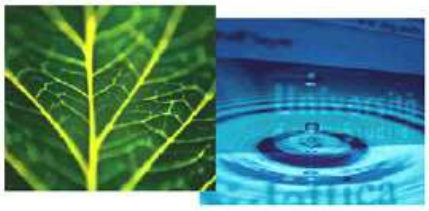
PhD Dissertation



International Doctoral School in
Information and Communication Technologies (ICT)
University of Trento, Italy

Privacy Preserving Enforcement of Sensitive Policies in Outsourced and Distributed Environments

Muhammad Rizwan Asghar

SUBMITTED TO THE DEPARTMENT OF
INFORMATION ENGINEERING AND COMPUTER SCIENCE (DISI)
IN THE PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Advisors:	Associate Prof. Dr. Bruno Crispo, University of Trento, Italy
	Dr. Giovanni Russello, The University of Auckland, New Zealand
Tutors:	Prof. Dr. Imrich Chlamtac, CREATE-NET and University of Trento, Italy
	Dr. Daniele Miorandi, CREATE-NET, Italy
Examiners:	Associate Prof. Dr. Alessandro Armando, FBK and University of Genova, Italy
	Dr. Ashish Gehani, SRI International, California, USA
	Prof. Dr. Pierangela Samarati, University of Milan, Italy

December 2013

To my family

Abstract

The enforcement of sensitive policies in untrusted environments is still an open challenge for policy-based systems. On the one hand, taking any appropriate security decision requires access to these policies. On the other hand, if such access is allowed in an untrusted environment then confidential information might be leaked by the policies. The key challenge is how to enforce sensitive policies and protect content in untrusted environments. In the context of untrusted environments, we mainly distinguish between outsourced and distributed environments. The most attractive paradigms concerning outsourced and distributed environments are cloud computing and opportunistic networks, respectively.

In this dissertation, we present the design, technical and implementation details of our proposed policy-based access control mechanisms for untrusted environments. First of all, we provide full confidentiality of access policies in outsourced environments, where service providers do not learn private information about policies during the policy deployment and evaluation phases. Our proposed architecture is such that we are able to support expressive policies and take into account contextual information before making any access decision. The system entities do not share any encryption keys and even if a user is deleted, the system is still able to perform its operations without requiring any action. For complex user management, we have implemented a policy-based Role-Based Access Control (RBAC) mechanism, where users are assigned roles, roles are assigned permissions and users execute permissions if their roles are active in the session maintained by service providers. Finally, we offer the full-fledged RBAC policies by incorporating role hierarchies and dynamic security constraints.

In opportunistic networks, we protect content by specifying expressive access control policies. In our proposed approach, brokers match subscriptions against policies associated with content without compromising privacy of subscribers. As a result, an unauthorised broker neither gains access to content nor learns policies and authorised nodes gain access only if they satisfy fine-grained policies specified by publishers. Our proposed system provides scalable key management in which loosely-coupled publishers and subscribers communicate without any prior contact. Finally, we have developed a prototype of the system that runs on real smartphones and analysed its performance.

Keywords: Policy Protection, Sensitive Policy Enforcement, Encrypted RBAC, Secure Opportunistic Networks, Encrypted CP-ABE Policies

Acknowledgements

It would not have been possible to write this dissertation without the support of several individuals. It gives me great pleasure to acknowledge all the people who helped me, in different ways, during the adventurous journey of my life.

First and foremost, I would like to extend my sincere gratitude to my Ph.D. advisor Associate Prof. Dr. Bruno Crispo. Next, I would like to sincerely thank my second Ph.D. advisor Dr. Giovanni Russello. Both of them introduced me to scientific research and also provided constant guidance and advice throughout my research work.

I would like to say thanks to my Ph.D. tutor Prof. Dr. Imrich Chlamtac. As a president of Create-Net, he also offered me a research position that funded me during the course of my Ph.D. Furthermore, I am thankful to my second Ph.D. tutor Dr. Daniele Miorandi who was head of the iNSPIRE area I was part of. He was always available to guide, encourage and support me during my stay at Create-Net.

I am grateful to the members of my Ph.D. assessment committee comprising of Dr. Ashish Gehani, Associate Prof. Dr. Alessandro Armando and Prof. Dr. Pierangela Samarati. Moreover, I am thankful to Dr. Ashish Gehani and SRI International for providing me opportunity to visit the Computer Science Laboratory in Menlo Park, California, USA.

I have been fortunate in making good friends in my academic life, too many to mention one by one. I am really thankful to all of my friends who joined me during coffee breaks, online conversations and social occasions (such as dinner parties and excursions), as well as all my colleagues at the University of Trento, Create-Net and SRI International. Specifically, I am grateful to those who reviewed and provided their input for improving the quality of my research work.

My doctoral studies were supported by EU FP7 COMPOSE (grant number 317862), EIT ICT Labs and EU FP7 ENDORSE (grant number 257063), which I gratefully acknowledge.

Above all, I am highly indebted to my family. My parents and brothers have given me their unequivocal support throughout my studies and they have always been by my side despite the distance. I would proudly mention my caring wife for her personal support and great patience at all times. I appreciate the spiritual support of my family. I love them very much.

Muhammad Rizwan Asghar
Trento, Italy
December 2013

Contents

List of Tables

List of Figures

List of Algorithms

List of Acronyms

ABE Attribute-Based Encryption.

ACL Access Control List.

AES Advanced Encryption Standard.

BDH Bilinear Diffie-Hellman.

BPM Business Process Management.

CC Contextual Condition.

CCE Contextual Condition Evaluation.

CDN Content Delivery Network.

CP-ABE Ciphertext-Policy Attribute-Based Encryption.

CW Chinese Wall.

DAC Discretionary Access Control.

DDH Decisional Diffie-Hellman.

DSoD Dynamic Separation of Duties.

DTN Delay Tolerant Network.

E-GRANT EnforcinG encRypted dynAmic security constraiNts in The cloud.

ECC Elliptic Curve Cryptography.

EHR Electronic Health Record.

ERBAC Encrypted Role-Based Access Control.

ERM Enterprise Resource Management.

ESPOON Enforcing Sensitive Policies in Outsourced envirOnmeNts.

ESPOON$_{ERBAC}$ Enforcing Sensitive Policies in Outsourced envirOnmeNts with Encrypted Role-Based Access Control.

HBDSoD History-Based Dynamic Separation of Duties.

IETF Internet Engineering Task Force.

IND-CPA INDistinguishable under Chosen Plaintext Attack.

IT Information Technology.

KB Kilo Byte.

KE Keyword Encryption.

KP-ABE Key-Policy Attribute Based Encryption.

MAC Mandatory Access Control.

ms milliseconds.

MSSE Multi-user Searchable Symmetric Encryption.

ObDSoD Object-Based Dynamic Separation of Duties.

OEM Outsourced Enforcement Module.

OpDSoD Operational Dynamic Separation of Duties.

PA Permission Assignment.

PBC Pairing-Based Cryptography.

PDP Policy Decision Point.

PEKS Public-key Encryption with Keyword Search.

PEP Policy Enforcement Point.

PIDGIN Privacy preserving Interest anD content sharinG in opportunIstic Networks.

PIP Policy Information Point.

PIR Private Information Retrieval.

PPT Probabilistic Polynomial Time.

PRES Proxy Re-Encryption with keyword Search.

QoS Quality of Service.

RA Role Assignment.

RBAC Role-Based Access Control.

RH Role Hierarchy.

SaaS Software-as-a-Service.

SDE Searchable Data Encryption.

SDSoD Simple Dynamic Separation of Duties.

SP Search Permission.

SR Search Role.

SRH Search Role Hierarchy.

TKMA Trusted Key Management Authority.

TTL Time To Live.

XACML eXtensible Access Control Markup Language.

XML eXtensible Markup Language.

Table of Notations

params	The public parameters
msk	The system wide master secret key
p and q	Two primes of size 1^k
$\mathbb{Z}_p^*$ and $\mathbb{Z}_q^*$	Cyclic groups
g	A generator
$\mathbb{G}$	A unique order subgroup of $\mathbb{Z}_p^*$
H	A collision-resistant hash function
f	A pseudorandom function
K_{u_i}	The client side key set
K_{s_i}	The server side key set
$c_i^*(e)$	The client encrypted element (by user i)
$c(e)$	The server encrypted element
$T_j^*(e)$	The client generated trapdoor (by user j)
$T^(e)$	The server generated trapdoor
CONDITION	The contextual condition that is represented as a tree
$\langle S, A, T \rangle$	A tuple representing the subject S can execute the action A on the target T
KS	Key Store

AT	Access Time
m, m_1 and m_2	Number of string attributes or Number of string comparisons in a contextual condition
n, n_1 and n_2	Number of numerical attributes or Number of numerical comparisons in a contextual condition
s	Size of a numerical attribute or Size of a numerical comparison
$ACT = (i, R)$	A role activation request that includes identity Requester i along with role R to be activated
$REQ = (R, A, T)$	An access request that includes role R a Requester is active in and action A to be taken over target T
L_r	List of roles
$\|L_r\|$	Number of roles in the list
L_p	List of permissions
$\|L_p\|$	Number of permissions in the list
G_{RH}	The role hierarchy graph
$\|G_{RH}\|$	Number of roles in the role hierarchy graph
Y	Number of actions in HBDSoD
Z	Number of domains in CW
c	Number of constraints
r	Number of records
A	A list of attributes
$\|A\|$	Number of attributes in the list
A_P^*	A list of attributes used to encrypt content
$\|A_P^*\|$	Number of attributes used to encrypt content

A_S^*	A list of attributes used to encrypt interest
$\|A_S^*\|$	Number of attributes used to encrypt interest
C	Content
$\|C\|$	Content size
I	A list of keywords a subscriber is interested in
$\|I\|$	Number of keywords a subscriber is interested in
T	A list of search tags associated with content
$\|T\|$	Number of search tags associated with content

Chapter 1

Introduction

The recent advancements in technology have changed the way how electronic data is stored and retrieved. Nowadays, individuals and enterprises are increasingly utilising remote services (such as Dropbox [1], Google Cloud Storage [2] and Amazon Simple Storage Service [3]), mainly for economical benefits. These services not only enable information sharing but also ensure availability of data from anywhere at any time. However, the growing use of remote services raises serious privacy issues by putting personal data at risk, particularly when the servers offering such services are untrusted. Unfortunately, servers get direct access to the data they store and process. For protecting sensitive data from servers in untrusted environments, data could be encrypted before leaving trusted boundaries. Regardless of whether the data is encrypted or not, the server will need to decide who will gain access to it. For regulating access to the data, access control policies could be specified. These are access control policies that will describe who can gain access to the data. State-of-the-art policy-based systems can ensure enforcement of these policies. However, the matter becomes complicated when sensitive policies, which may leak private information, have to be enforced in untrusted environments.

1.1 Motivation and Problem Statement

The enforcement of sensitive policies in untrusted environments is still an open challenge for policy-based systems. On the one hand, taking any appropriate security decision requires access to these policies. On the other hand, if such access is allowed in an untrusted environment then confidential information might be leaked by the policies. The key challenge is how to enforce sensitive policies and protect data in untrusted environments. This challenge arises from a fundamental question, i.e., *how can we establish trust in untrusted environments?* By establishing trust in untrusted environments, we will enable individuals and enterprises to leverage business models based on untrusted environments. At the

same time, we would be fostering trust of end-users by ensuring privacy and security of their personal data.

According to Gartner, the cloud-based security (including access management) services market will be worth $2.1 billion in 2013 and it will rise to $3.1 billion in 2015 [4]. This implies that security (access management in particular) of outsourced data is a key problem from a business analyst's point of view. It is important to know that outsourced environments are naturally untrusted. In the context of untrusted environments, we mainly distinguish two scenarios: (i) outsourced environments and (ii) distributed environments. The most attractive paradigms concerning outsourced and distributed environments are cloud computing and opportunistic networks, respectively.

1.1.1 Cloud Computing

Cloud computing is an emerging paradigm offering outsourced services to enterprises for storing and processing a huge amount of data at very competitive costs. It promises higher availability, scalability and more effective quality of service than in-house solutions. In cloud computing, the outsourced piece of data is within easy reach of cloud service providers. Unfortunately, one of the strong obstacles in widespread adoption of the cloud is to preserve confidentiality of the data [5]. There are several techniques that can guarantee confidentiality of data stored in outsourced environments while supporting basic search capabilities [6–15]. However, they do not support access control policies to regulate access to a particular subset of the stored data. State-of-the-art policy based mechanisms can work only when they are deployed and operated within a trusted domain [16]. In an untrusted environment, access policies may reveal sensitive information about the data they aim to protect.

To understand how access policies may reveal sensitive information in outsourced environments, let us imagine a scenario where a healthcare provider has outsourced its health record management services to a third party service provider. In this scenario, we do not trust the service provider to preserve data confidentiality. Therefore, we can encrypt health records before storing them in the outsourced environment. Furthermore, health records are associated with an access policy in order to prevent any unintended access. Let us consider the following access policy: *only a Cardiologist may access the health record*, which is attached to the health record. Even if the data is encrypted, a curious service provider may still infer private information about the patient's medical conditions. In the example policy, a curious service provider may easily deduce that the 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.

There are solutions that aim at providing the fine-grained access control on data stored

in outsourced environments [17–20]. However, those solutions are not suitable for scenarios where administrative actions are taken dynamically; this is because any administrative actions including updating access rights, adding users (or resources) and removing users (or resources) require re-distribution of new keys, as well as re-encryption of existing data with those keys. *The core research issue is to develop an efficient scheme with flexible key management that can enforce expressive access control policies in outsourced environments without revealing private information to service providers.*

1.1.2 Opportunistic Networks

Opportunistic networks are an emerging paradigm that has enabled individuals and enterprises to offer new services instantaneously. The fundamental reason behind this flexibility is that this paradigm aims at providing services without requiring any in-house Information Technology (IT) infrastructure [21]. Basically, opportunistic networks eliminate the need of any Internet connectivity.

In opportunistic networks, nodes can publish their own content and subscribe to others' content by indicating their interest. Any node can also act as a broker (also called a relay) that opportunistically receives content and interest, matches them and possibly delivers that content to other nodes. These opportunistic networks could be applied to the exchange of information in a wide range of domains from social media to military applications. Like cloud service providers, unauthorised brokers in opportunistic networks may infer private information from cleartext policies even when contents are encrypted.

Let us consider a battlefield scenario where soldiers are interested in sharing or acquiring sensitive information. We assume that there is no Internet connectivity in the battlefield. However, soldiers can exchange information via the short-range communication offered by smartphones. Soldiers can publish their content and subscribe for content of their interest. There are soldiers, known as brokers, who help to exchange content from one place to another. However, those soldiers must not be able to get access to content. For regulating access to content, a soldier, who is publishing, can encrypt content using state-of-the-art encryption techniques and specify an access policy describing which group of soldiers can get access. For instance, the policy could be *either a Soldier from the Infantry unit or a Major can get access.* Although the content is encrypted, soldiers serving as brokers and attackers (enemy having access to smartphones of brokers), may infer private information from cleartext policies, i.e., who will receive this content. Furthermore, subscription information (containing interest of subscribers) might compromise privacy of subscribers.

There are schemes that preserve predicate privacy [22, 23] and assume that the predicate is evaluated at the receiver's end. Shikfa *et al.* [24] propose a method that pro-

vides privacy and confidentiality in context-based forwarding. However, their proposed scheme disseminates information in one direction, i.e., from publishers to subscribers, without taking into account whether a subscriber is interested or not. In the context of publish-subscribe systems, there are many solutions that address privacy and security issues [25–27]. However, state-of-the-art techniques are mainly based on centralised solutions that cannot be applied to opportunistic networks, where each node may serve as a publisher, a broker and a subscriber. *The challenging research problem is to enable exchange of content and interest without (i) revealing content and its associated policies to unauthorised brokers and (ii) compromising the privacy of subscribers in opportunistic networks.*

1.2 Research Contributions

In this dissertation, we present the design, technical and implementation details of our proposed policy-based access control mechanisms for untrusted environments. In this section, we first discuss our research contributions in outsourced environments followed by advancements in opportunistic networks.

1.2.1 Enforcement of Encrypted Policies in Outsourced Environments

One of the main research goals is to enforce access control decisions while protecting access policies in outsourced environments. The core contributions concerning this part are as follows:

- We provide full confidentiality of access policies such that service providers in outsourced environments do not learn private information about policies during the policy deployment and evaluation phases.
- We support expressive access control policies, consider contextual conditions and take into account contextual information before making any access decision. In particular, our proposed solution is capable of handling complex policies involving non-monotonic boolean expressions and range queries.
- 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 data or access policies.
- For complex user management, we extend the basic policy enforcement mechanism to introduce the basic RBAC policies, where users are assigned roles, roles are assigned

permissions and users execute permissions if their roles are active in the session maintained by the service provider.

- The basic RBAC policies are augmented with role hierarchies by enabling role inheritance.
- Finally, we integrate dynamic security constraints (including Dynamic Separation of Duties and Chinese Wall) to provide the full-fledged RBAC policies in an outsourced environment. The full-fledged RBAC policies are enforced without revealing any private information to a curious service provider.

1.2.2 Enforcement of Encrypted Policies in Opportunistic Networks

The second research goal, which is even more challenging, is to propose a scheme that can enable exchange of content and interest without (i) revealing content and its associated policies to unauthorised brokers and (ii) compromising the privacy of subscribers. In the following, we describe main contributions related to the aforementioned goal:

- We protect content by specifying access control policies. In opportunistic networks, brokers match subscriber's interest against policies associated with content without compromising the subscriber's privacy (say, by learning their interest or attributes).
- In our proposed solution, an unauthorised broker neither gains access to content nor learns access policies and authorised nodes gain access only if they satisfy fine-grained policies specified by the publishers.
- The system provides scalable key management in which loosely-coupled publishers and subscribers communicate with each other without any prior contact.
- Finally, we have developed and analysed the performance of a prototype running on real smartphones in order to show the feasibility of our approach.

1.3 Organisation of the Dissertation

This dissertation consists of the follows chapters:

Chapter 2 proposes a policy-based access control mechanism that can deploy and enforce sensitive policies in an encrypted manner. The proposed mechanism maintains a clear separation between the security policies and the actual enforcement mechanism without loss of confidentiality. Moreover, we show performance overheads of the proposed algorithms.

Chapter 3 extends the proposed solution in Chapter 2 for supporting the basic RBAC policies. In this chapter, we also explain how the basic RBAC policies can incorporate role hierarchies. Furthermore, we provide a security analysis. Finally, we compare performance overheads incurred by access control mechanisms with and without RBAC models.

Chapter 4 explains how dynamic security policies (including Dynamic Separation of Duties and Chinese Wall) can be enforced and integrated with RBAC models. This chapter also shows performance overheads of the proposed algorithms.

Chapter 5 investigates how content could be encrypted and access control policies could be enforced in distributed environments, in particular in opportunistic networks. We propose a design and implement a scheme that can run on smartphones. Furthermore, we report some benchmarks of running the proposed cryptographic operations on smartphones.

Chapter 6 concludes the dissertation by summarising the chapters presented. It also points out some future research directions emerging from this work.

Appendix A reports a list of publications (with the corresponding abstracts) related to the work presented in this dissertation, as well as other publications.

Chapter 2

ESPOON: Enforcing Encrypted Security Policies in Outsourced Environments*

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.

The enforcement of sensitive policies in outsourced environments is still an open challenge for policy-based systems. On the one hand, taking the appropriate security decision requires access to the policies. However, if such access is allowed in an untrusted environment then confidential information might be leaked by the policies. Current solutions are based on cryptographic operations that embed security policies with the security mechanism. Therefore, the enforcement of such policies is performed by allowing the authorised parties to access the appropriate keys. We believe that such solutions are too rigid because they strictly intertwine authorisation policies with the enforcement mechanism. In this chapter, we address the issue of enforcing security policies in an outsourced environment while protecting the policy confidentiality. Our solution aims at providing a clear separation between security policies and the enforcement mechanism. The proposed technique does not reveal access policies and the access request.

*The preliminary version of this chapter has appeared in [28].

2.1 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 [29].

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 is 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 [30, 31] offer a protected data storage while supporting basic search capabilities performed on the server without revealing information about the stored data [6–15]. However, such solutions do not support access policies to regulate the access to a particular subset of the stored data.

2.1.1 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 [32, 33]. 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 [34], 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 [20]), i.e., it is honest to perform the required operations as described in the protocol but curious to learn information about stored or 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.

2.1.2 Research Contributions

In this chapter, we present a policy-based access control mechanism for outsourced environments where we support full confidentiality of access policies. We named our solution **Enforcing Sensitive Policies in Outsourced envirOnmeNts (ESPOON)**. One of the main advantages of ESPOON is that we maintain the clear separation between the security policies and the actual enforcement mechanism without loss of confidentiality. This can be guaranteed under the assumption that the service provider is honest-but-curious. Our approach allows us to implement the access control mechanism as an outsourced service with all the benefits associated with this business model without compromising the confidentiality of the policies. Summarising, the research contributions of our approach are threefold. First of all, the service provider does not learn private information about policies and the requester's attributes during the policy evaluation process. Second, ESPOON is capable of handling complex 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 the policies. As a proof-of-concept, we have implemented a prototype of our access control mechanism and analysed its performance to quantify the incurred overhead.

2.1.3 Chapter Outline

The rest of this chapter is organised as follows. In Section 2.2, we review the related work. Section 2.3 describes the proposed approach. Solution and algorithmic details are explained in Section 2.4 and Section 2.5, respectively. The performance overhead of the proposed solution is reported in Section 2.6. A discussion is provided in Section 2.7. Finally, we summarise this chapter in Section 2.8.

2.2 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 [30, 35–44]. 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 [6–15].

The idea of using an access control mechanism in an outsourced environment was initially explored in [19, 20, 33]. In this approach, De Capitani di 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 [32] De Capitani di 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 [45]) 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.* [46] employ the variant of Attribute-Based Encryption (ABE) proposed

in [47] (i.e., Ciphertext-Policy Attribute-Based Encryption (CP-ABE)) 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 [47–50]. Therefore, this mechanism is not suitable for guaranteeing confidentiality of access control policies in outsourced environments.

Related to the issue of the confidentiality of the access structure, the hidden credentials scheme presented in [51] 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 fulfil the access policy unless she possesses them. Bradshaw *et al.* [52] 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. Furthermore, hidden credentials schemes assume that the involved parties are online all the time to run the protocol.

The homomorphic encryption schemes [53–59] allow untrusted parties to perform mathematical operations on encrypted data without compromising the encryption. There are a number of issues with these schemes. The major issue is scalability. Unfortunately, state-of-the-art schemes are not suitable in practice for processing a huge amount of data due to computational limitations. Another problem is the key management. These schemes consider a single user that can perform the decryption. Basically, we are interested in schemes that can offer encryption and decryption in a multi-user setting, where each user should have her private key (i.e., different from other users).

The data could be distributed along with the sticky policy attached to it [60–62]. The data is basically encrypted with the sticky policy. For getting access to the data, the recipient needs to contact trusted authorities. The trusted authority grants access to the data by forwarding the decryption key to the recipient. Before sending the decryption key, the trusted authority verifies credentials of the recipient. Furthermore, this approach enables the trusted authority to take into account consent of the data owner before granting the access. However, approaches based on the sticky policies are not privacy preserving

because both policies and credentials are in cleartext.

Private Information Retrieval (PIR) protocols allow users to retrieve information without revealing queries to the server [63–70]. Basically, they can be deployed for fetching information from curious servers without compromising privacy of users, though they are computationally intensive. However, it is not clear how PIR protocols can help in a situation where the policy enforcement mechanism is delegated to third parties.

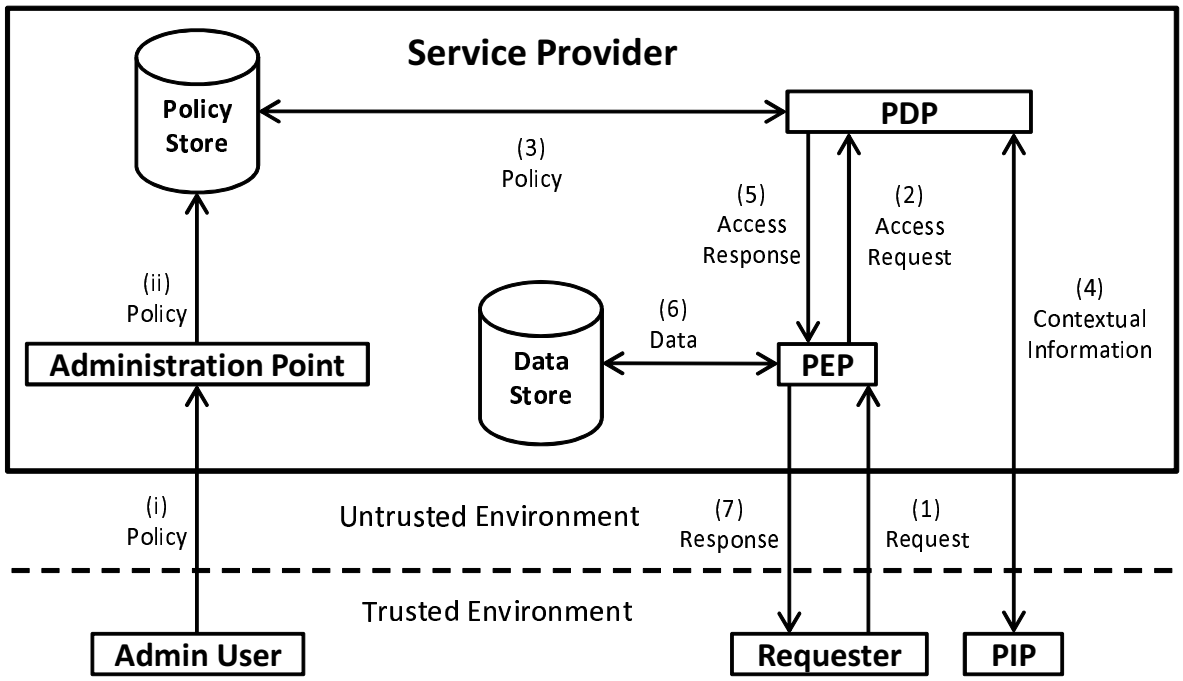


Figure 2.1: The ESPOON architecture for enforcing policies in outsourced environments

2.3 The ESPOON Approach

We propose Enforcing Sensitive Policies in Outsourced envirOnmeNts (ESPOON) that aims at providing a policy-based access control mechanism that can be deployed in an outsourced environment. Figure 2.1 illustrates the proposed architecture that has similar components as the widely accepted architecture for policy-based management proposed by Internet Engineering Task Force (IETF) [71]. In ESPOON, the **Admin User** deploys (i) access policies to the **Administration Point** that stores (ii) the policies in the **Policy Store**. Whenever a **Requester**, say a doctor, needs to access the data, a request is sent to the **Policy Enforcement Point (PEP)** (1). This request includes the Requester's identifier (subject), the requested data (target) and the action to be performed. The PEP

(2) forwards the access request to the **Policy Decision Point (PDP)**. The PDP (3) obtains the policies matching against the access request from the Policy Store and (4) retrieves the contextual information from the **Policy Information Point (PIP)**. The contextual information may include the environmental and Requester's attributes under which an access can be considered valid. For instance, a doctor should only access the data during office hours. For simplicity, we assume that the PIP collects all required attributes including the Requester's 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, then the PIP can request a trusted authority to sign the attributes before sending them to the PDP. The PDP evaluates the policies against the attributes provided by the PIP checking if the contextual information satisfies any policy conditions and sends to the PEP the access response (5). In case of *permit*, the PEP forwards the access action to the Data Store (6). 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 architecture for the policy-based access control is outsourced in an untrusted environment (see Figure 2.1). 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. Additionally, ESPOON guarantees that the confidentiality of the policies is protected while their evaluation is executed in the outsourced environment. This allows a more efficient evaluation of the 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 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 the confidentiality data is protected by one of the several techniques available for outsourced environments (see [30, 43, 44, 72]). However, to the best of our knowledge, there is no solution that can address the problem of guaranteeing policy confidentiality while allowing an efficient evaluation mechanism that is clearly separated from the security policies. Most of the techniques discussed in the related work section require the security mechanism to be tightly coupled 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.

2.3.1 The System Model

Before presenting the details of the scheme used in ESPOON, 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 the policies stored in the outsourced environment. An Admin User can deploy new policies or update/delete the policies already deployed.

- **Requester:** A Requester is a user that requests an access (e.g., read, write, search, etc.) over the data residing in the outsourced environment. Before the access is permitted, the policies deployed in the outsourced environment are evaluated.

- **Service Provider:** The Service Provider is responsible for managing the outsourced computation environment, where the ESPOON components are deployed and to store the data, and access policies. It is assumed the Service Provider is honest-but-curious, 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 (both the Admin User and Requester), the TKMA generates a key pair and securely transmits one part of the generated key pair to the user and the other to the Service Provider. The TKMA is deployed on the trusted environment. Although requiring a TKMA seems at odds with the needs of outsourced 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 time.

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

2.3.2 Representation of Policies

In this section, we provide an informal description of the policy representation used in our approach. In this chapter, we deal with only positive authorisation policies. This means that, as default no actions are allowed unless at least one authorisation policy can be applicable to the request.

```
if ⟨CONDITION⟩ then can ⟨S, A, T⟩
```

Figure 2.2: Representation of policies in ESPOON

In our approach, an authorisation policy is represented as a condition and a tuple as illustrated in Figure 2.2. This authorisation policy is interpreted as follows: if *CONDITION* is true then the subject S can execute the action A on the target T. At the time when a request is made, the information about the subject, the action that is requested and the target resource is collected by the Requester. The PIP collects several attributes representing the context in which the request is being executed and sends them to the PDP.

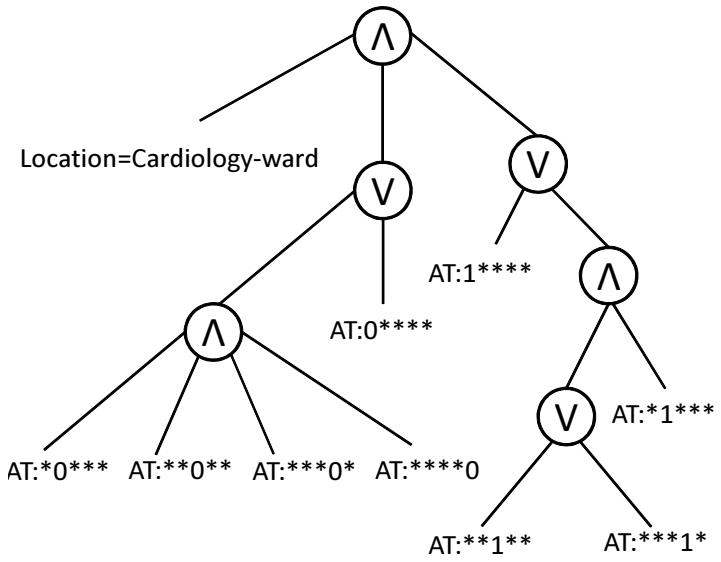


Figure 2.3: An example of a contextual condition illustrating $Location = Cardiology\text{-}ward$ and $AT > 9\#5$ and $AT < 17\#5$

The PIP collects and sends required contextual information to the PDP. To represent contextual conditions, we use the tree structure described in [47] 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 2.3 illustrates the tree structure representing this contextual condition, where access time is in a 5-bit representation (i.e., #5).

Let us consider *CONDITION* illustrated in Figure 2.3 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: $Location = Cardiology\text{-}ward$, $AT : 0 * * * *$, $AT : *1 * **$, $AT : * * 0 * *$, $AT : * * * 1 *$, $AT : * * * * 0$, 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.

In this policy representation, the $\langle S, A, T \rangle$ tuple and the leaf nodes in the condition tree are in clear text. Therefore, such information is easily accessible in the outsourced environment and may leak information about the data that the policies protect. In the following, we show how such representation can be protected while allowing the PDP to evaluate the policies against the request.

2.4 Solution Details of ESPOON

The main idea of our approach is to use an encryption scheme for protecting the confidentiality of the policies while allowing the PDP to perform the correct evaluation of the policies. We noticed that the operation performed by the PDP for evaluating policies is similar to the search operation executed in a database. In particular, in our case the condition of a policy is the query; and the data that is matched against the query is represented by the attributes that the Requester sends in the request.

As a starting point, we consider the multiuser Searchable Data Encryption (SDE) scheme proposed by Dong *et al.* in [30]. 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 [30] 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 for managing lifecycle of policies in outsourced environments. These phases include *initialisation*, ***policy deployment***, ***policy evaluation*** and *user revocation*. In the following, we provide the details of the algorithms used in each phase.

2.4.1 The Initialisation Phase

Before the policy deployment and policy evaluation phases, the SDE scheme needs to be initialised. This is required for generating the required key material. The following two algorithms that need to be run:

- The initialisation algorithm **Init** (Algorithm 2.1) is run by the TKMA. It takes as input the security parameter 1^k and outputs the public parameters $params$ and the master secret key set msk.
- The user key sets generation algorithm **KeyGen** (Algorithm 2.2) is run by the TKMA. It takes as input the master secret key set msk and the user (Admin User or Requester) identity i and generates two key sets K_{u_i} and K_{s_i}. The TKMA sends key sets K_{u_i} and K_{s_i} to the user i and the Key Store, respectively. Only the Administration Point, PDP and PEP are authorised to access the Key Store.

2.4.2 The Policy Deployment Phase

The policy deployment phase is executed when a new set of policies needs to be deployed on the Policy Store (or an existing version of policies needs to be updated). This phase is executed by the Admin User who edits the policies in a trusted environment. Before the policies leave the trusted environment, they need to be encrypted. Our policy representation consists of two parts: one for representing the condition and the other for the $\langle S, A, T \rangle$ tuple. Each part is encrypted using the following algorithms:

- The access policy condition encryption algorithm **ConditionEnc** (Algorithm 2.5) is run by the Admin User i. It takes as input a contextual condition and the user side key set K_{u_i} corresponding to Admin User i and outputs the encrypted contextual condition.

- The access policy $\langle S, A, T\rangle$ tuple encryption algorithm **SATEnc** (Algorithm 2.7) is run by the Admin User i. It takes as input the $\langle S, A, T\rangle$ tuple and K_{u_i} and outputs the client encrypted tuple $c_i^*(\langle S, A, T\rangle)$.

When the encrypted policy is sent to the outsourced environment, then another encryption round is performed. This is accomplished using the following algorithms:

- The access policy condition re-encryption algorithm **ConditionReEnc** (Algorithm 2.6) is run by the Administration Point. It takes as input the client encrypted contextual condition and the key K_{s_i} corresponding to the Admin User i and outputs the server encrypted contextual condition.
- The access policy $\langle S, A, T\rangle$ tuple re-encryption algorithm **SATReEnc** (Algorithm 2.8) is run by the Administration Point. It takes as input the client encrypted tuple $c_i^*(\langle S, A, T\rangle)$ and the key K_{s_i} corresponding to the Admin User i and outputs the re-encrypted tuple $c(\langle S, A, T\rangle)$.

The access policy can be now stored in the Policy Store. The stored policies do not reveal any information about the data because they are stored as encrypted.

