uOttawa

L'Université canadienne
Canada's university

**FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES**

uOttawa

L'Université canadienne
Canada's university

**FACULTY OF GRADUATE AND POSDOCTORAL STUDIES**

Kathryn Garson
AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.C.S.
GRADE / DEGREE

School of Information Technology and Engineering
FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Policy-Based Encryption and its Application in a Hospital System

TITRE DE LA THÈSE / TITLE OF THESIS

Carlisle Adams
DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

Anil Somayaji

Ali Miri

Gary W. Slater
Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

# Policy-Based Encryption and its Application in a Hospital System

by

**Kathryn Garson**

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
In partial fulfillment of the requirements
For the M.A.Sc. degree in Computer Science

School of Information Technology and Engineering
University of Ottawa

© Kathryn Garson, Ottawa, Canada, 2008

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

*Your file Votre référence*
*ISBN: 978-0-494-48634-4*
*Our file Notre référence*
*ISBN: 978-0-494-48634-4*

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

---

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

## Abstract

Hospitals are now using electronic medical records and computer applications in order to provide more efficient and thorough care for their patients. The Mobile Emergency Triage system provides doctors with decision support for emergency care by pulling information from a patient's health record and a medical literature database. In order to achieve compliance with privacy legislations PIPEDA and PHIPA, security and privacy measures must be put in place. A new encryption technology called policy-based encryption proves to be quite useful within a health care environment for providing both encryption and access control. We describe a current policy-based encryption system to be used by MET-$A^3$Support -- a multi-agent clinical decision support system. We offer a Java implementation of the policy-based encryption algorithms. We also give details about how this system will be integrated with MET-$A^3$Support to protect the privacy of patient health information.

## Acknowledgements

I would like to thank the entire MET Research Team including Wojtek Michalowski, Ken Farion, Szymon Wilk and Dympna O'Sullivan. We collaborated on the MET-$A^3$ Support system and they have given me much support and input on my work. Thanks especially to Carlisle Adams, my supervisor who introduced me to many interesting aspects of computer security and guided me through my projects. Thank you to Dr Ali Miri, Dr Anil Somayaji, and Dr Amiya Nayak for attending my thesis defense and providing helpful comments. I also want to thank my parents Paulette and Dave, my brother Brian, and Hong who have supported me and encouraged me.

# Table of contents

**LIST OF TABLES** ..... **I**

**LIST OF FIGURES** ..... **II**

**1 INTRODUCTION** ..... **1**

1.1 MOTIVATION ..... 1
*1.1.1 Network Encryption & Access Control* ..... *1*
*1.1.2 Encryption Only* ..... *3*
1.2 PRIVACY LEGISLATION IN HEALTH CARE ..... 4
1.3 APPLICATION OF PBC TO HOSPITAL SYSTEM ..... 7
*1.3.1 MET-A$^3$Support System* ..... *7*
*1.3.2 Requirements* ..... *8*
1.4 CONTRIBUTIONS ..... 9
*1.4.1 Extension to Policy-Based Encryption System* ..... *9*
*1.4.2 Implementation* ..... *10*
*1.4.3 Integration with Hospital System* ..... *11*
1.5 ORGANIZATION OF THESIS ..... 11

**2 BACKGROUND** ..... **12**

2.1 TOOLS & CONCEPTS ..... 12
*2.1.1 Cryptography* ..... *12*
*2.1.2 Certificate-Based PKC* ..... *13*
*2.1.3 Identity-Based Encryption* ..... *14*
*2.1.4 Policy-Based Encryption* ..... *14*
2.2 LITERATURE REVIEW ..... 16
*2.2.1 Public Key Cryptography with Access Control* ..... *16*
*2.2.2 Identity-Based Encryption* ..... *17*
*2.2.3 Policy-Based & Attribute-Based Encryption* ..... *20*
*2.2.4 Implementations and Applications* ..... *22*
2.3 LIMITATIONS OF CURRENT SOLUTIONS ..... 24
2.4 SUMMARY ..... 25

**3 POLICY BASED ENCRYPTION** ..... **27**

3.1 OVERVIEW ..... 27
3.2 A POLICY-BASED ENCRYPTION SYSTEM ..... 29
*3.2.1 Model* ..... *29*
*3.2.2 Bagga and Molva's System* ..... *30*
*3.2.3 Key Revocation* ..... *33*
3.3 EXTENSION ..... 35
3.4 SUMMARY ..... 36

**4 IMPLEMENTATION** ..... **38**

4.1 OVERVIEW ..... 38
*4.1.1 Packages Used* ..... *38*
*4.1.2 Implementing the algorithms* ..... *41*
4.2 JAVA CRYPTOGRAPHY ARCHITECTURE ..... 43
4.3 ISSUES AND SOLUTIONS ..... 46
*4.3.1 Policy and Credential Computation* ..... *47*
*4.3.2 Message Blocks and Ciphertext Blocks* ..... *48*
4.4 SUMMARY ..... 49

