Pr[\mathcal{A}'^{f_s(.)}(1^k) = 1]| < negl(k) \\ Pr[\mathcal{A}''^{f_s(.)}(1^k) = 1] < \tfrac{1}{2} + negl(k) \end{aligned} \tag{10}$$

Sum up $Succ_{RE}^{\mathcal{A}'}(k)$ and $Pr[\mathcal{A}''^{f_s(.)}(1^k) = 1]$:

$$\begin{aligned} 1 + negl(k) > & \tfrac{1}{2}Pr[\mathcal{A}(RA(\vec{L}_0), TD(\vec{r}_0)) = 0] + \\ & \tfrac{1}{2}Pr[\mathcal{A}(RA(\vec{L}_0), TD(\vec{r}_1)) = 1] + \\ & \tfrac{1}{2}Pr[\mathcal{A}(RA(\vec{L}_0), TD(\vec{r}_1)) = 0] + \\ & \tfrac{1}{2}Pr[\mathcal{A}(RA(\vec{L}_1), TD(\vec{r}_1)) = 1] \\ = & \tfrac{1}{2}Pr[\mathcal{A}(RA(\vec{L}_0), TD(\vec{r}_0)) = 0] + \\ & \tfrac{1}{2} + \\ & \tfrac{1}{2}Pr[\mathcal{A}(RA(\vec{L}_1), TD(\vec{r}_1)) = 1] + \\ = & \tfrac{1}{2} + Succ^{\mathcal{A}}(k) \end{aligned} \tag{11}$$

Therefore $Succ^{\mathcal{A}}(k) < \frac{1}{2} + negl(k)$.

7.4. Revealing Policy Structure

The policy structure reveals information about the operators, such as AND and OR, and the number of operands used in the contextual condition. To overcome this problem, dummy attributes could be inserted in the tree representing contextual conditions. Similarly, the PIP can send dummy attributes to the PDP at the time of policy evaluation to obfuscate the number of attributes required for evaluating any contextual condition.

8. Performance Analysis

In this section, we discuss a quantitative analysis of the performance of $ESPOON_{ERBAC}$. It should be noticed that here we are concerned about quantifying the overhead introduced by the encryption operations performed both at the trusted environment and the outsourced environment. In the following discussion, we do not take into account the latency introduced by the network communication.

8.1. Implementation Details

We have implemented $ESPOON_{ERBAC}$ in Java 1.6. We have developed all the components of the architecture required for performing the policy deployment and policy evaluation phases. For the cryptographic operations, we have implemented all the functions presented in Section 6. We have tested the implementation of $ESPOON_{ERBAC}$ on a single node based on an Intel Core2 Duo 2.2 GHz processor with 2 GB of RAM, running Microsoft Windows XP Professional version 2002 Service Pack 3.

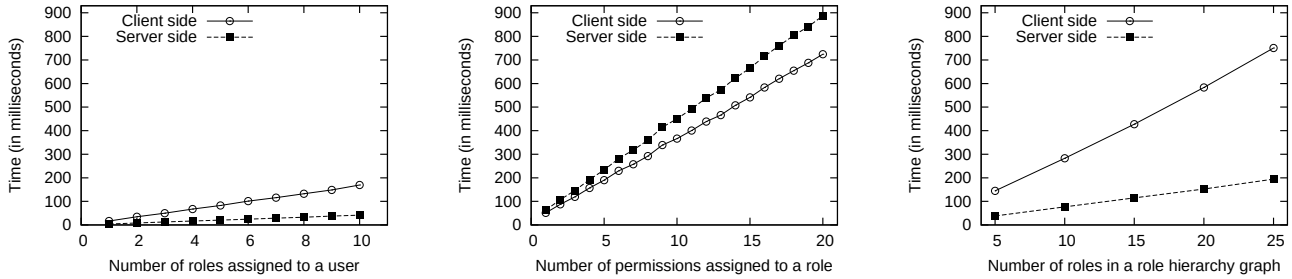
8.2. Performance Analysis of the Policy Deployment Phase

In this section, we analyse the performance of the policy deployment phase. In this phase, an Admin User encrypts policies and sends those encrypted policies to the Administration Point running in the outsourced environment. The Administration Point re-encrypts policies and stores them in the Policy Store in the outsourced environment. In the following, we analyse the performance of deploying (part of) policies including role assignment, permission assignment, contextual conditions and role hierarchy graph.

Role Assignment: In order to deploy a role assignment policy, an Admin User performs a first round of encryption on the client side (see Algorithm 5) and sends the client encrypted role assignment policy to the Administration Point. The Administration Point performs another round of encryption on the server side (see Algorithm 6) before storing role assignment policy in the Policy Store. Figure 10(a) shows performance overhead on the client side, as well as on the server side in order to deploy a role assignment policy. In this graph, we observe the performance by increasing number of roles in a role assignment policy. As we can expect, the performance overhead increases linearly with the linear increase in the number of roles in a role assignment policy. As we can notice, the graph grows linearly with the linear increase in the number of roles in the role assignment policy.