2.4.3 The Policy Evaluation Phase

The policy evaluation phase is executed when a Requester makes a request to access the data. Before the access permission is granted, the PDP evaluates the matching policies in the Policy Store on the Service Provider. The request contains the $\langle S, A, T\rangle$ tuple. This information is encrypted using the following algorithm before it leaves the trusted environment:

- The $\langle S, A, T\rangle$ request encryption algorithm **SATRequest** (Algorithm 2.12) is run by Requester j. It takes as input the $\langle S, A, T\rangle$ tuple and K_{u_j} and outputs the client encrypted tuple $T_j^*(\langle S, A, T\rangle)$.

The Requester sends the encrypted $\langle S, A, T\rangle$ tuple to the Service Provider. The policy evaluation phase on the Service Provider side starts with searching all the policies in the Policy Store matching against the Requester $\langle S, A, T\rangle$ tuple. This is accomplished by the following algorithm:

- The $\langle S, A, T\rangle$ tuple search algorithm **SATSearch** (Algorithm 2.13) is run by the PDP. It takes as input the client encrypted tuple $T_j^*(\langle S, A, T\rangle)$ from Requester j and all stored policies in the Policy Store $c(\langle S_i, A_i, T_i\rangle)_{1\leq i\leq n}$ and returns the matching tuples in the Policy Store.

If any match is found in the Policy Store then the PDP needs to match the contextual information against the access policy condition corresponding to the matched tuple. The PDP fetches the contextual information including Requester and environmental attributes from the PIP. The PIP encrypts the contextual information using the following algorithm:

- The attributes encryption algorithm **AttributesRequest** (Algorithm 2.14) is run by the PIP j. It takes as input the Requester and environmental attributes and K_{u_j} and outputs the encrypted attributes.

After receiving the contextual information from the PIP, the PDP matches the PIP attributes against the access policy condition. The PDP calls the following algorithm to evaluate the access policy condition:

- The access policy condition evaluation algorithm **ConditionEvaluation** (Algorithm 2.15) is run by the PDP. It takes as input a list of encrypted attributes, the key K_{s_j} corresponding to the PIP j and encrypted access policy condition tree and outputs *true* on successful policy evaluation and *false* otherwise.

2.4.4 The User Revocation Phase

The proposed solution offers revocation of a user (an Admin User or a Requester). For this purpose, the Administration Point runs the following algorithm:

- A user (an Admin User or a Requester) revocation algorithm **UserRevocation** (Algorithm 2.17) is run by the Administration Point. Given the user i, the Administration Point removes the corresponding server side key K_{s_i} from the Key Store.

2.5 Algorithmic Details of ESPOON

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

2.5.1 The Initialisation Phase

In this phase, the system is initialised and then the TKMA generates required keying material for entities in ESPOON. 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 2.1. 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 generator g such that $\mathbb{G}$ is the unique order q

Algorithm 2.1 Init

Description: *It generates the system level keying material including public parameters and the master secret.*

Input: A security parameter 1^k.

Output: The public parameters *params* 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: $params \leftarrow (\mathbb{G}, g, q, h, H, f)$
9: $msk \leftarrow (x, s)$
return $(params, msk)$

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 2.2 KeyGen

Description: *For each user, it generates two key sets: one for the user while other for the server.*

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: $K_{s_i} \leftarrow (i, x_{i2})$
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 2.2. 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).

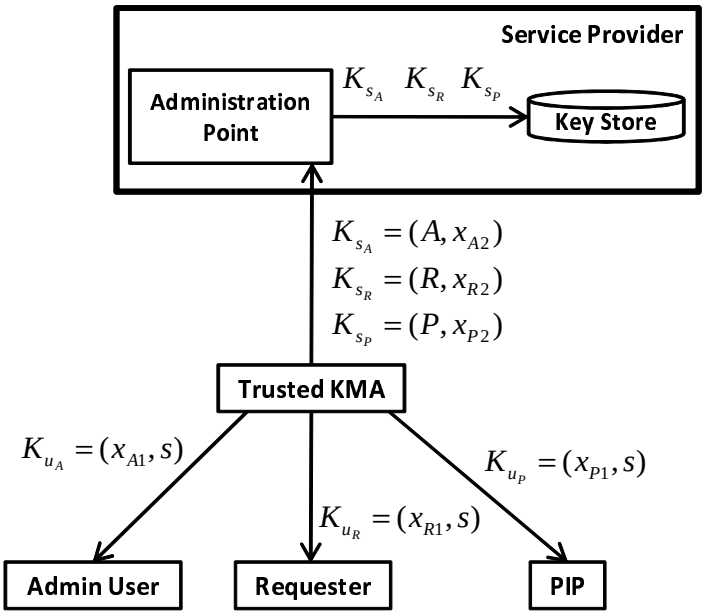


Figure 2.4: Distribution of keys in ESPOON

After running Algorithm 2.2, 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 Service Provider, respectively. The client side key set K_{u_i} serves as a private key for user i. The Administration Point of the Service Provider 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 2.4 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 Service Provider. 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.

2.5.2 The 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 Service Provider performs a second round of encryption. For performing a first round of encryption, an Admin User runs **ClientEnc** illustrated in Algorithm 2.3. **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

Algorithm 2.3 ClientEnc

Description: *It transforms the cleartext element into the client encrypted element.*

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 $c_i^*(e)$.

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

Algorithm 2.4 ServerReEnc

Description: *It transforms the client encrypted element into the server encrypted element.*

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 $c(e)$.

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

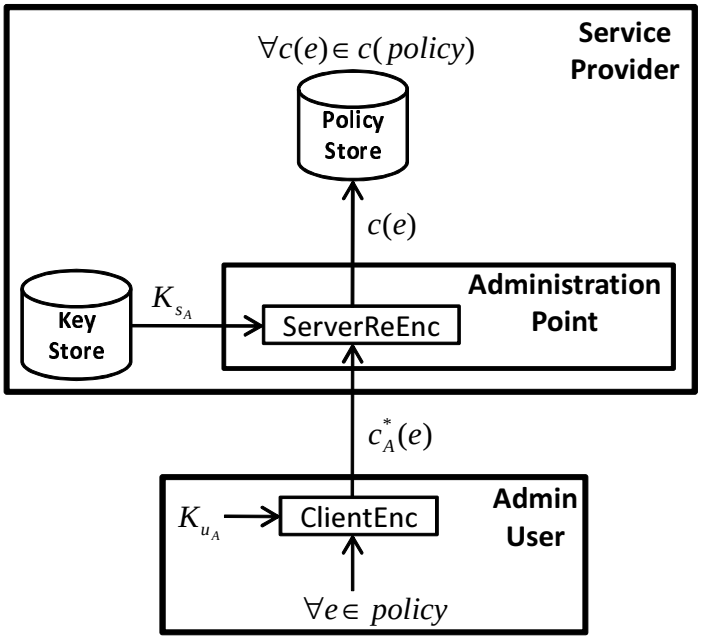


Figure 2.5: The policy deployment phase

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 2.5.

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 2.4. **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 2.5.

Algorithm 2.5 ConditionEnc

Description: *It transforms the cleartext condition into the client encrypted condition.*

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: $\quad c_i^*(e) \leftarrow$ call **ClientEnc** (e, K_{u_i}, *params*)
4: $\quad$ replace e of T_{C_i} with $c_i^*(e)$
5: **end for**
return T_{C_i}

Algorithm 2.6 ConditionReEnc

Description: *It transforms the client encrypted condition into the server encrypted condition.*

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: $\quad c(e) \leftarrow$ call **ServerReEnc** ($c_i^*(e)$, K_{s_i})
5: $\quad$ replace $c_i^*(e)$ of T_S with $c(e)$
6: **end for**
return T_S

Deployment of Contextual Conditions: The contextual condition can be deployed in two steps. In the first step, an Admin User performs a first round of encryption by running Algorithm 2.5. 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 2.3 (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 2.6. 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 2.4 (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 2.7 SATEnc

Description: *It transforms the cleartext tuple into the client encrypted tuple.*

Input: The $\langle S, A, T\rangle$ tuple, the client side key set K_{u_i} corresponding to Admin User i and the public parameters *params*.

Output: The client encrypted tuple $c_i^*(\langle S, A, T\rangle)$.

1: $c_i^*(S) \leftarrow$ call **ClientEnc** (S, K_{u_i}, *params*)
2: $c_i^*(A) \leftarrow$ call **ClientEnc** (A, K_{u_i}, *params*)
3: $c_i^*(T) \leftarrow$ call **ClientEnc** (T, K_{u_i}, *params*)
4: $c_i^*(\langle S, A, T\rangle) \leftarrow (c_i^*(S), c_i^*(A), c_i^*(T))$
return $c_i^*(\langle S, A, T\rangle)$

Algorithm 2.8 SATReEnc

Description: *It transforms the client encrypted tuple into the server encrypted tuple.*

Input: The client encrypted tuple $c_i^*(\langle S, A, T\rangle)$ and identity of Admin User i.

Output: The server encrypted tuple $c(\langle S, A, T\rangle)$.

1: $K_{s_i} \leftarrow KS[i]$ ▷ retrieve the server side key corresponding to Admin User i
2: $c(S) \leftarrow$ call **ServerReEnc** ($c_i^*(S)$, K_{s_i})
3: $c(A) \leftarrow$ call **ServerReEnc** ($c_i^*(A)$, K_{s_i})
4: $c(T) \leftarrow$ call **ServerReEnc** ($c_i^*(T)$, K_{s_i})
5: $c(\langle S, A, T\rangle) \leftarrow (c(S), c(A), c(T))$
return $c(\langle S, A, T\rangle)$

Deployment of a $\langle S, A, T\rangle$ ***Tuple:*** For deploying any $\langle S, A, T\rangle$ tuple, an Admin User performs the first round of encryption using her private key as illustrated in Algorithm 2.7, where each element including S, A and T is encrypted on the client side by running

ClientEnc (Algorithm 2.3) as shown in Line 1, Line 2 and Line 3, respectively. The Administration Point on the server side receives the client encrypted tuple and performs the second round of encryption using the server side key corresponding to the Admin User as illustrated in Algorithm 2.8, where the Administration Point first retrieves the server side key corresponding to Admin User i from the Key Store (see Line 1) and then re-encrypts $c_i^*(S)$, $c_i^*(A)$ and $c_i^*(T)$ by running **ServerReEnc** (Algorithm 2.4) as shown in Line 2, Line 3 and Line 4, respectively. Finally, the server encrypted tuple is stored in the Policy Store.

Algorithm 2.9 ClientTD

Description: *It transforms the cleartext element into the client generated trapdoor.*

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: $td_i^*(e) \leftarrow (t_1, t_2)$
return $td_i^*(e)$

Algorithm 2.10 ServerTD

Description: *It transforms the client generated trapdoor into the server generated trapdoor.*

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

Algorithm 2.11 Match

Description: *It matches the serer encrypted element against the server generated trapdoor.*

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**
return *true*
2: **else**
return *false*
3: **end if**

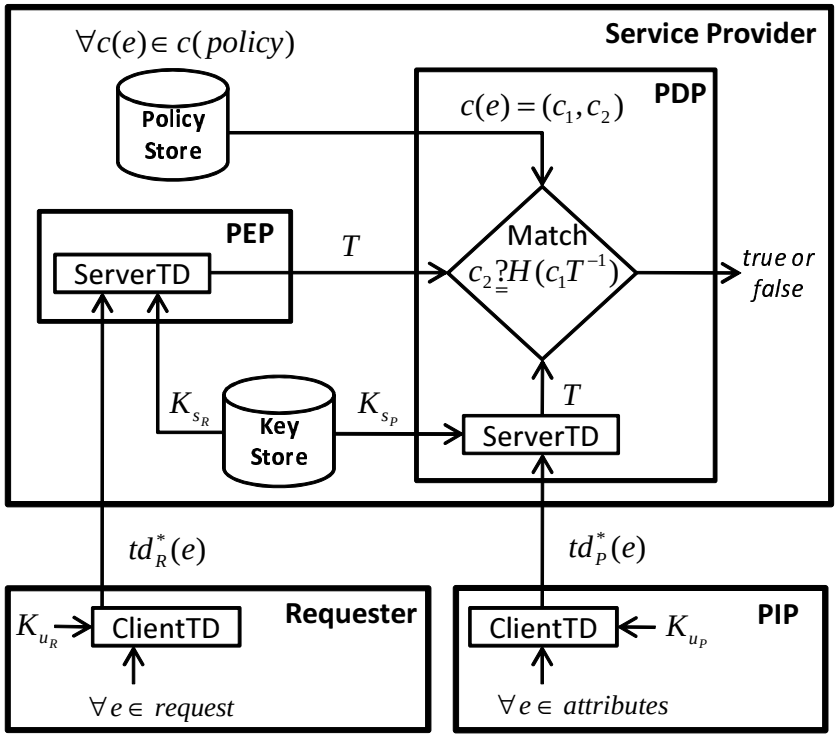


Figure 2.6: The policy evaluation phase

2.5.3 The Policy Evaluation Phase

The policy evaluation phase is executed when a Requester makes a request. In this phase, a Requester sends client generated trapdoors (using Algorithm 2.9) of a request to the PEP. The PEP converts client generated trapdoors into server generated trapdoors (using Algorithm 2.10) and sends them to the PDP. The PDP matches server encrypted trapdoors of the request with server encrypted elements of the policy (using Algorithm 2.11). 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 2.6. 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 2.9. **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 2.10 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).

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 2.11. **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 1) indicating that the match is successful. Otherwise, it returns *false* (Line 2).

In the following, we describe how to evaluate policies. For the evaluation of each policy, we follow general strategy as already described in this section and also illustrated in Figure 2.6.

Algorithm 2.12 SATRequest

Description: *It transforms the cleartext tuple into the client generated trapdoor tuple.*

Input: Tuple $\langle S, A, T\rangle$, the client side key set K_{u_i} corresponding to Requester j and the public parameters *params*.

Output: The client generated trapdoor tuple $td_j^*(\langle S, A, T\rangle)$.

1: $td_j^*(S) \leftarrow$ call **ClientTD** (S, K_{u_j}, *params*)
2: $td_j^*(A) \leftarrow$ call **ClientTD** (A, K_{u_j}, *params*)
3: $td_j^*(T) \leftarrow$ call **ClientTD** (T, K_{u_j}, *params*)
4: $td_j^*(\langle S, A, T\rangle) \leftarrow (td_j^*(S), td_j^*(A), td_j^*(T))$
return $td_j^*(\langle S, A, T\rangle)$

Generating Tuples: A Client Request: For making an access request, a Requester transforms the cleartext tuple into the trapdoor tuple as illustrated in Algorithm 2.12, which transforms each element in tuple including S, A and T into its corresponding trapdoor $td_j^*(S)$, $td_j^*(A)$ and $td_j^*(T)$ (using **ClientTD** illustrated in Algorithm 2.9)as shown in Line 1, Line 2 and Line 3, respectively. The Requester client side sends the trapdoor tuple to the PEP.

Searching a Tuple: When a Requester makes an access request, the PEP receives the client encrypted request and then it re-encrypts the request. The Service Provider first retrieves the server side key corresponding to Requester j as illustrated in Algorithm 2.13 Line 1. Next, it calls **ServerTD** (Algorithm 2.10) for each client encrypted element

Algorithm 2.13 SATSearch

Description: *It checks whether the access request matches with any encrypted tuple on the server side.*

Input: The client generated trapdoor tuple $td_j^*(\langle S, A, T\rangle)$, the identity of Requester j and a list (of size n) of encrypted policies stored on the server $c(\langle S_i, A_i, T_i\rangle)_{1\leq i\leq n})$.

Output: *true* or *false*.

1: $K_{s_j} \leftarrow KS[j]$ ▷ retrieve the server side key corresponding to Requester j
2: $td(S) \leftarrow$ call **ServerTD** ($td_j^*(S)$, K_{s_j})
3: $td(A) \leftarrow$ call **ServerTD** ($td_j^*(A)$, K_{s_j})
4: $td(T) \leftarrow$ call **ServerTD** ($td_j^*(T)$, K_{s_j})
5: **for** each encrypted tuple $c(\langle S, A, T\rangle)$ in $c(\langle S_i, A_i, T_i\rangle)_{1\leq i\leq n})$ **do**
6: $match_S \leftarrow$ call **Match** ($c(S)$, $td(S)$)
7: $match_A \leftarrow$ call **Match** ($c(A)$, $td(A)$)
8: $match_T \leftarrow$ call **Match** ($c(T)$, $td(T)$)
9: **if** $match_S \stackrel{?}{=} true$ and $match_A \stackrel{?}{=} true$ and $match_T \stackrel{?}{=} true$ **then**
return *true*
10: **end if**
11: **end for**
return *false*

including $td_j^*(S)$, $td_j^*(A)$ and $td_j^*(T)$ and calculates $td(S)$, $td(A)$ and $td(T)$ as shown in Line 2, Line 3 and Line 4, respectively. Then, the Service Provider checks if any encrypted tuple in the Policy Store matches with the encrypted access request (Line 5). For performing this match, all three encrypted elements are matching using **Match** (Algorithm 2.11) (Line 6-8). If all three elements are matched (Line 9), then this algorithm returns *true*. In case if no match is found, this algorithm returns *false*.

Algorithm 2.14 AttributesRequest

Description: *It transforms contextual attributes into trapdoors.*

Input: List of attributes contextual attributes L, the client side key set K_{u_j} corresponding to PIP j and the public parameters *params*.

Output: The client generated list of trapdoors of contextual attributes L_{C_j}.

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

Generating Contextual Attributes: The PIP runs **AttributesRequest** illustrated in Algorithm 2.14 to calculate client generated trapdoors of contextual information. **At-**

tributesRequest takes as input a list of contextual attributes L, the client side key set K_{u_j} corresponding to PIP j and the public parameters *params* and outputs the client generated list of trapdoors of contextual attributes L_{C_j}. First, it creates and initialises new list L_{C_j} (Line 1). For each attribute e in L (Line 2), it calculates the client generated trapdoor $td_j^*(e)$ by calling Algorithm 2.9 (Line 3) and adds $td_j^*(e)$ in L_{C_j} (Line 4).

Algorithm 2.15 ConditionEvaluation

Description: *It evaluates contextual condition and returns true on successful match and false otherwise.*

Input: The client generated list of trapdoors of contextual attributes L_{C_j}, the server encrypted contextual condition T_S and the identity of PIP j.

Output: *true* or *false*.

1: $K_{s_j} \leftarrow KS[j]$ ▷ retrieve the server side key corresponding to PIP j
2: $L_S \leftarrow \phi$
3: **for** each client generated trapdoor $td_j^*(e)$ in L_{C_j} **do**
4: $td(e) \leftarrow$ call **ServerTD** ($td_j^*(e)$, K_{s_j})
5: $L_S \leftarrow L_S \cup td_j^*(e)$
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: *break*;
17: **end if**
18: **end for**
19: **end for**
20: call **EvaluateTree** (*TREE.root*, *TREE*) ▷ see Algorithm 2.16
return *TREE.root.decision*

Evaluating Contextual Conditions: For evaluating any contextual condition, the PDP runs **ConditionEvaluation** illustrated in Algorithm 2.15. This algorithm takes as input the client generated list of trapdoors of contextual attributes L_{C_j}, the server encrypted contextual condition T_S and identity of PIP j and returns either *true* or *false*. First, it retrieves from the Key Store the server side key K_{s_j} (Line 1). Next, it creates and initialises a new list L_S (Line 2). For each client generated trapdoor $td_j^*(e)$ in L_{C_j}

(Line 3), it calculates the server generated trapdoor $td(e)$ by calling Algorithm 2.10 (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 2.11 (Line 14). Next, it evaluates non-leaf nodes of *TREE* by running Algorithm 2.16 (Line 20). Finally, it returns either *true* or *false* depending upon the evaluation of *TREE*.

Algorithm 2.16 EvaluateTree

Description: *Given a tree node, it recursively evaluates internal nodes of a policy tree and returns true if the policy tree is satisfied and false otherwise.*

Input: Node n and tree T.

Output: *true* or *false*.

```
1: if n.decision ≠ null then
       return n.decision
2: end if
3: for each child c of n in tree T do
4:     call EvaluateTree (c, T)                          ▷ recursive call
5: end for
6: t ← 0
7: m ← 0
8: for each child c of n in tree T do
9:     t ← t + 1
10:    if c.decision ?= true then
11:        m ← m + 1
12:    end if
13: end for
14: if (n.gate ?= AND and m ?= t) or (n.gate ?= OR and m ≥ 1) then
15:    n.decision ← true
16: else
17:    n.decision ← false
18: end if
       return n.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 1). For each child c of n in tree T (Line 3), it recursively calls **EvaluateTree** (Line 4). Next, it creates and initialises t (Line 6) and m (Line 7) indicating total children of n and a count of matched children, respectively. For each child c of n in tree T (Line 8), it counts total children (Line 9)

and matched children by checking made decisions (Line 11). 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 14). If so, it is set as *true* (Line 15) and *false* (Line 17) otherwise.

Algorithm 2.17 UserRevocation

Description: *It removes users from the system.*
Input: The user identity i.
Output: *true* or *false*.

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

2.5.4 The User Revocation Phase

In this phase, a user (an Admin User or a Requester) can be removed from the system. This phase consists of one algorithm called **UserRevocation** illustrated in Algorithm 2.17, which is run by the Administration Point. Given the user identity i, this algorithm checks whether the server side key set corresponding to user i exists in the Key Store (Line 1). If not then this algorithm returns *false* (Line 1), indicating that no such user exists. Otherwise, the server side key set K_{s_i} corresponding to user i is removed from the Key Store (Line 3-4) and finally this algorithm returns *true* (Line 4), indicating that user i has been removed from the system successfully.

2.6 Performance Analysis of ESPOON

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

2.6.1 Implementation Details of ESPOON

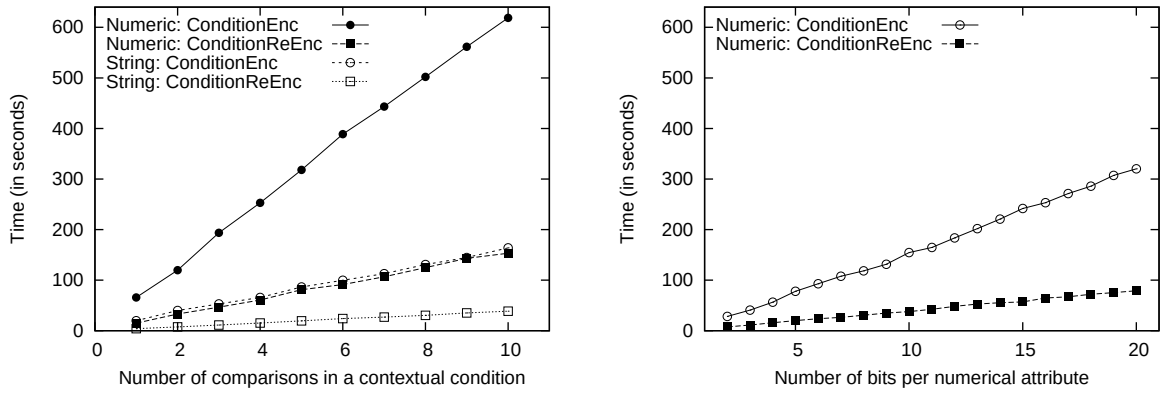
We have implemented ESPOON in Java 1.6. We have developed all the components of the architecture required in the management lifecycle of ESPOON policies in outsourced environments. In particular, we have implemented all the algorithms presented in Section

2.5. We have tested the implementation of ESPOON 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. The number of iterations performed for each of the following results is 1000.

2.6.2 Performance Analysis of the Policy Deployment Phase

In this section, we analyse the performance of the policy deployment phase. In this phase, access policies are first encrypted at the Admin User side (that is a trusted domain) and then sent over to the Administration Point running in the outsourced environment. The Administration Point re-encrypts the policies and stores them in the Policy Store in the outsourced environment. The policy contains two parts (i) a contextual condition and (ii) a $\langle S, A, T \rangle$ tuple. In the following, we discuss performance overheads of deploying both parts.

Deploying a Contextual Condition: Our policy representation consists of the tree representing the policy condition and the $\langle S, A, T \rangle$ tuple describing what action A a subject S can perform over the target T. In the tree representing contextual conditions, leaf nodes represent string comparisons (for instance, $Location = Cardiology\text{-}ward$) 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.



(a) (b)

Figure 2.7: Performance overhead of deploying contextual conditions: (a) numerical and string comparisons and (b) size of a numerical attribute

The performance overhead of deploying contextual conditions is illustrated in Figure 2.7. Figure 2.7(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 steeper 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 "$attributeName_i < 15\#4$".[1]

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 contextual 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 2.7(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 $O(m + n \cdot s)$ 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.

Table 2.1: Performance overhead of encrypting the $\langle S, A, T\rangle$ tuple during the policy deployment

Algorithm Name	**SATEnc**	**SATReEnc**
Time (in milliseconds)	46.44	11.65

Deploying a $\langle S, A, T\rangle$ ***Tuple:*** As for the $\langle S, A, T\rangle$ tuple, the average encryption time taken by the **SATEnc** (Algorithm 2.7) and **SATReEnc** (Algorithm 2.8) are shown in Table 2.1. The time complexity of the encryption of the $\langle S, A, T\rangle$ tuple during the policy deployment phase is constant because it does not depend on any parameters.

During the policy deployment phase, the encryption operations performed on the Admin User side take more time to encrypt the access policy than the Service Provider side to re-encrypt the same policy (either **ConditionReEnc** or **SATReEnc**). This is because the **ConditionEnc** and **SATEnc** algorithms perform more complex cryptographic operations, such as generation of random number and hash calculations, than the respective algorithms on the Service Provider side.

[1]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.

2.6.3 Performance Analysis of the Policy Evaluation Phase

In this section, we analyse the performance of the policy evaluation phase. In this phase, a Requester encrypts the $\langle S, A, T\rangle$ tuple before sending to the PEP running in the outsourced environment. The PEP re-encrypts and forwards it to the PDP. The PDP has to select the set of policies that are applicable to the request. Once the PDP has found the policies then the PDP will evaluate if the attributes in the contextual information satisfy any of the conditions of the selected policies. In the following, we discuss performance overhead of generating the encrypted $\langle S, A, T\rangle$ tuple, searching the requested $\langle S, A, T\rangle$ tuple in the policy store and evaluating contextual conditions.

Table 2.2: Performance overhead of generating the $\langle S, A, T\rangle$ request

Algorithm Name	**SATRequest**
Time (in milliseconds)	47.07

The $\langle S, A, T\rangle$ ***Request Tuple:*** To make a request, it is necessary to generate the $\langle S, A, T\rangle$ tuple representing the subject S requesting to perform action A on target T. The $\langle S, A, T\rangle$ tuple needs to be transformed into *trapdoors* before it is sent over to the PEP. The trapdoors will be used for performing the encrypted policy evaluation in the outsourced environment. The trapdoor representation does not leak information on the element of the $\langle S, A, T\rangle$ tuple. This phase takes approximately 47.07 milliseconds (ms) as shown in Table 2.2.

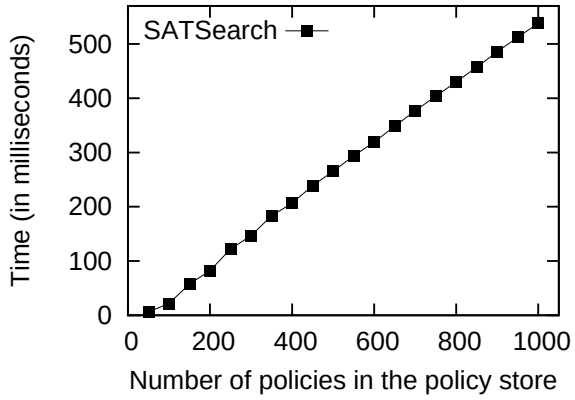
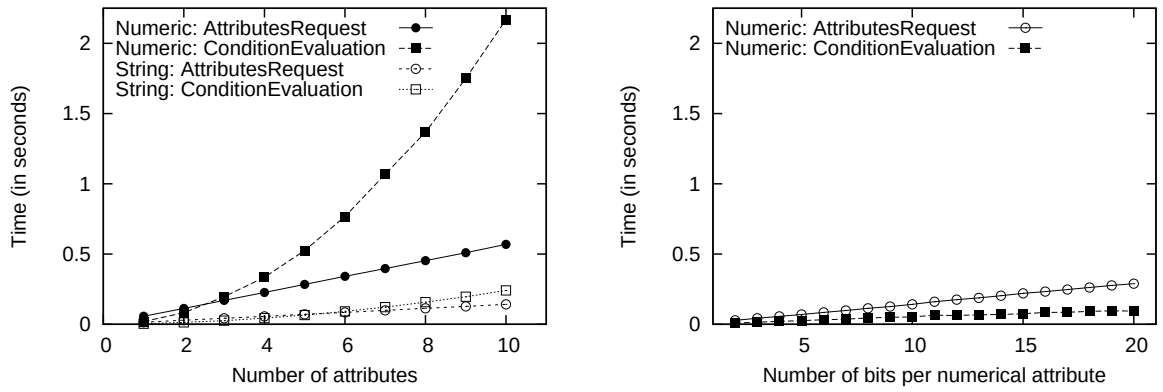


Figure 2.8: Performance overhead of searching a $\langle S, A, T\rangle$ tuple

Searching a $\langle S, A, T\rangle$ ***Tuple:*** Once the PDP gets the request, it re-encrypts and then performs an encrypted search in the Policy Store in order to find any matching $\langle S, A, T\rangle$ tuples. Figure 2.8 shows the performance overhead on the Service Provider side. In

our experiment, we varied the number of encrypted policies stored in the Policy Store ranging from 50 to 1000. As we can observe, it takes 0.5 ms on average for performing an encrypted match operation between the $\langle S, A, T \rangle$ tuple of the request and the $\langle S, A, T \rangle$ tuple in the Policy Store. This means that on average it takes half a second for finding a matching policy in the Policy Store with 1000 policies.



(a) (b)

Figure 2.9: Performance overhead of evaluating contextual conditions: (a) numerical and string attributes and (b) size of a numerical attribute

Evaluating Contextual Conditions: If any match is found in the Policy Store then the PDP needs to fetch the contextual information from the PIP. The PIP is responsible to collect and send the required contextual information that includes 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 2.14). 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 2.9 shows the performance overhead of evaluating contextual conditions. In particular, Figure 2.9(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

2.9(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 2.15. 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.

Figure 2.9(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 m_1 trapdoors in a request are matched against m_2 leaf nodes in the tree resulting in a $O(m_1 \cdot m_2)$ 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 $n \cdot s$ different trapdoors. These trapdoors then need to be matched against the leaf nodes representing the numerical comparisons. Figure 2.9(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. If there are n_1 numerical attributes in the request and n_2 different numerical comparisons (where each numerical attribute or numerical comparison is of size s), the complexity of evaluating numerical conditions will be $O(n_1 \cdot n_2 \cdot s^2)$ in the worst case. In general, the complexities of generating trapdoors for conditions and evaluating contextual conditions are $O(m + n \cdot s)$ and $O(m_1 \cdot m_2 + n_1 \cdot n_2 \cdot s^2)$, respectively.

Table 2.3 provides a summary of time complexity of each phase in the lifecycle of ESPOON.

Table 2.3: Summary of time complexity of each phase in the lifecycle of ESPOON

Phase Name	**Complexity in the Worst Case**
Deployment of contextual condition	$O(m + n \cdot s)$
Attributes request	$O(m + n \cdot s)$
Evaluation of contextual condition	$O(m_1 \cdot m_2 + n_1 \cdot n_2 \cdot s^2)$

2.7 Discussion

2.7.1 Data Protection

In this chapter, we have focused on how to enforce sensitive security policies in outsourced environments. For the data protection, we may employ existing encryption techniques, such as the proxy encryption scheme [30] or schemes based on ABE [47,49]. In [73], we have discussed how to protect data using the proxy encryption scheme. In this dissertation, we have covered the topic of data protection using CP-ABE [47] in Chapter 5.

2.7.2 Revealing Policy Structure

The access policy structure reveals information about the operators, such as AND and OR, and the number of operands used in the access policy condition. To overcome this problem, dummy attributes may be inserted in the tree structure of the access policy. Similarly, the PIP can send dummy attributes to the PDP at the time of policy evaluation to obfuscate the number of attributes required in a request.

2.7.3 Collusion Attack

In ESPOON, we assume that multiple users can collude; however, they cannot gain more than what each user can access individually because each one has her own private key and combination of those keys do not reveal any further information. On the other hand, a user and the Service Provider can collude together to gain unauthorised access to the data by combining their keys, where they can recover the master secret. For withstanding against this kind of collusion, one possibility is to assume multiple instances of the Service Provider and split the server side key such that each instance gets one share. The main drawback of this approach is that it cannot work if all instances of the Service Provider are compromised. Another approach is to provide protection with an extra layer of encryption say by employing Key-Policy Attribute Based Encryption (KP-ABE) [49], which is collusion-resistant.

2.7.4 On the Impossibility of Cryptography Alone for Privacy-Preserving Cloud Computing

Van Dijk and Juels argue in [74] that cryptography alone is not sufficient for preserving the privacy in the cloud environment. They prove that in multi-client settings it is impossible to control how information is released to clients with different access rights. Basically, in their threat model clients do not mutually trust each other. In our settings, users are mutually trusted: our main contribution is to protect the confidentiality of access policies (and therefore of the data) from the Service Provider.

2.8 Chapter Summary

In this chapter, we have presented the ESPOON architecture to support a policy-based access control mechanism for outsourced environments. Our approach separates the security policies from the actual enforcement mechanism while guaranteeing the confidentiality of the policies when given assumptions hold (i.e., the Service Provider is honest-but-curious). The main advantage of our approach is that policies are encrypted but it still allows the PDP to perform the policy evaluation without knowing the policies. Second, ESPOON is capable of handling complex policies 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 the policies.

From performance and management perspectives, ESPOON might be suitable for handling access policies of small to medium enterprises. However, both performance and management will be cumbersome if ESPOON has to be deployed for handling access policies of large enterprises having a large number of users, thus requiring complex user management. In the next chapter, we propose architecture that can enforce sensitive policies of large enterprises having a large number of users.

Chapter 3

ESPOON$_{ERBAC}$: Enforcing Encrypted RBAC Policies in Outsourced Environments*

For complex user management, large enterprises employ 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 chapter, we aim at filling this gap by proposing **Enforcing Sensitive Policies in Outsourced envirOnmeNts with Encrypted Role-Based Access Control (ESPOON$_{ERBAC}$)** for enforcing RBAC policies in outsourced environments. ESPOON$_{ERBAC}$ is based on ESPOON (discussed in Chapter 2). Basically, ESPOON$_{ERBAC}$ extends ESPOON in order to enforce RBAC policies in an encrypted manner, where a curious service provider do not learn private information about sensitive RBAC policies. We have implemented ESPOON$_{ERBAC}$ and provided its performance evaluation showing a limited overhead, thus confirming viability of our approach.

3.1 Introduction

According to [77], RBAC is the most widely used security model. RBAC [16] makes decisions based on roles a user is active in. However, it cannot be deployed in outsourced environments because it assumes a trusted infrastructure in order to regulate access on data. In RBAC models, RBAC policies may leak information about the data they aim

*The preliminary version of this chapter has appeared in [75, 76].

to protect. In [28], we propose ESPOON that aims at enforcing authorisation policies in outsourced environments. In [76], we extend ESPOON to support RBAC policies and role hierarchies but our solution does not outsource all operations because we assume presence of the Company RBAC Manager in trusted environments for the role assignment.

3.1.1 Research Contributions

In this chapter, we present an RBAC mechanism for outsourced environments where we support full confidentiality of RBAC policies. We named our solution **Enforcing Sensitive Policies in Outsourced envirOnmeNts with Encrypted Role-Based Access Control (ESPOON$_{ERBAC}$)**. ESPOON$_{ERBAC}$ is based on ESPOON. Like ESPOON, ESPOON$_{ERBAC}$ can enforce RBAC policies without revealing private information to the service provider that is assumed honest-but-curious. Summarising, the research contributions in this chapter are threefold.

1. The service provider does not learn private information about RBAC policies and the requester's attributes during the policy deployment or evaluation processes.

2. We extend the basic RBAC policies to support role hierarchies. The curious service provider enforces role hierarchy without revealing information about roles in the role hierarchy graph.

3. 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 ESPOON$_{ERBAC}$ mechanism and analysed its performance to quantify the overhead incurred by cryptographic operations used in the proposed scheme.

3.1.2 Chapter Outline

The rest of this chapter is structured as follows. Section 3.2 reviews the related work. In Section 3.3, we present the proposed architecture of ESPOON$_{ERBAC}$. Section 3.4 and Section 3.5 focus on solution details and algorithmic details, respectively. Security analysis of ESPOON$_{ERBAC}$ is provided in Section 3.6. In Section 3.7, we analyse the performance overhead of ESPOON$_{ERBAC}$. Finally, Section 3.8 summarises this chapter.

3.2 Related Work

RBAC [16] 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 [16], 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 [78–84]. 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. In [85], Crampton and Khambhammettu discuss delegation in RBAC. Unfortunately, existing solutions [16, 78–85] assume a trusted infrastructure to regulate access on data and they cannot be applied to outsourced environments, where a curious service provider might leak sensitive policies.

Mandatory Access Control (MAC) is a strict model of access control that takes a hierarchical approach to control access to resources [86]. In MAC, access to resources is controlled by the system administrator. MAC assigns security labels to resources. Discretionary Access Control (DAC) is a type of access control in which resource owners control access to their resources [87]. In DAC, each resource object has an Access Control List (ACL) that contains a list of users or groups who can gain access to the resource object. Like traditional RBAC, both MAC and DAC assume a trusted infrastructure in order to regulate access to the resources.

eXtensible Access Control Markup Language (XACML) is a standard that defines an access control policy language and a processing model specifying how to evaluate access requests against deployed access control policies [71, 88]. The XACML policy language is based on eXtensible Markup Language (XML). For making any access decision, XACML considers that access control policies and access requests are in cleartext. Unfortunately,

cleartext policies and access requests may reveal private information.

In [28], we 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 [76], we extended ESPOON to support RBAC policies and role hierarchies. However, in [76] 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.

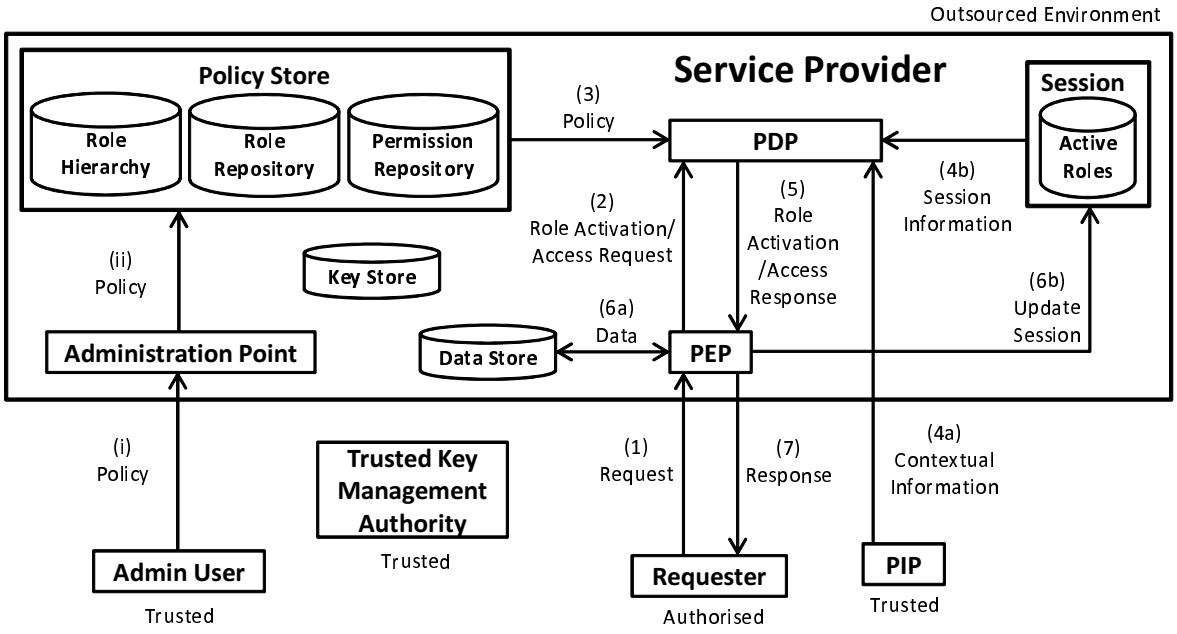


Figure 3.1: The ESPOON$_{ERBAC}$ architecture for enforcing RBAC policies in outsourced environments

3.3 The ESPOON$_{ERBAC}$ Approach

ESPOON$_{ERBAC}$ aims at providing RBAC mechanism that can be deployed in an outsourced environment. Figure 3.1 illustrates the proposed architecture that has similar components to the widely accepted architecture for the policy-based management proposed by IETF [71]. 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 PEP. This request includes the Requester's identifier and the requested role. The PEP forwards (2) the role activation request to the 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 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*, 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 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

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

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*.

Since ESPOON$_{ERBAC}$ is based on ESPOON, we use the same system model as already considered in ESPOON (see Section 2.3.1).