**5 MET-A3SUPPORT SYSTEM ........ 50**

5.1 PRIVACY OF PATIENT MEDICAL INFORMATION ........ 50
5.2 INTEGRATING POLICY-BASED ENCRYPTION INTO A MULTI-AGENT SYSTEM ........ 52
*5.2.1 System Architecture ........ 52*
*5.2.2 Organization of Security ........ 54*
*5.2.3 Protecting Patient Information ........ 55*
*5.2.4 Agent Communication and Encryption ........ 56*
*5.2.5 Policies and Credentials ........ 58*
5.3 WORKFLOW ........ 59
5.4 SECURITY IN THE MET-A3SUPPORT SYSTEM ........ 60
*5.4.1 Authentication Mechanism ........ 61*
*5.4.2 Audit Logs ........ 63*
*5.4.3 Integrity Mechanism ........ 63*
*5.4.4 File Purging ........ 64*
5.5 SUMMARY ........ 65

**6 CONCLUSION ........ 66**

6.1 CONTRIBUTIONS ........ 66
6.2 FUTURE WORK ........ 67
*6.2.1 SEMs ........ 68*
*6.2.2 Testing ........ 68*
*6.2.3 Policies and Credentials ........ 70*
6.3 LESSONS LEARNED ........ 71

**BIBLIOGRAPHY ........ 73**

**GLOSSARY OF TERMS ........ 79**

**JAVA CODE FOR PBCPROVIDER ........ 81**

PBCCIPHER.JAVA ........ 81
PBCKEY.JAVA ........ 88
PBCKEYPAIRGENERATOR.JAVA ........ 89
PBCKEYPARAMETERS.JAVA ........ 90
PBCPRIVATEKEY.JAVA ........ 93
PBCPROVIDER.JAVA ........ 94
PBCPUBLICKEY.JAVA ........ 95
PBCSYSTEMPARAMETERS.JAVA ........ 97
TEST.JAVA ........ 99

## List of Tables

Table 2-1 Comparison of public key encryption systems ... 16
Table 3-1: Policy Look-up ... 28
Table 4-1: Java Packages Used in Implementation ... 40
Table 6-1: Initial Performance Testing of Encryption System ... 69

## List of Figures

Figure 3-1: Policy-Based Encryption ........................................ 30
Figure 3-2: Key Revocation by Updating Policy Time Period ........................................ 33
Figure 4-1: UML Class Diagram for PBCProvider ........................................ 44
Figure 5-1: MET-$A^3$Support Architecture ........................................ 53
Figure 5-2: Agent Using Encryption Service ........................................ 57
Figure 5-3: Task Diagram ........................................ 60

# Chapter 1

# 1 Introduction

The topic of this thesis is exploring policy-based encryption and its application in an electronic hospital system. It will detail the extension to a current policy-based encryption system in order to make it a more complete system to be used in a particular project, the implementation of it, and the integration with a hospital software system. Having the opportunity to provide security technologies for a hospital system project, we wanted to explore the options for a suitable encryption and access control system.

## *1.1 Motivation*

Our motivation comes from the need to provide security for patient information in a hospital system used on a wireless network. This multi-agent system will use patient medical information to provide decision support in an emergency room setting. Patient records will be pulled from a Hospital Information System to be stored and transmitted within our MET-$A^3$Support system. Storing and transmitting such sensitive data requires the use of security technologies. We look at various encryption and access control technologies to address the security needs as discussed next.

### 1.1.1 Network Encryption & Access Control

The Wired Equivalent Privacy (WEP) protocol is intended to provide confidentiality on wireless networks however many weaknesses have been found that could lead to a relatively easy attack[12]. It is not considered to be secure against any more than a casual eavesdropper. For our purposes we need more than this in order to comply with our first privacy goal of protecting sensitive data with a high level of security. Wi-Fi Protected Access (WPA and WPA2) are improvements over WEP although some vulnerabilities still exist[63]. Our hospital system is using the same wireless network as other systems,

therefore it would be desirable to use an encryption and access control method which will isolate our system, such as VPN.

A Virtual Private Network (VPN) is another consideration for securing our wireless network. IPSec and SSL can each be used to implement a VPN. IPSec has been known to be complex to configure and manage and we will not consider it for this implementation[2]. SSL meanwhile has been used as a much easier alternative and has implementations that have proven to be secure.

SSL/TLS uses a handshake protocol to establish shared keys between a client and server. All communications between client and server use the shared key to encrypt data. This creates an 'encrypted channel' between the client and server. SSL provides confidentiality and data integrity through cryptography. The handshake protocol allows for authentication of the server, so the client is assured of a secure connection with the proper server. OpenVPN uses SSL/TLS technology and has been shown to offer the SSL security properties of authentication, confidentiality, and data integrity[40]. Client software for the VPN would be set up on all tablet PCs and server software on MET servers. Users accessing the network would do so by logging in using the VPN software.

Encrypting all transmissions on the network will secure data from anyone who is not logged onto the network. It is still necessary to implement an access control system to manage permissions and access to patient data for employees who access the software. Role-Based Access Control (RBAC) is used to restrict access to resources to only those who have the allowed permissions[29]. The access control is based on the user's role. Each role has permissions assigned to it and each user is assigned a role. If there are any changes to a user's status, that user's role can be changed without affecting the rest of the system. Only the assignments of users to roles need to be changed for changes in the users' permissions. Changes to role definitions would usually be less frequent and can be made without having to assign changes to each individual user. RBAC is a desirable form of access control for sensitive information and security software.

RBAC will provide protection from unauthorized access. It would also allow us to limit information disclosure to a strictly 'need to know basis'. Coupled with SSL network encryption this would cover most of our privacy goals. However we may not need two separate security systems to make our hospital system secure as we will discuss next.