During the policy deployment phase, the encryption algorithm on the client side (Algorithm 3) takes more time that of the server side (Algorithm 4) as shown in Figure 10. The encryption algorithm on the client side takes more time because it performs more complex cryptographic operations such as random number generation and hash calculation as illustrated in Algorithm 3. However, any policy is deployed very rarely; whereas, it may be evaluated quite frequently. Therefore, the performance overhead of the policy evaluation phase (discussed in Section 8.3) is of great importance.

Permission Assignment: For deploying permissions to a role, an Admin User performs a first round of encryption on the client side (see Algorithm 7) and sends both the client encrypted role and client encrypted permissions to the Administration Point, where each permission contains both an action and a target. The Administration Point generates the server encrypted role and server encrypted permissions after performing a second round of encryption on the server side (see Algorithm 8). Figure 10(b) shows the performance overhead of deploying a permission assignment policy. This graph illustrates the



(a) (b) (c)

Figure 10: Performance overhead of deploying RBAC policies: (a) roles assigned to a user, (b) permissions to a role and (c) a role hierarchy graph

performance of deploying a permission assignment policy for a role with a number of permissions ranging from 1 to 20. As we can expect, the performance overhead increases linearly with the linear increase in the number of permissions in the permission assignment policy.

Contextual Conditions: Both role assignment and permission assignment policies include a contextual condition as we can see in Figure 2 and Figure 3, respectively. The contextual condition is represented as a tree structure as illustrated in Figure 4. During the policy deployment phase, an Admin User encrypts each leaf node of the tree (see Algorithm 9) while the Administration Point re-encrypts each leaf node (see Algorithm 10) and finally stores the tree in the Policy Store either in the Role Repository or the Permission Repository.

In the tree representing contextual conditions, leaf nodes represent string comparisons (for instance, *Location = Cardiology-ward*) and/or numerical comparisons (for instance, *AccessTime > 9*). A string comparison is always represented by a single leaf node while a numerical comparison may require more than one leaf nodes. In the worst case, a single numerical comparison, represented as s bits, may require s separate leaf nodes. Therefore, numerical comparisons have a major impact on the encryption of a policy at deployment time.

Figure 11(a) illustrates the performance overhead of deploying numerical and string comparisons. In this graph, we increase the number of string comparisons and numerical comparisons present in the contextual condition of a policy. As the graph, the time taken by deployment functions on the client side and the server side grow linearly with the number of comparisons in the contextual condition. The numerical comparisons have a stepper line because one numerical comparison of size s may be equivalent to s string comparisons in the worst case. For string comparisons, we have used "$attributeName_i$=$attributeValue_i$", where i varies from 1 to 10. For numerical comparisons, we have used "$attributeName_i$ < 15#4".[2]

To check how the size of the bit representation impacts on the encryption functions during the deployment phase, we have performed the following experiment. We fixed the number of numerical comparisons in the contextaul condition to only one and increased the size s of the bit representation from 2 to 20 for the comparison "$attributeName < 2^s - 1$. Figure 11(b) shows the performance overhead of the encryption during the policy deployment phase on the client side, as well as on the server side. We can see that the policy deployment time incurred grows linearly with the increase in the size s of a numerical attribute. In general, the time complexity of the encryption of the contextual conditions during the policy deployment phase is $O(m + ns)$ where m is the number of string comparisons, n is the number of numerical comparisons, and s represents the number of bits in each numerical comparison.

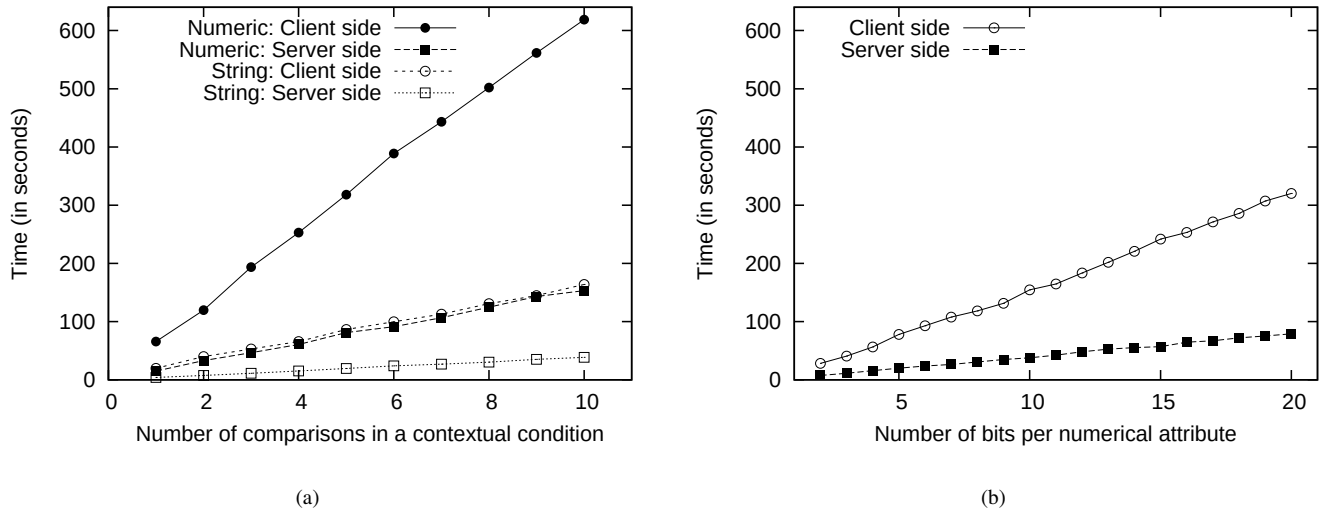
Role Hierarchy Graph: The PDP may search for a base role of the one in the access request *REQ* since a derived role inherits all permissions from its base role. For supporting this search, we deploy a role hierarchy graph. For deploying a role hierarchy graph, an Admin User performs the first round in order to generate the client encrypted trapdoor, as well as to calculate the client generated trapdoor of each role in the graph (see Algorithm 11). The Admin User sends the client generated role hierarchy graph to the Administration Point. The Administration Point performs the second round to generate the server encrypted trapdoor, as well as to calculate the server generated trapdoor of each role in the graph (see Algorithm 12). The PDP matches the trapdoor of role in *REQ* with the server encrypted role and if this match is successful, it finds trapdoors of the base roles. The trapdoors of base roles are required in order to perform search in the list of server encrypted roles in the Permission Repository.