```
if ⟨CONDITION⟩ then ⟨USER⟩ can be active in ⟨{R_1, R_2, ..., R_n}⟩
```

Figure 3.2: RBAC Policy: Role assignment

```
if ⟨CONDITION⟩ then ⟨R⟩ can execute ⟨{(A_1, T_1), (A_2, T_2), ..., (A_n, T_n)}⟩
```

Figure 3.3: RBAC Policy: Permission assignment

3.3.1 Representation of RBAC Policies and 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 3.2 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 3.3 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.

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 (as already explained in Section 2.3.2).

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

Figure 3.4: RBAC Policy: Role hierarchy

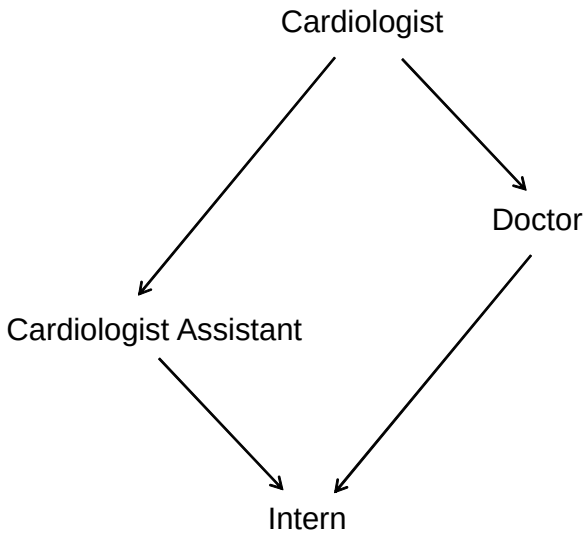


Figure 3.5: An example of a role hierarchy graph illustrating that Cardiologist and Doctor roles are derived from Intern while the Cardiologist role is derived from Cardiologist and Doctor roles

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 Service Provider. In ESPOON$_{ERBAC}$, the PDP traverses in the role hierarchy graph to find base roles. Figure 3.4 illustrates how we represent a role hierarchy graph. In Figure 3.4, 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 3.5.

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.

3.4 Solution Details of ESPOON$_{ERBAC}$

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$_{ERBAC}$, we extend ESPOON. In the following, we describe core phases in ESPOON$_{ERBAC}$.

3.4.1 The 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 2.3), 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 3.5), 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 Service Provider. 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.

3.4.2 The Policy Evaluation Phase

A Requester can make a role activation request ACT. Before sending ACT to the Service Provider, 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 Service Provider. 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 2.3). 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 roles, 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 encrypted 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.

3.5 Algorithmic Details of ESPOON$_{ERBAC}$

In this section, we provide details of algorithms used in core phases (including the policy deployment phase and the policy evaluation phase) for managing lifecycle of policies. The following algorithms (along with ESPOON algorithms described in Chapter 2, Section 3.5) constitute the proposed schema.

3.5.1 The Policy Deployment Phase

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 Section 2.5.2 and also illustrated in Figure 2.5.

Algorithm 3.1 RoleAssignment:ClientEnc

Description: *It transforms the cleartext role assignment list into the client encrypted role assignment list.*

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: $c_i^*(r) \leftarrow$ call **ClientEnc** (r, K_{u_i}, *params*) ▷ see Algorithm 2.3
4: $L_{C_i} \leftarrow L_{C_i} \cup c_i^*(r)$
5: **end for**
return (j, L_{C_i})

Algorithm 3.2 RoleAssignment:ServerReEnc

Description: *It re-encrypts the client encrypted role assignment list and generates the server encrypted role assignment list.*

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: $c(r) \leftarrow$ call **ServerReEnc** ($c_i^*(r)$, K_{s_i}) ▷ see Algorithm 2.4
5: $L_S \leftarrow L_S \cup c(r)$
6: **end for**
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:ClientEnc** illustrated in Algorithm 3.1. 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 a list L_{C_i} (Line 1). For each role in L (Line 2), it generates client encrypted role by calling **ClientEnc** illustrated in Algorithm 2.3 (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:ServerReEnc** illustrated in Algorithm 3.2. 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:ServerReEnc**, the Administration Point first retrieves the server side key K_{s_i} corresponding to Admin User i (Line 1). It creates and initialises a 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 2.4 (Line 4) and updates L_S by adding the server encrypted role (Line 5).

Algorithm 3.3 PermissionAssignment:ClientEnc

Description: *It transforms the cleartext permission assignment list into the client encrypted permission assignment list.*

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**
return ($c_i^*(r)$, L_{C_i})

Deployment of Permission Assignment Policies: An Admin User can assign permissions to a role. In order to deploy policies regarding permissions assignment to roles,

Algorithm 3.4 PermissionAssignment:ServerReEnc

Description: *It re-encrypts the client encrypted permission assignment list.*

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: $c(action) \leftarrow$ call **ServerReEnc** $(c_i^*(action), K_{s_i})$
6: $c(target) \leftarrow$ call **ServerReEnc** $(c_i^*(target), K_{s_i})$
7: $L_S \leftarrow L_S \cup (c(action), c(target))$
8: **end for**
return $(c(r), L_S)$

an Admin User runs Algorithm 3.3. 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 2.3 (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 3.4. 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 2.4 (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).

Deployment of a 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

Algorithm 3.5 RoleHierarchy:ClientEnc

Description: *It encrypts the role hierarchy graph.*

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 2.9
5: replace r of G_{C_i} with $(c_i^*(r), td_i^*(r))$
6: **end for**
return G_{C_i}

Algorithm 3.6 RoleHierarchy:ServerReEnc

Description: *It re-encrypts the client generated role hierarchy graph.*

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 2.10
6: replace $(c_i^*(r), td_i^*(r))$ of G_S with $(c(r), td(r))$
7: **end for**
return G_S

fulfil requested permissions. For this purpose, the PDP needs a trapdoor of each base role so that it can match this trapdoor against encrypted 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 3.5. 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 2.3 (Line 3) and the client trapdoor by calling **ClientTD** (Line 4) illustrated in Algorithm 2.9 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 3.6. 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 2.4 (Line 4) and the server trapdoor by calling **ServerTD** (Line 5) illustrated in Algorithm 2.10 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).

3.5.2 The Policy Evaluation Phase

The policy evaluation phase is executed when a Requester makes a request either ACT or REQ. 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 2.6.

Searching a Role: A Requester can make a role activation request ACT and sends it to the Service Provider. In order to grant ACT, the Service Provider runs **SearchRole** illustrated in Algorithm 3.7. 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 2.10 (Line 2). For each server encrypted role $c(r)$ in L_S (Line 3), it performs matching against $td(r)$ by calling Algorithm 2.11 (Line 4). If any match is successful (Line 5), it returns *true*

Algorithm 3.7 SearchRole

Description: *It checks whether the requested role is in the role assignment list of the Requester.*

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} ← KS[i]                          ▷ retrieve the server side key corresponding to Requester i
2: td(r) ← call ServerTD (td_i^*(r), K_{s_i})
3: for each server encrypted role c(r) in L_S do
4:     match ← call Match (c(r), td(r))                                  ▷ see Algorithm 2.11
5:     if match ≟ true then
         return true
6:     end if
7: end for
       return false
```

(Line 5), meaning that ACT is granted. Otherwise, it returns *false* (Line 7).

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 Service Provider checks if the Requester is already in the role in REQ. For this purpose, the Service Provider runs Algorithm 3.7, where L_S shows a list of active roles in the session. Furthermore, the PDP also runs Algorithm 3.7 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.

Searching a Permission: A Requester can send REQ for executing certain permissions. The PEP on the Service Provider 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 3.7. After a role is matched in the Permission Repository, the PEP searches the permission in REQ by running Algorithm 3.8. 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 2.10. 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 2.11. If both matches are

Algorithm 3.8 SearchPermission

Description: *It checks whether the requested permission is present in the list of permissions assigned to the Requester.*

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**
return *true*
8: **end if**
9: **end for**
return *false*

successful (Line 7) for any permission $(c(action), c(target))$ in L_S, it returns *true* (Line 7). Otherwise, it returns *false* (Line 9).

Algorithm 3.9 SearchRoleHierarchyGraph

Description: *It checks whether the Requester's role is inherited from any base role in the role hierarchy graph.*

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**
return *true*
4: **end if**
5: **end for**
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 3.9 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 2.11 (Line 2). If any match is found (Line 3), it returns *true* (Line 3). Otherwise, it returns *false* (Line 5).

3.6 Security Analysis

In this section, we provide a combined security analysis of ESPOON$_{ERBAC}$ and ESPOON because ESPOON$_{ERBAC}$ is built on the top of ESPOON. In other words, ESPOON$_{ERBAC}$ uses algorithms presented in Chapter 2. Therefore, we have not provided any security analysis of ESPOON in Chapter 2. In this section, we analyse the security of the policy deployment phase that includes the Role Assignment (RA) encryption (Algorithms 3.1 and 3.2), the Permission Assignment (PA) encryption (Algorithms 3.3 and 3.4), the Contextual Condition (CC) encryption (Algorithms 2.5 and 2.6), and the Role Hierarchy (RH) encryption (Algorithms 3.5 and 3.6). We then analyse the security of the policy evaluation phase that include Search Role (SR) (Algorithms 2.9 and 3.7), Search Permission (SP) (Algorithms 2.9 and 3.8), Contextual Condition Evaluation (CCE) (Algorithms 2.14 and 2.15) and Search Role Hierarchy (SRH) (Algorithms 2.9, 2.10 and 3.9).

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

3.6.1 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 the policy deployment phase are based on **ClientEnc** (Algorithm 2.3) and **ServerReEnc** (Algorithm 2.4). It is equivalent to encrypting a single keyword in the SDE scheme [30]. Dong *et al.* [30] show that the single Keyword Encryption (KE) 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.* [30] prove the following theorem about the single 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:*

$$Succ^{\mathcal{A}}_{KE,S}(k) = Pr\left[\, b' = b \;\middle|\; \begin{array}{c} (params, msk) \leftarrow Init(1^k) \\ (K_u, K_s) \leftarrow KeyGen(msk, U) \\ w_0, w_1 \leftarrow \mathcal{A}^{ClientEnc(K_u,\cdot)}(K_s) \\ b \xleftarrow{R} \{0,1\} \\ c_i^*(w_b) = ClientEnc(x_{i1}, w_b) \\ b' \leftarrow \mathcal{A}^{ClientEnc(K_u,\cdot)}(K_s, c_i^*(w_b)) \end{array} \right] \tag{3.1}$$

$$< \tfrac{1}{2} + negl(k)$$

Proof. See Theorem 1 in [30].

3.6.2 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: $\vec{C}^{(i)} = C(r_0^1), \ldots, C(r_0^i), C(r_1^{i+1}), \ldots, C(r_1^t)$. The success probability of the adversary in distinguishing the encryption of the two lists of roles is defined as:

$$Succ_{\mathcal{A}}(k) = \frac{1}{2}Pr[A(\vec{C}^0) = 0] + \frac{1}{2}Pr[A(\vec{C}^t) = 1] \tag{3.2}$$

In the following, we show that breaking the RA scheme reduces to breaking the KE game. In the KE game from [30], 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 $i \xleftarrow{r} [1, t]$. It then uses r_0^i, r_1^i to challenge the single keyword encryption KE game. The adversary gets back c_b^i as the result, where c_b^i is the encryption of either r_0^i or r_1^i. $\mathcal{A}'$ uses this result to construct a hybrid vector $(c_0^1, \ldots, c_0^{i-1}, c_b^i, c_1^{i+1}, \ldots, c_1^t)$ 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:

$$Succ^i_{\mathcal{A}}(k) = \frac{1}{2}Pr[A(\vec{C}^{(i)}) = 0] + \frac{1}{2}Pr[A(\vec{C}^{(i-1)}) = 1] \tag{3.3}$$

Since i is randomly chosen, it holds that:

$$\begin{aligned} Succ_{\mathcal{A}'}(k) &= \sum_{i=1}^{t} Succ^{i}_{\mathcal{A}}(t) \cdot \tfrac{1}{t} \\ &= \tfrac{1}{2t} Pr[A(\vec{C}^0) = 0] + \sum_{i=1}^{t-1}(Pr[A(\vec{C}^i) = 0] \\ &\quad + Pr[A(\vec{C}^i) = 1]) + \tfrac{1}{2} Pr[A(\vec{C}^t) = 1] \\ &= \tfrac{1}{t}(\tfrac{1}{2} Pr[A(\vec{C}^0) = 0] + \tfrac{1}{2} Pr[A(\vec{C}^t) = 1]) + \tfrac{t-1}{2t} \\ &= \tfrac{1}{t} Succ_{\mathcal{A}}(k) + \tfrac{t-1}{2t} \end{aligned} \tag{3.4}$$

Because the success probability of $\mathcal{A}'$ to break the single keyword encryption scheme KE 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.

3.6.3 Security of Algorithms in the Policy Evaluation Phase

We now analyse the security of SR, SP, CCE and SRH. These algorithms require the Service Provider to take some client input (i.e., trapdoors computed using Algorithm 2.9), process it (i.e., re-encrypt it using Algorithm 2.10), 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 Service Provider. We follow the concept of non-adaptive indistinguishability security introduced for encrypted databases by [38] and adapted by [30] 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 Service Provider. 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 Service Provider 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 Service Provider 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 Service Provider 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 Service Provider 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_{u_i} as: $V_{K_{u_i}}(q^{u_i}) = tr(q^{u_i})$*

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

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

$$Pr\left[\, b' = b \;\middle|\; \begin{array}{l} (params, msk) \leftarrow Init(1^k) \\ (K_u, K_s) \leftarrow KeyGen(msk, U) \\ \mathcal{H}_{io}, \mathcal{H}_{i1} \leftarrow \mathcal{A}(K_s) \\ b \xleftarrow{R} \{0,1\} \\ b' \leftarrow \mathcal{A}(K_s, V_{K_u}(\mathcal{H}_{ib})) \end{array} \right] < \frac{1}{2} + negl(k) \tag{3.5}$$

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:

$$\begin{aligned} Succ^{\mathcal{A}}(k) &= \tfrac{1}{2}Pr[\mathcal{A}(RA(\vec{L}_0), TD(\vec{r}_0)) = 0] + \\ &\quad \tfrac{1}{2}Pr[\mathcal{A}(RA(\vec{L}_1), TD(\vec{r}_1)) = 1] \\ &< \tfrac{1}{2} + negl(k) \end{aligned} \tag{3.6}$$

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 3.1 to encrypt it as $RA(L_j^{u_j})$.
  - For each Search Role request $q_j^{u_j}$, run **ClientTD** (Algorithm 2.9) 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 RA 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:

$$\begin{aligned} \tfrac{1}{2} + negl(k) &> Succ_{RA}^{\mathcal{A}'}(k) \\ &= \tfrac{1}{2}Pr[\mathcal{A}((RA(\vec{L}_0), TD(\vec{r}_1))) = 0] + \\ &\quad \tfrac{1}{2}Pr[\mathcal{A}((RA(\vec{L}_1), TD(\vec{r}_1))) = 1] \end{aligned} \tag{3.7}$$

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:

$$\begin{aligned} Pr[\mathcal{A}''^{f_s(.)}(1^k) = 1] = \\ \tfrac{1}{2}Pr[\mathcal{A}(RA(\vec{L}_0), TD(\vec{r}_0)) = 0] + \\ \tfrac{1}{2}Pr[\mathcal{A}(RA(\vec{L}_0), TD(\vec{r}_1)) = 1] \end{aligned} \tag{3.8}$$

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:

$$Pr[\mathcal{A}''^{f_s(.)}(1^k) = 1] = \frac{1}{2} \tag{3.9}$$

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

$$\begin{aligned} |Pr[\mathcal{A}''^{f_s(.)}(1^k) = 1] - Pr[\mathcal{A}'^{f_s(.)}(1^k) = 1]| < negl(k) \\ Pr[\mathcal{A}''^{f_s(.)}(1^k) = 1] < \tfrac{1}{2} + negl(k) \end{aligned} \tag{3.10}$$

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

$$\begin{aligned} 1 + negl(k) > \tfrac{1}{2}Pr[\mathcal{A}(RA(\vec{L}_0), TD(\vec{r}_0)) = 0] + \\ \tfrac{1}{2}Pr[\mathcal{A}(RA(\vec{L}_0), TD(\vec{r}_1)) = 1] + \\ \tfrac{1}{2}Pr[\mathcal{A}(RA(\vec{L}_0), TD(\vec{r}_1)) = 0] + \\ \tfrac{1}{2}Pr[\mathcal{A}(RA(\vec{L}_1), TD(\vec{r}_1)) = 1] \\ = \tfrac{1}{2}Pr[\mathcal{A}(RA(\vec{L}_0), TD(\vec{r}_0)) = 0] + \\ \tfrac{1}{2} + \\ \tfrac{1}{2}Pr[\mathcal{A}(RA(\vec{L}_1), TD(\vec{r}_1)) = 1] + \\ = \tfrac{1}{2} + Succ^{\mathcal{A}}(k) \end{aligned} \tag{3.11}$$

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

3.7 Performance Analysis of ESPOON$_{ERBAC}$

In this section, we discuss a quantitative analysis of the performance of ESPOON$_{ERBAC}$. In particular, we focus on performance of the modules that have been modified as compared to the ESPOON architecture presented in Chapter 2. 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.

3.7.1 Implementation Details of ESPOON$_{ERBAC}$

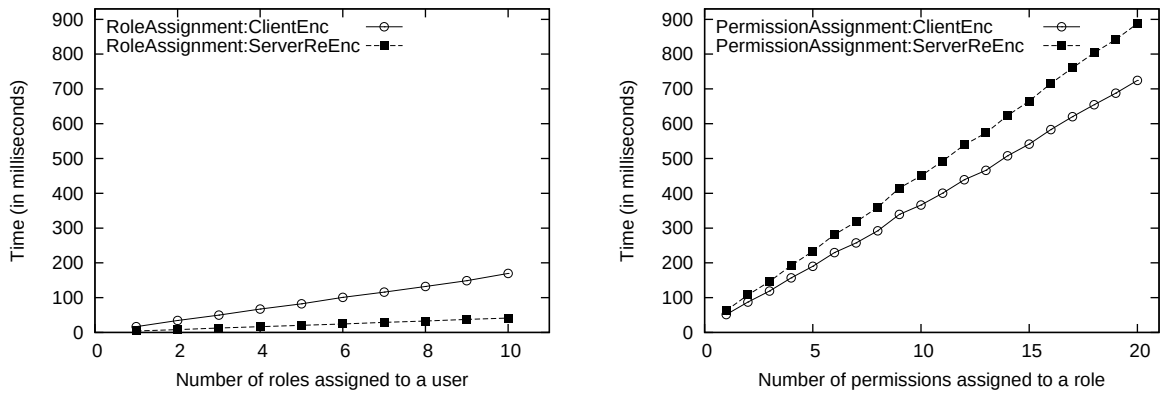
We have implemented ESPOON$_{ERBAC}$ in Java 1.6. We have developed all the components of the architecture required for performing the policy deployment and policy evaluation phases. For the cryptographic operations, we have implemented all the functions presented in Section 3.5. We have tested the implementation 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. The number of iterations performed for each of the following results is 1000.

3.7.2 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 the role assignment list, the permission assignment and the role hierarchy graph (as shown in Figure 3.6.

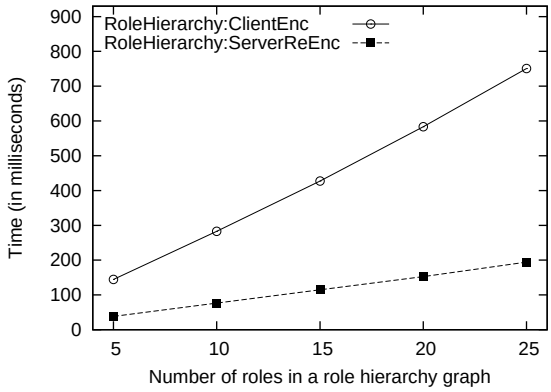
The Role Assignment List: In order to deploy a role assignment list, an Admin User performs a first round of encryption on the client side (see Algorithm 3.1) and sends the client encrypted role assignment list to the Administration Point. The Administration Point performs another round of encryption on the server side (see Algorithm 3.2) before storing the role assignment list in the Policy Store. Figure 3.6(a) shows performance overhead on the client side, as well as on the server side in order to deploy a role assignment list. In this graph, we observe the performance by increasing number of roles in a role assignment list. As we can expect, the performance overhead increases linearly with the linear increase in the number of roles in a role assignment list. As we can notice, the graph grows linearly with the linear increase in the number of roles in the role assignment list L_r. Asymptotically, the complexity of this phase is $\Theta(|L_r|)$.

During the policy deployment phase, the encryption algorithm on the client side (Algorithm 2.3) takes more time that of the server side (Algorithm 2.4) as shown in Figure 3.6. The encryption algorithm on the client side takes more time because it performs more



(a)

(b)



(c)

Figure 3.6: Performance overhead of deploying RBAC policies: (a) a list of roles assigned to a user, (b) a list of permissions to a role and (c) a role hierarchy graph

complex cryptographic operations such as random number generation and hash calculation as illustrated in Algorithm 2.3. 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 3.7.3) is of great importance.

The Permission Assignment List: For deploying permissions assigned to a role, an Admin User performs a first round of encryption on the client side (see Algorithm 3.3) 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 3.4). Figure 3.6(b) shows the performance overhead of deploying a permission assignment list. This graph illustrates the performance of deploying a permission assignment list 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 list L_p. Asymptotically, the complexity of this phase is $\Theta(|L_p|)$.

Contextual Conditions: Both the role assignment and the permission assignment lists include a contextual condition as we can see in Figure 3.2 and Figure 3.3, respectively. The performance of contextual condition is already analysed in Chapter 2, Section 2.6.2 (see Figure 2.7).

The 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 3.5). 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 3.6). 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 3.6(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 G_{RH}. Asymptotically, the complexity of this phase is $\Theta(|G_{RH}|)$.

Table 3.1: Performance overhead of encrypting requests during the policy evaluation phase

Request Type	**Time (in milliseconds)**
ACT	16.353
REQ	47.069

3.7.3 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 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 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 2.9). This trapdoor generation of role takes 16.353 ms as illustrated in Table 3.1. 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 ms as illustrated in Table 3.1. 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 the 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 3.7. Figure 3.7(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

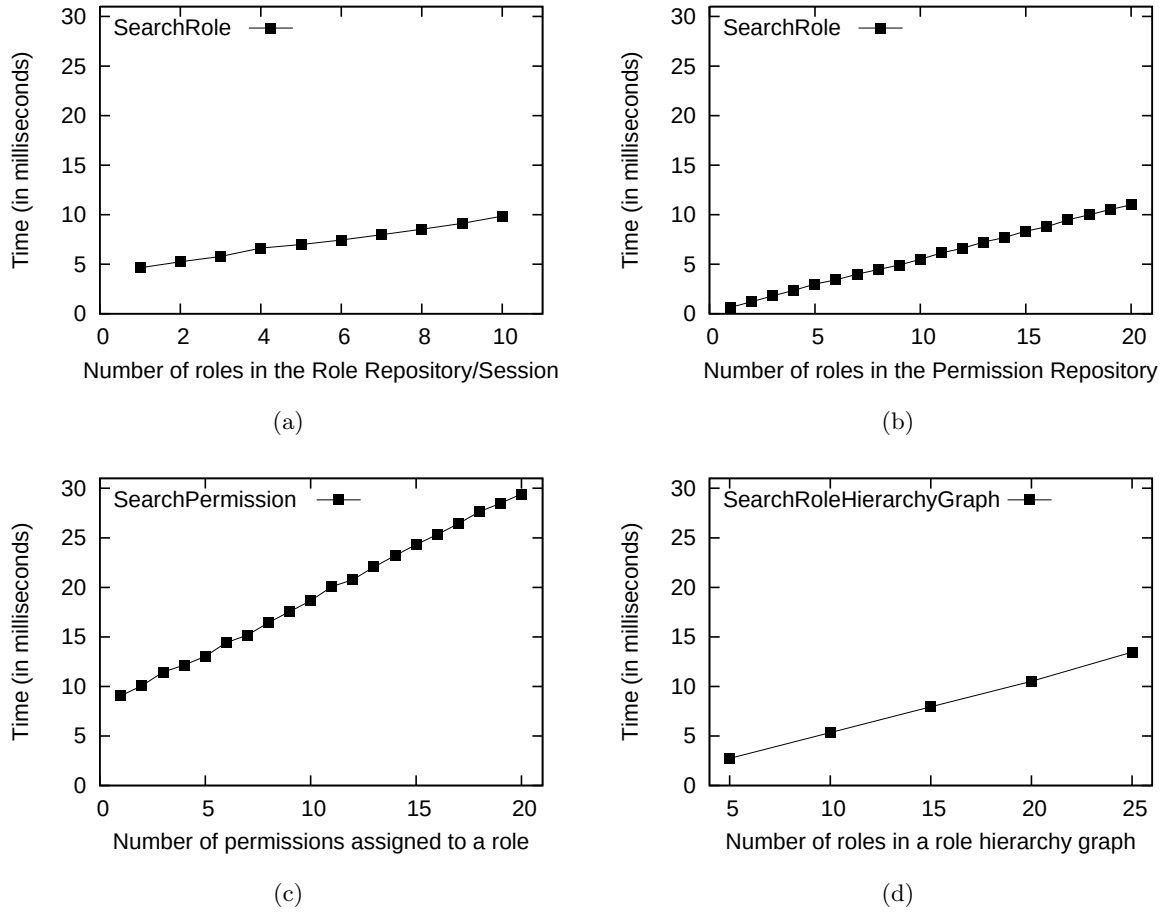


Figure 3.7: Performance overhead of evaluating RBAC policies: (a) searching roles in the Role Repository/Session, (b) searching a role in the Permission Repository, (c) checking the list of permissions assigned to a role and (d) searching a role in the role hierarchy graph

ms to generate the server encrypted trapdoor of role in ACT while it takes approximately 0.6 ms 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 3.7(a)). Asymptotically, the complexity of this phase is $O(|L_r|)$.

Searching a Role in the 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 3.7(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 3.7 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 3.7(b)) takes less time than searching a role in the Role Repository or Session (as illustrated in Figure 3.7(a)). Asymptotically, the complexity of this phase is $O(|L_r|)$.

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 3.8). 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 ms. The trapdoor generation of the requested permission, containing an action and a target, takes 8 ms. Next, the PDP match (server generated trapdoors of) this requested permission with the list of (sever encrypted) permissions assigned to the found role. Figure 3.7(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 ms. Asymptotically, the complexity of this phase is $O(|L_p|)$.

Evaluating Contextual Conditions: For evaluating the role assignment (illustrated

in Figure 3.2) or the permission assignment (illustrated in Figure 3.3) policies, the PDP may need to evaluate contextual conditions. This part has already been discussed in Chapter 2, Section 2.6.3 (see Figure 2.9).

Searching in 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 3.7(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 G_{RH}. Asymptotically, the complexity of this phase is $O(|G_{RH}|)$.

Table 3.2: Summary of time complexity of each phase in the lifecycle of ESPOON$_{ERBAC}$

Phase Name	**Complexity in the Worst Case**
Deployment of the role assignment list	$\Theta(\|L_r\|)$
Deployment of the permission assignment list	$\Theta(\|L_p\|)$
Deployment of the role hierarchy graph	$\Theta(\|G_{RH}\|)$
Searching a role	$O(\|L_r\|)$
Searching a permission	$O(\|L_p\|)$
Searching in the role hierarchy graph	$O(\|G_{RH}\|)$

Table 3.2 provides a summary of time complexities of different phases in the lifecycle of ESPOON$_{ERBAC}$.

Comparing ESPOON$_{ERBAC}$ with ESPOON: We compare the performance overheads of the policy evaluation of ESPOON$_{ERBAC}$ with that of ESPOON [28]. 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 3.3.1. 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 2.3 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 2.3) at the role activation time while the condition *RequesterName=<NAME>* can be enforced at the permission grant time.

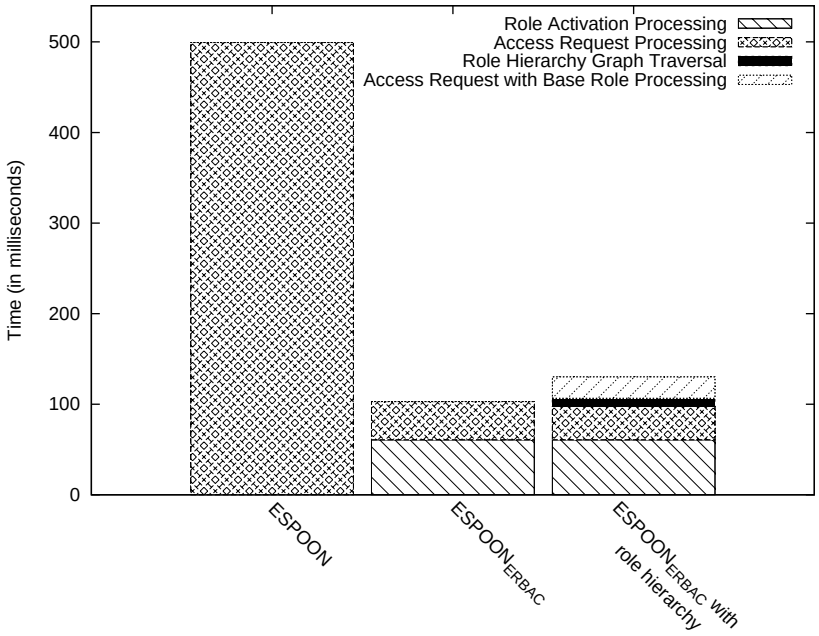


Figure 3.8: Performance comparison of ESPOON and ESPOON$_{ERBAC}$

Figure 3.8 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 ms. 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 ms 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 3.8, grating the permission takes up to 42 ms. 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 3.8 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 3.8. 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.