### 1.1.2 Encryption Only

It may be unnecessary to employ both encryption and access control as separate technologies in our system. We present options we considered that combine encryption and access control functionality.

Traditional encryption methods such as public key cryptography (PKC) are cumbersome to apply to access control. Managing keys in this system has often been a limiting factor in real world applications. For our system, we need a way to encrypt medical records for access by multiple recipients, who are not necessarily known at encryption time. PKC doesn't offer this flexibility on its own; intended recipients are known and their keys are used in the encryption process. The key management problem is the main limitation. Users have two keys, their public key and private key. Certificates are created to associate a public key with a person, which is verifiable. These certificates are published in public look-up lists. Someone wishing to encrypt a document for another will have to look up the recipient's public key. The user should then validate the public key which consists of checking that the certificate was signed by an appropriate authority and that the public key has not expired or been revoked. The validation of the public key must be done regularly to ensure its status has not changed. This step can be inefficient and it is also not clear how to encrypt for a role, where there are multiple unknown recipients. If we wanted to use PKC, we would have to add access control methods to allow for encrypting on roles[28]. However this adds unnecessary complexity to the system. It would be easier to encrypt all transmissions on the network and use traditional access control methods. Thus PKC does not offer a simple solution for this case.

Identity-based encryption is a form of public key cryptography that offers the same security with more flexibility in the representation of the public key. In this form of encryption, the public key is a well-known value, such as an email address, which is unique to a person. There is no need to have a certificate binding the identity to the public key since the public key already reflects the identity of the user. Therefore there is no need for look-up lists to manage certificates and revocation. This system removes some of the burden from users. However it is still not clear how to encrypt for multiple unknown recipients.

Policy-based encryption is a more generalized form of identity-based encryption. Instead of encrypting on a value unique to one person, a policy dictating access control rules is used to encrypt a document. Then, any person who satisfies those rules can decrypt. Just like identity-based encryption, the key management problem is avoided. Policies are created based on an organization's access control rules. A user can decrypt when they prove they have permissions that satisfy the policy. Again this can be done by associating permissions with a user account and having the user complete a login. Policy-based encryption also allows for encrypting for multiple recipients. By encrypting on a policy, multiple people who fit that policy can decrypt. This type of encryption provides encryption and role-based access control in one package.

## *1.2 Privacy Legislation in Health Care*

When looking to apply security technologies to health care systems, we must look at the requirements for protection of personal health information. Such requirements are outlined in privacy laws specific to countries and provinces. We focus on the privacy legislation in Canada and more specifically in the province of Ontario, where we wish to use the policy-based encryption in a hospital system. These privacy guidelines are not restricted to Canada. In the US, title II of the Health Insurance Portability and Accountability Act (HIPAA 1996) outlines standards for security and privacy of patient health records[33]. They encourage the use of electronic data interchange in the health care system while protecting information. Physical and technical safeguards similar to the

PIPEDA safeguards discussed next are outlined. Similarly in Europe they have the EU Data Protection Directive (1995) [28].

In Canada, the Personal Information Protection and Electronic Documents Act (PIPEDA) is a law that describes guidelines for the protection of personal information[51]. Personal information is described as any information about an identifiable person that is recorded in any form, which does not include the name, title, business address or telephone number of an employee or organization. The law gives rights to individuals to

- know how and why an organization collects, uses, or discloses personal information
- expect the organization to handle their information appropriately and use it only for purposes for which they have consented
- know who in the organization will be in charge of their personal information
- know that the information held by the organization will be accurate and up-to-date
- complain about how an organization handles their personal information
- obtain access to their information and make corrections
- **expect that the organization will protect their personal information with proper security measures**

PIPEDA is a nation wide act that does not limit itself to health care or personal health information. The Personal Health Information Protection Act (PHIPA) is privacy legislation in the province of Ontario which builds on PIPEDA[51] and describes guidelines specific to health care settings. It is essentially a replacement for PIPEDA in this sector. Both Acts are based on 10 privacy principles described in the Canadian Standards Association Model Privacy Code[26]. The 10 principles form the basis of the code which describes the minimum requirements for protection of personal information. Principle 7 is of particular interest to us as it describes the safeguards that should be in place to protect sensitive data.

Here we highlight the aspects of Principle 7 that pertain to our project:

1. **Sensitive information should be protected with a higher level of security**: Medical records are always considered to contain highly sensitive data. For our project we need to ensure that all patient data is secured by the best available methods. The sensitive information needs to be protected from unauthorized users during both transmission and storage.

2. **Methods of protection should include technological methods such as the use of passwords and encryption**: We should investigate different encryption and authentication methods to find which would be most suitable for a health care environment such as the one for the MET-A$^3$Support project. The most secure password scheme may be great for privacy but may not be feasible in a health care emergency environment. We need to find technological solutions that are practical.

3. **Limit access to a 'need to know' basis**: Access control methods need to be used. We can limit access to individual documents of a medical record for both read and write permissions. Employee roles within the hospital and permissions associated to those roles can be used in determining a good access control model.

4. Both PHIPA and PIPEDA specify that medical records should be **protected from loss, theft, unauthorized access, disclosure, copying, use, or modification** regardless of what format they are in. Electronic records must be protected by access control and include an integrity mechanism. Furthermore, we must have an audit functionality to ensure that no malicious use of the system goes unnoticed.