In our experiment, we consider a role hierarchy graph in which each role R_i extends role R_{i+1} for all values of i from 0 to $n - 1$ where n indicates the total number of nodes and varies from 5 to 25. Figure 10(c) shows the performance overhead of encrypting a role hierarchy graph both on the client side and the server side. The graph grows linearly with the number of roles in a role hierarchy graph.

8.3. Performance Analysis of the Policy Evaluation Phase

In this section, we analyse the performance of the policy evaluation phase. In this phase, a Requester sends the encrypted request to the PEP running in the outsourced environment. The

[2]It should be noted that using the comparison less than 15 in a 4-bit representation represents the worst case scenario requiring 4 leaf nodes.



(a) (b)

Figure 11: Performance overhead of deploying contextual conditions: (a) numerical and string comparisons and (b) size of a numerical attribute

Table 1: Performance overhead of encrypting requests during the policy evaluation phase

Request Type	Time (in milliseconds)
ACT	16.353
REQ	47.069

PEP forwards the encrypted request to the PDP. The PDP has to select the set of policies that are applicable to the request. The PDP may require contextual information in order to evaluate the selected policies. In the following, we calculate the performance overhead of generating requests, search a role (in the Role Repository, in the Active Roles repository or in the Permission Repository), searching a permission, evaluating contextual conditions and searching a role in a role hierarchy graph.

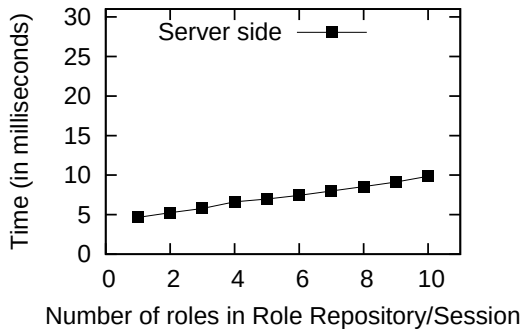
Generating Requests: A Requester may send the role activation request *ACT*. In order to generate *ACT*, a Requester calculates the client generated role (see Algorithm 13). This trapdoor generation of role takes 16.353 milliseconds as illustrated in Table 1. After a Requester is active in a role, she may make an access request *REQ* . A Requester has to calculate trapdoor for each element (including role, action and target) in *REQ*. The *REQ* generation takes 47.069 milliseconds as illustrated in Table 1. We can see that *REQ* generation takes 3 times of *ACT* generation because *REQ* has to calculate 3 trapdoors while *ACT* has to generate only a single trapdoor. The request generation does not depend on any parameters and can be considered constant.

Searching a Role in Role Repository/Session: In order to grant *ACT*, the PDP needs to search roles in the Role Repository. For searching a role, the PDP first calculates the server generated trapdoor of role in *ACT* and then matches this server encrypted trapdoor with server encrypted roles in the role assignment list as illustrated in Algorithm 16. Figure 12(a) shows the performance overhead (in the worst case) of performing this search. In this graph, we can observe that it grows linearly with increase in number of roles. As the graph indicates, the search function takes initial approximately 4 milliseconds to generate the server encrypted trapdoor of role in *ACT* while it takes approximately 0.6 milliseconds to perform encrypted match.

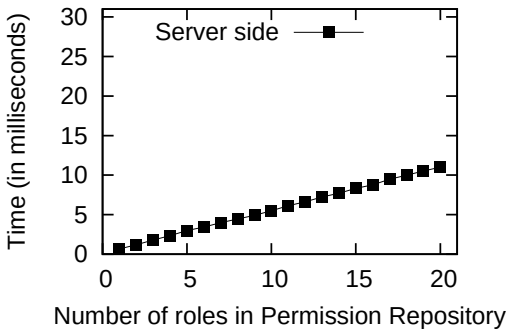
The PDP grants *ACT* by adding the server encrypted role of the Requester in the Active Roles repository of the Session. This implies that the Session maintains a list of active roles. Once a Requester makes an access request *REQ*, the PDP has to search in the Session if she is already active in role indicated in *REQ*. The performance overhead of searching a role in session is same as it incurs for searching a role in the Role Repository (shown in Figure 12(a)).