3.8 Chapter Summary

In this chapter, we have presented the ESPOON$_{ERBAC}$ architecture that can enforce sensitive RBAC policies in an encrypted manner, where users are assigned roles and users can execute permissions if they are active in a session that manages lists of roles different users are active in. For structuring sensitive roles within an organisation, ESPOON$_{ERBAC}$ also supports the role hierarchies in RBAC. The RBAC policy is enforced such that it does not reveal information about roles and permissions managed in the outsourced environment.

In order to cope with the real-world business requirements, Sandhu *et al.* [16] propose an RBAC constraint model that includes both static and dynamic security constraints. The static security constraints can easily be enforced by existing ESPOON$_{ERBAC}$ and ESPOON architectures. However, the challenging issue is to support dynamic security constraints in outsourced environments, where the access histories are managed by the curious Service Provider. In the next chapter, we investigate how to manage the access histories in order to enforce dynamic security constraints without leaking private information to the curious Service Provider.

Chapter 4

E-GRANT: Dynamic Security Constraints in RBAC*

Cloud computing is an emerging paradigm offering outsourced services to enterprises for storing and processing huge amount of data at very competitive costs. For leveraging the cloud to its fullest potential, organisations require security mechanisms to regulate access on data, particularly at runtime. One of the strong obstacles in widespread adoption of the cloud is to preserve confidentiality of the data. In fact, confidentiality of the data can be guaranteed by employing existing encryption schemes; however, access control mechanisms might leak information about the data they aim to protect. State of the art access control mechanisms can statically enforce constraints such as static separation of duties. The major research challenge is to enforce constraints at runtime, i.e., enforcement of dynamic security constraints (including Dynamic Separation of Duties and Chinese Wall) in the cloud. The main challenge lies in the fact that dynamic security constraints require notion of sessions for managing access histories that might leak information about the sensitive data if they are available as cleartext in the cloud. In this chapter, we present E-GRANT: an architecture able to enforce dynamic security constraints without relying on a trusted infrastructure, which can be deployed as Software-as-a-Service (SaaS). In EnforcinG encRypted dynAmic security constraiNts in The cloud (E-GRANT), sessions' access histories are encrypted in such a way that enforcement of constraints is still possible. As a proof-of-concept, we have implemented a prototype and provided a preliminary performance analysis showing a limited overhead.

*The final version of this chapter will appear in [89].

4.1 Introduction

With its cost-effective model, cloud-based services are very attractive for enterprises and government sectors. Initially developed as a cheap storage solution (monthly \$0.085/GB and \$0.095/GB, as of October 2013, offered by Google [2] and Amazon [3], respectively), the cloud paradigm today is able to offer affordable software solutions. The term Software-as-a-Service (SaaS) is used to indicate software products offered as a service through the cloud. Several vendors have adopted this model to offer their products at a more affordable price. Classes of software products available as SaaS range from document management tools (such as Google Drive [90]) to image processing tools (such as Adobe Photoshop [91]). Recently, even Business Process Management (BPM) solutions have become available as SaaS from major players in this field, such SAP with its Business ByDesign [92]. BPM solutions are at the core of modern organisations to coordinate the activities within their departments and streamline customers' requests. As empirical studies have demonstrated [93], the use of BPM solutions increases the productivity of the organisation and customer satisfaction.

One of the crucial aspects of BPM systems is the enforcement of access control decisions for assigning human resources to execute tasks within a business process. If this control is too restrictive then it could hamper the productivity of the overall business process. On the other hand, a very lax approach might undermine the confidentiality of sensitive data (when accessed by unauthorised users), resulting in serious consequences for the organisation. In a BPM system, the access control mechanism has to take into account business-related notions such as conflict-of-interests. Typical examples are that of an employee able to execute two tasks that might lead to fraudulent actions and that of an employee executing the same task over two different sets of data that could be in conflict with each other. Over the years, a huge amount of research effort has been put on this topic. The results have culminated in identifying and enforcing dynamic security constraints [16, 94–96].

If dynamic security constraints are to be correctly enforced, the system needs to maintain history of all actions executed by the entities that it controls, as well as contextual information of the requester (e.g., time and location). When the system receives a new request, it checks whether allowing the current request violates any constraints in view of the earlier actions performed by the same (group of) requesters. State of the art enforcement techniques [80, 97–99] rely on a trusted infrastructure, which expects information to be in cleartext. That is, the history of actions, contextual information, and constraints are all stored in cleartext to be readily accessible.

With the move towards outsourced solutions, the trust assumptions in the manage-

ment of the infrastructure do not hold any longer. The cloud providers that have control over the hardware, where data and security constraints are deployed (and enforced), could easily have access to them. The data can be protected using encryption techniques; however, state-of-the-art enforcement techniques [80, 97–99] cannot preserve confidentiality of dynamic security constraints because they expect all information in cleartext at both deployment and enforcement times. The problem here is that learning about the security constraints might leak information about the data itself. There are some cryptographic techniques that can enforce static security constraints in outsourced environments [20, 28, 33, 75, 76]. Unfortunately, there is no cryptographic solution that can enforce dynamic security constraints in the cloud.

4.1.1 Research Contributions

In this chapter, we want to fill this gap and propose an enforcement mechanism for dynamic security constraints that can be offered either as a stand-alone SaaS solution or integrated with other SaaS products that require the enforcement of these constraints. The main idea is to outsource the enforcement of constraints without revealing sensitive information to the untrusted infrastructure. To the best of our knowledge, we are first to address the problem of enforcing dynamic security constraints in outsourced environments. We named our solution E-GRANT. E-GRANT can enforce constraints while taking into account contextual information (such as time and location of the user) without revealing any information to cloud providers. In our mechanism, an administrator can specify constraints with contextual conditions including non-monotonic boolean expressions and range queries. In E-GRANT, constraints as well as session information are encrypted. The encryption scheme we use is such that it does not require users to share any encryption keys. In case a user leaves the organisation, the system is still able to perform its operations without requiring re-encryption of constraints or access histories managed by the session. Finally, we have implemented a prototype of E-GRANT and analysed its performance to quantify the incurred overhead.

4.1.2 Chapter Outline

The rest of this chapter is organised as follows. Section 4.2 reviews the related work. Section 4.3 provides an overview of the dynamic security constraints supported in E-GRANT. Section 4.4 describes the E-GRANT architecture. Section 4.5 and Section 4.6 focus on solution and algorithmic details of the E-GRANT architecture, respectively. In Section 4.7, we provide details about information disclosure in E-GRANT and the type of collusion attack that our solution is subjected to. Section 4.8 describes implementation details

and analyses the performance overhead of the E-GRANT prototype. Finally, Section 4.9 concludes this chapter.

4.2 Related Work

There is a significant amount of research on enforcing dynamic security constraints including Dynamic Separation of Duties (DSoD) [16, 94, 95, 100–102] and Chinese Wall (CW) [96, 103]. State of the art solutions including *RCL 2000* [98], *GTRBAC* [80], *MFOTL* [104] and [97, 99, 105, 106] mainly focus on formally specifying the constraints. They assume a trusted infrastructure in order to enforce the constraints. There are some approaches that extend the enforcement mechanisms for taking into account contextual information such as time and location while making the access decision [78–81]. However, none of the existing approaches are applicable when the enforcement mechanism is delegated to a third party that is not trusted. These approaches operate on the constraints that are stored in cleartext. Unfortunately, these constraints may leak information about the internal policies of an organisation and can result in serious implications if not adequately protected.

There are some approaches for enforcing static security constraints in outsourced environments [20, 28, 33, 75, 76]. The idea of delegating the access control mechanism to an outsourced environment has initially been explored by De Capitani di Vimercati *et al.* in [20] and extended it in [33]. Their proposed solution is based on the key derivation method [45], where each user has a key capable of decrypting resources she is authorised to access. The main drawback of this type of approaches is that they tightly couple security policies with the enforcement mechanism; therefore, any changes in the security policies require to generate new keys and to redistribute them to the users.

In [28], we propose ESPOON that aims at providing a clear separation between security policies and the enforcement mechanism. ESPOON enforces authorisation policies in outsourced environments. In ESPOON, a data owner may attach an authorisation policy with her data while storing it on the server running in the outsourced environment. A data consumer may request for the data and get access if the authorisation policy corresponding to the requested data is satisfied, where the evaluation is performed also by the server running in the outsourced environment. ESPOON does not consider concept of roles at all. In [75, 76], we extend ESPOON for supporting an encrypted version of the RBAC model and propose ESPOON$_{ERBAC}$. Users can be associated to roles and get access rights based on the role hierarchies that are managed by the server. In ESPOON$_{ERBAC}$, it is possible to enforce static security constraints, such as static separation of duties; however, it is not possible to delegate the enforcement of dynamic security constraints, such as

History-Based Dynamic Separation of Duties (HBDSoD) and CW. The main issue is that the proposed architecture in [75,76] lacks to manage encrypted session management, necessary for enforcing dynamic security constraints in outsourced environments.

The security policy enforcement is mainly based on encrypted matching schemes in untrusted environments. There are number of schemes that address encrypted matching in outsourced environments [35, 36, 38, 44, 47, 49]. Song *et al.* [35] are the first to propose an encrypted matching scheme, where documents and requests are encrypted using symmetric keys. The main drawback of this scheme is that it is a single-user scheme. Multi-user Searchable Symmetric Encryption (MSSE) [38] is the first scheme to support encrypted matching in multi-user settings. In the MSSE scheme, a data owner controls the search access by granting and revoking the search privileges to the users within her group by employing the symmetric encryption. The issue with scheme is that it requires redistribution of secret to all users once a user is revoked. Boneh *et al.* [36] are the first to propose the encrypted matching scheme in the public settings; however, it is not a multi-user scheme. Shao *et al.* [44] introduce Proxy Re-Encryption with keyword Search (PRES) scheme that is a combination of proxy re-encryption and Public-key Encryption with Keyword Search (PEKS). In PEKS, a delegation key is generated for the target user. The target user re-encrypts the ciphertext with the delegated key. The re-encryption algorithm outputs another ciphertext corresponding to the public key of the target user. That is why, this scheme high performance overhead for re-encrypting ciphertext.

There are schemes based on ABE including CP-ABE [47] and KP-ABE [49]. In CP-ABE, policies are attached with ciphertext; while, in KP-ABE, attributes are attached with ciphertext. The main issue is that both schemes leave policies and attributes in cleartext, respectively. Unfortunately, policies and attributes in cleartext may reveal private information about the encrypted data.

4.3 Dynamic Security Constraints in E-GRANT

E-GRANT focuses mainly on enforcing dynamic security constraints. There are two variants of dynamic security constraints: (i) DSoD [16, 94, 95] and (ii) CW [96]. Both DSoD and CW can be implemented by maintaining access history for each entity active in the system [107]. At each new request, the system has to check that none of the defined constraints are violated by granting the received request with respect to the earlier actions performed by the same (group of) requesters. With each variant of constraints, it is possible to specify contextual conditions i.e., enforcing constraints while taking into account contextual information, such as time and location of the requester. In the following, first we briefly explain both variants and then we describe contextual conditions.

4.3.1 Dynamic Separation of Duties

DSoD constraints [16, 94, 95] aim at providing multi-user control over the resources when there is any conflict-of-interest for completing a business process. In the following, we provide a brief description of each category of DSoD varying from coarse-grained to fine-grained levels, as discussed in [108].

Simple Dynamic Separation of Duties (SDSoD) In SDSoD, a user may be a member of two mutually exclusive roles but must not be active in both roles simultaneously.

Object-Based Dynamic Separation of Duties (ObDSoD) In ObDSoD, a user may be active in mutually exclusive roles simultaneously, but must not act in both roles upon a single object.

Operational Dynamic Separation of Duties (OpDSoD) In OpDSoD, a user may be active in mutually exclusive roles simultaneously, but must not get authorised to execute all actions of a business process.

HBDSoD In HBDSoD, a user may be active in mutually exclusive roles simultaneously, but the user must not get authorised to execute all actions of a business process involving the same object. For example, a user active in both clerk and manager roles can either issue or approve a particular instance of the *purchase order*. HBDSoD combines ideas behind ObDSoD and OpDSoD, requiring a detailed access history on each object. Thus, it is the most fine-grained category of DSoD.

4.3.2 Chinese Wall

A CW constraint [96] prevents users to access an object belonging to a domain which is in conflict-of-interest with other domain whose object is previously accessed by the same (group of) users. In other words, a CW constraint aims at providing confidentiality by preventing illegitimate information flow between domains that are in conflict-of-interest. For instance, let us consider the consultant organisation that provides services to companies that are in conflict-of-interest, say Google and Microsoft. The CW constraint will help the consultant organisation to enforce the policy that an employee can work at either Google or Microsoft but cannot work at both companies.

4.3.3 Contextual Conditions

In E-GRANT, both DSoD and CW constraints can be enforced under a certain context [28, 75, 78–81]. The context can be specified as contextual conditions, which are evaluated

at runtime by collecting contextual information. Usually, contextual information includes, but not limited to, the requester's location and the access time. As an example of a HBDSoD constraint with contextual conditions, we can consider the case where a user active in two mutually exclusive roles. For instance, two roles clerk and manager cannot issue and approve the same instance of the purchase order *on the same day from the same sub-office*.

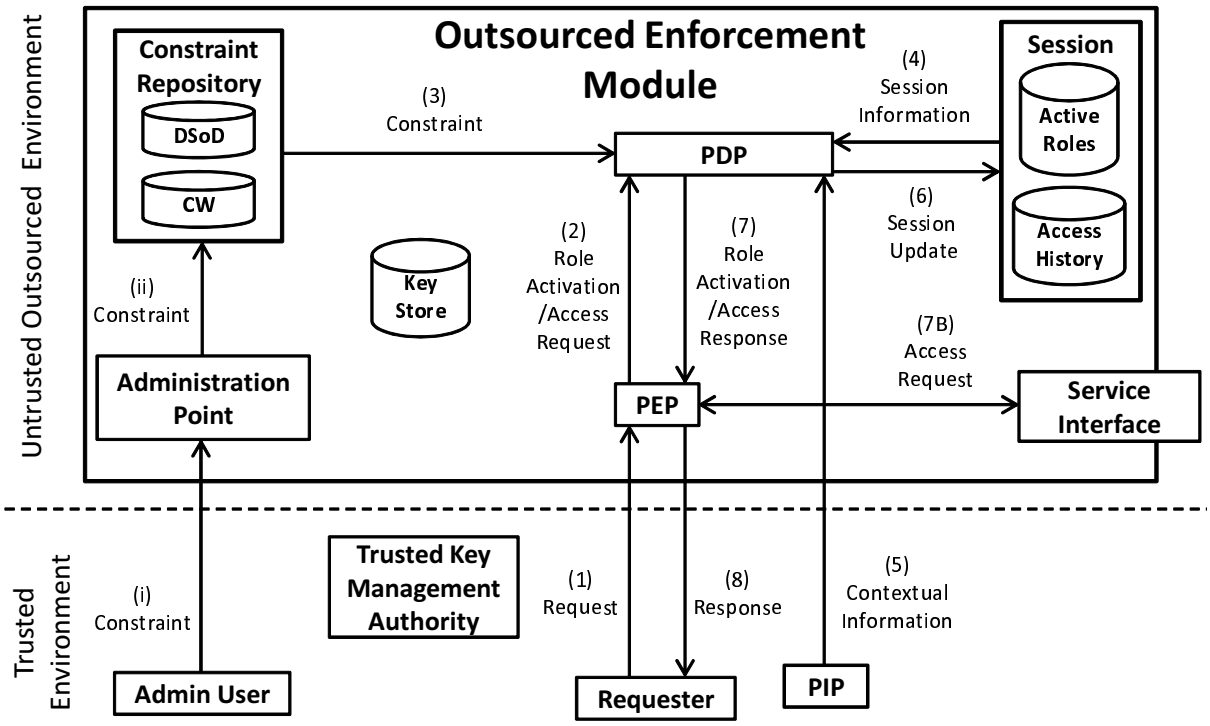


Figure 4.1: The E-GRANT architecture for enforcing dynamic security constraints in outsourced environments

4.4 The E-GRANT Architecture

The E-GRANT architecture aims at enforcing dynamic security constraints in outsourced environments in such a way that contents of constraints, contextual conditions, session information for maintaining access histories and contents of the request are not revealed to cloud providers because they are encrypted. Therefore, the enforcement mechanism can be deployed in the cloud without the need of fully trusting administrators of cloud providers. Our main goal here is to protect the confidentiality of information used by the enforcement mechanism for taking its access control decisions. The rationale behind

this is that even if the data is protected (e.g., encrypted) a curious administrator might learn information about the data by inspecting the constraints and access histories that are typically deployed in cleartext. Figure 4.1 illustrates the E-GRANT architecture containing the following entities:

Admin User An Admin User is responsible for deploying, updating and deleting dynamic security constraints.

Requester A Requester is a user that can make requests to access resources and execute actions in the system.

Outsourced Enforcement Module (OEM) It is responsible for storing and enforcing dynamic security constraints. In E-GRANT, the OEM is deployed as SaaS in the outsourced environment, managed by the cloud provider. We assume that the cloud provider is *honest-but-curious* (as assumed in [20, 33]): that is, it allows the components to follow the protocol for performing requested actions but curious to deduce information about contents of constraints, access histories and requests.

Trusted Key Management Authority (TKMA) The TKMA is a trusted authority responsible for generating keys used for protecting data stored on the OEM. For each user (be it an Admin User or a Requester), the TKMA generates the client key set and the server key set that are sent to the user and the OEM, respectively. The OEM stores all server side key sets in the Key Store and is responsible for revoking users. The TKMA is only the minimal infrastructure that is run within a trusted environment. However, the TKMA can be kept offline because it generates the key only once when any user gets registered with the system.

In E-GRANT, an Admin User can deploy new constraints and update (or delete) existing constraints. For deploying new constraints, an Admin User sends the (i) Constraint to the OEM as shown in Figure 4.1. The Administration Point is a component of the OEM that receives (i) and then stores it in the Constraint Repository (ii), which is managed by the OEM.

A Requester can send a (1) Request to the OEM as illustrated in Figure 4.1. The PEP of the OEM receives (1) and then identifies whether (1) is a role activation request or an access request. The PEP forwards the (2) Role Activation/Access Request to the PDP of the OEM. The PDP is the core component that can grant the request after evaluating the deployed constraints. For evaluating constraints, the PDP fetches the (3) Constraint from the Constraint Repository and the (4) Session Information from the Session component of the OEM. The Session component maintains two repositories including Active Roles and the Access History. Active Roles is a repository that keeps record of roles that have

been activated for a Requester while the Access History is a repository that maintains what information has been accessed by a Requester. The Session Information can include information about active roles or the access history; thus, it plays a vital role in evaluating the constraints.

The constraints could be enforced under some contextual conditions. A PDP evaluates contextual conditions after collecting contextual information, such as time and information about the Requester, e.g., her location. The Policy Information Point (PIP) is a trusted entity that provides (5) Contextual Information to the PDP. The contextual information must satisfy contextual conditions for the successful enforcement of constraints.

After the evaluation, the PDP sends the (7) Role Activation/Access Response to the PEP. The response in (7) is either *allow* or *deny* depending on the PDP evaluation as explained in Section 4.5. In case of *allow*, the PDP updates the session with the role activation or access information by sending the (6) Session Update message to the Session. The PDP forwards its decision to the PEP. If the decision is *allow*, the PEP forwards (7B) Access Request to the Service Interface. Finally, the PEP may send the (8) Response to the Requester.

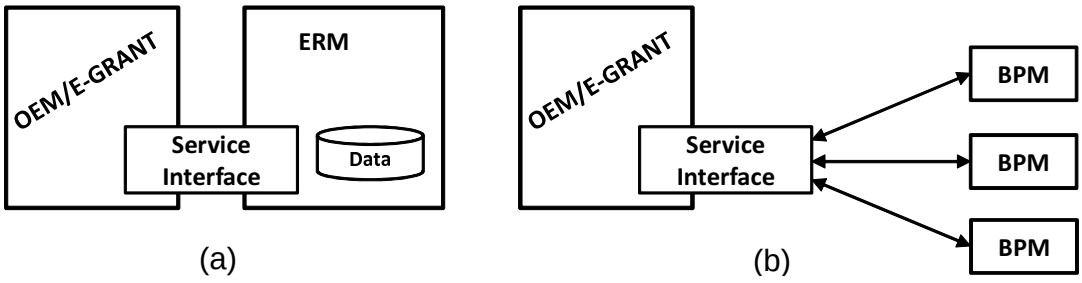


Figure 4.2: Integration of E-GRANT with other services by (a) directly importing the Service Interface (b) remotely invoking the Service Interface

The Service Interface is a programmable interface that can be used for integrating E-GRANT with other services. The Service Interface can be used as an entry point for forwarding access requests to the PEP for other services. Figure 4.2 shows two possible configurations. In Figure 4.2(a), the E-GRANT OEM is integrated with an Enterprise Resource Management (ERM) SaaS. In this scenario, the OEM can be used for receiving users' requests, enforcing security constraints and forwarding the granted requests to the ERM. Another option is shown in Figure 4.2(b), where several BPM SaaS instances remotely invoke the Service Interface of the OEM for making access control requests. It should be noted that the mechanisms used by other services to protect their data is out of the scope of E-GRANT. E-GRANT is solely responsible for the enforcement of encrypted security constraints. In the following section, we will provide a detailed description on

how encrypted security constraints are deployed and enforced by the OEM.

4.5 Solution Details of E-GRANT

E-GRANT aims at enforcing dynamic security constraints in outsourced environments. The main idea behind E-GRANT is to employ the encryption scheme for protecting constraints and the sessions while delegating the enforcement mechanism to the OEM. The encryption scheme is based on the proxy re-encryption proposed by Dong *et al.* [30]. Due to lack of space, we omit details of some operations (including enforcement of SDSoD, ObDSoD and OpDSoD) and cover the most complex operations offered by E-GRANT including enforcement of HBDSoD and CW. In the following, we describe how constraints, as well as requests are represented and then we provide technical details for enforcing constraints in an encrypted way.

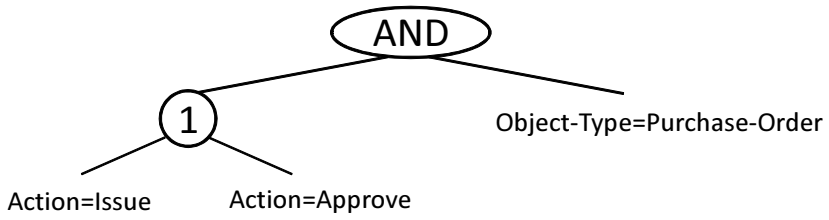


Figure 4.3: An example of HBDSoD where a Requester's *action can be 1-of-(Issue,Approve) AND Object-Type is Purchase-Order*

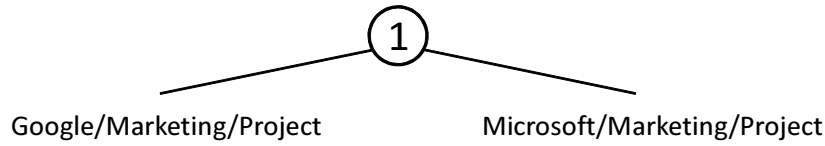


Figure 4.4: An example of CW illustrating two domains that are in conflict-of-interest

4.5.1 Representation of Constraints

For representing both DSoD and CW constraints, we use the tree structure proposed by Bethencourt *et al.* in [47], which they used for representing CP-ABE policies. Internal nodes of the tree represent AND, OR or threshold gates (e.g., 2 out of 3) while leaf nodes represent values of the condition predicates of a constraint. Figure 4.3 illustrates an example of the HBDSoD constraint, where a Requester can execute either *issue* or

approve but not both actions on the same instance of the *purchase order*. Similarly, we can express the CW constraint. Figure 4.4 illustrates an example of the CW constraint, where a Requester can work exclusively on instance of either Google's marketing project or Microsoft's marketing project.

4.5.2 Representation of a Request

The access request can be represented as a tuple $REQ = \langle R, A, O, I \rangle$, where R is role of the Requester, A indicates the action to be taken, O and I describe type of the object being accessed and its instance identifier, respectively. For instance, consider a Requester, active in a role *manager*, takes the *approve* action over the instance of a *purchase order*. The object type O may be a fully qualified name that may include the domain hierarchy an object type may belong to. For example, consider a CW constraint, where a Requester (employed by a consultant organisation) cannot work on instances belonging to both Google's marketing project and Microsoft's marketing project. Here, the object type O is *Project* while the domain hierarchy is: *Google/Marketing* and *Microsoft/Marketing*. In case, if it is the role activation request then a Requester just needs to send her role. Thus, the access request is more complex than the role activation request; therefore, we will focus more on the access request in rest of the chapter.

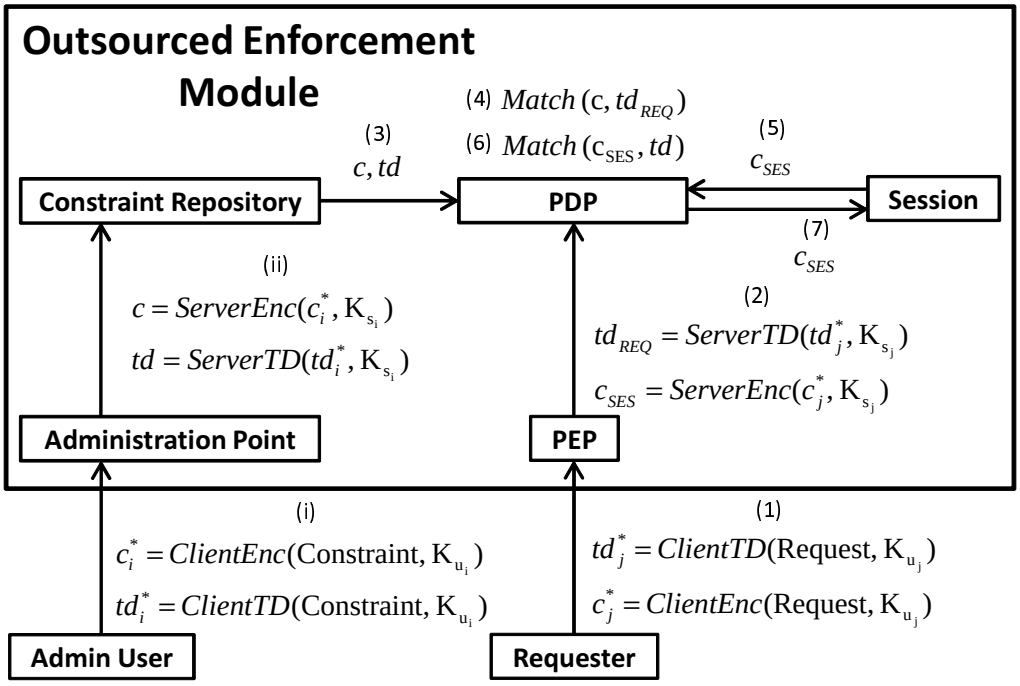


Figure 4.5: The detailed E-GRANT architecture

4.5.3 Technical Details of E-GRANT

In this section, we provide technical details of the E-GRANT architecture as illustrated in Figure 4.5. The detail of the algorithms in Figure 4.5 can be found in Chapter 2 (Section 2.5) while the detail of each phase in the enforcement lifecycle of dynamic security constraints can be found in Section 4.6.

Initialisation: E-GRANT is based on the proxy re-encryption scheme proposed by Dong *et al.* [30], where each user (including an Admin User and a Requester) gets a client side key set from the TKMA while the OEM as a proxy server also receives a server side key set corresponding to that user. The OEM maintains all these key sets in a Key Store, which can be accessed by different components of the OEM including the Administration Point, the PDP and the PEP.

Constraint Deployment: For deploying a constraint, an Admin User performs the first round of encryption using the client side key set. In this round of encryption, each leaf node of the constraint tree is encrypted while non-leaf nodes representing AND, OR or threshold gates are in cleartext. Next, an Admin User sends the user encrypted tree to the Administration Point of the OEM as shown in Figure 4.5 Step (i). After the first round of encryption, constraints are protected but they cannot be enforced yet as they are not in common format. To convert constraints into a common format, the Administration Point of the OEM performs the second round of encryption using the server side key set corresponding to the same Admin User who performed the first round of encryption as shown in Figure 4.5 Step (ii). In fact, the second round of encryption by the Administration Point serves as a proxy re-encryption. The common format implies that the constraints get encrypted with the master secret key, which is known neither to any users nor to the OEM. Like the first round of encryption, each leaf node of the tree representing the security constraint is re-encrypted. Finally, the re-encrypted constraints are stored by the Constraint Repository.

If an encrypted request satisfies any encrypted deployed constraint (i.e., Figure 4.5 Step (4)), then the session information is required to be matched against elements of the constraint (i.e., Figure 4.5 Step (6)). That is, the session information is matched with those elements of the constraint that are not present in the request. For example, let us consider the SDSoD constraint, where a user may be a member of two mutually exclusive roles clerk and manager but must not be active in both roles simultaneously. Let us assume that the requester's role is clerk. Since the requester's role is matched against the same role in the constraint, the OEM will consult the session to check if the same user is active in manager's role. For performing such a check, OEM requires trapdoors of

the constraint because only trapdoors could be matched with the encrypted information. That is why, trapdoors are stored along with the encrypted constraint at deployment time. For calculating these trapdoors, an Admin User performs the first round of trapdoor generation using the client side key set for each leaf node in the request (as shown in Figure 4.5 Step (i)) while the OEM performs the second round of trapdoor generation using the server side key set corresponding to that Admin User (as shown in Figure 4.5 Step (ii)). The trapdoor representation does not leak any information.

Making a Request: For making a request, a Requester generates *REQ* and transforms it into trapdoors using the client side key set for each element in the request. That is, there is a trapdoor for each element in *REQ*. Finally, *REQ* is sent over to the PEP of the OEM as shown in Figure 4.5 Step (1).

Constraint Evaluation: The deployed constraints are checked when the OEM receives a request from any Requester. The request is not in the common format yet and requires another round of the trapdoor generation. In the second round of trapdoor generation, the PEP generates the server side trapdoors for each element in *REQ* (i.e., Figure 4.5 Step (2)). After completing the second round of trapdoor generation, the PEP forwards the request to the PDP. The PDP fetches encrypted constraints from the Constraint Repository (i.e., Figure 4.5 Step (3)) and matches it against the encrypted request (i.e., Figure 4.5 Step (4)). If the constraint is satisfied, then certain elements of the constraint (i.e., all elements except one that is present in the request) are required to be matched against the session information.

Contextual Conditions: Optionally, constraints may include contextual conditions (already discussed in detail in Chapter 2). For the evaluation of contextual conditions, the PDP might require contextual information, which is fetched from the PIP. The PIP performs the first round of trapdoor generation using the client side key set[1]. Let us consider that the required contextual information is current office hour and location of the Requester. We represent each string attribute as a single element. The numerical attributes are represented as a bag of bits, where each numerical attribute of size s-bit is represented by s elements (in the worst case). For the simplicity, we assume that there are total 8 office hours (from 9:00 AM to 5:00 PM) that can be represented with three bits. For instance, the first office hour can be represented as: $t : **0$, $t : *0*$ and $t : 0**$; and the last (8th) office hour can be represented as: $t : **1$, $t : *1*$ and $t : 1**$. Similarly,

[1]The PIP is considered as a user and gets the client side key set in the same way as a normal user (an Admin User or a Requester) does.

the location of the Requester can be represented as: *location* : *office*. While performing the first round of trapdoor generation, a trapdoor is generated for each element of contextual information. For instance, in the example where contextual information includes office hours (say the first hour) and location of the Requester (say office), a trapdoor is generated for each element including $t : **0$, $t : *0*$, $t : 0**$ and *location* : *office*. After performing the first round of trapdoor generation, the PIP sends contextual information to the PDP. The PDP performs the second round of trapdoor generation for each element of contextual information so that a match can be performed.

While performing the encrypted match between the encrypted session information and the encrypted constraint/request, the OEM does not reveal contents. If contextual information is required to be matched, it is matched in the same way as other elements of the constraint/request are matched against the session information. After checking the session information (i.e., Figure 4.5 Step (6)), if the constraint is not satisfied, the access is permitted and the role activation (or the access) response is sent from the PDP to the PEP as *allow*. Otherwise, the access is denied and the role activation (or the access) response is sent from the PDP to the PEP as *deny*.

Updating the Session: If the evaluation is successful, the PDP updates the session to maintain the access history, as well as active roles. For updating the session, the PDP requires the request (and contextual information). The Requester may send encrypted request along with the trapdoors of the request as shown in Figure 4.5 Step (1). Alternatively, the PDP/PEP can collect this information after the PDP evaluation is succeeded. In both cases, the OEM performs the second round of encryption and finally updates the Session with the encrypted request as shown in Figure 4.5 Step (7). If the requested action is the access request, the PEP additionally forwards it to the Service Interface. Finally, the PEP may send a response to the Requester.

User Revocation: In E-GRANT, users (both Admin Users and Requesters) do not share any keys and even if a compromised user is removed, there is no need to re-encrypt deployed constraints or re-distribute keys. For removing a user from the system, the Administration Point of the OEM takes the user identifier and then removes the server side key corresponding to that user from the Key Store.

4.6 Algorithmic Details of E-GRANT

In this section, we identify all phases describing the enforcement lifecycle of dynamic security constraints in outsourced environments. For each of these phases, we list all of

its algorithms in detail. In fact, these algorithms constitute the proposed schema that is based on [30].

4.6.1 The Initialisation Phase

In this phase, the system is initialised by the TKMA. During the system initialisation, the system level master key and public parameters are generated. This phase consists of only one algorithm called **Init** illustrated in Algorithm 2.1. After running this algorithm, the TKMA publicises the public parameters $params = (\mathbb{G}, g, q, h, H, f)$ and keeps securely the master secret key $msk = (x, s)$.

4.6.2 The Key Generation Phase

During the key generation phase, the keying material is generated for each user including an Admin User and a Requester by the TKMA. This phase consists of only one algorithm called **KeyGen** and is illustrated in Algorithm 2.2. After running the **KeyGen** algorithm, the TKMA generates two key sets: K_{u_i} and K_{s_i} corresponding to user i. The TKMA securely transmits K_{u_i} and K_{s_i} to the user i and the OEM, respectively. Each user i receives the user side key set K_{u_i} and stores it securely as it serves as the private key for her. The Administration Point of the OEM receives the server side key set K_{s_i} corresponding to user i and inserts it in the Key Store, where the Key Store is updated as: $KS \leftarrow KS \cup K_{s_i}$. The Key Store of the OEM is initialised as: $KS \leftarrow \Phi$.

Algorithm 4.1 ClientGeneratedConstraint

Description: *It transforms cleartext constraints into the (encrypted) client generated constraints, which are sent to the Administration Point as shown in Figure 4.1 Step (i).*

Input: The constraint tree SCT, the client side key set K_{u_i} corresponding to Admin User i and the public parameters $params$.

Output: The client generated constraint tree SCT_{C_i}.