These safeguards form the basis of our motivation for applying policy-based encryption to a hospital system. In the next section we describe our hospital system and the requirements we have for protecting the privacy of patient records using encryption and access control in the form of policy-based encryption.

## 1.3 Application of PBC to Hospital System

The health care environment has privacy needs for protecting electronic patient records using security technologies. We will apply the policy-based encryption to a hospital system to meet the privacy requirements detailed above.

### 1.3.1 MET-A$^3$Support System

The Mobile Emergency Triage (MET) is a software system providing anytime and anywhere decision support (A$^3$Support) for Emergency Department triage decision-making[45]. Patient medical records and clinical medical information are used to help doctors by providing them with evidence-based decision support. A doctor will use the system by giving symptoms as input and receiving decision support for diagnosis and treatment. The system will be run on Motion Computing C5 tablet PCs connected to a wireless network[47]. The system will be used in a trial in the emergency department of an Ottawa hospital.

We will integrate the policy-based encryption system with the multi-agent clinical decision support system to protect patient medical information. We will use policies that are created based on the type of document it will protect. Decryption keys will be managed with user accounts and accessed by providing login information. The staff will not have to actively participate in encryption procedures, other than providing login information. The policy-based encryption provides access control for patient medical information, preventing unauthorized access and limiting access on a need to know basis. It also provides a high level of security by encrypting all patient identifying information. The policy-based encryption offers ease of use for staff and provides encryption and access control, which makes it a good addition to our system to comply with privacy and usability requirements.

A typical scenario of a physician using the MET-A$^3$Support would be to diagnose a new patient in the emergency room. The physician will visit the patient while carrying their tablet PC. After logging in, the physician inputs the patient's symptoms. The system will

take the input symptoms, the patient's record from the Hospital Information System (HIS), and clinical medical information from an electronic library of medical literature, and will process the information to produce a suggested diagnosis. The result is displayed on the tablet PC to the physician. The physician can then either make changes to inputs to get a different diagnosis or accept the diagnosis and continue. The system can then provide treatment suggestions based on that diagnosis, and evidence from the medical literature to support that information.

### 1.3.2 Requirements

The privacy guidelines emphasize need for encryption and access control; however we must also take into consideration usability needs of the work environment. In an emergency room, electronic security systems and procedures cannot get in the way of medical tasks. The staff members using the hospital systems are not necessarily trained in computer security. This is similar to the needs of many organizations where employees are not trained in computer security but are required to do tasks each day involving sensitive information. In our system, we do not want to ask staff to manage encryption procedures and keys. The encryption should be done automatically and the decryption managed by the user logging into the system once each session.

Our requirements for a security system to meet our goals of using it in our hospital MET-$A^3$Support system are:

1. Encryption for multiple unknown recipients: we want to be able to encrypt patient information for multiple recipients without knowing who specifically they will be at the time of encryption. For example, individual people are not known but employee roles are specified as the encryption rule.
2. Access control based on employee roles: we want to be able to specify access control based on employee roles and not on individual employees
3. Usability: This encompasses a few aspects of the encryption system that we want to be automated to reduce the work placed on the user. For ease of key

management we want the encryption to be automatically done with a policy chosen based on the type of information protected, and the decryption to be handled by the user logging into the system (at which point their decryption keys will be associated with their login session and applied automatically when a request for information is made).

4. One system: we want one encryption and access control system to be able to fit our requirements.

Other security measures to deal with modification and theft will be discussed separately since they are not part of the main body of work which is focused on the encryption. However to ensure the security of patient information in the MET-A$^3$Support system, we need to address all concerns. We will discuss authentication mechanisms, audit log, integrity mechanisms and purging of files in section 6 with the conclusion.

## 1.4 Contributions

The first contribution of this work is the extension of a policy-based encryption system proposed by Bagga and Molva[7] to allow this PBC system to be used in our hospital system to provide security for sensitive information. We also contribute the implementation of such a policy-based system following the Java Cryptographic Architecture framework. Then we show how it can be used in a real world scenario by providing details of how we integrate our policy-based encryption system into the MET-A$^3$Support hospital system and how it can be used to provide security for patient information.

### 1.4.1 Extension to Policy-Based Encryption System

Bagga and Molva describe a policy-based encryption system in [7][8]. It originally is derived from the work of Boneh and Franklin who developed an identity-based encryption scheme[15]. Chen, Harrison, Solera and Smart extend their work to use multiple identities[24]. Smart further extends the work of [24] to allow for more

expressive access control[25]. Bagga and Molva's scheme is then an extension of Smart's work and allows for encryption on a full access control policy. We choose this policy-based encryption system as the basis of our work for a number of reasons. Some other similar systems are based on a threshold number of constraints being met, which is not suitable for our system. The structure of the policy being a Boolean expression can allow for it to be passed as a string and parsed, which can be used more easily than a policy requiring more advanced structures such as trees. Finally, their work builds on an encryption system for which a public Java implementation is offered by the NUI Maynooth Crypto Group. By reusing code, our implementation can be done efficiently.

In order to use this system in our trial, we will need to add some functionality by adding expressivity in the policy, allowing multiple actions. We would need to add the ability to specify what action can be taken on a resource once a user is granted access, including being able to specify read-only and write access and is controlled by opening the document in the allowed mode. We will add the allowed action to the policy combined with the appropriate conditions. This additional expressivity will allow us to provide better protection for sensitive information since we can control who is allowed to edit the resource.