Searching a Role in Permission Repository: After finding the role of *REQ* in the list of active roles, the PDP has to search if the same role has the requested permission. For this purpose, the PDP has first to search the role of *REQ* in the Permission Repository and if any match is found, it has to search the requested permission in the list of permissions assigned to the found role. Figure 12(b) shows the performance overhead (in the worst case) of searching a role in the Permission Repository. The graph grows linearly with the increase in the number of roles in the Permission Repository. The PDP runs Algorithm 16 but with a slight modification of ignoring the server trapdoor generation (in Line 2) as it is already generated when the role of *REQ* is searched in the session. This is why, searching a role in the Permission Repository (as illustrated in Figure 12(b)) takes less time than searching a role in the Role Repository or Session (as illustrated in Figure 12(a)).

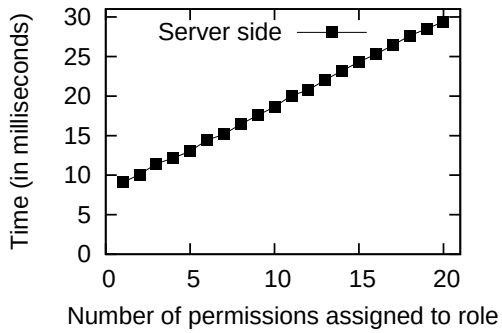
Searching a Permission: After a role is found in the Permission Repository, the PDP searches the requested permission in the list of permissions assigned to the found role (see Algorithm 17). Before searching the list of permissions, the PDP has to calculate server generated trapdoors of both the action and the target present in *REQ*. As we explained earlier, a single trapdoor generation on the server side takes approximately 4 milliseconds. The trapdoor generation of the requested permission, containing an action and a target, takes 8 milliseconds. Next,



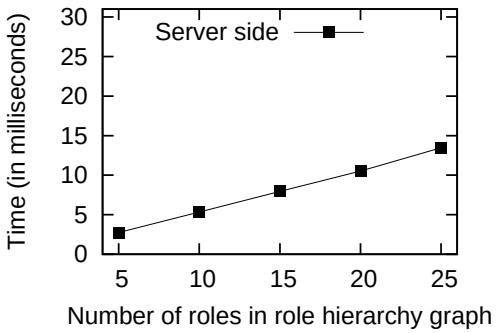
(a)



(b)

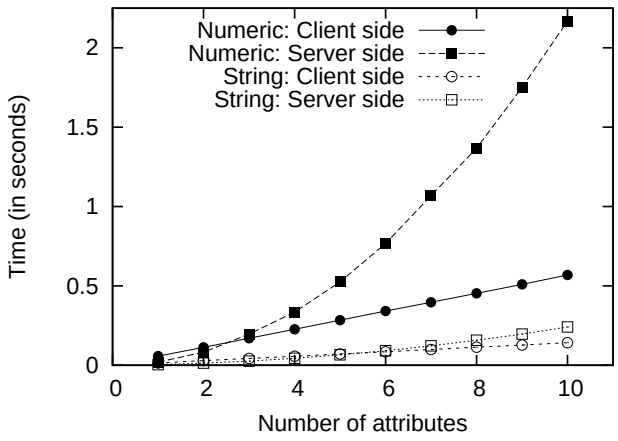


(c)

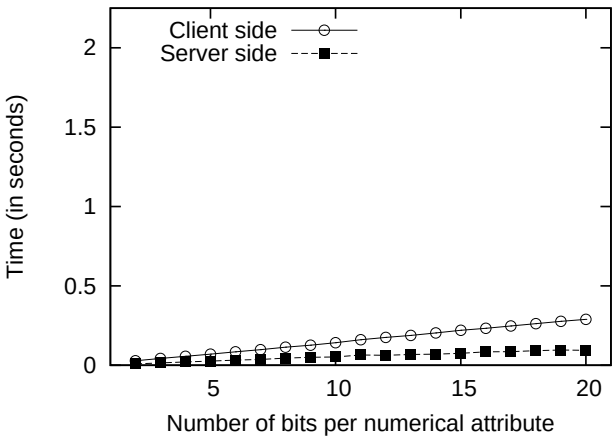


(d)

Figure 12: Performance overhead of evaluating RBAC policies



(a)



(b)

Figure 13: Performance overhead of evaluating contextual conditions

the PDP match (server generated trapdoors of) this requested permission with the list of (sever encrypted) permissions assigned to the found role. Figure 12(c) shows the performance overhead (in the worst case) of searching server generated trapdoor of permission with a list of server encrypted permissions. The graph grows linearly with the increase in the number of permissions in the list. For each permission match, the PDP performs (at most) two encrypted matches each incurring approximately 0.6 milliseconds.