1: $SCT_{C_i} \leftarrow SCT$
2: **for** each leaf-node element e in tree SCT_{C_i} **do**
3: $c_i^*(e) \leftarrow$ call **ClientEnc** (e, K_{u_i}, $params$)
4: $td_i^*(e) \leftarrow$ call **ClientTD** (e, K_{u_i}, $params$)
5: $ug(e) \leftarrow (c_i^*(e), td_i^*(e))$
6: replace e of SCT_{C_i} with $ug(e)$
7: **end for**
return SCT_{C_i}

4.6.3 The Constraint Deployment Phase

During this phase, a constraint is deployed by an Admin User. Each constraint is deployed in two phases; therefore, this phase consists of two algorithms: Algorithm 4.1 and Algorithm 4.2 called **ClientGeneratedConstraint** and **ServerGeneratedConstraint**, respectively. The constraint is first transformed into a tree structure as already explained in Section 4.5. After performing transformation, each leaf node of this tree SCT is encrypted (by running **ClientEnc** described in Chapter 2 as Algorithm 2.3) and client generated trapdoors (by running **ClientTD** described in Chapter 2 as Algorithm 2.9) are also calculated using the client side key set K_{u_i} corresponding to Admin User i as shown in Algorithm 4.1, which is run by the Admin User. Finally, the client generated constraint SCT_{C_i} is sent over to the Administration Point of the OEM as illustrated in Figure 4.1 Step (i).

Algorithm 4.2 ServerGeneratedConstraint

Description: *It re-encrypts the client generated constraints into the server generated constraints, which are finally deployed by the Administration Point as shown in Figure 4.1 Step (ii).*

Input: The client generated constraint tree SCT_{C_i} and Admin User i.

Output: The server generated constraint tree SCT_S.

1: $K_{s_i} \leftarrow KS[i]$ ▷ retrieve the server side key corresponding to Admin User i
2: $SCT_S \leftarrow SCT_{C_i}$
3: **for** each leaf-node client generated element $ug(e) = (c_i^*(e), td_i^*(e))$ in tree SCT_S **do**
4: $c(e) \leftarrow$ call **ServerReEnc** ($c_i^*(e)$, K_{s_i})
5: $td(e) \leftarrow$ call **ServerTD** ($td_i^*(e)$, K_{s_i})
6: $sg(e) \leftarrow (c(e), td(e))$
7: replace $ug(e)$ of SCT_S with $sg(e)$
8: **end for**
return SCT_S

The Administration Point of the OEM receives the client encrypted constraint SCT_{C_i} and performs another round of encryption (by running **ServerReEnc** described in Chapter 2 as Algorithm 2.4) and the trapdoor generation (by running **ServerTD** described in Chapter 2 as Algorithm 2.10) using the server side key set K_{s_i} corresponding to Admin User i as shown in Algorithm 4.2. After running Algorithm 4.2, the Administration Point stores the server generated constraints in the Constraint Repository on the OEM as illustrated in Figure 4.1 Step (ii).

Algorithm 4.3 ClientGeneratedRequest

Description: *It transforms the cleartext request into the client generated request, which is sent to the PEP as shown in Figure 4.1 Step (1).*

Input: The request REQ containing list of elements, the client side key set K_{u_i} corresponding to Requester i and the public parameters $params$.

Output: The client generated request REQ_{C_i}.

1: $REQ_{C_i} \leftarrow REQ$
2: **for** each element e in list REQ_{C_i} **do**
3: $td_i^*(e) \leftarrow$ call **ClientTD** (e, K_{u_i}, $params$)
4: $c_i^*(e) \leftarrow$ call **ClientEnc** (e, K_{u_i}, $params$)
5: $req_i^*(e) \leftarrow (td_i^*(e), c_i^*(e))$
6: replace e of REQ_{C_i} with $req_i^*(e)$
7: **end for**
return REQ_{C_i}

4.6.4 The Request Phase

In this phase, Requester i makes a request REQ, which is enciphered using her private key set K_{u_i}. This phase consists of one algorithm called **ClientGeneratedRequest** illustrated in Algorithm 4.3 in which each element in REQ (assuming REQ also includes contextual information) is transformed into a trapdoor (by running **ClientTD** described in Chapter 2 as Algorithm 2.9). Furthermore, each element in REQ is encrypted (by running **ClientEnc** described in Chapter 2 as Algorithm 2.3) because it is required to be stored in the session provided it is granted. Finally, the client request REQ_{C_i} is sent over to the OEM.

4.6.5 The Constraint Evaluation and Session Update Phase

This is the core phase in which constraints are evaluated and the session is updated with the information within the request, provided the request is granted. This phase consists of one algorithm called **ConstraintEval-SessionUp** illustrated in Algorithm 4.4, which is run by the PEP of the OEM. After receiving the client request REQ_{C_i}, the PEP first retrieves the server side key K_{s_i} corresponding to Requester i (Line 1). The PEP then performs the second round of trapdoor generation (by running **ServerTD** described in Chapter 2 as Algorithm 2.10) for each element in REQ_{C_i} (Line 2-6). After performing the second round of trapdoor generation, the server generated request REQ_S is matched against the deployed constraint SCT_S (Line 7-12), where it is mainly checked if the encrypted tree *EncryptedTree* of the deployed constraint SCT_S is satisfied by the encrypted request REQ_S (Line 12). The detail how *EncryptedTree* is matched against

Algorithm 4.4 ConstraintEval-SessionUp

Description: *It fetches the encrypted constraints (see Figure 4.1 Step (3)), transforms the client request into the server generated request, then matches constraints with the request.*

Input: The server generated constraint tree SCT_S, the list of client generated trapdoor REQ_{C_i}, Requester i and session S.

Output: *true* or *false*.

1: $K_{s_i} \leftarrow KS[i]$ ▷ retrieve the server side key corresponding to Requester i
2: $REQ_S \leftarrow REQ_{C_i}$
3: **for** each client generated request element $req_i^*(e).td_i^*(e)$ in list REQ_S **do**
4: $td(e) \leftarrow$ call **ServerTD** ($td_i^*(e)$, K_{s_i})
5: replace $req_i^*(e).td_i^*(e)$ of REQ_S with $td(e)$
6: **end for**
7: $EncryptedTree \leftarrow SCT_S$
8: Add field *decision* to each node of *EncryptedTree*
9: **for** each node n in tree *EncryptedTree* **do**
10: $n.decision \leftarrow null$ ▷ initialise decision field with *null*
11: **end for**
12: call **CheckTreeSatisfiability** ($EncryptedTree.root$, $EncryptedTree$, REQ_S)
13: **if** $EncryptedTree.root.decision \stackrel{?}{=} true$ **then**
14: $TrapdoorList \leftarrow$ extract trapdoors from *EncryptedTree* that needs to be searched in session s
15: $record\text{-}found \leftarrow false$
16: **for** each record r in session S **do**
17: **for** each server encrypted element $c(e)$ in r to be matched with $td(e)$ in $TrapdoorList$ **do**
18: $match \leftarrow$ call **Match** ($child.c(e)$, $REQ_S.td(e)$)
19: **if** $match \stackrel{?}{=} false$ **then**
20: break;
21: **end if**
22: **end for**
23: **if** $match \stackrel{?}{=} true$ **then**
24: $record\text{-}found \leftarrow true$
25: break;
26: **end if**
27: **end for**
28: **if** $record\text{-}found \stackrel{?}{=} true$ **then return** *false*
29: **end if**
30: **end if**

▷ steps for updating session

31: $r \leftarrow \phi$
32: **for** each client encrypted request element $req_i^*(e).c_i^*(e)$ in list REQ_S **do**
33: $c(e) \leftarrow$ call **ServerReEnc** ($c_i^*(e)$, K_{s_i})
34: $r \leftarrow r \cup c(e)$
35: **end for**
36: $S \leftarrow S \cup r$ ▷ session updation
return *true*

Algorithm 4.5 CheckTreeSatisfiability

Description: *It checks whether the encrypted constraint satisfies the encrypted request.*

Input: The root node n of encrypted constraint tree *EncryptedTree* and the list of server generated trapdoors of request REQ_S.

Output: *true* or *false*.

1: **if** $n \stackrel{?}{=} null$ **then**
   return *true* ▷ if *null* constraint then it trivially satisfies the request
2: **end if**
3: **if** $n.decision \neq null$ **then**
   return $n.decision$ ▷ if decision is already made then return it
4: **end if**
5: **if** $isLeaf(n) \stackrel{?}{=} true$ **then**
6: $n.decision \leftarrow$ call **Match** ($n.c(e)$, $REQ_S.td(e)$)
   return $n.decision$ ▷ if it is leaf node then perform matching and return its decision
7: **end if**
8: $k' \leftarrow 0$
9: **for** each *child* of n in *EncryptedTree* **do** ▷ if it is non-leaf node then call this function recursively for each of its child
10: **if** call **CheckTreeSatisfiability** (*child*, *EncryptedTree*, REQ_S) $\stackrel{?}{=} true$ **then**
11: $k' \leftarrow k' + 1$
12: **end if**
13: **end for**
14: **if** ($n.gate \stackrel{?}{=} OR$ and $k' \geq 1$) or $n.k \stackrel{?}{=} k'$ **then**
15: $n.decision \leftarrow true$ ▷ set decision as *true* if (a) node's gate is OR and one of its child is satisfied or (b) the number of children n has is equal to number of satisfied elements, i.e., the case of both AND and *threshold* gates
16: **else**
17: $n.decision \leftarrow false$
18: **end if**
   return $n.decision$

REQ_S is provided in Algorithm 4.5.

If SCT_S is matched against REQ_S (Line 13), then the certain trapdoors of the deployed constraint are extracted (Line 14) and then matched against records in the Active Roles repository (in case of role activation request) or the Access History repository (in case of access request) of the session (Line 15-27). If the match (in Line 28) is successful (assuming the constraint with 1-out-of-n condition for roles or actions), no action is taken and *false* is returned (Line 28), indicating that the session is not updated; otherwise, each element of REQ_S is re-encrypted (by running **ServerReEnc** described in Chapter 2 as Algorithm 2.4) and then the session is updated with the encrypted information of active roles or the access history (Line 31-36) and *true* is returned (Line 36), indicating that the session is updated by running Algorithm 4.4.

4.7 Discussion

This section provides the discussion about security aspects of E-GRANT including information disclosure and the collusion attack.

4.7.1 Information Disclosure

In E-GRANT, a curious OEM may deduce the structure of security constraints. That is, a curious OEM may learn what gates (AND, OR and k-of-n) are used in security constraints. However, the most important information is actually contents of security constraints that are not revealed to the OEM. To partially resolve the problem of revealing structure, we may include some dummy elements in the constraint. Furthermore, a curious OEM may also deduce how many elements are present (but does not learn about contents of elements) in the request or contextual information; once again, the Requester or the PIP can include some dummy elements in order to obfuscate the number of elements present in the request or contextual information, respectively.

4.7.2 Collusion Attack

In E-GRANT, a single compromised user (either an Admin User or a Requester) may recover the master secret key by colluding with the OEM. One way to withstand the collusion attack is to split the client side key set into two parts; where, one part is given to the user while the other part is managed by the organisation gateway to access the OEM. In this case, the organisation gateway is assumed trusted. The other way to withstand the collusion attack is to consider the trusted hardware for storing the client side key set.

4.8 Performance Analysis of E-GRANT

In this section, we show the effectiveness of E-GRANT for enforcing dynamic security constraints by quantifying the performance overhead incurred by the cryptographic operations performed at both the client and the server sides. During this performance evaluation, we are not taking into account the latency introduced by the network. In the following, we first describe implementation details of the prototype we have developed. Next, we show the performance evaluation of: (i) deploying dynamic security constraints, (ii) making a request, (iii) evaluating dynamic security constraints and (iv) finally updating session with the information within the request.

4.8.1 Implementation Details of E-GRANT

We have developed a prototype of E-GRANT for enforcing dynamic security constraints. The prototype is implemented in Java 1.6. For this prototype, we have designed all the components of the architecture required for deploying and evaluating constraints. In short, we have implemented all algorithms presented in Section 4.6.

We have tested our E-GRANT prototype 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. The values of the execution time shown in the following graphs are averaged over 1000 iterations.

4.8.2 Performance Analysis of Deploying Dynamic Security Constraints

In this section, we analyse the performance of deploying dynamic security constraints. In order to deploy a constraint, an Admin User performs on the client side the first round of encryption and the trapdoor generation for each element in the constraint as explained in Section 4.5 (see Algorithm 4.1) and sends the client generated constraint to the OEM. The Administration Point of the OEM receives the client generated constraint and performs the second round of encryption and the trapdoor generation for each element in the client generated constraint (see Algorithm 4.2). Finally, the server generated constraint is sent to the Constraint Repository of the OEM.

We measure the performance of deploying both types of security constraints including HBDSoD and CW. The simplest HBDSoD constraint is defined with two actions at least, meaning a user cannot execute both actions. For increasing complexity of the HBDSoD constraint, we can consider more than two actions using the following notation: $HBDSoD(Ya)$, where Y (≥ 2) denotes the number of actions in the constraint. Similarly, the simplest CW constraint is defined at the object level, meaning a user cannot access an instance of an object whose instance has already been accessed. In order to increase the

complexity of the CW constraint, we can include the domain hierarchy. Generally, the CW constraint can be represented as: $CW(Zd/o)$, where Z (≥ 0) denotes the number of domains that may be present in the domain hierarchy. If the constraint is at the object level, the value of Z will be 0 and constraint would become CW(o). However, if the constraint includes any domains, then the value of Z will be more than 0. For instance, if there is one domain then the constraint would be represented as CW(d/o). Similarly, if there are two domains (i.e., one domain and one subdomain) in the domain hierarchy of an object then the constraint would be represented as CW(2d/o) and so on. Asymptotically, the complexities of deploying HBDSoD and CW constraints are $\Theta(Y)$ and $\Theta(Z)$, respectively.

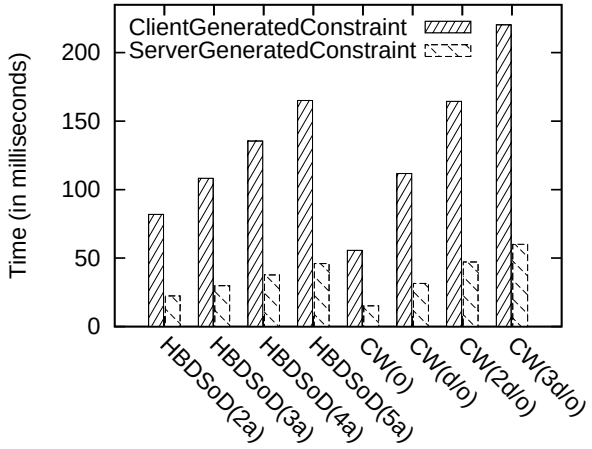


Figure 4.6: Performance overhead of deploying dynamic security constraints

Figure 4.6 indicates the performance overhead incurred by deploying constraints on both the client and the server sides. During the performance evaluation, we consider both HBDSoD and CW constraints, each with varying level of complexity, where number of actions in the HBDSoD constraint are varied from 2 to 5 (with step size 1) and number of domains in the CW constraint are varied from 0 to 3 (with step size 1), respectively. As we can expect, the performance overhead of each type of constraint grows linearly if we gradually increase its complexity. Furthermore, we can observe that algorithms on the client side take more time as compared to that of the server side for deploying any type of constraints. This is mainly due to the fact the client side performs more complex cryptographic operations such as random number generations and hash calculations (as shown in Algorithm 2.3 and Algorithm 2.9 in Chapter 2) than the respective algorithms on the server side (as shown in Algorithm 2.4 and Algorithm 2.10 in Chapter 2). However, these operations are executed only when the Admin User has to deploy a new constraint or update existing ones. On the other hand, constraints are evaluated every time a request

is made. Thus, the performance of generating requests and evaluating constraints, which are measured in the following sections, is of great importance, considering the fact that it will impact the latency for providing access to the data.

4.8.3 Performance Analysis of Generating Requests

In this section, we analyse the performance of generating access requests on the Requester's client side. To make the access request, a Requester has to generate the $REQ = \langle R, A, O, I \rangle$ tuple representing that role R is requesting to perform action A on instance I of object type O. Each element of REQ is transformed into trapdoors, necessary for performing the match against encrypted HBDSoD or CW constraints deployed on the OEM. The trapdoor representation does not leak information on elements of REQ. Furthermore, each element of REQ is also encrypted, necessary for storing the REQ tuple as encrypted in the session after REQ is granted. The time required to generate such a tuple (by running Algorithm 4.3) is around 120 ms as shown in the graph of Figure 4.7.

The PDP might need contextual information to make the decision whether the requested action is permitted based on deployed constraints. One way to provide such information is to send the required contextual information together with the REQ tuple. In this case, the client side of the Requester takes the responsibility to generate the trapdoors of contextual information. The other option is to let the PDP requests contextual information to the PIP (running in the trusted environment) when such information is needed. The former option requires fewer interactions because the PDP has already all required information. However, this comes at the price for the Requester's client side of generating extra encrypted data (the trapdoor representation for contextual information). The latter option requires more interaction since the PDP has to contact the PIP. However, this happens only if contextual information is really required by the PDP.

In our experiments, we considered case in which the contextual information is included with every REQ tuple. We selected two types of contextual information: the time and the location of the Requester. As we explained in Section 4.5, the time t is represented as three elements indicating the office hour while the location l is represented as a single string element.

The graph in Figure 4.7 shows the performance overhead incurred at the Requester's client when the REQ tuple contains the value of time t ($REQ(t)$ in the graph) and location l ($REQ(l)$ in the graph). As can be seen in the graph, when the value of time is added to the REQ tuple, there is more performance overhead to be incurred as compared to that of the location because the time value t is represented as three elements, requiring generation of three trapdoors. On the other hand, the value l of the location is represented by just a single element, requiring generation of only a single trapdoor. We also measured the

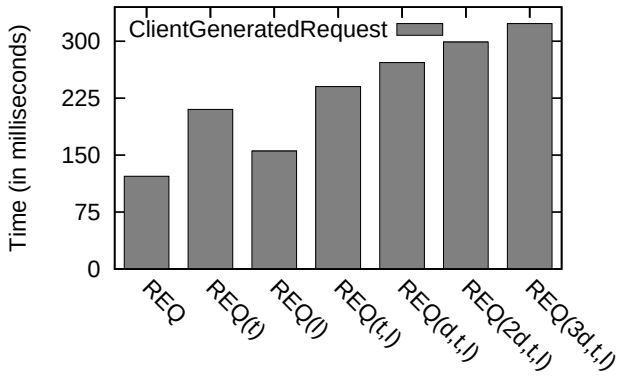


Figure 4.7: Performance overhead of generating access requests on the Requester's client side

case in which both time and location trapdoors are generated with the *REQ* tuple and the overhead is combination of two previous cases ($REQ(t, l)$ in the graph).

When CW constraints are enforced, it might be needed to include additional information about the target resource within the *REQ* tuple. This additional information is the domain hierarchy an object type may belong to. In the domain hierarchy, there may be multiple levels of domains. The trapdoors representing this information need also to be generated by the Requester's client. We performed experiments where together with the time and location, also domain information have been added to the *REQ* tuple. Moreover, we also varied the depth of the domain hierarchy from one domain level (represented as $REQ(t, l, d)$) to three levels (represented as $REQ(t, l, 3d)$). The last three values in Figure 4.7 provide the measurements for these cases. As it is quite obvious, the performance overhead of generating these requests increases linearly with the increase in domains levels. However, it should be noticed that even in the worst case (where time, location and three domain levels are inserted in the *REQ* tuple), the average time for generating a request is still below 325 ms. In the worst case, the request generation phase takes $\Theta(Z)$.

4.8.4 Performance Analysis of Evaluating Dynamic Security Constraints

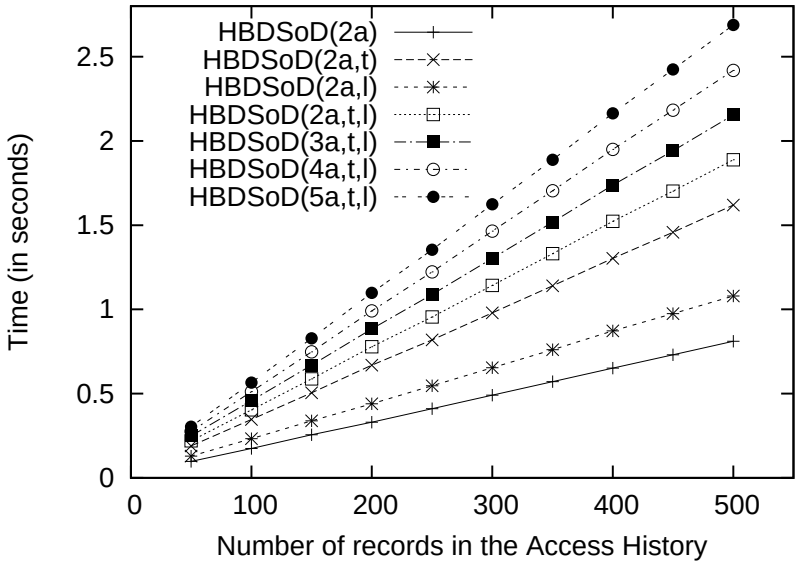
In this section, we analyse performance of evaluating security constraints on the OEM. For evaluating constraints, the request coming from the Requester is first transformed into the common format by performing the second round of trapdoor generation (see Algorithm 4.4). During the trapdoor generation, each client generated trapdoor is transformed into the server generated trapdoor as illustrated in Algorithm 2.10 of Chapter 2. This second round of encryption is necessary to perform the matching between the

trapdoors of the request and the encrypted constraints. In the following, we analyse the performance overheads of evaluating both HBDSoD and CW constraints.

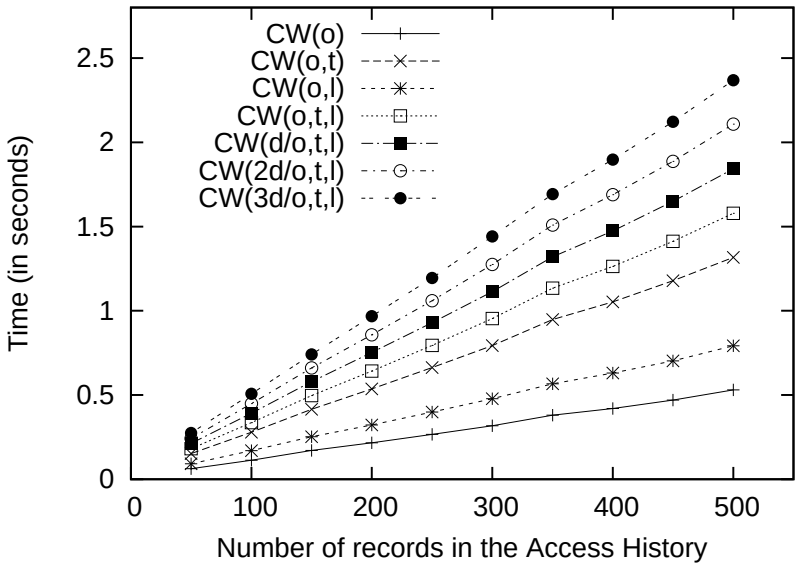
Evaluating HBDSoD Constraints: First of all, let us make a concrete example of the enforcement of HBDSoD constraints to understand what operations are executed at the OEM. Let us assume a Requester makes a request *REQ* for executing the action *approve* on the object type *purchase order*. As an example of a HBDSoD constraint, let us consider one that limits a Requester to execute only one action out of the two actions *issue* and *approve* that can be executed on a particular instance of a *purchase order*. First, the PDP matches the object type in *REQ* with the object type of the deployed constraints in the Constraint Repository. If the match is successful, the PDP will match the action in *REQ* with one of the action specified in the HBDSoD constraint. On the second successful match, the PDP has to check that the Requester has not executed the *issue* action on this specific instance of *purchase order* in the past. To perform this check, the PDP searches in the Access History to find all records where the object type and instance match with that of *REQ* tuple. If such a record is found then the PDP checks if the action value in the records matches the k-out-of-n condition of the HBDSoD constraint. In particular, in our example it means the PDP searches in the Access History to find any records containing action *approve*. If this is the case, the constraint is violated and the PDP will not grant the action. Otherwise, the Requester can *issue* the *purchase order*.

From the above example, it is clear that the performance of enforcing a constraint depends on three main factors. The first factor is the number of constraints deployed in the Constraint Repository. When a request arrives, the PDP has to find in the repository a matching constraint. Finding a matching constraint clearly depends on the number of constraints in the repository. The second factor is the number of elements specified in the constraint. These elements can include two or more actions that could be executed only once by a Requester on a given instance of an object. Moreover, also contextual information can be taken into account. Finally, the other major factor is the number of records in the Access History that the PDP has to search to check whether a given constraint is violated or not. Asymptotically, the enforcement of HBDSoD constraints takes $O(Y \cdot c \cdot r)$, where C is the number of constraints deployed in the repository and r is the number of records in the Access History.

To measure the performance overhead, we performed the following experiments. We deployed 100 different HBDSoD constraints in the repository such that the one that matches the incoming request is the last one. This, of course, represents the worst case scenario. We also believe that 100 different constraints is way beyond the typical needs



(a)



(b)

Figure 4.8: Performance overhead of evaluating dynamic security constraints (a) HBDSoD and (b) CW on the OEM

of an enterprise. To study how the complexity of the constraint specification and number of records in the Access History affect the performance of the constraint evaluation, we execute several runs of our experiments varying the constraint complexity and number of records. Figure 4.8(a) shows the evaluation time in seconds in different settings. As we can observe in Figure 4.8(a), the evaluation time increases with the increase in the number of actions in the constraint (from 2 actions up to 5) and when contextual information such as time t and/or location l of the Requester are also considered. Similarly, the evaluation time increases with the increase in the number of records in the Access History.

Evaluating CW Constraints: A CW constraint enforces that a Requester cannot gain access to two mutually exclusive objects. When a request *REQ* tuple is received, the PDP has to search the CW constraints relevant to the object type specified in the request tuple. Basically, the object type in the request tuple has to match one of the object types specified in a CW constraint. If a match is found, the PDP has to search in the Access History for a record containing the object type specified in the constraint that is not matched with that of the *REQ* tuple (and that is relevant to the Requester). If such a record is found, it means the constraint is violated; that is, the Requester has accessed in the past a object type that is in conflict with the one specified in the current request. In this case, the action in the request will not be permitted. The CW constraints can be specified at the level of object types. However, a fine-grained specification may be achieved if the domain hierarchy, objects may belong to, is also taken into account. In this case, we assume that *REQ* and records in the Access History repository have the domain information at the same level (where level indicates number of domains) as is present in the constraint, where each element of the domain information in *REQ* will be matched with the corresponding element in the constraint.

As for the HBDSoD constraints, the time for evaluating the CW constraints depends on the number of deployed constraints in the repository, the complexity of the constraint specification and the number of records in the Access History. Thus, the asymptotic complexity can be calculated as $O(Z \cdot c \cdot r)$. To measure the actual overhead, we performed a similar set of experiments as conducted for HBDSoD constraints. We deployed 100 different CW constraints and considered the worst case scenario. We then changed the number of elements in the constraint and the number of records in the Access History. The results are shown in Figure 4.8(b).

The above results clearly show that there is a penalty to be paid for the enforcement of encrypted constraints in outsourced environments. The execution time varies from 100 ms to 2.5 seconds as number of records in the Access History increase from 100 to 500. To be fair, our experiments have been executed with very basic hardware. We expect that our

Table 4.1: Summary of time complexity of each phase in the lifecycle of E-GRANT

Phase Name	**Complexity in the Worst Case**
Deployment of HBDSoD constraints	$\Theta(Y)$
Deployment of CW constraints	$\Theta(Z)$
Generation of requests	$\Theta(Z)$
Evaluation of HBDSoD constraints	$O(Y \cdot c \cdot r)$
Evaluation of CW constraints	$O(Z \cdot c \cdot r)$

solution would be able to perform better with more dedicated resources, such as servers deployed in a cloud infrastructure. Moreover, all the executions have been performed as a centralised solution. Clearly, having in these settings a single PEP and a single PDP to process all the incoming requests represent a bottleneck. To solve this problem, we are planning to develop a distributed version of our proposed architecture that can be deployed on multiple nodes and adapted to the actual request demand.

Table 4.1 provides a summary of time complexities of different phases in the lifecycle of E-GRANT.

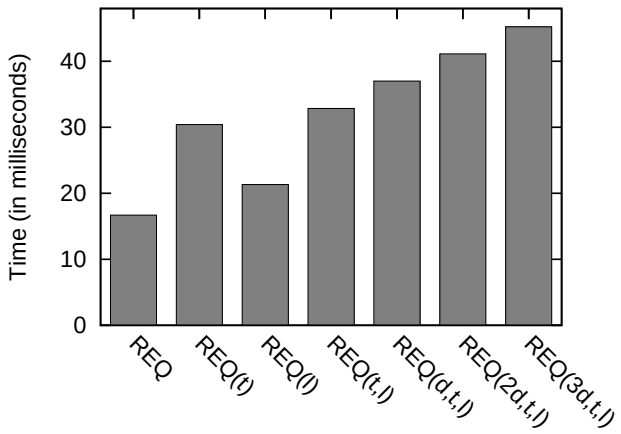


Figure 4.9: Performance overhead of updating the Session with the request data

4.8.5 Performance Analysis of Session Update

After the PDP checks that the current request is not violating any deployed constraints and the request is granted, the Access History in the Session needs to be updated with the information in the executed request. The session update is managed by the PEP that executes the second round of encryption before storing the encrypted data in the Session (see Algorithm 4.4). Figure 4.9 shows the performance overhead of encrypting the request for storing it in the Session. The graph shows the execution time of different formats of

the *REQ* tuple: that is, from the basic format containing only subject, action and target information to more complex ones having time, location and a domain hierarchy of objects up to three levels.

4.9 Chapter Summary

In this chapter, we have proposed E-GRANT, an architecture for enforcing dynamic security constraints as an outsourced service running in the cloud. The main contribution of E-GRANT is that it supports the enforcement of encrypted security constraints while maintaining the encrypted session in the cloud. In this way, cloud providers learn neither about the information stored by the session nor about the content of security constraints being enforced. The proposed approach provides a scalable key management, where users do not share any encryption keys. If users leave the organisation or keys get compromised, they can be revoked without requiring re-distribution of keys and re-encryption of deployed constraints.

The combination of Chapter 2, Chapter 3 and Chapter 4 offers the full-fledged RBAC model that can support role hierarchies and the constraint model. This full-fledged RBAC model can be outsourced such that the Service Provider cannot learn private information about sensitive policies being enforced. The data (or policy) outsourcing follows the traditional client-server model, where there are two main roles, a client and a server. Our proposed solutions assume that both client and server roles run in different spaces. The issue of enforcement of sensitive policies becomes quite challenging if we consider a distributed model, where each peer can play multiple roles simultaneously. Unfortunately, our existing proposals do not work because the underlying assumption becomes invalid, i.e., both a client and a server run in the same space in distributed settings. In the next chapter, we investigate privacy and security issues in enforcing sensitive security policies in distributed environments.

Chapter 5

PIDGIN: Enforcing Security Policies in Distributed Environments*

Opportunistic networks have recently received considerable attention from both industry and researchers. These networks can be used for many applications without the need for a dedicated IT infrastructure. In the context of opportunistic networks, the application to content sharing in particular has attracted specific attention. To support content sharing, opportunistic networks may implement a publish-subscribe system in which users may publish their own content and indicate interest in others' content through subscription. Using a smartphone, any user can act as a broker by opportunistically forwarding both published content and interest within the network. Unfortunately, despite their provision of this great flexibility, opportunistic networks raise serious privacy and security issues. Untrusted brokers can not only compromise the privacy of subscribers by learning their interest but also can gain unauthorised access to the disseminated content.

There are solutions that can regulate access to content by specifying access policies. However, access policies may reveal information about content they aim to protect. This chapter addresses the research challenges inherent to the exchange of content and interest without: (i) revealing content and its associated policies to unauthorised brokers and (ii) compromising the privacy of subscribers. Specifically, this chapter presents an interest and content sharing solution that addresses these security challenges and preserves privacy in opportunistic networks. We demonstrated the feasibility and efficiency of this solution by implementing a prototype and analysing its performance on real smart phones.

*The final version of this chapter will appear in [89].

5.1 Introduction

In the last few years, the usage of smartphones has grown dramatically and is predicted to increase even more in coming years [109]. Considering the pervasive nature of smartphones, mobile opportunistic networks could be leveraged to share information. Several of the concepts behind opportunistic networks originate from Delay Tolerant Networks (DTNs) that offer flexible content sharing without requiring a dedicated IT infrastructure [21]. Haggle [110], an example of such a network architecture, allows smartphones to opportunistically share content via short-range communication [111]. To share content, opportunistic networks such as Haggle implement a publish-subscribe system in which nodes can publish their own content and subscribe to others' content by indicating their interest. Any node can also act as a broker (also called a relay) that opportunistically receives content and interest, matches them, and possibly delivers that content to other nodes.

The opportunistic networks could be applied to the exchange of information in a wide range of domains from social media to military applications. However, such networks also present serious privacy and security issues, particularly the need for an approach to the exchange of content and interest that neither (i) reveals content and its associated policies to unauthorised brokers nor (ii) compromises the privacy of subscribers.

For the regulation of access to content, cryptographic approaches such as ABE which include CP-ABE [47] and KP-ABE [49] offer fine-grained control over content but leak information about the policies and attributes that protect that content, respectively. To protect these policies, state-of-the-art solutions exist to enforce sensitive policies in outsourced environments [28, 75, 112]. However, such solutions assume that the outsourced server does not collude with any client. Thus, these solutions cannot be applied in opportunistic network settings in which nodes communicate in a peer-to-peer fashion, i.e., serving as both a client and a server.

5.1.1 Research Contributions

This chapter presents **Privacy preserving Interest anD content sharinG in opportunIstic Networks (PIDGIN)**, an interest and content sharing scheme that preserves privacy. In PIDGIN,

- brokers match subscriber's interest against policies associated with content without compromising the subscriber's privacy (say, by learning attributes or interest).
- an unauthorised broker neither gains access to content nor learns access policies, and authorised nodes gain access only if they satisfy fine-grained policies specified

by the publishers.

- the system provides scalable key management in which loosely-coupled publishers and subscribers communicate with each other without any prior contact.

As a proof-of-concept, we have developed and analysed the performance of a prototype running on real smartphones in order to show the feasibility of our approach.

5.1.2 Chapter Outline

The rest of this chapter is organised into the following sections. Section 5.2 provides a brief overview of opportunistic networks, describes the motivating scenario, and lists some of the major research challenges for interest and content sharing in opportunistic networks with guaranteed preservation of privacy. In Section 5.3, we draw the system model. Next, we describe the proposed scheme in Section 5.4. Section 5.5 elaborates PIDGIN's details. In Section 5.6, we provide the concrete construction. Section 5.7 analyses PIDGIN from a security perspective. In Section 5.8, we report the outcomes of the performance analysis. Section 5.9 is dedicated for discussion. Section 5.10 reviews the related work. Finally, we conclude in Section 5.11.

5.2 Opportunistic Networks and Research Challenges

In this section, we provide a brief overview of opportunistic networks, a motivating scenario, and the major research challenges in opportunistic networks that we address.

5.2.1 Overview of Opportunistic Networks

Conceptually, opportunistic networks originate from DTNs that enable content exchange between nodes in a publish-subscribe fashion, generally via short-range communication. In a typical opportunistic network, such as Haggle, a subscriber node subscribes interest while a publisher node publishes content to its neighbouring nodes [111]. These neighbouring nodes are intermediate nodes, known as brokers, that epidemically disseminate interest and content within the network. A resolution takes place when a broker node finds a match between the interest of a subscriber and the tags associated with published content. As a result of resolution, a broker forwards content to the subscriber. In the following section, we consider a motivating scenario that can further help to understand opportunistic networks and research challenges concerning privacy and confidentiality.