### 1.4.2 Implementation

The policy based encryption system by Bagga and Molva has no public implementation. We offer an implementation in Java to allow for use across multiple platforms. The implementation follows the JCA framework allowing for easily integrating security functions into other systems. Using the encryption algorithms becomes very straight forward without having to know the details of how the encryption algorithms work. We provide details about our implementation and the problems we have encountered in the process of implementing this encryption system in Java.

### 1.4.3 Integration with Hospital System

The policy-based encryption will be used in our MET-A$^3$Support system to provide security of patient medical information. We provide details about how we use the encryption in our system. We describe the MET-A$^3$Support multi-agent system including the system architecture and where we place the encryption functionality. The encryption services are used at the entry point of our system where patient information is brought in from the Hospital Information System and at the client device where the patient information is presented to the user (a physician or other staff member). We describe the nature of the policies and credentials and how these are used to protect patient identifying information. Our work on integrating this encryption with a software system provides an example of a practical use of the policy-based encryption.

## *1.5 Organization of Thesis*

In Chapter 2 we give some background information to be able to understand the tools and concepts of policy-based encryption. We also include a peer review and discuss limitations of these current solutions. Chapter 3 is the main chapter on policy-based encryption, giving details about this type of encryption as well as the system proposed by Bagga and Molva[7]. Their system will be the one we extend and use in our implementation. Chapter 4 contains the implementation details of the encryption algorithms in Java as well as details of how the system follows the JCA framework. Problems encountered are discussed as well. The integration of our encryption system with the MET-A$^3$Support system is detailed in chapter 5. We use this system to show how patient electronic medical records can be protected using the policy-based encryption system we have implemented. We conclude with chapter 6 by giving a summary of contributions and discussing future work to be done on this work. We also briefly discuss other security measures to use in the MET-A$^3$Support system in order to assure security is met.

# Chapter 2

# 2 Background

The purpose of this chapter is to review the concepts needed for understanding the content of this thesis and to review other encryption and access control technologies providing similar level of security as policy-based encryption. We also give examples of encryption systems used in hospital settings.

## *2.1 Tools & Concepts*

### 2.1.1 Cryptography

The basic principle of cryptography is to change a message into something that is illegible, by means of an encryption algorithm, which cannot be restored without knowing a secret value. A plaintext is a message (string) that will be encoded using an encryption algorithm and a key, resulting in a ciphertext. The ciphertext can be restored by using a decryption algorithm with a key, resulting in the original plaintext. In each encryption system, the message space and ciphertext space are described, to know what form the plaintext must be in and what form to expect the ciphertext to be produced in.

There are two main types of cryptography, private key (symmetric) and public key (asymmetric). We will use the term public key cryptography or PKC (public key cryptography) to refer to asymmetric cryptography and public key to refer to the key used in the encryption system. Symmetric cryptography requires both the sender and recipient to share the same key. The sender encrypts under that key and sends it to the recipient, who can decrypt using the shared key. The main problem with this system is of how to distribute the private key securely between both parties.

In public key cryptography each user has a private key and a public key which are mathematically related but cannot be derived from each other. A trusted authority (TA) is set up to manage keys, generate keys for the user, and handle new requests. The private key is kept secret by its owner. The public key is associated to one user, is not kept secret and is made publicly known. Someone wishing to encrypt something for a user will encrypt it under the recipient's public key. Only the person who has the corresponding private key can decrypt the document. This type of system does not have the key distribution problem of symmetric cryptography, since the public key can be shared in the open and is the only key needed to encrypt with. Most encryption systems use a combination of symmetric and asymmetric cryptography, using the asymmetric cryptography system to set up a shared key. The shared key can be encrypted under the recipients' public keys and sent to them securely. Then the shared key can be used in a symmetric encryption system for further communication. This has the advantage that symmetric key systems are generally faster.

Public key encryption systems can also have signature schemes. The private key can be used to 'encrypt' a message to produce a signature. This type of encryption algorithm is called a signature algorithm. Only the person in possession of the private key could have created this signature. Anyone can verify the signature by 'decrypting' it using the corresponding public key. The decryption algorithm is called a verification algorithm. Digitally signing a document or resource in this way acts as a regular signature by ink on paper documents.

### 2.1.2 Certificate-Based PKC

Most public key cryptography systems are managed using certificates. A public key is generated and contained in a certificate which is signed by a trusted authority. The signature of the TA is proof that the public key belongs to the user named in the certificate. Users can check the validity of the public key by checking the signature of the trusted authority. These certificates are published in a list allowing anyone to look up a

person's public key in the list. To encrypt a document for a user, the sender will check for that recipient's public key certificate, encrypt with the public key and send it to them. The private key is kept secret by the owner. Only the owner of the private key can decrypt something that is encrypted under their public key. These certificate based schemes usually also provide a signature scheme.