Evaluating Contextual Conditions: For evaluating role assignment (illustrated in Figure 2) or permission assignment (illustrated in Figure 3) policies, the PDP may need to evaluate contextual conditions. For evaluating contextual conditions, the PDP needs to fetch contextual information from the PIP. The The PIP is responsible to collect and send the required contextual information that include information about the Requester (for instance, Requester's location or Requester's age) or the environment in which the request is made (for instance, time or temperature). The PIP transforms these attributes into trapdoors before sending to the PDP (as illustrated in Algorithm 18). For each single string attribute (for instance, *Location* := *Cardiology-ward*), the PIP generates a single trapdoor. For each numerical attribute of size s-bit (for instance, *AccessTime* =: *10#5*), the PIP generates s trapdoors. Figure 13(a) shows the performance overhead of generating trapdoors by the PIP on the client side for both numerical and string attributes. In our experiment, we vary number of attributes (both string and numeric) from 1 to 10. As we can see, the graph grows linearly with the increase in number of attributes. For numerical attributes, the curve of trapdoor generation on the client side is steeper than that of the string attributes because numerical attribute is of size s bits where s is set to 4. This means that each numerical attribute requires 4 trapdoors; on the other hand, a string attribute requires only a single attribute. We observe also the behaviour of generating client trapdoors for a numerical attribute of varying size. Figure 13(b) shows behaviour of generating on the client side trapdoors of a numerical attribute of varying size ranging from 2 to 20 bits. This graph grows linearly with the increase in number of bits, representing size of a numerical attribute.

After receiving trapdoors of contextual information, the PDP may evaluate a contextual condition. To evaluate the tree representing a contextual condition, the PDP matches contextual information against the leaf nodes in the tree, as illustrated in Algorithm 20. To quantify the performance overhead of this encrypted matching, we have performed the following test. First, we have considered two cases: the first case is the one in which the PIP provides only string attributes and the contextual condition contains only string comparisons; in the second, the PIP provides only numerical attributes and the contextual condition consists only of numerical comparisons. For both cases, the number of attributes varies together with the number of comparisons in the tree. In particular, if the PIP provides n different attributes then the contextual condition will contain n different comparisons.

Figure 13(a) shows also the performance overhead of evaluating string and numerical comparisons on the server side. As we can see, the condition evaluation for numerical attributes has a steeper curve. This can be explained as follows. For the first case, for each string attribute only a single trapdoor is generated. A string comparison is represented as a single leaf node in the tree representing a contextual condition. This means that n trapdoors in a request are matched against m leaf nodes in the tree resulting in a $O(nm)$ complexity (however, in our experiments the number of attributes and the number of comparisons are always the same). For the case of the numerical attributes, we have also to take in to consideration the bit representation. In particular, for a give numerical attribute represented as s bits, we need to generate s different trapdoors. This means that n numerical attributes in a request will be converted in to ns different trapdoors. These trapdoors then need to be matched against the leaf nodes representing the numerical comparisons. Figure 13(b) shows the performance overhead of evaluating a numerical comparison where the size of a numerical attribute varies from 2 to 20. As we have discussed for the policy deployment phase, in the worst case scenario, a numerical comparison for a s-bit numerical attribute requires s different leaf nodes. In a tree with m different numerical comparisons, this means that the ns trapdoors need to be matched against ms resulting in $O(nms^2)$ complexity.

Searching a Role Hierarchy Graph: The PDP may search a role in the role hierarchy graph. For performing this search, we consider a role hierarchy graph in which each role R_i extends role R_{i+1} for all values of i from 0 to $n-1$ where n indicates the total number of nodes and varies from 5 to 25. Figure 12(d) shows the performance overhead of searching a role in the role hierarchy graph deployed on the server side. As we can expect, the graph grows linearly with the number of roles in a role hierarchy graph.

Comparing $ESPOON_{ERBAC}$ with ESPOON: We compare the performance overheads of the policy evaluation of $ESPOON_{ERBAC}$ with that of *ESPOON* [1]. Before we show the comparison, we see how policies are expressed in both $ESPOON_{ERBAC}$ and *ESPOON*. The $ESPOON_{ERBAC}$ policies are explained in Section 4.2. The *ESPOON* policy is expressed as a $\langle S, A, T\rangle$ tuple with a *CONDITION*, meaning if *CONDITION* holds then subject S can take action A over target T. For comparing the performance overheads, we consider *ESPOON* policies with 50 unique subjects and each subject has 10 unique actions and targets where each $\langle S, A, T\rangle$ tuple's condition is the conjunction (AND) of the contextual condition illustrated in Figure 4 and *RequesterName=<NAME>*. That is, a subject can execute action over the target provided subject's name is equal to one specified in the condition, subject's location is cardiology-ward and time is between 9 AM and 5 PM. Similarly, we consider $ESPOON_{ERBAC}$ policies with 50 unique roles and each role has 10 unique permissions, where each user can get active in 5 roles. The introduction of RBAC simplifies the roles and permission management because we can enforce possible conditions at role activation time instead of enforcing them at the permission grant time. For instance, we can enforce location and time checks (i.e., the condition illustrated in Figure 4) at the role activation time while the condition *RequesterName=<NAME>* can be enforced at the permission

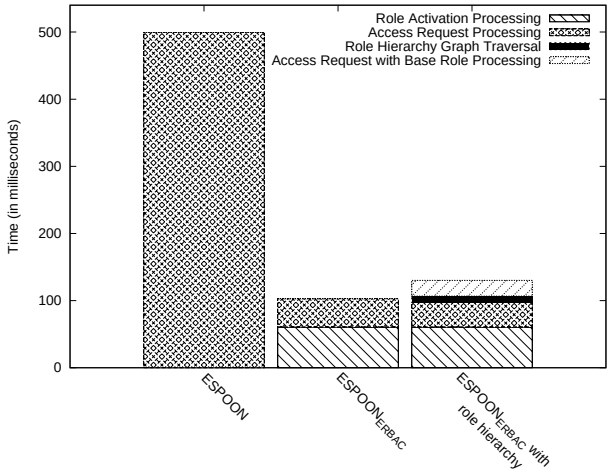


Figure 14: Performance comparison of *ESPOON* and *ESPOON*$_{ERBAC}$

grant time.