Curiosity - A Military Mission: Let us consider a battlefield scenario for a mission

called *Curiosity* in which soldiers are equipped with smartphones. During the mission, a scout collects some sensitive information about the enemy (for instance, an image of the enemy's position) using her smartphone camera. After acquiring this sensitive information, a scout desires to share it with other soldiers. For this reason, she may tag the image with the mission name, i.e., *Curiosity*. Unfortunately, there is no Internet connectivity on the battlefield and the only way to share is to use the short-range communication offered by smartphones. Therefore, the scout would like to share the image with other soldiers using their smartphones. We assume that the soldiers are interested in getting information about the mission and subscribe using their smartphones.

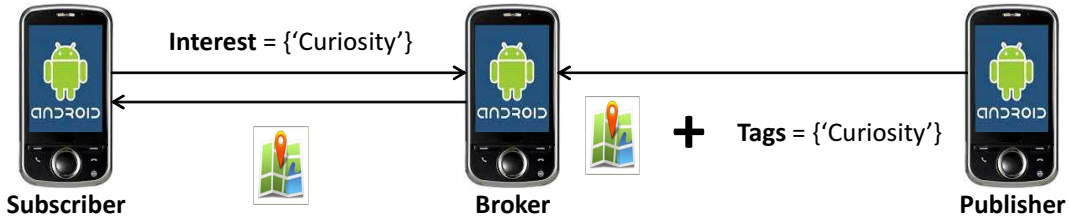


Figure 5.1: An example of content sharing in an opportunistic network

5.2.2 Motivating Scenario

Haggle: A Possible Solution: To exchange information in such scenarios, we can leverage opportunistic networks, such as Haggle. Using Haggle, the scout publishes the image with *Curiosity* as a tag. Any solider can show interest in *Curiosity* by subscribing, as illustrated in Figure 5.1. Here, we assume that someone as a broker receives both interest and image along with the tag. Whenever that happens, the broker checks whether the interest of a subscriber matches any tag associated with the image. If so, the broker forwards the image to the subscriber(s).

Privacy and Confidentiality Issues: First of all, to preserve confidentiality, the information about the *Curiosity* mission should be shared only within a particular group of soldiers. Each content item is associated with an access policy that indicates who should have access to it. For example, information about the *Curiosity* mission might have a policy (P) that content is shared with either a *Major* or a *Soldier* from the *Infantry* unit. Even if the content (i.e., image) is encrypted, the policy itself could reveal sensitive information. That is, an enemy may infer useful information from the fact that some contents are sent to a *Major* or a *Soldier* from the *Infantry* unit. Outsiders (i.e., enemies) and insiders (i.e., soldiers) serving as brokers may gain unauthorised access to contents. Furthermore, the interest of subscribers and the tags associated with content may also

reveal sensitive information. Therefore, in addition to the content itself, its associated tags, policies, and subscription information (i.e., interests) should also be protected.

This scenario motivates the need to tackle the security and privacy issues that we generally face in opportunistic networks. In the following section, we list some major research challenges inherent to these issues that we address in this chapter.

5.2.3 Research Challenges

To guarantee the preservation of privacy for interest and content sharing in opportunistic networks, the following major research challenges related to both (i) privacy and confidentiality (i.e., *C1-C3*) and (ii) functionality (i.e., *C4-C5*) need to be addressed:

C1 In the presence of unauthorised brokers, how do we regulate access to disseminated content and preserve confidentiality of content and associated policies?

C2 In the presence of curious brokers, how does the network exchange content without compromising the privacy of its subscribers?

C3 How can a subscriber subscribe to content without exposing her interest to untrusted brokers?

C4 In order to minimise the flood of unnecessary traffic on the communication network, how do we ensure that a subscriber receives content if and only if authorised to decrypt?

C5 Assuming the loosely-coupled nature of the publish-subscribe model, how do we address the challenges above (i.e., *C1-C4*) without sharing any keys between publishers and subscribers?

5.3 The System Model

Before presenting our threat model and assumptions, we identify the entities involved in the system:

A Publisher is a node that can publish the content.

A Subscriber is a node that can subscribe interest.

A Broker is a node that may receive and disseminate both content and interest. It evaluates whether any content matches known interest. On successful evaluation, it forwards content to the subscribers.

Trusted Key Management Authority (TKMA) is an offline trusted entity that distributes keying material (including private keys and/or public parameters) to all nodes out of the band (usually once in the lifetime of a node, typically when the node is initialised).

The Threat Model. We assume that brokers are *honest-but-curious*, i.e., they honestly follow the protocol, but remain curious to learn about content and interest. Also, we assume that brokers may collude. Furthermore, we consider that the TKMA is fully trusted and plays a role at the time of system initialisation. Last but not least, we assume only passive adversaries and do not consider active adversaries that can manipulate the exchanged information.

5.4 The Proposed Idea

In this section, we describe the proposed scheme for preserving privacy during interest and content sharing in opportunistic networks. As a starting point, we consider some basic schemes that partially address research challenges listed in Section 5.2.3. Next, we gradually address all research challenges and finally describe the proposed scheme.

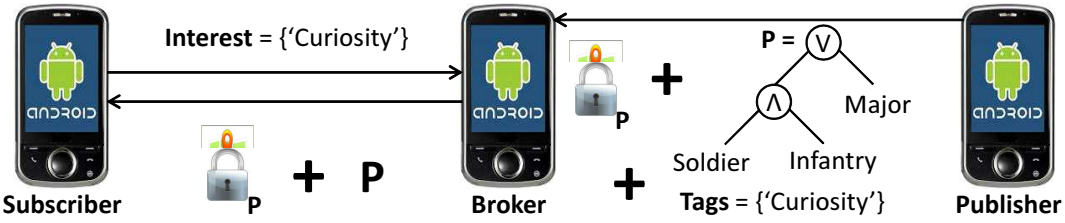


Figure 5.2: Regulating access to content using CP-ABE policies

5.4.1 Scheme I: Regulate Access on Content

To preserve the confidentiality of content, a publisher might specify who can gain access. A possible approach for the publisher could be to regulate access on content by employing ABE, such as CP-ABE [47] or KP-ABE [49]. ABE offers fine-grained policies for content access. In this scheme, we consider CP-ABE because it enables a publisher to exert control over access to content, as described in the use case scenario. In contrast, in KP-ABE, a key generation authority exerts control over who can access content. Figure 5.2 illustrates this scheme in which the image is encrypted according to the policy: *either a Major or a Solider from the Infantry unit can get access.* The policy is expressed as a tree whose leaf nodes represent the attributes; non-leaf nodes denote the AND, OR and threshold gates.

In this scheme, a broker forwards content to the subscribers if a subscriber's interest matches with any tag associated with the content.

This approach preserves the confidentiality of disseminated contents without providing access to unauthorised brokers. This scheme, however, has a drawback. A broker might send content to subscribers who might not be able to decrypt it. In fact, a broker's role is merely to match the interest of subscribers against tags associated with content without checking whether a subscriber has access authorisation. For instance, consider a subscriber who is a soldier but neither a *Major* nor a member of the *Infantry* unit.

In summary, this scheme resolves the access control problem *(C1)* while raising the problem of a communication network flooded with unnecessary traffic *(C4)*.

5.4.2 Scheme II: Perform an Authorisation Check

This scheme extends Scheme I and resolves the flooding problem *C4*. In this scheme, a subscriber may send attributes and interest to brokers so that a broker can perform an authorisation check prior to forwarding the contents. To perform the authorisation check, a broker matches leaf nodes in the policy tree with the subscriber's attributes. If there is a match, a leaf node will be marked as satisfied. After evaluating leaf nodes, a broker evaluates intermediate nodes (including AND, OR and threshold) in the policy. A broker will forward encrypted content to subscriber if and only if (i) the root node of the policy is marked as satisfied and (ii) the interest matches to the tags.

This scheme targets both the access control problem *(C1)* and the flooding problem *(C4)*. However, it still raises some privacy issues. First, both the cleartext attributes of subscribers and the cleartext CP-ABE policies can compromise the privacy of subscribers, i.e., *C2*. For example, the enemy may learn from policies that there is some information intended for a *Major*. Second, the cleartext interest of a subscriber may also leak information, i.e., *C3*. For instance, the enemy may learn that this content or interest concerns the *Curiosity* mission.

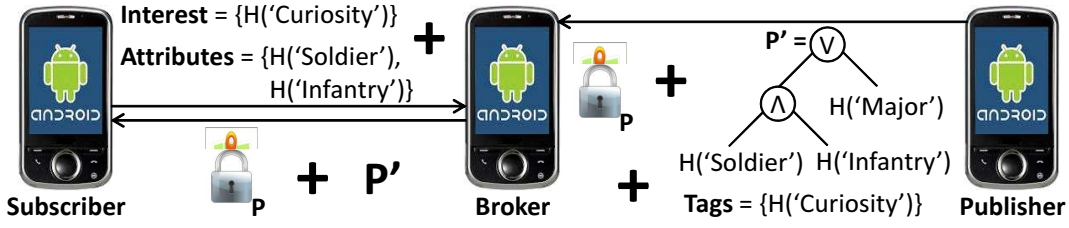


Figure 5.3: Private information is hidden through replacement of leaf nodes in the CP-ABE policy, tags, attributes and interest items with their corresponding hashes

5.4.3 Scheme III: Hide Private Information Using a Hash

In order to partially overcome the issue of subscriber privacy *(C2)*, a subscriber and a publisher may hash both attributes and leaf nodes in the policy tree, respectively. Similarly, a subscriber's interest could be protected by calculating the hash values of interest items and tags associated with contents. In this scheme, a broker forwards encrypted content to subscribers if and only if (i) the hash value of the interest matches the hash value of the tag (i.e., h('Curiosity')) and (ii) hash values of attributes (i.e., {h('Soldier'), h('Infantry')}) satisfy the policy P' whose leaf nodes are also hashed, as shown in Figure 5.3.

Unfortunately, this scheme is vulnerable to a pre-computed dictionary attack. That is, the enemy may pre-calculate a list of hashes for possible attributes (and leaf nodes in the policy tree) and a list of hashes for potential interest items (and tags). The pre-calculated list of hashes may easily reveal the original attributes (and leaf nodes in the policy tree) and interest (and tags).

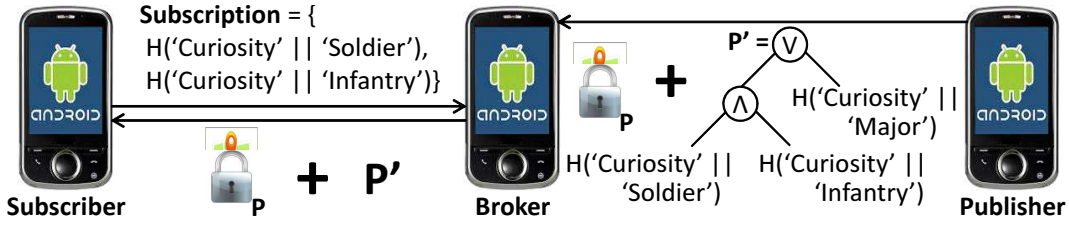


Figure 5.4: Hardening against a pre-computed dictionary attack through concatenation a pair of (i) a leaf node in the CP-ABE policy and a tag (ii) an attribute and an interest item, then calculation of the hash on the final string

5.4.4 Scheme IV: Hardening Against a Pre-Computed Dictionary Attack

To harden against the pre-computed dictionary attack, a publisher may replace each leaf node in the policy with a hash of a concatenated pair of a tag and an attribute. Similarly, a subscriber may subscribe using the hash of a concatenated pair of an interest item and an attribute (i.e., {H('Curiosity' || 'Soldier'), H('Curiosity' || 'Infantry')}) as illustrated in Figure 5.4. In this scheme, a broker just needs to check whether the items in a subscription satisfy the hashed policy P'. Upon successful evaluation, the broker will forward the content to subscribers. The advantage of this scheme is that it not only hardens against pre-computed dictionary attacks but also decreases the number of comparisons performed at the broker's end as compared to Scheme III. This is because a broker performs integrated checks that cover both authorisation and interest matching

simultaneously in contrast to Scheme III in which a broker performs two different checks: one to check the authorisation and one to match the interest. Though it enlarges the key space (which could be computationally extensive), this scheme is still vulnerable to a pre-computed dictionary attack.

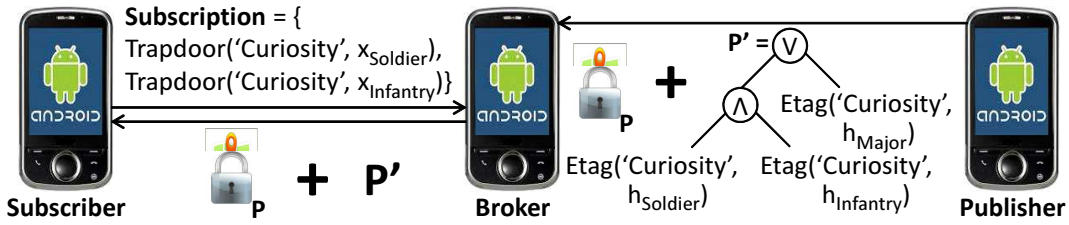


Figure 5.5: The PIDGIN scheme protecting the content, the policy, the tags associated with content, and the subscriber's interest and attributes

5.4.5 PIDGIN: The Proposed Scheme

Our proposed scheme, PIDGIN, aims at addressing all research challenges (i.e., *C1-C5*) listed in Section 5.2.3. The main idea behind PIDGIN is regulation of access to content using CP-ABE and extension of cleartext CP-ABE policies with the PEKS scheme [36] to protect attributes, interest, tags and leaf nodes in the policy tree. The PEKS scheme consists of four basic functions including **Keygen**, **Etag**[1], **Trapdoor** and **Test**. For each attribute, we run **Keygen** to calculate a key pair consisting of both public (i.e., $h_{Soldier}$) and private (i.e., $x_{Soldier}$) keys corresponding to a given attribute (i.e., *Soldier*). To protect policies and tags, a publisher can replace each leaf node in the policy tree with the **Etag** function of the PEKS scheme, which takes as input a tag (i.e., *Curiosity*) and the public key of the attribute as shown in Figure 5.5. A subscriber protects attributes and interest by replacing each interest item in the subscription list with a **Trapdoor** function which takes as input an interest item (i.e., *Curiosity*) and the private key (i.e., generated by the PEKS scheme) corresponding to the attribute.

A broker performs encrypted matching between encrypted policies and encrypted subscriptions. It runs the **Test** function, a building block that matches a trapdoor to an encrypted tag. If an encrypted tag in the policy tree P' matches with any encrypted trapdoor in the subscription list, the tree node is marked as satisfied. The broker evaluates all nodes in the policy tree starting from leaf nodes to root. If the root is satisfied, the broker will forward content along with the encrypted policy to the subscribers.

[1]The Etag function is called PEKS in [36].

5.5 Technical Details of PIDGIN

5.5.1 Initialisation and Key Generation Phases

During the initialisation phase, the system is set up to initialise both CP-ABE and PEKS schemes. In PIDGIN, the TKMA generates and distributes keys during the key generation phase. The TKMA generates a private set of attributes (i.e., CP-ABE private key) and sends it securely to the subscriber out of the band. The TKMA publishes the public part of attributes (i.e., CP-ABE public key) to all publishers. Since the attributes are protected using the PEKS scheme, the TKMA also generates a pair of keys corresponding to each attribute. Similar to the CP-ABE key distribution, the TKMA sends the private and public parts of the PEKS key pair to the subscriber and publishers, respectively. The major difference between the CP-ABE private key set and the PEKS private key set is that the former is unique for each user, while the latter is not.

5.5.2 The Publisher's Encryption Phase

To protect the content and preserve the privacy of subscribers, a publisher encrypts content with CP-ABE policies and protects those policies as well. The contents could be encrypted with a symmetric key, such as Advanced Encryption Standard (AES), which is further encrypted with the CP-ABE policy. Since the CP-ABE policy may compromise the privacy of subscribers, the CP-ABE policies are encrypted using PEKS. While encrypting CP-ABE policies using PEKS, PIDGIN also incorporates tags that are associated with content.

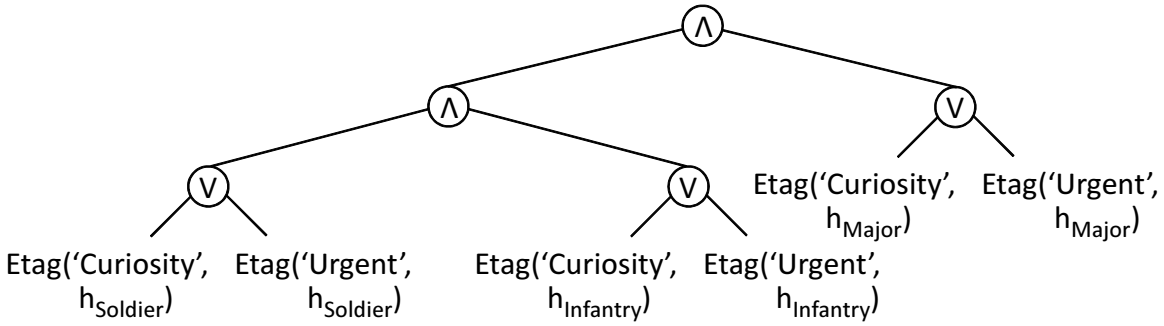


Figure 5.6: The extended CP-ABE policy with two tags, i.e., 'Curiosity' and 'Urgent'.

To extend CP-ABE policies for PEKS, a publisher considers each leaf node in the policy tree as well as number of tags that are associated with contents. If there is only a single tag then a publisher replaces the leaf node with the **Etag** function as already illustrated in Figure 5.5. The **Etag** function takes a tag keyword to be encrypted and the

public key corresponding to the leaf node under consideration. After running the **Etag** function, a publisher gets an encrypted tag. The **Etag** function does not leak information about the tags or leaf nodes in the policy tree. In the case that there is more than one tag then a publisher runs the **Etag** function for each tag item and encrypts it with the public key corresponding to the leaf node under consideration. Finally, the leaf node attribute is replaced with the subtree where all newly generated Etags corresponding to tags are disjuncted using OR. Figure 5.6 illustrates an example of the policy involving two tags, i.e., 'Curiosity' and 'Urgent'.

5.5.3 The Subscriber's Encryption Phase

In order to protect the interest of a subscriber and its attributes, a subscriber encrypts each interest item using the private key (i.e., generated by the PEKS scheme) corresponding to the attribute. PIDGIN considers that a subscriber might have multiple attributes and interest items. Generally, each interest item is encrypted with each private key (i.e., generated by the PEKS scheme) that corresponds to the attribute. Figure 5.5 describes the case in which a subscriber holds two attributes and subscribes with a single interest item. Let us assume that a subscriber has two interest items, say 'Curiosity' and 'Urgent', while holding attributes Solider and Infantry. The subscription list would contain four items including **Trapdoor**('Curiosity', $x_{Soldier}$), **Trapdoor**('Curiosity', $x_{Infantry}$), **Trapdoor**('Urgent', $x_{Soldier}$) and **Trapdoor**('Urgent', $x_{Infantry}$). The trapdoor representation does not leak information about the interest item and the attribute.

5.5.4 The Broker's Matching Phase

A broker opportunistically exchanges both content and subscriptions. Once a broker receives both the encrypted subscription and the encrypted content along with the encrypted policies, it evaluates whether the encrypted subscription satisfies the encrypted policy. For this evaluation, the broker runs a matching function that recursively evaluates the encrypted policy tree. The **Test** function matches each encrypted leaf node in the policy against the encrypted interest item in the subscription.

The **Test** function returns either *TRUE* or *FALSE*, indicating whether the encrypted tag is matched with the trapdoor or not, respectively. By running the **Test** function, a broker does not learn about the tag or the interest item because both are encrypted and they are matched in an encrypted manner. If an encrypted tag in the policy tree matches with any trapdoor in the subscription list, that node is marked as satisfied. After evaluating leaf nodes, a broker can evaluate intermediate AND, OR and threshold nodes in the policy tree to finally identify whether the root node of the policy tree is satisfied or

not. If the root node is satisfied, the broker will forward content along with the encrypted policy to the subscriber.

5.5.5 The Subscriber's Decryption Phase

Once a subscriber receives the encrypted content along with the encrypted policy, it first recovers the original CP-ABE policy. For this recovery, either leaf node (if a single tag, see Figure 5.2) or a subtree of tags (if more than one tag, see Figure 5.6) is replaced with their corresponding attribute. Before sharing the encrypted interest, a subscriber builds the subscription history as a lookup table containing an attribute and its corresponding trapdoor. If the trapdoor is matched with any encrypted tag in the leaf node of the policy, the subscription history will be looked up to find the attribute corresponding to the matched trapdoor. Next, a leaf node (if a single tag) or a subtree of tags (if more than one tag) will be replaced with the found attribute. If no match is found, then a dummy attribute will be placed. This recovers the original CP-ABE policy (i.e., one shown in Figure 5.2) that can finally be used by the CP-ABE decryption function to get the symmetric key that is required for decryption of the contents.

5.6 Concrete Constructions of PIDGIN

In this section, we provide some definitions and details of core functions used in different phases of the PIDGIN lifecycle.

5.6.1 Definitions

The Policy Structure. We assume a policy tree P that represents an access structure. Each non-leaf node represents an AND, an OR or a threshold gate. Let us consider that num_x denotes number of children of a node x and k_x represents the threshold value. For OR and AND gates, k_x is 1 and num_x, respectively. For the threshold gate, the value of k_x is: $0 < k_x \leq num_x$. Let us consider that $\mathbf{parent}(x)$ represents the parent of a node x, $\mathbf{att}(x)$ denotes the attributes associated with leaf node x, and $\mathbf{index}(x)$ returns the number associated with a node x, with nodes numbered from 1 to num.

Bilinear Maps. Let $\mathbb{G}_1$ and $\mathbb{G}_2$ be two multiplicative cyclic groups of prime order p. Let g be a generator of $\mathbb{G}_1$ and $e : \mathbb{G}_1 \times \mathbb{G}_1 \rightarrow \mathbb{G}_2$ be a bilinear map. The bilinear map e satisfies the following properties:

- Computability: given $g, h \in \mathbb{G}_1$, there is a polynomial time algorithm to compute $e(g, h) \in \mathbb{G}_2$.

- Bilinearity: $\forall u, v \in \mathbb{G}_1$ and $a, b \in \mathbb{Z}_p$, we have $e(u^a, v^b) = e(u, v)^{ab}$.
- Non-degeneracy: if g is a generator of $\mathbb{G}_1$ then $e(g, g)$ is a generator of $\mathbb{G}_2$, where $e(g, g) \neq 1$.

Notice that the bilinear map e is symmetric since $e(g^a, g^b) = e(g, g)^{ab} = e(g^b, g^a)$.

Hash Functions. We consider the hash functions:

$$H_1 : \{0,1\}^* \rightarrow \mathbb{G}_1$$
$$H_2 : \mathbb{G}_2 \rightarrow \{0,1\}^{\log p}$$

Lagrange Coefficient. We define the Lagrange coefficient $\Delta_{i,A}$ for $i \in \mathbb{Z}_p$ and a set A of elements in $\mathbb{Z}_p$:

$$\Delta_{i,A}(x) = \prod_{j \in A, j \neq i} \frac{x - j}{i - j}$$

5.6.2 Construction Details of PIDGIN

Init(1^K). The init algorithm takes as input the security parameter k that determines the size of p. It randomly picks two exponents $\alpha, \beta \in \mathbb{Z}_p$ and outputs the public key $PK = (\mathbb{G}_1, g, h = g^\beta, e(g, g)^\alpha)$ and the master key $MK = (\beta, g^\alpha)$. The public key PK is published while the master key MK is kept securely by the TKMA. Moreover, two stores, the Search Key Secret Store ($SKSS$) and the Search Key Public Store ($SKPS$), which are managed by the TKMA, are initialised as:

$$SKSS \leftarrow \phi$$
$$SKPS \leftarrow \phi$$

KeyGen(MK, A). The key generation algorithm is run by the TKMA. It takes as input a list of attributes A and outputs a CP-ABE decryption key and a set of search key pairs. To generate the decryption key, it first chooses a random $r \in \mathbb{Z}_p$ and then a random $r_j \in \mathbb{Z}_p$ for each attribute $j \in A$. Next, it computes the decryption key as:

$$DK = (D = g^{(\alpha + r)/\beta},$$
$$\forall \in A : D_j = g^r \cdot H_1(j)^{r_j}, D'_j = g^{r_j})$$

Before the generation of a search key pair for an attribute $j \in A$, a search key store (either $SKSS$ or $SKPS$) can be looked up. If the search key pair already exists, then the

public and private keys will be collected from $SKPS$ or $SKSS$, respectively. Otherwise, the algorithm chooses a random $x_j \in \mathbb{Z}_p^*$, calculates $h_j = g^{x_j}$, and updates both private and public key stores as:

$$SKSS \leftarrow SKSS \cup (j, x_j)$$
$$SKPS \leftarrow SKPS \cup (j, h_j)$$

Next, it computes the search key secret as: $SKS = (\forall \in A : x_j)$. Finally, the $SKPS$ is publicised while the decryption key DK and the search key secret SKS are securely transmitted to the subscriber.

Etag(PK, h_i, t)**.** The **Etag** algorithm encrypts a given tag t with h_i. It chooses a random $r \in \mathbb{Z}_p^*$ and computes $z = e(H_1(t), h^r)$. Next, it computes $A = g^r$ and $B = H_2(z)$ and outputs the encrypted tag as: $ET = (A, B)$.

Pub-Enc$(PK, SKPS, C, P, T)$**.** The publisher encryption algorithm encrypts content C under the access policy P with a list of tags T. It also encrypts P. In reality, it randomly generates a symmetric key K and encrypts C as $\{C\}_K$ and then encrypts K under P. To encrypt K under P, it chooses a polynomial q_x for each node x in a top-down manner, starting from the root R, such that it sets degree d_x one less than the threshold value k_x, i.e., $d_x = k_x - 1$. Starting from the root R, it chooses a random $s \in \mathbb{Z}_p$, sets $q_R(0) = s$ and chooses other d_R points randomly. For any other non-root node x, it sets $q_R(0) = q_{parent(x)}(index(x))$ and chooses other d_x points randomly. Let Y be the set of leaf nodes in P. The ciphertext is computed as:

$$CT = (\tilde{E} = Ke(g, g)^{\alpha s}, E = h^s,$$
$$\forall y \in Y : E_y = g^{q_y(0)}, E'_y = H_1(att(y))^{q_y(0)})$$

Next, the policy P is encrypted as follows. For each leaf node i, it looks up the corresponding private secret key h_i from the $SKPS$. Then, it runs **Etag**(h_i, t) for each tag $t \in T$ and combines all encrypted tags corresponding to an attribute to form an OR subtree. The original leaf node attribute is replaced with this OR subtree. If only one tag exists in T, the original attribute is replaced with the output of the **Etag** function. This basically generates the encrypted policy P'. Finally, this algorithm returns $PE = (P', CT, \{C\}_K)$.

Trapdoor(x_i, t)**.** The **Trapdoor** algorithm encrypts interest item t using x_i. It returns the encrypted interest item $TD = H_1(t)^{x_i}$.

Sub-Enc(I, SKS)**.** The subscriber encryption algorithm encrypts interest I using the

attributes SKS. For each interest item $t \in I$, it runs **Trapdoor**(x_i, t) using search key secret x_i corresponding to each attribute $i \in SKS$. A subscriber also maintains a history of subscription HS to keep track of all trapdoors belonging to a subscription. HS is initialised as $HS \leftarrow \phi$ and updated as:

$$\forall i \in SKS : HS \leftarrow HS \cup (i, TD_i)$$

HS maintains each trapdoor with its corresponding attribute. Finally, this algorithm publicises $SE = (TD_1, TD_2, \ldots, TD_{|I|.|SKS|})$ and keeps HS securely.

Test(ET, TD)**.** The **Test** algorithm takes the encrypted tag and trapdoor and returns *TRUE* if $H_2(e(TD, A) \stackrel{?}{=} B$ is *TRUE* and *FALSE* otherwise.

Bro-Match(P', SE)**.** This algorithm takes the publisher encrypted policy P' and the subscriber encrypted interest SE and returns *TRUE* if they match and *FALSE* otherwise. To perform the match, a broker runs **Test**(ET_i, TD_j) for each leaf node i in P' and trapdoor $TD_j \in SE$. If an encrypted leaf node matches with any trapdoor, it is marked as satisfied (i.e., *TRUE*). After evaluating leaf nodes, the algorithm evaluates intermediate nodes (AND, OR and threshold). After this evaluation, if the root node of the encrypted policy P' is satisfied, that is, *TRUE*, then this algorithm returns *TRUE* and *FALSE* otherwise.

Sub-Dec(PE, HS, DK) This algorithm decrypts the policy P' and then decrypts the encrypted contents PE. First, it matches encrypted leaf nodes with a trapdoor in HS by running **Test**. If a match is found, the corresponding attribute is selected from HS. The leaf node (if a single tag) or a subtree of encrypted tags conjuncted with OR (if more tags) will be replaced with the selected attribute. If no match is found, then a dummy attribute will be placed. This recovers the original policy, which will be used to decrypt the symmetric key: if node x is a leaf node then we assume $i = att(x)$ and run the following function if $i \in A$:

$$\begin{aligned} DecryptNode(CT, DK, x) &= \frac{e(D_i, E_x)}{e(D'_i, E'_x)} \\ &= \frac{e(g^r.H(i)^{r_i}, g^{q_x(0)})}{e(g^{r_i}, H(i)^{q_x(0)})} \\ &= e(g, g)^{rq_x(0)} \end{aligned}$$

If $i \notin A$ then $DecryptNode(CT, DK, x) = \perp$. For a non-leaf node x, the algorithm runs $DecryptNode(CT, DK, z)$ for each child z of x and stores output as F_z. Let A_x be an

arbitrary k_x-sized set of child nodes z such that $F_z \neq \perp$. If no such set exists then the node was not satisfied and the function returns $\perp$. Otherwise, it computes:

$$F_x = \prod_{z \in A_x} F_z^{\Delta_{i,A'_x(0)}}$$

(where $i = index(z)$ and $A'_x = index(z) : z \in A_x$)

$$= \prod_{z \in A_x} (e(g,g)^{r.q_z(0)})^{\Delta_{i,S'_x}(0)}$$
$$= \prod_{z \in A_x} (e(g,g)^{r.q_{parent(z)}(index(z))})^{\Delta_{i,A'_x}(0)}$$

(by construction)

$$= \prod_{z \in A_x} (e(g,g)^{r.q_x(0)})^{\Delta_{i,A'_x}(0)}$$
$$= (e(g,g)^{r.q_x(0)}$$

(using polynomial interpolation)

If the tree is satisfied by A, we set

$$G = DecryptNode(CT, DK, R)$$
$$= e(g,g)^{rq_R(0)}$$
$$= e(g,g)^{rs}$$

The symmetric key is decrypted by computing:
$\tilde{E}/(e(E,D)/G) = \tilde{E}/(e(h^s, g^{(\alpha+r)/\beta})/e(g,g)^{rs}) = K.$

Finally, K is used to decrypt $\{C\}_K$ in order to access contents C.

5.7 Security Analysis of PIDGIN

In PIDGIN, the contents are encrypted using a symmetric key, which is encrypted with the CP-ABE policy. The leaf nodes in the policy tree are further encrypted using **Etag** as proposed in PEKS by Boneh *et al.* [36]. The PEKS is semantically secure against a chosen keyword attack in the random oracle model, assuming that the Bilinear Diffie-Hellman

(BDH) problem is hard (for proof, see Theorem 3.1 in [36]). However, the CP-ABE policy structure is not protected and leaks information about number of attributes or tags used. This leak could partially be tackled by inclusion of some dummy attributes at the cost of an increase in complexity. In PIDGIN, brokers may collude but they cannot gain access to contents, policies or subscriptions. If a broker colludes with a subscriber, they together learn no more information than is already available to the subscriber alone. In the case that two subscribers collude to receive content that each of them alone cannot get otherwise, our scheme prevents such collusion attacks because each subscriber's (CP-ABE) decryption key includes a randomness value that will prevent access to the content.

5.8 Performance Analysis of PIDGIN

As a proof-of-concept, we have developed a prototype of PIDGIN. The prototype is based on an extension of the open source libfenc library [113] written in the C language, a library of functional encryption that includes CP-ABE. Since we proposed to extend CP-ABE with PEKS, we have implemented PEKS in C using the Pairing-Based Cryptography (PBC) library [114], which is an underlying library also required by the libfenc library. PBC is based on Elliptic Curve Cryptography (ECC). The curve we use in our experimentation is of type A. After extending the CP-ABE with PEKS (on the x86 architecture), we cross-compiled it for the ARM architecture to test our prototype on a Samsung Galaxy SIII smartphone (Android version 4.1.2, kernel version 3.0.31, 1 GB RAM, and 1.4 GHz processor). For the deployment of this prototype, we cross-compiled both GMP [115] (the GNU Multiple Precision arithmetic library required by PBC) and PBC libraries for the ARM architecture and installed both on the smartphone. The presented results are averaged over 20 runs.

In our analysis, we have not considered battery consumption because the prototype of PIDGIN we have developed so far requires some optimisations that we have suggested in Section 5.9. However, our future plan is to analyse battery consumption after implementing possible optimisations.

5.8.1 Initialisation and Key Generation Phases

During the initialisation phase, the system-level keying material is generated. During the key generation phase, both search and decryption keys are generated for a given set of attributes. Both phases could be run on a PC because keys are distributed out of the band. However, we consider running both phases on a smartphone (with specifications already described above). The initialisation phase takes 108.5 ms. The generation time of search keys grows linearly with increase in number of attributes as illustrated in Figure

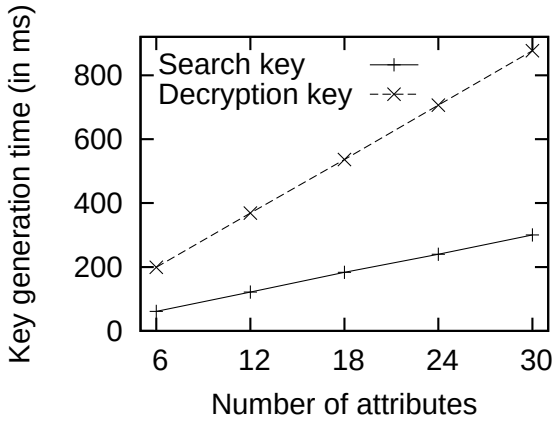


Figure 5.7: Effect of attributes on the key generation time

5.7, where 30 search keys take 300 ms (i.e., an average of 10 ms per attribute). Similarly, the key generation time of decryption keys also grows linearly with increase in number of attributes, where 30 decryption keys take approximately 877 ms (i.e., an average of 29.25 ms per attribute). Asymptotically, the complexity of the key generation is $\Theta(|A|)$, where $|A|$ indicates number of attributes in list A.