### 2.1.3 Identity-Based Encryption

Identity-based encryption is a form of public key cryptography with a public key and private key for each user[57]. The public key consists of a string which is unique to a person. One example of a public key could be an email address. The public key is well known and can be shared easily. Public lists publishing public key certificates are not necessary. Since the public key identity string is uniquely tied to one user, there is no need for a TA to provide a signature to tie the public key and user together. The private key is generated by the user proving their identity to a TA, usually by following standard authentication measures such as logging into an account. The private key can be generated per request or once and then kept secret by the user. In this system, users do not need to manage their own public key. To encrypt a message for a recipient, the sender will specify the identity string to encrypt on (e.g.: email address) and send the encrypted message to the recipient. The recipient will authenticate themselves (for example, by providing a username and password) to the TA and receive the corresponding decryption key. The TA can also be called a Private Key Generator (PKG) which will solely handle requests for generating a private key

### 2.1.4 Policy-Based Encryption

Policy-based encryption was derived from identity-based encryption using the idea that any string can be used as the public key. In the case of policy-based encryption, a string representing a policy is used as the public key. Instead of the string representing a specific identity, it becomes a set of rules specifying multiple people who could be allowed to decrypt it. The policy can be a set of access control rules, represented by different data structures such as a logical Boolean expression, lists, or trees. The private

key generated for each user is a set of credentials each representing a permission that user has. The credentials are presented as the private key to decrypt a resource that is protected by being encrypted by a policy. If the credentials satisfy the conditions in the policy, then the resource is decrypted.

Attribute-based systems are synonymous with policy based encryption in that they encrypt on sets of conditions. Some attribute-based systems do not have a structure set up to handle a full policy-like set of rules, instead relying simply on a threshold number of conditions being met. We refer to attribute-based systems as ones that encrypt on multiple conditions without a structure to enforce a specific policy. In these systems, either all conditions match, or a threshold number of them. There is no way to specify a logical expression. Attribute-based systems which have the structure in place to process a logical expression policy will be referred to as policy-based systems.

The differences between IDE and policy-based systems are not just how the public key is represented but how the encryption and decryption processes work. In policy-based encryption, each condition in the policy is computed independently during the encryption process, as opposed to IDE which treats the entire public key as one string. This allows for multiple private keys corresponding to the same public key in policy-based encryption. The policy is only satisfied if a certain set of conditions are satisfied, and there may be more than one combination of credentials which satisfy it. For example, in a simple policy stating <Doctor, role> OR <Nurse, role>, Alice with the credential of Doctor, and Bob with the credential of Nurse, can both decrypt even though their private keys are completely different. Therefore, define a policy as a set of rules that are computed independently as opposed to an identity which is treated as one string, even if that identity represents a group or role.

In a policy-based system, the policies can be made public and managed by an entity in charge of access control rules. A policy can be applied based on the type of resource being protected or by user choice. A user generates their decryption keys by authenticating to a TA. This can be done per session, or per decryption request. The user

does not necessarily have to manage their own private key if the system automatically applies it for them at the time of request.

| | **Public Key Encryption (asymmetric encryption)** | **Identity-Based Encryption** | **Policy-Based Encryption** |
|---|---|---|---|
| **1. Encryption for multiple unknown recipients (i.e.: a role)** | Not easily – either encrypt multiple copies, use a common (group) public key, or use additional access control measures | Not easily – either associate an identity with multiple people and give each access to generating private key, or add additional access control measures | Yes |
| **2. Access control based on employee roles** | No | No | Yes |
| **3. Usability** | | | |
| User manages public key, look up lists | Yes | No | No |
| User manages private key | Yes – private key generated at same time as public key, must be kept private | Not necessarily – can have private key generated at log-in | Not necessarily – can have private key generated at log-in |
| **4. One System (for both encryption and access control)** | No | No | Yes |

**Table 2-1 Comparison of public key encryption systems**

## 2.2 Literature Review

### 2.2.1 Public Key Cryptography with Access Control

As mentioned before, traditional public key cryptography does not inherently provide role-based access control. Wilkinson, Hearn, and Wiseman describe an access control system that uses public key encryption to control access to documents[28]. The

documents are encrypted under a group public key. Members of that group can decrypt by using the group private key. They also describe how symmetric key cryptography can be used in a similar manner. However they conclude that asymmetric cryptography will provide better protection from key compromises at proxies.

This type of scheme could work for providing encryption and role-based access control that we are looking for. However it still suffers from key management problems. The group public keys need to be created and managed in look up lists. Users must wishing to encrypt will have to know how to choose a public key and validate it before using it to encrypt with. It is also not clear how to encrypt a document for multiple groups or roles.

### 2.2.2 Identity-Based Encryption

Identity-based cryptography began in 1984 with a signature scheme proposed by Shamir[57]. Its security relied on the popular RSA algorithm. This encryption scheme provided only signature functionality and not full encryption; however it laid the foundation for future identity-based schemes. It wasn't until 2001 that a full identity-based encryption system was proposed, and two distinct systems were described.

Cocks developed an identity-based encryption system using Quadratic Residues[25]. This proved to be less efficient than the Boneh and Franklin system discussed next. The encryption processes the plaintext bit by bit which creates a much longer ciphertext and reduces the efficiency of transmitting messages[4]. Boneh, Gentry and Hamburg [16] later developed an efficient identity-based encryption without using pairings. It also relied on the quadratic residuosity problem. The ciphertext is shorter than in Cocks, but the encryption and decryption processes are slower. For efficiency reasons, these encryptions systems have not been widely used in practice and therefore we will not be using them in our work.