Figure 14 shows the performance overheads of evaluating *ESPOON* and *ESPOON*$_{ERBAC}$ policies. In *ESPOON*, a requester's subject is matched with one in the repository of 500 entries (i.e., 50 subjects each with 10 actions and targets). If there is any match, requester's action and target are matched and then condition is evaluated. In the worst case, in *ESPOON*, the access request processing can take approximately up to 500 milliseconds. On the other hand, in *ESPOON*$_{ERBAC}$, a requester first gets active in a role provided condition holds. The role activation can take approximately up to 60 milliseconds for a user that can get active in 5 roles. After the role activation, a requester can be granted permissions assigned to its role. However, first the active role is searched in the session and then the permission can be granted if the condition associated with that permission holds. As we can see in Figure 14, grating the permission takes up to 42 milliseconds. The reason why *ESPOON*$_{ERBAC}$ performance is better than that of *ESPOON* because (i) all possible conditions are enforced at the role activation time and (ii) introduction of roles simplified the roles and permissions management.

We also consider the effect of role hierarchies on the *ESPOON*$_{ERBAC}$ performance. In a role hierarchy, we assume that a role can inherit all permissions from its base role. This simplifies the role management and permission assignment to roles. In our experimentation, we consider 50 roles where each role has 5 permissions. Furthermore, there is a role hierarchy graph containing 25 roles, which is necessary for finding inheritance relationship between roles. Figure 14 shows a very slight performance gain to evaluate the access request in case of role hierarchy in *ESPOON*$_{ERBAC}$. Since the permission can be associated with base role, we need to traverse in the role hierarchy graph to find base roles. The performance of traversing in the role hierarchy graph is shown in Figure 14. Finally, the requested permission is granted if associated even with any base roles. The role hierarchy may improve performance but in the worst case it incurs higher overhead. However, the performance of *ESPOON*$_{ERBAC}$ with role hierarchy is still better than that of *ESPOON*.

9. Conclusions and Future Work

In this paper, we have presented the *ESPOON*$_{ERBAC}$ architecture to support RBAC policies for outsourced environments. Our approach separates the security policies from the actual enforcing mechanism while guaranteeing the confidentiality of RBAC policies assuming the SP is honest-but-curious. The main advantage of our approach is that RBAC policies are encrypted but it still allows the PDP to perform the policy evaluation without revealing contents of requests or policies. Second, *ESPOON*$_{ERBAC}$ is capable of handling complex contextual conditions involving non-monotonic boolean expressions and range queries. Finally, the authorised users do not share any encryption keys making the process of key management very scalable. Even if a user key is deleted or revoked, the other entities are still able to perform their operations without requiring re-encryption of RBAC policies.

As future directions of our research, we are working on integrating a secure audit mechanism in *ESPOON*$_{ERBAC}$. The mechanism should allow the SP to generate genuine audit logs without allowing the SP to get information about both the data and the policies. However, an auditing authority must be able to retrieve information about who accessed the data and what policy was enforced for any access request made. Another direction of our work is towards the extension of the encrypted search and match capabilities to handle the case of negative authorisation policies and policies for long-lived sessions where the conditions need to be continuously monitored and the attributes of the request can be dynamically updated.

Acknowledgment

The work of the first and third authors was supported by the EU FP7 research grant 257063 (project ENDORSE) while the work of the fourth author was supported by the Italian MIUR PRIN (project Autonomous Security).

REFERENCES

[1] Muhammad Rizwan Asghar, Mihaela Ion, Giovanni Russello, and Bruno Crispo. ESPOON: Enforcing Encrypted Security Policies in Outsourced Environments. In *The Sixth International Conference on Availability, Reliability and Security*, ARES'11, pages 99–108, August 2011.

[2] Muhammad Rizwan Asghar, Mihaela Ion, Giovanni Russello, and Bruno Crispo. ESPOON$_{ERBAC}$: Enforcing security policies in outsourced environments. *Elsevier Computers & Security (COSE)*, 35:2–24, 2013. Special Issue of the International Conference on Availability, Reliability and Security (ARES).

[3] Muhammad Rizwan Asghar, Giovanni Russello, and Bruno Crispo. Poster: *ESPOON*$_{ERBAC}$: Enforcing security policies in outsourced environments with encrypted rbac. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 841–844, New York, NY, USA, 2011. ACM.

[4] Mikhail J. Atallah, Marina Blanton, Nelly Fazio, and Keith B. Frikken. Dynamic and efficient key management for access hierarchies. *ACM Trans. Inf. Syst. Secur.*, 12:18:1–18:43, January 2009.