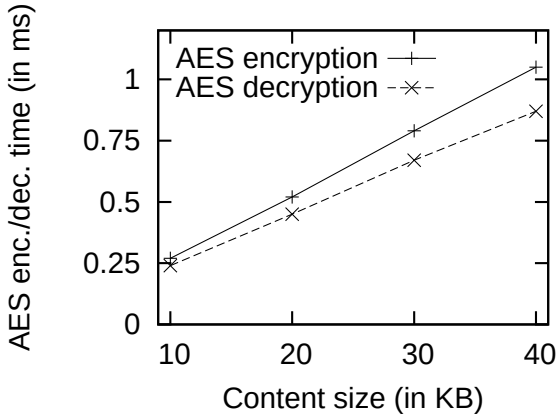


Figure 5.8: Effect of content size on the AES encryption/decryption time

5.8.2 The Publisher's Encryption Phase

In this phase, a publisher encrypts content with a randomly generated symmetric key. In our prototype we use AES keys. The symmetric key is encrypted with the CP-ABE policy. The CP-ABE policy is extended with tags that are also encrypted. Figure 5.8 shows the symmetric encryption time, which grows linearly with the increase in size of content (C). Encryption of a piece of content of size 40 Kilo Byte (KB) takes 0.105 ms (i.e., an average of 0.026 ms per KB). To measure the performance overhead for the encryption

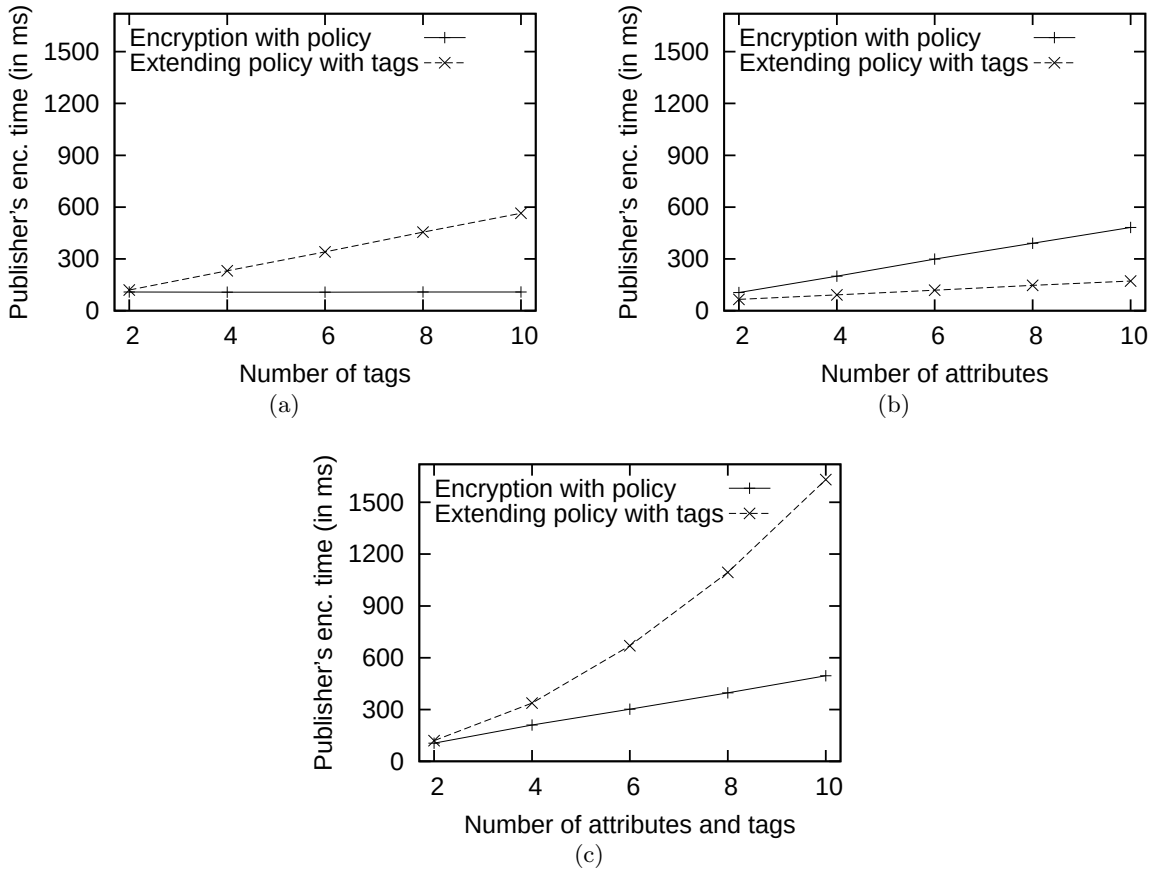


Figure 5.9: Effect of (a) tags, (b) attributes and (c) both tags and attributes on publisher's encryption time

time, we varied the numbers of tags and/or attributes (A_P^*), as shown in Figure 5.9. In Figure 5.9(a) and Figure 5.9(b), we observe the effect of tags and attributes on publisher's encryption time, respectively. In Figure 5.9(a), we observe effect of tags (ranging from 2 to 10) while keeping the number of attributes constant (i.e., 2 attributes - the minimum attributes required to make AND/OR policy). As we can expect, the time to extend a policy with tags grows linearly with increase in number of tags. In Figure 5.9(b), we observe the effect of attributes (ranging from 2 to 10) in a policy while considering a single tag. The time for encryption of the symmetric key with the policy grows linearly with increase in number of attributes. Since the number of attributes increases, it also linearly increases the time to extend the policy with tags. In Figure 5.9(c), we show the most complex case in which we increase both attributes and tags simultaneously. The growth of the time needed to extend a policy with tags is quadratic, depending on the number of attributes and the number of tags. In our experimentation, we considered the number of tags as equal to the number of attributes. In a policy with 2 attributes each with 2 tags, it takes approximately 120 ms to extend the policy tags, while in a policy with 10 attributes with 10 tags each, it takes approximately 1632 ms. Generally, the asymptotic complexity of publisher's encryption is $\Theta(|A_P^*| \cdot |T| + |C|)$.

5.8.3 The Subscriber's Encryption Phase

Figure 5.10 shows the performance overhead incurred during the encryption (see Figure 5.10(a)) and decryption phases (see Figure 5.10(b) and Figure 5.10(c)). In the subscriber's encryption phase, a subscriber encrypts the subscription, which is based on the number of interest items (I) and attributes (A_S^*). In our experimentations, we observed the effect of how different values for the number of attributes and interest separately and together affect the subscription's encryption time. To observe the effect of the number of attributes, we increased the attributes from 2 to 10 while keeping interest items constant (i.e., 1 interest item). Generation of trapdoors for 10 attributes with a single interest item each took approximately 106 ms. Second, we observed the effect of number of interest items on the subscription's time by increasing interest items from 2 to 10 while keeping attributes constant (i.e., 2 attributes conjuncted with either AND or OR). The subscriber took approximately 284 ms to encrypt an interest containing 10 items. As illustrated in Figure 5.10(a), attributes alone or interest items alone linearly affect the subscriber's encryption time. However, we also consider the case when we see effects of both attributes and interest items together. For this purpose, we assumed that number of attributes is equal to that of interest items; that is, if there are two attributes, it means there are two interest items per attribute. Similarly, we assumed 10 attributes with 10 interest items each, which took 1063 ms. The combined effect of attributes and interest items indicates

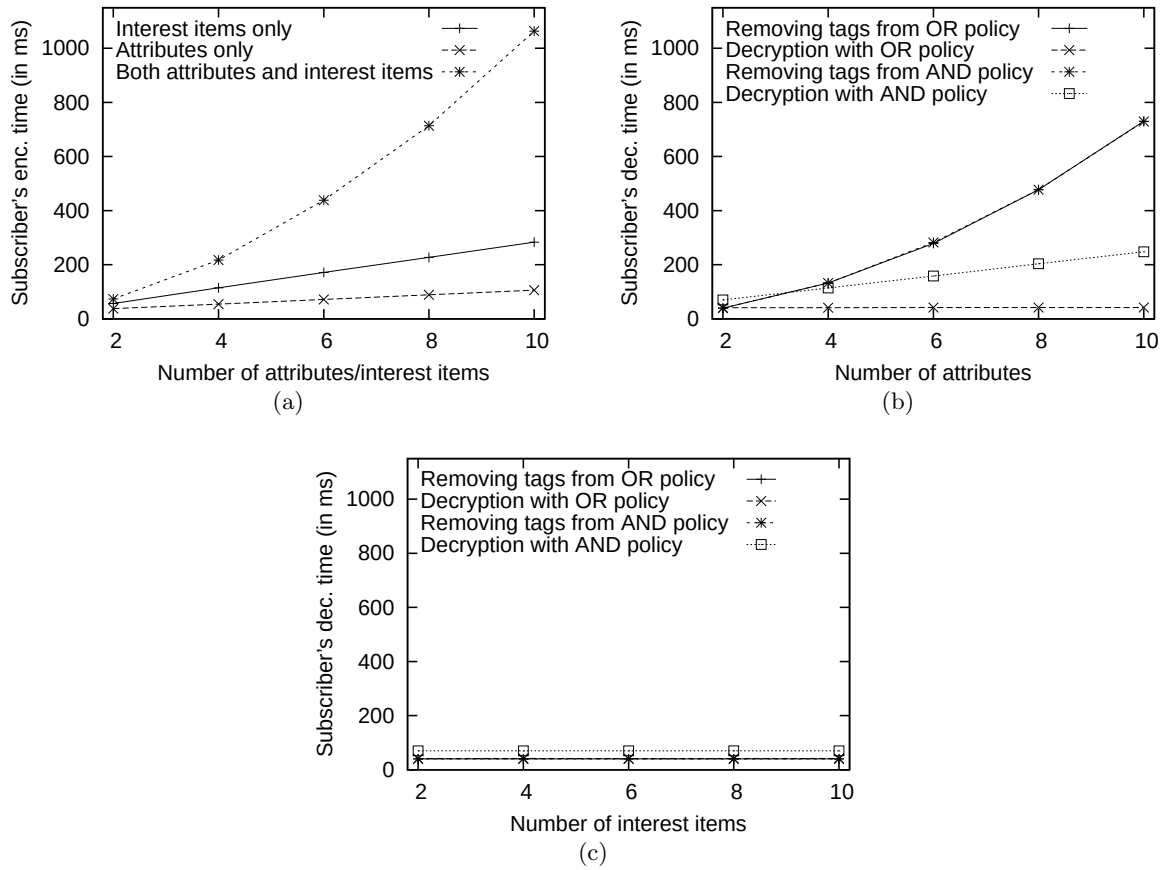


Figure 5.10: Effect of (a) attributes/interest items on the subscriber's encryption time and effect of (b) attributes and (c) tags on the subscriber's decryption time

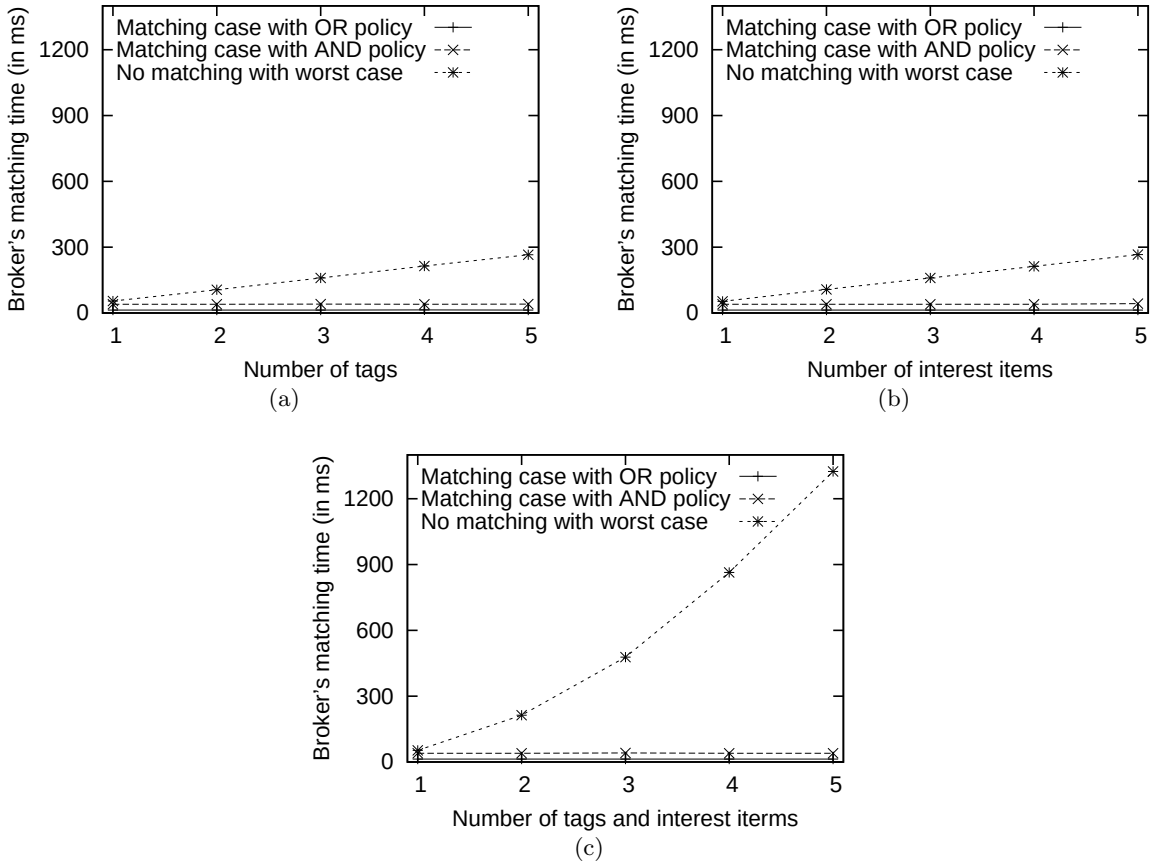


Figure 5.11: Effect of (a) tags, (b) interest items and (c) both tags and interest items on the broker's encrypted matching time

that its growth has quadratic effect on the subscriber's encryption time as shown in Figure 5.10(a). The asymptotic complexity of the subscriber's encryption is: $\Theta(|A_S^*| \cdot |I|)$.

5.8.4 The Broker's Matching Phase

This is the key phase in the lifecycle of PIDGIN. During this phase, a broker matches the encrypted subscription against the encrypted policy associated with the encrypted content. In our analysis, we observe the effect of the numbers of tags and interest items separately and together while keeping the number of attributes constant (i.e., 2 attributes, necessary to have OR or AND policy). Furthermore, we consider the matching case with both OR and AND policies, as well as a zero match case which is the worst case situation. Figure 5.11 shows the performance analysis of this phase. In Figure 5.11(a), we observe the effect of number of tags on the matching time while keeping the number of interest items as constant, i.e., 1. As the graph shows, the matching time increases linearly with

the increase in number of tags. Similarly, we measure the effect of the number of interest items on the matching time while keeping the number of tags constant, i.e., 1. As Figure 5.11(b) indicates, the matching time grows linearly with the increase in the number of interest items. In both Figure 5.11(a) and Figure 5.11(b), the OR policy takes less time as compared to that of the AND policy when we consider the matching case because we use a short circuit evaluation (explained in Section 5.9.2) to evaluate both OR and AND gates. Finally, we consider the most complex case in which we increase the number of tags and the number of interest items together (equally) with 2 attributes. Similar to Figure 5.11(a) and Figure 5.11(b), it takes less time to evaluate the OR policy as compared to that of the AND policy. Next, we consider the worst case in which there are 5 tags and 5 interest items with a 2-attribute policy conjuncted using OR. Since there are 2 attributes in the policy tree with 5 tags each, there will be 10 leaf nodes in the encrypted policy. Furthermore, 2 attributes with 5 interest items each will make 10 trapdoors in the subscription list. The broker checks whether any encrypted leaf node in the policy matches with any trapdoor in the subscription list. In this worst case, the broker runs the **Test** function 100 times, thus taking approximately 1324 ms. In addition to this experiment, we measured the overhead for running the **Test** function and discovered that it takes 13.28 ms. This implies that the real overhead comes from the **Test** function, that is, in fact, a bilinear pairing operation. Hence, the matching operation is dependent on how efficient the bilinear pairing is. The best and worst case complexities of this phase are $\Omega(1)$ and $O(|A_P^*| \cdot |T| \cdot |A_S^*| \cdot |I|)$, respectively.

5.8.5 The Subscriber's Decryption Phase

A subscriber receives the encrypted content (along with the encrypted policy) from the broker if the encrypted interest satisfies the encrypted policy associated with the encrypted content. During the decryption phase, first a subscriber strips off the tags from the policy and then performs decryption with the policy to recover the symmetric key, which is finally used to decrypt the contents. Figure 5.10(b) and Figure 5.10(c) show the effect of the number of attributes and interest items, respectively, on the subscriber's decryption time. In Figure 5.10(b), where we increase attributes from 2 to 10 while keeping the number of interest items constant i.e., 1, we consider both OR and AND policies to see the effect of attributes on the stripping of tags from the policy. Also, we show the performance overhead for the decryption that recovers the symmetric key. In Figure 5.10(c), we describe the case in which the number of interest items are increased from 2 to 10 (but attributes are kept constant i.e., 2), assuming the matching case, i.e., both the publisher and the subscriber are using the same tags and interest items, respectively. Here, the overhead does not increase with the increase in the number of interest items

because we have implemented the short circuit evaluation to evaluate AND, OR and threshold gates. In fact, the trapdoor in the subscription matches interest items against tags in the policy, thus making policy evaluation successful without requiring further matches. Finally, the encrypted contents are decrypted using the symmetric key, which is recovered after we perform the CP-ABE decryption. Figure 5.8 shows the time required for decryption of the content using the AES key. Decryption of a piece of content of size 40 KB takes 0.87 ms (i.e., an average of 0.22 ms per KB). Overall, the complexity of subscriber's decryption is: $O(|A_P^*| \cdot |T| \cdot |A_S^*| \cdot |I| + |C|)$ in the worst case and $\Omega(|C|)$ in the best case.

Table 5.1: Summary of time complexity of each phase in the lifecycle of PIDGIN

Phase Name	**Best Case**	**Worst Case**
Key generation	$\Theta(\|A\|)$	
Publisher encryption	$\Theta(\|A_P^*\| \cdot \|T\| + \|C\|)$	
Subscriber encryption	$\Theta(\|A_S^*\| \cdot \|I\|)$	
Broker matching	$\Omega(1)$	$O(\|A_P^*\| \cdot \|T\| \cdot \|A_S^*\| \cdot \|I\|)$
Subscriber decryption	$\Omega(\|C\|)$	$O(\|A_P^*\| \cdot \|T\| \cdot \|A_S^*\| \cdot \|I\| + \|C\|)$

Table 5.1 summarises time complexity of each phase in the lifecycle of PIDGIN.

Table 5.2: Space overhead of generating encrypted tags and trapdoors

Function	**Size (in Bytes)**
Encrypted Tag (by a publisher)	256
Trapdoor (by a subscriber)	128

5.9 Discussion

5.9.1 Storage Analysis of PIDGIN

As we explained in Section 5.8, the curve we use in our experimentation is of type A. Using this curve, the space complexity of an encrypted tag a trapdoor are 256 and 128 Bytes, respectively. Table 5.2 shows space overhead of generating encrypted tags and trapdoors.

5.9.2 Optimisation and Scalability

Optimisation using Short Circuit Evaluation. The real bottleneck is matching at brokers a set of encrypted policies against encrypted subscriptions. The large scale

matching requires efficiency and some optimisations. One of the optimisations at brokers is implementation of short circuit evaluation for evaluating internal (i.e., non-leaf) nodes of the encrypted policy tree including OR and AND gates. That is, if the node is an OR gate then a broker can stop its evaluation and mark it satisfied once a single child node is satisfied, without performing further matches. Similarly, a broker can mark an AND gate unsatisfied when a single child node is marked unsatisfied. The short circuit evaluation can significantly reduce number of encrypted matches at brokers. This might be useful for the large policies involving a number of children in the policy tree. However, this might not speed up the performance when the set of policies or the number of subscriptions is very large.

Scalability. For matching a large set of encrypted policies against a large number of encrypted subscriptions, PIDGIN can take into account additional information that can drastically improve the overall performance. That is, a publisher can specify the content creation date while a broker can log time when the content was received. A subscriber can take advantage of this extra information by expressing additional constraints in subscription. For instance, a subscriber can express her subscription as: *all pieces of content matching with my interest, where the content is created or received in last two hours.* The content creation date and the content received time may help brokers to check whether subscriptions satisfy the published contents, without requiring encrypted matching. Furthermore, a publisher can publish content with Time To Live (TTL), meaning brokers should remove that particular piece of content after expiration of TTL. Similarly, subscribers also can include TTL with subscriptions to indicate that brokers can remove subscriptions from the network after expiration of TTL. The inclusion of TTL, in both the content and the subscription, will reduce both computation and storage needs.

5.9.3 Key Management

Deployment in Practical Scenarios. There are various options to setup the TKMA, an offline trusted entity that distributes keying material. It mainly depends on the scenario for which PIDGIN is deployed. For instance, for the military scenarios, it can be administrated by a military headquarter; similarly, in organisations, the admin department can manage it. However, it is challenging to setup the TKMA for various civilian applications. For those kinds of applications, the town or city administration could be one option. For emerging scenarios, such as social events, the organising authorities (such as event organisers) might own the TKMA.

Distributed TKMA. Without loss of generality, we can make the TKMA distributed.

There are two main types of keys that are generated by the TKMA, the CP-ABE and the search keys. There are alraedy solutions for setting up multi-authority ABE [116, 117], where the CP-ABE key authorities can be distributed. Whereas, the key authority for generating the search keys is inherently distributed.

5.10 Related Work

The problem of encrypted matching in opportunistic networks is an instance of the wider problem of a search over encrypted data. Song *et al.* [35] propose a search scheme over encrypted data based on symmetric keys. The symmetric nature of the scheme rules out its applicability where mobile nodes communicate with each other without any prior contacts. The PEKS scheme [36] supports a search on encrypted data in the public key setting. In PIDGIN, we use the PEKS scheme as a building block; moreover, its usage in isolation does not solve privacy and confidentiality issues in opportunistic networks because it lacks the ability to regulate access on content while providing collusion-resistant decryption keys.

The ABE schemes can regulate access to content while guaranteeing collusion resistance. However, both variants of ABE including CP-ABE [47] and KP-ABE [49] do not protect the policies and attributes associated with content, respectively. In PIDGIN, we use CP-ABE [47] as a building block but only after we protect the policies because the original CP-ABE scheme does not specifically protect them. The complimentary KP-ABE [49] scheme does not protect attributes. While, Goyal *et al.* leave the problem of encrypted attributes as open [49], we address this challenging issue in this chapter.

ESPOON [28] can protect security policies in outsourced environments. In [75], we propose ESPOON$_{ERBAC}$ that extends ESPOON with Encrypted Role-Based Access Control (ERBAC) that is deployable in outsourced environments. However, these solutions [28, 75, 112] assume no collusion between a user and a server. Thus, none of these solutions [28, 75, 112] are applicable to opportunistic networks in which each node can serve as all three roles including publisher, broker and subscriber.

There are schemes that protect policies [22, 23, 118, 119] and assume that the policy is evaluated at the receiver's end. Furthermore, schemes offering hidden credentials [51] and hidden policies [120] assume direct interaction between the sender and the receiving parties. Unfortunately, all such schemes cannot work in opportunistic networks where policy enforcement is delegated to untrusted brokers.

Shikfa *et al.* [24] propose a method that provides privacy and confidentiality in context-based forwarding. However, their method is a different dimension of work than ours. In fact, their proposed scheme disseminates information in one direction, i.e., from publishers,

without taking into account whether a subscriber is interested or not. In other words, it does not provide opportunity for a subscriber to subscribe. Moreover, our proposed scheme regulates access to content while offering more expressive and fine-grained policies as compared to the one proposed in [24].

Nabeel *et al.* [121] provide a solution for preserving privacy in content based publish-subscribe systems. In their approach, brokers in outsourced environments make routing decisions without knowing the content. However, they assume that subscribers get registered with publishers prior to any communication and publishers share the symmetric key with subscribers. This solution cannot work in opportunistic network settings where loosely-coupled publishers and subscribers do not require any registration or key sharing with each other.

In the context of publish-subscribe systems, there are many solutions that address privacy and security issues [25–27]. However, state-of-the-art techniques are mainly based on centralised solutions that cannot be applied to opportunistic networks, where each node may serve as a publisher, a broker and a subscriber.

5.11 Chapter Summary

This chapter presented PIDGIN, a privacy preserving interest and content sharing scheme for opportunistic networks. In PIDGIN, access policies are enforced by brokers such that they neither learn content and associated policies nor compromise privacy of subscribers. To show the feasibility of our approach, we implemented PIDGIN and evaluated its performance by measuring the overhead incurred by cryptographic operations when run on a smartphone.

In Chapter 2-4, we have investigated how to enforce sensitive security policies in outsourced environments while this chapter has addressed how sensitive policies can be enforced in distributed environments. Hence, we covered enforcement of sensitive security policies in both outsourced and distributed environments. In the next chapter, we summarise our contributions and highlight some directions for future work.

Chapter 6

Conclusions and Future Work

In this dissertation, we have addressed a fundamental issue of establishing trust in untrusted environments by protecting access policies and data. In particular, we have investigated how to enforce sensitive policies in outsourced and distributed environments. In our approach, the data is encrypted under expressive access control policies that are attached with the encrypted data. Our proposed mechanisms enforce those policies such that private information is not revealed during the policy deployment and evaluation phases. Furthermore, we offer the full-fledged RBAC mechanism (including role hierarchies and dynamic security constraints) for large enterprises with complex user management.

In our work, we have presented some motivational scenarios. Besides what we have considered, there could be other application scenarios as well. For data outsourcing, we can imagine investigation and security agencies that might require data protection, as well as secure enforcement of sensitive policies. Similarly, we can apply our policy enforcement mechanism in opportunistic networks to report and control crimes in developing countries, where the Internet connectivity is poor or unaffordable. In developed countries, we can think of more sophisticated use cases, such as partially offloading the central Content Delivery Network (CDN) by employing our proposed mechanism so that subscribers can download content from neighbourhood, thus reducing the burden on the centralised server.

In this chapter, we briefly summarise the research contributions of the dissertation and outline some future directions emerging from this work.

6.1 Summary of the Contributions

The core contributions of this dissertation are stated as follows:

ESPOON: Enforcement of Sensitive Policies in Outsourced Environments. In Chapter 2, we have addressed the challenging issue of enforcing sensitive policies in out-

sourced environments while protecting confidentiality of access policies. Our proposed solution, ESPOON, provides a clear separation between security policies and the enforcement mechanism. ESPOON does not reveal private information about access policies or the access request. In fact, we implement ESPOON as an outsourced service without compromising the confidentiality of policies under the assumption that the service provider is honest-but-curious. Furthermore, ESPOON supports contextual conditions and incorporates contextual information during the policy evaluation phase. Contextual conditions are expressive because they include non-monotonic boolean expressions and range queries. The system entities do not share any keys; therefore, if a user is deleted or revoked, the system is still able to perform its operations without requiring any re-encryption of policies. Last but not least, we have implemented a prototype of ESPOON to measure overheads incurred by cryptographic operations during the policy deployment and evaluation phases.

ESPOON$_{ERBAC}$: Supporting RBAC Policies and Role Hierarchies. In Chapter 3, we have extended ESPOON with RBAC policies and proposed ESPOON$_{ERBAC}$. In ESPOON$_{ERBAC}$, users are assigned roles and permissions are assigned to roles. A user can execute the permission if she is active in a role managed by the session maintained in outsourced environments. Besides the basic RBAC policies, ESPOON$_{ERBAC}$ incorporates roles hierarchies, where roles can be inherited. For developing prototype of ESPOON$_{ERBAC}$, we have extended prototype of ESPOON. Finally, we have measured the computational overheads incurred by ESPOON$_{ERBAC}$ operations.

E-GRANT: Facilitating RBAC with Dynamic Constraints. In Chapter 4, we have focused on the enforcement of dynamic security constraints without revealing sensitive information to the untrusted infrastructure. The dynamic constraints include DSoD and CW. For enforcement of dynamic constraints, we have developed E-GRANT. E–GRANT can seamlessly be integrated with ESPOON$_{ERBAC}$. Finally, we have developed the prototype and reported performance overhead of E-GRANT.

We would like to mention that ESPOON, ESPOON$_{ERBAC}$ and E-GRANT can be deployed as SaaS.

PIDGIN: Protecting Privacy and Confidentiality in Opportunistic Networks. In Chapter 5, we have investigated how to exchange content and interest without (i) providing any access to unauthorised brokers and compromising privacy of subscribers. The solution we propose is PIDGIN that aims at regulating access by encrypting content

using CP-ABE policies. The CP-ABE policies are very expressive and specify who can gain access to content. In PIDGIN, CP-ABE policies and tag associated with content are further encrypted using the PEKS scheme. Therefore, brokers match subscriber's interest against content polices without compromising privacy of subscribers. Furthermore, unauthorised brokers do not gain access to content and nodes gain access to content if they satisfy fine-grained policies specified by the publishers. Moreover, the system provides a scalable key management, where loosely-coupled publishers and subscribers do not share any keys. Finally, we have developed a prototype of PIDGIN and analysed the performance of involved cryptographic algorithms by running PIDGIN on smartphones.

6.2 Future Directions

The research work described in this dissertation can be extended along several directions.

Accountable Access Control Mechanisms. In this dissertation, we have proposed how sensitive policies can be enforced. One possible direction of future research is to explore ways of making the enforcement architecture accountable in untrusted environments, thus preventing service providers (or brokers) to repudiate the operations that have been performed. The mechanism should allow service providers to generate genuine audit logs without revealing private information about both data and access policies. However, an auditing authority must be able to retrieve information about who accessed the data and what policy was enforced against any access request.

Negative Authorisation Policies and Conflict Resolution. In our proposed solutions, we have considered positive authorisation policies in untrusted environments. It would be interesting to investigate how to support negative authorisation policies. Since negative and positive authorisation policies might raise conflicts, conflict resolution of policies in untrusted environments might be another interesting topic of research.

Making Policy Outsourcing Distributed. Another substantial part of our future research aims at re-engineering the architecture in a distributed manner in order to run several instances of the proposed system on multiple nodes of the service provider. One of the key aspects here is to adapt the number of instances to the actual request load for offering a reasonable Quality of Service (QoS) without over-provisioning the resources.

Scalable and Collusion-Resistant Access Control Models. Generally, access control models in the literature are only either scalable or collusion-resistant. In our view,

proposing a scalable and collusion-resistant access model for outsourced environments is still an open challenge. Besides that, developing an efficient cryptographic construction and implementing it efficiently are also among open research challenges.

Protection of Policy Structure. In our proposed mechanisms, we express an access control policy as a tree, where leaf nodes of the tree are encrypted while internal nodes (including AND, OR and threshold gates) are in cleartext. Protection of this policy structure is also an open challenge. More specifically, it is a challenging issue to support expressive access control policies such that service providers do not learn any information about structure of policies being enforced.

Key Revocation in Distributed Settings. In distributed settings (including opportunistic networks), revoking a key is quite problematic. The issue is that one cannot inform all nodes about keys that have been revoked because there is no centralised authority for management of key revocation. That is, the key revocation information could epidemically be disseminated only through nodes, say from a group of nodes to other nodes in the network. We believe that investigating an approach to efficiently address the key revocation problem would make distributed networks more practical.

Efficient Pairing Implementation. As evident from the performance evaluation, the real bottleneck is the overhead incurred by pairing operations at brokers in opportunistic networks. Basically, an efficient pairing implementation would drastically improve the performance of the system. As future work, we would investigate possible optimisations and the use of an efficient pairing implementation, such as the one proposed in [122]. Alternatively, we can consider implementation of cryptographic constructs at processor level, i.e., support of pairing operations in a cryptographic processor.

6.3 Closing Remarks

This work has appeared in international journals, conferences and workshops (See Appendix A). In particular, the basic architecture for enforcing sensitive security policies in outsourced environments has been presented in [28]. The proposed architecture has been extended to support RBAC style of access policies in outsourced environments and is described in [75, 76]. The work on enforcing RBAC in outsourced environments has further been extended by incorporating security constraints in RBAC, which is presented in [89]. The data protection issues have been tackled in [73, 123]. The scenario based security and privacy issues have been listed in [124]. For brevity reasons, we have included

only the research work that fall within the core topic of this dissertation and excluded some published work [73, 123, 124]. Finally, the issue of policies and data protection in distributed environments has been analysed and addressed in [125].

Bibliography

[1] "Dropbox." https://www.dropbox.com/. Last Accessed: October 30, 2013.

[2] Google, "Google cloud storage pricing." https://cloud.google.com/pricing/cloud-storage, February 2013. Last Accessed: October 30, 2013.

[3] Amazon, "Amazon simple storage service (Amazon S3)." http://aws.amazon.com/s3/#pricing, February 2013. Last Accessed: October 30, 2013.

[4] Gartner, "Gartner says cloud-based security services market to reach $2.1 billion in 2013." http://www.gartner.com/newsroom/id/2616115, October 2013. Last Accessed: October 30, 2013.

[5] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, pp. 50–58, Apr. 2010.

[6] B. Hore, S. Mehrotra, M. Canim, and M. Kantarcioglu, "Secure multidimensional range queries over outsourced data," *The VLDB Journal*, vol. 21, no. 3, pp. 333–358, 2012.

[7] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, (New York, NY, USA), pp. 965–976, ACM, 2012.

[8] C. Bösch, R. Brinkman, P. Hartel, and W. Jonker, "Conjunctive wildcard search over encrypted data," in *Secure Data Management* (W. Jonker and M. Petkovic, eds.), vol. 6933 of *Lecture Notes in Computer Science*, pp. 114–127, Springer Berlin Heidelberg, 2011.

[9] N. Cao, C. Wang, M. Li, K. Ren, and W. Lou, "Privacy-preserving multi-keyword ranked search over encrypted cloud data," in *INFOCOM, 2011 Proceedings IEEE*, pp. 829–837, 2011.

[10] M. Li, S. Yu, N. Cao, and W. Lou, "Authorized private keyword search over encrypted data in cloud computing," in *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pp. 383–392, 2011.

[11] Y. Yang, H. Lu, and J. Weng, "Multi-user private keyword search for cloud computing," in *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pp. 264–271, 2011.

[12] B. Zhu, B. Zhu, and K. Ren, "PEKSrand: Providing predicate privacy in public-key encryption with keyword search," in *Communications (ICC), 2011 IEEE International Conference on*, pp. 1–6, 2011.

[13] J. Li, Q. Wang, C. Wang, N. Cao, K. Ren, and W. Lou, "Fuzzy keyword search over encrypted data in cloud computing," in *INFOCOM, 2010 Proceedings IEEE*, pp. 1–5, 2010.

[14] C. Wang, N. Cao, J. Li, K. Ren, and W. Lou, "Secure ranked keyword search over encrypted cloud data," in *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, pp. 253–262, 2010.

[15] Y. Yang, F. Bao, X. Ding, and R. H. Deng, "Multiuser private queries over encrypted databases," *Int. J. Appl. Cryptol.*, vol. 1, pp. 309–319, Aug. 2009.

[16] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-Based Access Control Models," *Computer*, vol. 29, no. 2, pp. 38–47, 1996.

[17] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, G. Livraga, S. Paraboschi, and P. Samarati, "Enforcing dynamic write privileges in data outsourcing," *Elsevier Computers & Security (COSE)*, 2013.