Also in 2001, Boneh and Franklin developed an identity-based encryption system using the Weil pairing. The Weil pairing is an example of a bilinear map over elliptic curves. It

is a mathematical function which maps points between two groups. See [4] for a complete definition of the Weil pairing. Since then, most identity-based encryption and policy-based encryption systems proposed have been based on this scheme using bilinear pairings over elliptic curves[15]. In their system the public key is an identity string. The private key is generated by a PKG when the user has authenticated properly, and consists of the hash of the identity string multiplied by the PKG's secret master key. The encryption and decryption both use computations involving a bilinear mapping, provided by the Weil pairing. Although the identity string can be any arbitrary string, there is no structure in place to handle a string representing multiple roles or policies. The identity string is treated as one string in computations in the encryption and decryption processes. Their system cannot be used as a full access control encryption system. The security of most identity-based encryption schemes including Boneh and Franklin's rely on the Bilinear Diffie-Helman computational problem or variations of it[4].

Baek and Zheng describe a threshold identity-based encryption based on Boneh and Franklin's work[5]. In this system, the user can distribute parts of his private key to different decryption servers using Shamir's secret-sharing technique[56]. In order to decrypt a resource, the user sends the ciphertext to each server and receives back from each a decryption share. If they reach a threshold number of shares, they will be able to receive the whole plaintext. This has potential applications in mediated encryption. Since the parts of the private key are distributed, the user must have cooperation with the PKGs in order to decrypt. This process is called mediated encryption and the PKGs play the role of Security Mediators (SEMs). If the user should have his permissions revoked, the PKG simply doesn't allow the user to decrypt. This offers more control than in a system where once the user has their full private key there is no easy way to revoke it.

Hierarchical schemes were developed as a solution to a problem, which arose with having only one PKG available in an encryption system. That one PKG must handle all requests for keys in the system. In a system with a small number of users there is no problem, but with many users the PKG may be overloaded with work. Horwitz and Lynn suggest a hierarchy of PKGs, where each computes the private keys only for the entities in level

below them[37]. Users in the system have a tuple of identities to represent their public key, which consists of the identities of each PKG in their ancestry. For example, a user Bob could have the ID tuple ($ID_{Bob}$, $ID_{EmergencyDept}$,$ID_{Hospital}$) where his ID, the ID of his emergency department PKG, and the ID of his Hospital PKG are contained in his public key. Howitz and Lynn however did not have a fully functioning system. Gentry and Silverburg developed a full Hierarchical Identity-Based Encryption system using multiple PKGs based on Boneh and Franklin's IBE Scheme[30].

Chen, Harrison, Soldera and Smart show how to use multiple PKGs in Boneh and Franklin's IBE Scheme to allow for a simple access control structure[24]. The system allows for creating virtual keys by having conjunctions of multiple existing public keys. These multiple keys are created by using the multiple PKGs along with a person's identity. The person's identity is combined with one PKG's public key are added with the same person's identity combined with another PKG's public key. By adding these keys together, the public key becomes a simple access rights structure. For example, to encrypt for a person Bob who works at a bank as well as at a security firm, encrypting on his identity with each of the PKGs for the bank and security firm ensures that only Bob who works at both can decrypt. This creates a simple access control structure in conjunctive normal form. Smart [59] extended the work of [24] to allow for a more expressive access control. It allows encrypting for multiple identities in a conjunctive and disjunctive normal form. This formed the basis of a policy-based encryption system discussed in the next section.

A number of signature schemes have also been proposed but will not be discussed here since we focus on encryption systems. A few signature schemes have been proposed based on the bilinear pairings [18][17][13]. Also a number of identity-based signature schemes have been developed from pairing based identity-based encryption schemes [22][35][65][3]. There has also been work done on identity-based signcryption schemes which provide encryption and signature at the same time [19][44][43].

A couple of encryption schemes have varied from the usual form derived from Boneh and Franklin's original identity-based system and resulted in more efficient computations. Boneh and Boyen describe an encryption system where the encryption requires no bilinear map computation and the decryption requires at most two[14]. This results in a much faster system. They describe both a hierarchical IBE system like Gentry and Silverburg, plus another efficient system. Both are not able to perform encryptions based on policies providing access control and so will not be considered for our project. Similarly, Zhang, Safavi-Naini, and Susilo [66] created an identity-based signature scheme that requires less pairing operations than [18].

Baek, Newmarch, Safavi-Naini and Susilo describe a number of these identity-based encryption and signature schemes in their paper "A survey of Identity-Based Cryptography"[4].

### 2.2.3 Policy-Based & Attribute-Based Encryption

Bagga and Molva further extend the work of Smart [59] to propose a policy-based cryptography scheme including an encryption scheme and signature scheme[7][8]. They propose using a policy as a public key to encrypt a document. The policy can be represented by a Boolean expression in a combination of conjunctive normal form and disjunctive normal form. A user obtains their decryption key based on their credentials and can decrypt if these credentials satisfy the policy rules. This system is the one used as the basis of this thesis work for reasons outlined below. They also describe how to make this scheme collusion-resistant in [6].