[5] Joonsang Baek, Reihaneh Safavi-Naini, and Willy Susilo. Public key encryption with keyword search revisited. In Osvaldo Gervasi, Beniamino Murgante, Antonio Lagan, David Taniar, Youngsong Mun, and Marina Gavrilova, editors, *Computational Science and Its Applications ICCSA 2008*, volume 5072 of *Lecture Notes in Computer Science*, pages 1249–1259. Springer Berlin / Heidelberg, 2008.

[6] J. Bethencourt, A. Sahai, and B. Waters. Ciphertext-policy attribute-based encryption. In *Security and Privacy, 2007. SP '07. IEEE Symposium on*, pages 321 –334, may 2007.

[7] Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. Public key encryption with keyword search. In Christian Cachin and Jan Camenisch, editors, *Advances in Cryptology - EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 506–522. Springer Berlin / Heidelberg, 2004.

[8] Dan Boneh and Brent Waters. Conjunctive, subset, and range queries on encrypted data. In Salil Vadhan, editor, *Theory of Cryptography*, volume 4392 of *Lecture Notes in Computer Science*, pages 535–554. Springer Berlin / Heidelberg, 2007.

[9] Robert W. Bradshaw, Jason E. Holt, and Kent E. Seamons. Concealing complex policies with hidden credentials. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 146–157, New York, NY, USA, 2004. ACM.

[10] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proceedings of the 13th ACM conference on Computer and communications security*, CCS '06, pages 79–88, New York, NY, USA, 2006. ACM.

[11] Sabrina De Capitani di Vimercati, Sara Foresti, Sushil Jajodia, Stefano Paraboschi, Gerardo Pelosi, and Pierangela Samarati. Preserving confidentiality of security policies in data outsourcing. In *Proceedings of the 7th ACM workshop on Privacy in the electronic society*, WPES '08, pages 75–84, New York, NY, USA, 2008. ACM.

[12] Sabrina De Capitani di Vimercati, Sara Foresti, Sushil Jajodia, Stefano Paraboschi, and Pierangela Samarati. A data outsourcing architecture combining cryptography and access control. In *Proceedings of the 2007 ACM workshop on Computer security architecture*, CSAW '07, pages 63–69, New York, NY, USA, 2007. ACM.

[13] Sabrina De Capitani di Vimercati, Sara Foresti, Sushil Jajodia, Stefano Paraboschi, and Pierangela Samarati. Over-encryption: management of access control evolution on outsourced data. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 123–134. VLDB Endowment, 2007.

[14] Changyu Dong, Giovanni Russello, and Naranker Dulay. Shared and searchable encrypted data for untrusted servers. *Journal of Computer Security*, 19(3):367–397, 2011.

[15] Philippe Golle, Jessica Staddon, and Brent Waters. Secure conjunctive keyword search over encrypted data. In Markus Jakobsson, Moti Yung, and Jianying Zhou, editors, *Applied Cryptography and Network Security*, volume 3089 of *Lecture Notes in Computer Science*, pages 31–45. Springer Berlin / Heidelberg, 2004.

[16] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *Proceedings of the 13th ACM conference on Computer and communications security*, CCS '06, pages 89–98, New York, NY, USA, 2006. ACM.

[17] Jason E. Holt, Robert W. Bradshaw, Kent E. Seamons, and Hilarie Orman. Hidden credentials. In *Proceedings of the 2003 ACM workshop on Privacy in the electronic society*, WPES '03, pages 1–8, New York, NY, USA, 2003. ACM.

[18] Yong Hwang and Pil Lee. Public key encryption with conjunctive keyword search and its extension to a multi-user system. In Tsuyoshi Takagi, Tatsuaki Okamoto, Eiji Okamoto, and Takeshi Okamoto, editors, *Pairing-Based Cryptography Pairing 2007*, volume 4575 of *Lecture Notes in Computer Science*, pages 2–22. Springer Berlin / Heidelberg, 2007.

[19] James B.D. Joshi, Elisa Bertino, Usman Latif, and Arif Ghafoor. A generalized temporal role-based access control model. *IEEE Transactions on Knowledge and Data Engineering*, 17:4–23, 2005.

[20] Seny Kamara and Kristin Lauter. Cryptographic cloud storage. In Radu Sion, Reza Curtmola, Sven Dietrich, Aggelos Kiayias, Josep Miret, Kazue Sako, and Francesc Seb, editors, *Financial Cryptography and Data Security*, volume 6054 of *Lecture Notes in Computer Science*, pages 136–149. Springer Berlin / Heidelberg, 2010.

[21] Young-Gab Kim and Jongin Lim. Dynamic activation of role on rbac for ubiquitous applications. In *Proceedings of the 2007 International Conference on Convergence Information Technology*, ICCIT '07, pages 1148–1153, Washington, DC, USA, 2007. IEEE Computer Society.

[22] Emil Lupu and Morris Sloman. Reconciling role based management and role based access control. In *Proceedings of the second ACM workshop on Role-based access control*, RBAC '97, pages 135–141, New York, NY, USA, 1997. ACM.

[23] Shivaramakrishnan Narayan, Martin Gagné, and Reihaneh Safavi-Naini. Privacy preserving ehr system using attribute-based infrastructure. In *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*, CCSW '10, pages 47–52, New York, NY, USA, 2010. ACM.