[18] M. Raykova, H. Zhao, and S. Bellovin, "Privacy enhanced access control for outsourced data sharing," in *Financial Cryptography and Data Security* (A. Keromytis, ed.), vol. 7397 of *Lecture Notes in Computer Science*, pp. 223–238, Springer Berlin Heidelberg, 2012.

[19] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati, "A data outsourcing architecture combining cryptography and access control," in *Proceedings of the 2007 ACM workshop on Computer security architecture*, CSAW '07, (New York, NY, USA), pp. 63–69, ACM, 2007.

[20] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. 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, pp. 123–134, VLDB Endowment, 2007.

[21] L. Pelusi, A. Passarella, and M. Conti, "Opportunistic networking: data forwarding in disconnected mobile ad hoc networks," *Communications Magazine, IEEE*, vol. 44, no. 11, pp. 134–141, 2006.

[22] E. Shen, E. Shi, and B. Waters, "Predicate privacy in encryption systems," in *Theory of Cryptography* (O. Reingold, ed.), vol. 5444 of *Lecture Notes in Computer Science*, pp. 457–473, Springer Berlin Heidelberg, 2009.

[23] J. Katz, A. Sahai, and B. Waters, "Predicate encryption supporting disjunctions, polynomial equations, and inner products," *Journal of Cryptology*, vol. 26, no. 2, pp. 191–224, 2013.

[24] A. Shikfa, M. Önen, and R. Molva, "Privacy and confidentiality in context-based and epidemic forwarding," *Computer Communications*, vol. 33, no. 13, pp. 1493 – 1504, 2010.

[25] S. Choi, G. Ghinita, and E. Bertino, "A privacy-enhancing content-based publish/-subscribe system using scalar product preserving transformations," in *Database and Expert Systems Applications* (P. G. Bringas, A. Hameurlain, and G. Quirchmayr, eds.), vol. 6261 of *Lecture Notes in Computer Science*, pp. 368–384, Springer Berlin Heidelberg, 2010.

[26] N. Shang, M. Nabeel, F. Paci, and E. Bertino, "A privacy-preserving approach to policy-based content dissemination," in *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pp. 944–955, 2010.

[27] M. Srivatsa and L. Liu, "Secure event dissemination in publish-subscribe networks," in *Distributed Computing Systems, 2007. ICDCS '07. 27th International Conference on*, pp. 22–22, 2007.

[28] M. R. Asghar, M. Ion, G. Russello, and B. Crispo, "ESPOON: enforcing encrypted security policies in outsourced environments," in *The Sixth International Conference on Availability, Reliability and Security*, ARES'11, pp. 99–108, IEEE Computer Society, August 2011.

[29] K. Ondo and M. Smith, "Outside it: the case for full it outsourcing," *Healthcare financial management : journal of the Healthcare Financial Management Association*, vol. 60, no. 2, pp. 92–98, 2006.

[30] C. Dong, G. Russello, and N. Dulay, "Shared and searchable encrypted data for untrusted servers," *Journal of Computer Security*, vol. 19, no. 3, pp. 367–397, 2011.

[31] S. Kamara and K. Lauter, "Cryptographic cloud storage," in *Financial Cryptography and Data Security* (R. Sion, R. Curtmola, S. Dietrich, A. Kiayias, J. Miret, K. Sako, and F. Sebé, eds.), vol. 6054 of *Lecture Notes in Computer Science*, pp. 136–149, Springer Berlin / Heidelberg, 2010.

[32] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, G. Pelosi, and P. Samarati, "Preserving confidentiality of security policies in data outsourcing," in *Proceedings of the 7th ACM workshop on Privacy in the electronic society*, WPES '08, (New York, NY, USA), pp. 75–84, ACM, 2008.

[33] S. De Capitani Di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati, "Encryption policies for regulating access to outsourced data," *ACM Trans. Database Syst.*, vol. 35, pp. 12:1–12:46, May 2010.

[34] G. Russello, C. Dong, and N. Dulay, "Authorisation and conflict resolution for hierarchical domains," *Policies for Distributed Systems and Networks, IEEE International Workshop on*, vol. 0, pp. 201–210, 2007.

[35] D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *Security and Privacy, 2000. S P 2000. Proceedings. 2000 IEEE Symposium on*, pp. 44–55, 2000.

[36] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano, "Public key encryption with keyword search," in *Advances in Cryptology - EUROCRYPT 2004* (C. Cachin and J. Camenisch, eds.), vol. 3027 of *Lecture Notes in Computer Science*, pp. 506–522, Springer Berlin / Heidelberg, 2004.

[37] P. Golle, J. Staddon, and B. Waters, "Secure conjunctive keyword search over encrypted data," in *Applied Cryptography and Network Security* (M. Jakobsson, M. Yung, and J. Zhou, eds.), vol. 3089 of *Lecture Notes in Computer Science*, pp. 31–45, Springer Berlin / Heidelberg, 2004.

[38] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," in *Proceedings of the 13th ACM conference on Computer and communications security*, CCS '06, (New York, NY, USA), pp. 79–88, ACM, 2006.

[39] Y. Hwang and P. Lee, "Public key encryption with conjunctive keyword search and its extension to a multi-user system," in *Pairing-Based Cryptography - Pairing 2007*

(T. Takagi, T. Okamoto, E. Okamoto, and T. Okamoto, eds.), vol. 4575 of *Lecture Notes in Computer Science*, pp. 2–22, Springer Berlin / Heidelberg, 2007.

[40] D. Boneh and B. Waters, "Conjunctive, subset, and range queries on encrypted data," in *Theory of Cryptography* (S. Vadhan, ed.), vol. 4392 of *Lecture Notes in Computer Science*, pp. 535–554, Springer Berlin / Heidelberg, 2007.

[41] P. Wang, H. Wang, and J. Pieprzyk, "Threshold privacy preserving keyword searches," in *SOFSEM 2008: Theory and Practice of Computer Science* (V. Geffert, J. Karhumäki, A. Bertoni, B. Preneel, P. Návrat, and M. Bieliková, eds.), vol. 4910 of *Lecture Notes in Computer Science*, pp. 646–658, Springer Berlin / Heidelberg, 2008.

[42] J. Baek, R. Safavi-Naini, and W. Susilo, "Public key encryption with keyword search revisited," in *Computational Science and Its Applications - ICCSA 2008* (O. Gervasi, B. Murgante, A. Laganà, D. Taniar, Y. Mun, and M. Gavrilova, eds.), vol. 5072 of *Lecture Notes in Computer Science*, pp. 1249–1259, Springer Berlin / Heidelberg, 2008.

[43] H. S. Rhee, J. H. Park, W. Susilo, and D. H. Lee, "Trapdoor security in a searchable public-key encryption scheme with a designated tester," *Journal of Systems and Software*, vol. 83, no. 5, pp. 763 – 771, 2010.

[44] J. Shao, Z. Cao, X. Liang, and H. Lin, "Proxy re-encryption with keyword search," *Information Sciences*, vol. 180, no. 13, pp. 2576 – 2587, 2010.

[45] M. J. Atallah, M. Blanton, N. Fazio, and K. B. Frikken, "Dynamic and efficient key management for access hierarchies," *ACM Trans. Inf. Syst. Secur.*, vol. 12, pp. 18:1–18:43, January 2009.

[46] S. Narayan, M. Gagné, and R. Safavi-Naini, "Privacy preserving EHR system using attribute-based infrastructure," in *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*, CCSW '10, (New York, NY, USA), pp. 47–52, ACM, 2010.

[47] J. Bethencourt, A. Sahai, and B. Waters, "Ciphertext-Policy Attribute-Based Encryption," in *Security and Privacy, 2007. SP '07. IEEE Symposium on*, pp. 321 –334, May 2007.

[48] A. Sahai and B. Waters, "Fuzzy Identity-Based Encryption," in *Advances in Cryptology - EUROCRYPT 2005* (R. Cramer, ed.), vol. 3494 of *Lecture Notes in Computer Science*, pp. 557–557, Springer Berlin / Heidelberg, 2005.

[49] V. Goyal, O. Pandey, A. Sahai, and B. 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, (New York, NY, USA), pp. 89–98, ACM, 2006.

[50] R. Ostrovsky, A. Sahai, and B. Waters, "Attribute-based encryption with non-monotonic access structures," in *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, (New York, NY, USA), pp. 195–203, ACM, 2007.

[51] J. E. Holt, R. W. Bradshaw, K. E. Seamons, and H. Orman, "Hidden credentials," in *Proceedings of the 2003 ACM workshop on Privacy in the electronic society*, WPES '03, (New York, NY, USA), pp. 1–8, ACM, 2003.

[52] R. W. Bradshaw, J. E. Holt, and K. E. Seamons, "Concealing complex policies with hidden credentials," in *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, (New York, NY, USA), pp. 146–157, ACM, 2004.

[53] C. Gentry, *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford, CA, USA, 2009. AAI3382729.

[54] M. Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," in *Advances in Cryptology - EUROCRYPT 2010* (H. Gilbert, ed.), vol. 6110 of *Lecture Notes in Computer Science*, pp. 24–43, Springer Berlin Heidelberg, 2010.

[55] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) LWE," in *Foundations of Computer Science (FOCS), 2011 IEEE 52nd Annual Symposium on*, pp. 97–106, 2011.

[56] M. Naehrig, K. Lauter, and V. Vaikuntanathan, "Can homomorphic encryption be practical?," in *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, CCSW '11, (New York, NY, USA), pp. 113–124, ACM, 2011.

[57] C. Gentry and S. Halevi, "Implementing Gentry's fully-homomorphic encryption scheme," in *Advances in Cryptology - EUROCRYPT 2011* (K. Paterson, ed.), vol. 6632 of *Lecture Notes in Computer Science*, pp. 129–148, Springer Berlin Heidelberg, 2011.

[58] "An implementation of homomorphic encryption." `https://github.com/shaih/HElib`. Last Accessed: October 14, 2013.

[59] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Advances in Cryptology - EUROCRYPT 99* (J. Stern, ed.), vol. 1592 of *Lecture Notes in Computer Science*, pp. 223–238, Springer Berlin Heidelberg, 1999.

[60] M. Mont, S. Pearson, and P. Bramhall, "Towards accountable management of identity and privacy: sticky policies and enforceable tracing services," in *Database and Expert Systems Applications, 2003. Proceedings. 14th International Workshop on*, pp. 377–382, 2003.

[61] D. W. Chadwick and S. F. Lievens, "Enforcing "sticky" security policies throughout a distributed application," in *Proceedings of the 2008 Workshop on Middleware Security*, MidSec '08, (New York, NY, USA), pp. 1–6, ACM, 2008.

[62] S. Pearson and M. C. Mont, "Sticky policies: An approach for managing privacy across multiple parties," *Computer*, vol. 44, no. 9, pp. 60–68, 2011.

[63] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan, "Private information retrieval," *J. ACM*, vol. 45, pp. 965–981, Nov. 1998.

[64] "Private information retrieval." `http://crypto.stanford.edu/pir-library/`. Last Accessed: October 30, 2013.

[65] S. Yekhanin, "Private information retrieval," *Commun. ACM*, vol. 53, pp. 68–73, Apr. 2010.

[66] P. Williams and R. Sion, "Usable PIR," in *NDSS*, The Internet Society, 2008.

[67] I. Goldberg, "Improving the robustness of private information retrieval," in *Security and Privacy, 2007. SP '07. IEEE Symposium on*, pp. 131–148, 2007.

[68] J. Camenisch, M. Dubovitskaya, and G. Neven, "Oblivious transfer with access control," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, (New York, NY, USA), pp. 131–140, ACM, 2009.

[69] J. Camenisch, M. Dubovitskaya, R. Enderlein, and G. Neven, "Oblivious transfer with hidden access control from attribute-based encryption," in *Security and Cryptography for Networks* (I. Visconti and R. Prisco, eds.), vol. 7485 of *Lecture Notes in Computer Science*, pp. 559–579, Springer Berlin Heidelberg, 2012.

[70] F. Olumofin and I. Goldberg, "Revisiting the computational practicality of private information retrieval," in *Financial Cryptography and Data Security* (G. Danezis,

ed.), vol. 7035 of *Lecture Notes in Computer Science*, pp. 158–172, Springer Berlin Heidelberg, 2012.

[71] 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`.

[72] C. Dong, G. Russello, and N. Dulay, "Shared and searchable encrypted data for untrusted servers," in *Data and Applications Security XXII* (V. Atluri, ed.), vol. 5094 of *Lecture Notes in Computer Science*, pp. 127–143, Springer Berlin Heidelberg, 2008.

[73] M. R. Asghar, G. Russello, B. Crispo, and M. Ion, "Supporting complex queries and access policies for multi-user encrypted databases," in *The ACM Workshop on Cloud computing security workshop*, CCSW '13, November 2013.

[74] M. Van Dijk and A. Juels, "On the impossibility of cryptography alone for privacy-preserving cloud computing," in *Proceedings of the 5th USENIX conference on Hot topics in security*, HotSec'10, (Berkeley, CA, USA), pp. 1–8, USENIX Association, 2010.

[75] M. R. Asghar, M. Ion, G. Russello, and B. Crispo, "ESPOON$_{ERBAC}$: Enforcing security policies in outsourced environments," *Elsevier Computers & Security (COSE)*, vol. 35, pp. 2–24, 2013. Special Issue of the International Conference on Availability, Reliability and Security (ARES).

[76] M. R. Asghar, G. Russello, and B. 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, pp. 841–844, ACM, 2011.

[77] A. C. O'Connor and R. J. Loomis, "Economic analysis of Role-Based Access Control," tech. rep., National Institute of Standards and Technology, December 2010. Available at: `http://csrc.nist.gov/groups/SNS/rbac/documents/20101219_RBAC2_Final_Report.pdf`.

[78] J. B. D. Joshi, E. Bertino, A. Ghafoor, and Y. Zhang, "Formal foundations for hybrid hierarchies in gtrbac," *ACM Trans. Inf. Syst. Secur.*, vol. 10, pp. 2:1–2:39, Jan. 2008.

[79] Y.-G. Kim and J. Lim, "Dynamic activation of role on RBAC for ubiquitous applications," in *Proceedings of the 2007 International Conference on Convergence*

Information Technology, ICCIT '07, (Washington, DC, USA), pp. 1148–1153, IEEE Computer Society, 2007.

[80] J. B. Joshi, E. Bertino, U. Latif, and A. Ghafoor, "A generalized temporal role-based access control model," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, pp. 4–23, 2005.

[81] M. Strembeck and G. Neumann, "An integrated approach to engineer and enforce context constraints in RBAC environments," *ACM Trans. Inf. Syst. Secur.*, vol. 7, pp. 392–427, August 2004.

[82] G. Neumann and M. 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, (New York, NY, USA), pp. 65–79, ACM, 2003.

[83] E. Bertino, P. A. Bonatti, and E. Ferrari, "Trbac: A temporal role-based access control model," *ACM Trans. Inf. Syst. Secur.*, vol. 4, pp. 191–233, Aug. 2001.

[84] E. Lupu and M. Sloman, "Reconciling role based management and role based access control," in *Proceedings of the second ACM workshop on Role-based access control*, RBAC '97, (New York, NY, USA), pp. 135–141, ACM, 1997.

[85] J. Crampton and H. Khambhammettu, "Delegation in role-based access control," *International Journal of Information Security*, vol. 7, no. 2, pp. 123–136, 2008.

[86] S. De Capitani di Vimercati and P. Samarati, "Mandatory access control policy (mac)," in *Encyclopedia of Cryptography and Security* (H. C. van Tilborg and S. Jajodia, eds.), pp. 758–758, Springer US, 2011.

[87] S. De Capitani di Vimercati, "Discretionary access control policies (dac)," in *Encyclopedia of Cryptography and Security* (H. C. van Tilborg and S. Jajodia, eds.), pp. 356–358, Springer US, 2011.

[88] S. Godik, A. Anderson, B. Parducci, P. Humenn, and S. Vajjhala, "OASIS eXtensible Access Control 2 Markup Language (XACML) 3," tech. rep., Tech. rep., OASIS, 2002.

[89] M. R. Asghar, G. Russello, and B. Crispo, "E-GRANT: Enforcing encrypted dynamic security constraints in the cloud," 2013. (In submission).

[90] Google, "Introducing google drive... yes, really." http://googleblog.blogspot.it/2012/04/introducing-google-drive-yes-really.html, April 2012. Google Official Blog, Last Accessed: April 4, 2013.

[91] P. Pehrson, "Adobes new SAAS model." http://www.paulpehrson.com/2011/04/11/adobes-new-software-as-a-service-model/, April 2011. Last Accessed: April 4, 2013.

[92] SAP, "SAP business bydesign." http://www.sap.com/solutions/technology/cloud/business-by-design/highlights/index.epx. Last Accessed: September 17, 2013.

[93] M. Kohlbacher, "The effects of process orientation on customer satisfaction, product quality and time-based performance," October 2009. presented at the 29th Annual International Conference of the Strategic Management Society in Washington D.C., USA.

[94] G. Kong and J. Li, "Research on RBAC-based Separation of Duty Constraints," *Journal of Information and Computing Science*, vol. 2, pp. 235–240, August 2007.

[95] M. Nash and K. Poland, "Some conundrums concerning separation of duty," in *Research in Security and Privacy, 1990. Proceedings., 1990 IEEE Computer Society Symposium on*, pp. 201 –207, May 1990.

[96] D. Brewer and M. Nash, "The chinese wall security policy," in *Security and Privacy, 1989. Proceedings., 1989 IEEE Symposium on*, pp. 206 –214, May 1989.

[97] J. Crampton and H. Khambhammettu, "A framework for enforcing constrained RBAC policies," in *Computational Science and Engineering, 2009. CSE '09. International Conference on*, vol. 3, pp. 195 –200, August 2009.

[98] G.-J. Ahn and R. Sandhu, "Role-based authorization constraints specification," *ACM Trans. Inf. Syst. Secur.*, vol. 3, pp. 207–226, November 2000.

[99] V. Gligor, S. Gavrila, and D. Ferraiolo, "On the formal definition of separation-of-duty policies and their composition," *Security and Privacy, IEEE Symposium on*, vol. 0, p. null, 1998.

[100] "Dynamic Separation of Duties," in *Encyclopedia of Cryptography and Security* (H. C. van Tilborg and S. Jajodia, eds.), pp. 369–369, Springer US, 2011.

[101] G.-J. Ahn and R. Sandhu, "The rsl99 language for role-based separation of duty constraints," in *Proceedings of the Fourth ACM Workshop on Role-based Access Control*, RBAC '99, (New York, NY, USA), pp. 43–54, ACM, 1999.

[102] R. S. Sandhu, "Separation of duties in computerized information systems," in *DBSec*, pp. 179–190, 1990.

[103] S. De Capitani di Vimercati and P. Samarati, "Chinese Wall," in *Encyclopedia of Cryptography and Security* (H. C. van Tilborg and S. Jajodia, eds.), pp. 202–203, Springer US, 2011.

[104] D. Basin, F. Klaedtke, and S. Müller, "Monitoring security policies with metric first-order temporal logic," in *Proceedings of the 15th ACM symposium on Access control models and technologies*, SACMAT '10, (New York, NY, USA), pp. 23–34, ACM, 2010.

[105] A. Armando, S. Ranise, F. Turkmen, and B. Crispo, "Efficient run-time solving of RBAC user authorization queries: pushing the envelope," in *Proceedings of the second ACM conference on Data and Application Security and Privacy*, CODASPY '12, (New York, NY, USA), pp. 241–248, ACM, 2012.

[106] J. Crampton, "Specifying and enforcing constraints in role-based access control," in *Proceedings of the Eighth ACM Symposium on Access Control Models and Technologies*, SACMAT '03, (New York, NY, USA), pp. 43–50, ACM, 2003.

[107] V. C. Hu, D. F. Ferraiolo, and D. R. Kuhn, "Assessment of access control systems," tech. rep., National Institute of Standards and Technology, September 2006. Available at: `http://iris.nyit.edu/~kkhoo/Spring2008/Topics/Topic10/AssessmentofAccessControlSys2006_NISTIR-7316.pdf`.

[108] A. Schaad, P. Spadone, and H. Weichsel, "A case study of separation of duty properties in the context of the austrian "elaw" process.," in *Proceedings of the 2005 ACM symposium on Applied computing*, SAC '05, (New York, NY, USA), pp. 1328–1332, ACM, 2005.

[109] Emarketer, "Smartphones, tablets drive faster growth in ecommerce sales - mobile will take a greater percentage of total ecommerce retail sales." `http://www.emarketer.com/Article/Smartphones-Tablets-Drive-Faster-Growth-Ecommerce-Sales/1009835`, April 2013. Last accessed: September 17, 2013.

[110] "Haggle: An EU Funded Project." `http://www.haggleproject.org/`, June 2010. Last accessed: August 7, 2013.

[111] E. Nordström, P. Gunningberg, and C. Rohner, "A search-based network architecture for mobile devices," tech. rep., Department of Information Technology, Uppsala University, 2009.

[112] A. Kapadia, P. P. Tsang, and S. W. Smith, "Attribute-based publishing with hidden credentials and hidden policies," in *NDSS*, The Internet Society, 2007.

[113] "libfenc: The functional encryption library." `https://code.google.com/p/libfenc/`. Last Accessed: October 30, 2013.

[114] B. Lynn, "PBC: The Pairing-Based Cryptography Library." `http://crypto.stanford.edu/pbc/`. Last Accessed: October 30, 2013.

[115] "GMP: The GNU Multiple Precision Arithmetic Library." `https://gmplib.org/`. Last Accessed: October 30, 2013.

[116] M. Chase and S. S. Chow, "Improving privacy and security in multi-authority attribute-based encryption," in *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, (New York, NY, USA), pp. 121–130, ACM, 2009.

[117] M. Chase, "Multi-authority Attribute Based Encryption," in *Theory of Cryptography* (S. Vadhan, ed.), vol. 4392 of *Lecture Notes in Computer Science*, pp. 515–534, Springer Berlin Heidelberg, 2007.

[118] T. Nishide, K. Yoneyama, and K. Ohta, "Attribute-based encryption with partially hidden encryptor-specified access structures," in *Applied Cryptography and Network Security* (S. Bellovin, R. Gennaro, A. Keromytis, and M. Yung, eds.), vol. 5037 of *Lecture Notes in Computer Science*, pp. 111–129, Springer Berlin Heidelberg, 2008.

[119] J. Lai, R. Deng, and Y. Li, "Fully secure cipertext-policy hiding CP-ABE," in *Information Security Practice and Experience* (F. Bao and J. Weng, eds.), vol. 6672 of *Lecture Notes in Computer Science*, pp. 24–39, Springer Berlin Heidelberg, 2011.

[120] K. Frikken, M. Atallah, and J. Li, "Attribute-based access control with hidden policies and hidden credentials," *Computers, IEEE Transactions on*, vol. 55, no. 10, pp. 1259–1270, 2006.

[121] M. Nabeel, N. Shang, and E. Bertino, "Efficient privacy preserving content based publish subscribe systems," in *Proceedings of the 17th ACM symposium on Access*

Control Models and Technologies, SACMAT '12, (New York, NY, USA), pp. 133–144, ACM, 2012.

[122] G. Grewal, R. Azarderakhsh, P. Longa, S. Hu, and D. Jao, "Efficient implementation of bilinear pairings on ARM processors," in *Selected Areas in Cryptography* (L. Knudsen and H. Wu, eds.), vol. 7707 of *Lecture Notes in Computer Science*, pp. 149–165, Springer Berlin Heidelberg, 2013.

[123] M. R. Asghar, M. Ion, G. Russello, and B. Crispo, "Securing data provenance in the cloud," in *Open Problems in Network Security* (J. Camenisch and D. Kesdogan, eds.), vol. 7039 of *Lecture Notes in Computer Science*, pp. 145–160, Springer Berlin Heidelberg, 2012.

[124] M. R. Asghar and D. Miorandi, "A holistic view of security and privacy issues in smart grids," in *Smart Grid Security* (J. Cuellar, ed.), vol. 7823 of *Lecture Notes in Computer Science*, pp. 58–71, Springer Berlin Heidelberg, 2013.

[125] M. R. Asghar, A. Gehani, B. Crispo, and G. Russello, "PIDGIN: Privacy-preserving interest and content sharing in opportunistic networks," 2013. (In submission).

Appendix A

Research Publications

A.1 Related Publications

In International Journals

1. **Muhammad Rizwan Asghar**, Mihaela Ion, Giovanni Russello, Bruno Crispo, ***ESPOON$_{ERBAC}$: Enforcing Security Policies in Outsourced Environments***, *Elsevier Computers & Security (COSE), volume 35, pages 2-24, 2013.* ***One of three papers from ARES 2011 invited to this journal.***

   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 Control (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 chapter, 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 evalua-

tion showing a limited overhead, thus confirming viability of our approach.

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

2. **Muhammad Rizwan Asghar**, Mihaela Ion, Giovanni Russello, Bruno Crispo, ***E-GRANT: Enforcing Encrypted Dynamic Security Constraints in the Cloud****, 2013. (In submission).*

   Abstract: Cloud computing is an emerging paradigm offering outsourced services to enterprises for storing and processing huge amount of data at very competitive costs. For leveraging the cloud to its fullest potential, organisations require security mechanisms to regulate access on data, particularly at runtime. One of the strong obstacles in widespread adoption of the cloud is to preserve confidentiality of the data. In fact, confidentiality of the data can be guaranteed by employing existing encryption schemes; however, access control mechanisms might leak information about the data they aim to protect. State of the art access control mechanisms can statically enforce constraints such as static separation of duties. The major research challenge is to enforce constraints at runtime, i.e., enforcement of dynamic security constraint (including Dynamic Separation of Duties and Chinese Wall) in the cloud. The main challenge lies in the fact that dynamic security constraints require notion of sessions for managing access histories that might leak information about the sensitive data if they are available as cleartext in the cloud. In this chapter, we present E-GRANT: an architecture able to enforce dynamic security constraints without relying on a trusted infrastructure, which can be deployed as SaaS. In E-GRANT, sessions' access histories are encrypted in such a way that enforcement of constraints is still possible. As a proof-of-concept, we have implemented a prototype and provide a preliminary performance analysis showing a limited overhead, thus confirming the feasibility of our approach.

   Keywords: Secure Cloud Services, Sensitive Dynamic Constraints, Encrypted DSoD, Encrypted Chinese Wall, SaaS Enforcement Mechanism

In International Conferences and Workshops

3. **Muhammad Rizwan Asghar**, Ashish Gehani, Giovanni Russello, Bruno Crispo, ***PIDGIN: Privacy-preserving Interest and Content Sharing in Opportunistic Networks****, 2013. (In submission).*

   Abstract: Opportunistic networks have recently received considerable attention

from both industry and researchers. These networks can be used for many applications without the need for a dedicated IT infrastructure. In the context of opportunistic networks, the application to content sharing in particular has attracted specific attention. To support content sharing, opportunistic networks may implement a publish-subscribe system in which users may publish their own content and indicate interest in other content through subscription. Using a smartphone, any user can act as a broker by opportunistically forwarding both published content and interest within the network. Unfortunately, despite their provision of this great flexibility, opportunistic networks raise serious privacy and security issues. Untrusted brokers can not only compromise the privacy of subscribers by learning their interest but also can gain unauthorised access to the disseminated content. This chapter addresses the research challenges inherent to the exchange of content and interest without: (i) compromising the privacy of subscribers and (ii) providing unauthorised access to untrusted brokers. Specifically, this chapter presents an interest and content sharing solution that addresses these security challenges and preserves privacy in opportunistic networks. We demonstrated the feasibility and efficiency of this solution by implementing a prototype and analysing its performance on real smart phones.

Keywords: Secure Opportunistic Networks, Privacy-preserving Content Sharing, Sensitive Policy Enforcement, Encrypted CP-ABE Policies, Secure Haggle

4. **Muhammad Rizwan Asghar**, Giovanni Russello, Bruno Crispo, Mihaela Ion, ***Supporting Complex Queries and Access Policies for Multi-user Encrypted Databases***, *In Proceedings of The 5th ACM Workshop on Cloud Computing Security Workshop (CCSW) in conjunction with the 20th ACM Conference on Computer and Communications Security (CCS), Berlin, Germany, November 2013.*

Abstract: Cloud computing is an emerging paradigm offering companies (virtually) unlimited data storage and computation at attractive costs. It is a cost-effective model because it does not require deployment and maintenance of any dedicated IT infrastructure. Despite its benefits, it introduces new challenges for protecting the confidentiality of the data. Sensitive data like medical records, business or governmental data cannot be stored unencrypted on the cloud. Companies need new mechanisms to control access to the outsourced data and allow users to query the encrypted data without revealing sensitive information to the cloud provider. State-of-the-art schemes do not allow complex encrypted queries over encrypted data in a multi-user setting. Instead, those are limited to keyword searches or conjunctions

of keywords. This chapter extends work on multi-user encrypted search schemes by supporting SQL-like encrypted queries on encrypted databases. Furthermore, we introduce access control on the data stored in the cloud, where any administrative actions (such as updating access rights or adding/deleting users) do not require re-distributing keys or re-encryption of data. Finally, we implemented our scheme and presented its performance, thus showing feasibility of our approach.

Keywords: Encrypted Databases, Complex Encrypted Queries, Access Control, Data Outsourcing

5. **Muhammad Rizwan Asghar**, Daniele Miorandi, ***A Holistic View of Security and Privacy Issues in Smart Grids***, *In Jorge Cuellar, editor, Smart Grid Security, volume 7823 of Lecture Notes in Computer Science, pages 58-71, Springer Berlin Heidelberg, 2013.*

   Abstract: The energy system is undergoing a radical transformation. The coupling of the energy system with advanced information and communication technologies is making it possible to monitor and control in real-time generation, transport, distribution and consumption of energy. In this context, a key enabler is represented by smart meters, devices able to monitor in near real-time the consumption of energy by consumers.

   If, on one hand, smart meters automate the process of information flow from endpoints to energy suppliers, on the other hand, they may leak sensitive information about consumers. In this chapter, we review the issues at stake and the research challenges that characterise smart grids from a privacy and security standpoint.

   Keywords: Privacy, Data Security, Smart Meters, Smart Grids, Prosumers

6. **Muhammad Rizwan Asghar**, Mihaela Ion, Giovanni Russello, Bruno Crispo, ***Securing Data Provenance in the Cloud***, *In Jan Camenisch and Dogan Kesdogan, editors, Open Problems in Network Security, volume 7039 of Lecture Notes in Computer Science, pages 145-160, Springer Berlin Heidelberg, 2012.*

   Abstract: Cloud storage offers the flexibility of accessing data from anywhere at any time while providing economical benefits and scalability. However, cloud stores lack the ability to manage data provenance. Data provenance describes how a particular piece of data has been produced. It is vital for a post-incident investigation, widely used in healthcare, scientific collaboration, forensic analysis and legal proceedings. Data provenance needs to be secured since it may reveal private information about the sensitive data while the cloud service provider does not guarantee confidential-

ity of the data stored in dispersed geographical locations. This chapter proposes a scheme to secure data provenance in the cloud while offering the encrypted search.

Keywords: Secure Data Provenance, Encrypted Cloud Storage, Security, Privacy

7. **Muhammad Rizwan Asghar**, Giovanni Russello, 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. ACM, 2011.*

   Abstract: The enforcement of security policies is an open challenge in environments where the IT infrastructure has been outsourced to a third party. Although the outsourcing allows companies to gain economical benefits and scalability, it imposes the threat of leaking the private information about the sensitive data managed and processed by untrusted parties. In this work, we propose an architecture to enforce Role-Based Access Control (RBAC) style of authorisation policies in outsourced environments. As a proof of concept, we have implemented a demo and measured the performance overhead incurred by the proposed architecture.

   Keywords: Encrypted RBAC, Encrypted Policy Enforcement, Data Outsourcing, Security, Privacy

8. **Muhammad Rizwan Asghar**, Mihaela Ion, Giovanni Russello, Bruno Crispo, ***ESPOON: Enforcing Encrypted Security Policies in Outsourced Environments***, *In The Sixth IEEE International Conference on Availability, Reliability and Security, ARES'11, pages 99-108. IEEE Computer Society, August 2011 (Full paper* ***acceptance rate: 20%****).*

   Abstract: The enforcement of security policies in outsourced environments is still an open challenge for policy-based systems. On the one hand, taking the appropriate security decision requires access to the policies. However, if such access is allowed in an untrusted environment then confidential information might be leaked by the policies. Current solutions are based on cryptographic operations that embed security policies with the security mechanism. Therefore, the enforcement of such policies is performed by allowing the authorised parties to access the appropriate keys. We believe that such solutions are far too rigid because they strictly intertwine authorisation policies with the enforcing mechanism.

   In this paper, we want to address the issue of enforcing security policies in an untrusted environment while protecting the policy confidentiality. Our solution ESPOON is aiming at providing a clear separation between security policies and the

enforcement mechanism. However, the enforcement mechanism should learn as less as possible about both the policies and the requester attributes.

Keywords: Encrypted Policies, Policy Protection, Sensitive Policy Evaluation, Data Outsourcing, Cloud Computing, Privacy, Security

A.2 Other Publications

In International Conferences and Workshops

9. Soudip Roy Chowdhury, Muhammad Imran, **Muhammad Rizwan Asghar**, Sihem Amer-Yahia, Carlos Castillo, ***Tweet4act: Using Incident-Specific Profiles for Classifying Crisis-Related Messages***, *In Proceedings of The 10th International Conference on Information Systems for Crisis Response and Management (ISCRAM), Baden-Baden, Germany, May 2013.*

10. **Muhammad Rizwan Asghar**, Giovanni Russello, ***ACTORS: A Goal-Driven Approach for Capturing and Managing Consent in e-Health Systems***, *In 2012 IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY), pages 61-69, 2012.*

11. **Muhammad Rizwan Asghar**, Giovanni Russello, ***Flexible and Dynamic Consent-Capturing***, *In Jan Camenisch and Dogan Kesdogan, editors, Open Problems in Network Security, volume 7039 of Lecture Notes in Computer Science, pages 119-131, Springer Berlin Heidelberg, 2012.*

Appendix B

Vitae

Muhammad Rizwan Asghar was born in Faisalabad, Pakistan on July 26, 1983. He is a Researcher at CREATE-NET (an International Research Centre based in Trento, Italy) since September 2010. For pursuing his Ph.D., he joined the Security Group at the Department of Information Engineering and Computer Science (DISI), University of Trento, Italy in November 2010. In his Ph.D. research, he investigated privacy preserving enforcement of sensitive policies in outsourced and distributed environments (presented in this dissertation), under the supervision of Associate Prof. Dr. Bruno Crispo and Dr. Giovanni Russello.

He was a Visiting Fellow in the Computer Science Laboratory at the Stanford Research Institute (SRI), California, USA from July to December 2012. Prior to joining CREATE-NET, he was a Research Assistant at the University of Trento from September 2009 to August 2010. He received his M.Sc. degree in Information Security Technology from the Department of Mathematics and Computer Science, Eindhoven University of Technology (TU/e), The Netherlands in 2009 and carried out his research on "DRM Convergence: Interoperability between DRM Systems" as a Master Thesis Student at Ericsson Research Eurolab, Germany. He obtained his B.Sc. (Honours) degree in Computer Science from Punjab University College of Information Technology (PUCIT), University of the Punjab, Lahore, Pakistan in 2006. During his career, he served also as a Software Engineer at international software companies.

His research interests include access control, applied cryptography, cloud computing, security and privacy.

Homepage: http://disi.unitn.it/~asghar