[24] Gustaf Neumann and Mark Strembeck. An approach to engineer and enforce context constraints in an rbac environment. In *Proceedings of the eighth ACM symposium on Access control models and technologies*, SACMAT '03, pages 65–79, New York, NY, USA, 2003. ACM.

[25] Alan C. O'Connor and Ross J. Loomis. Economic analysis of role-based access control. Technical report, National Institute of Standards and Technology, December 2010. Available at: `http://csrc.nist.gov/groups/SNS/rbac/documents/20101219_RBAC2_Final_Report.pdf`.

[26] K. Ondo and M. Smith. Outside it: the case for full it outsourcing. *Healthcare financial management : journal of the Healthcare Financial Management Association*, 60(2):92–98, 2006.

[27] Rafail Ostrovsky, Amit Sahai, and Brent Waters. Attribute-based encryption with non-monotonic access structures. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 195–203, New York, NY, USA, 2007. ACM.

[28] Hyun Sook Rhee, Jong Hwan Park, Willy Susilo, and Dong Hoon Lee. Trapdoor security in a searchable public-key encryption scheme with a designated tester. *Journal of Systems and Software*, 83(5):763 – 771, 2010.

[29] Giovanni Russello, Changyu Dong, and Naranker Dulay. Authorisation and conflict resolution for hierarchical domains. *Policies for Distributed Systems and Networks, IEEE International Workshop on*, 0:201–210, 2007.

[30] Amit Sahai and Brent Waters. Fuzzy identity-based encryption. In Ronald Cramer, editor, *Advances in Cryptology EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 557–557. Springer Berlin / Heidelberg, 2005.

[31] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *Computer*, 29:38–47, February 1996.

[32] Jun Shao, Zhenfu Cao, Xiaohui Liang, and Huang Lin. Proxy re-encryption with keyword search. *Information Sciences*, 180(13):2576 – 2587, 2010.

[33] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, SP '00, pages 44–55, Washington, DC, USA, 2000. IEEE Computer Society.

[34] Mark Strembeck and Gustaf Neumann. An integrated approach to engineer and enforce context constraints in rbac environments. *ACM Trans. Inf. Syst. Secur.*, 7:392–427, August 2004.

[35] Peishun Wang, Huaxiong Wang, and Josef Pieprzyk. Threshold privacy preserving keyword searches. In Viliam Geffert, Juhani Karhumki, Alberto Bertoni, Bart Preneel, Pavol Nvrat, and Mria Bielikov, editors, *SOFSEM 2008: Theory and Practice of Computer Science*, volume 4910 of *Lecture Notes in Computer Science*, pages 646–658. Springer Berlin / Heidelberg, 2008.

[36] R. Yavatkar, D. Pendarakis, and R. Guerin. Ietf rfc 2753: A framework for policy based admission control, January 2000. Available at: `http://docstore.mik.ua/rfc/rfc2753.html`.

Vitae

Muhammad Rizwan Asghar received his B.Sc. (Hons.) degree in Computer Science from University of the Punjab, Lahore, Pakistan, in 2006. In 2009, he obtained his M.Sc. degree in Information Security Technology from Eindhoven University of Technology, the Netherlands. He joined Create-Net (an international research center based in Trento, Italy) in 2010. Currently, he is a Ph.D. candidate at University of Trento, Italy. His research interests include access controls, applied cryptography, cloud computing, security and privacy.

Mihaela Ion received her B.Sc. in Information Technology and M.Sc. in Computer Science from International University in Germany. During her studies, she conducted various research projects with University of Marseille in France, SAP Waldorf and IBM Research Boeblingen in Germany. She joined CREATE-NET in 2007 where she's been working on various EU and Italian projects. Her research topics include data confidentiality in publish/subscribe systems, privacy for e-health applications, distributed identity and trust management. She is currently a Ph.D. candidate at the University of Trento working on security of publish/subscribe systems.

Giovanni Russello is a lecturer at the University of Auckland, New Zealand, and leads the Security technical group within the iNSPIRE area at CREATE-NET in Trento, Italy. Giovanni received his M.Sc. (summa cum laude) in Computer Science from University of Catania, Italy in 2000. In 2006, he obtained his Ph.D. from the Eindhoven University of Technology. After obtaining his Ph.D., Giovanni moved to the Policy Group in the Department of Computing at Imperial College London. Giovanni's research interests include policy-based security systems, privacy and confidentiality in cloud computing, smartphone security, and applied cryptography.

Bruno Crispo received his Ph.D. in Computer Science from University of Cambridge, UK in 1999, having awarded a M.Sc. in Computer Science from University of Turin, Italy in 1993. He is an associate professor at University of Trento since September 2005. Prior to that he was Associate Professor at Vrije Universiteit in Amsterdam, He is Co-Editor of the Security Protocol International Workshop proceedings since 1997. He is member of ACM and senior member of IEEE. His main research interests spans across the field of security and privacy. In particular his recent work focus on the topic of security protocols, access control in very large distributed systems, distributed policy enforcement, embedded devices and smartphone security and privacy and privacy-breaching malware detection. He has published more than 100 papers in international journals and conferences on security related topics.