One of the first attribute-based systems is proposed by Sahai and Waters [54]. A document can be encrypted with a set of attributes. A person can decrypt only if their set of credentials fulfills a threshold value number of attributes. This type of system works well with biometrics authentication schemes which naturally have a tolerance error. In biometric authentication, a biometric reading is taken of the user and compared to a previously computed template. A range of values close to the template value are

considered to be accepted, which is called error tolerance. Similarly, Chase describes a multi-authority attribute-based encryption system that uses a threshold number of attributes to allow decryption[23]. Each TA in this system monitors certain attributes and distributes keys accordingly. Both of these systems are not what we want for our system. Instead of a threshold number of attributes met, we want specific set of attributes in the form of a logical expression. For example, consider a policy stating a clinical physician during a certain time period or an administrative staff member during a certain period can have access to a document. This policy should not be allowed to be fulfilled by someone who has the attributes clinical physician and administrative staff, without the necessary time period. In a threshold system with no structure controlling the format of the policy, this rule would not be enforced properly.

Another system proposed by Goyal, Pandey, Sahai, Waters [32] operates in the reverse way. A document is encrypted with a public key of attributes, and the user's private key has an access structure built in. This is the opposite of what we are looking for in our system, with the structure (policy) being in the public key.

Kapadia, Tsang and Smith propose an attribute-based encryption system that allows for role based access control[39]. Their system relies on hidden credentials and policies. In our case, the policies will most likely be public since there are a finite set of policies created based on well-known rules. It would be better to have a system that doesn't rely on having secret policies. We also want to be able to have an expressive policy to include rules that aren't necessarily in the form of attributes, for example we want to specify a read-only or write permissions in the policy. For that reason, full policy-based encryption systems would be better to use than attribute-based systems. Most attribute-based encryption schemes aren't as efficient as the Bagga and Molva scheme discussed next.

Bradshaw, Holt, and Seamons describe another hidden credentials system in [20]. This is similar to policy-based encryption in that users must have the correct credentials in order to decrypt. They use multiple encryptions to essentially represent Boolean expressions in conjunctive and disjunctive normal forms as in Bagga and Molva's scheme. The multiple

layers of encryption reduce the efficiency of the system. They propose a solution to make it more efficient in [36] however it still suffers from collusion where users can collaborate to produce decryption keys from a collection of their credentials[6].

Bethencourt, Sahai and Waters describe another system with a policy used to encrypt[11]. The ciphertext-policy attribute-based encryption system encrypts a message on a set of attributes represented by a tree structure with inner nodes as AND or OR gates and leaves as policy conditions. A user with proper credentials can decrypt. This is similar to the Bagga and Molva system except for the representation of the policy. Similarly, Pirretti, Traynor, McDaniel, and Waters developed a policy-based encryption system[53]. The policy structure is controlled by thresholds organized into logical expressions to allow for a fully expressive access control policy. They offer an attribute based encryption API library written in C on their website http://siis.cse.psu.edu/attribute.html. At the time of writing, we were unaware of their work. They offer another solution for a policy-based encryption system that could be applied to a hospital system.

### 2.2.4 Implementations and Applications

To our knowledge, there is currently no public implementation of a policy-based encryption system. The NUI Maynooth Crypto Group has made available an implementation of the Boneh and Franklin identity-based encryption system [21]. The implementation is done in Java using a modified Tate Pairing instead of the Weil pairing originally proposed for the algorithms. They provide detailed steps for how to use the identity-based encryption and decryption. Along with the code for the identity-based encryption, they provide supporting code for bilinear pairing computations and elliptic curve arithmetic operations. We used some of these libraries in the implementation of our policy-based encryption in Java. Unfortunately the identity-based encryption implementation cannot be used whole in our system since it doesn't fit all of our requirements, notably being able to encrypt for multiple recipients and having access control easily built in.

Other public implementations of identity-based encryption systems also exist. A group of people at Stanford including Boneh and Franklin provide an implementation of the identity-based encryption system designed by them[60]. The implementation was developed under Debian GNU/Linux using C/C++. The code for their Stanford IBE system is available on their website: http://crypto.stanford.edu/ibe/download.html. They also offer a pairing-based cryptography library (in C) providing mathematical operations necessary in most pairing-based systems. Shamus Software [58] also provides a library called MIRACL using C/C++ that has Boneh and Franklin's IBE scheme. Their code is available at http://www.shamus.ie/.

Voltage is a great example of a company using the encryption technologies discussed here for security and privacy solutions in health care environments. They offer an identity-based encryption for email messaging that is currently being used in hospitals in the United States[64]. Similarly, Secure Computing offers policy-based cryptography products. They implemented a token based authentication system with audit logs to ensure HIPAA compliance for a system in a hospital[55].

Mont, Bramhall, and Harisson from the Hewlett Packard Lab in Bristol, UK have developed a messaging service using identity-based cryptography for a hospital[46]. Their scheme uses the fact that any string can be used to encrypt on including a role. When a user wants to send a message they choose a key to encrypt it under, and send the encrypted message to a recipient along with the key used to encrypt it under. The recipient communicates with the TA and asks for the corresponding decryption key. If the recipient has necessary permissions they will receive the decryption key. The sender can choose a role as the encryption key. Thus, anyone who doesn't belong to the role cannot receive the decryption key and see the message. This provides a secure email system for use within the hospital that also allows for encrypting documents sent using the service. However, we would need an encryption service outside of an email system to use in our hospital system. We need to protect patient records both when transmitting across the network and when storing on the servers.

In “The RETSINA MAS: a Case Study” Sycara et al describe a RETSINA multi-agent system and their experiences working on the project[62]. The way they integrate security into their system is comparable to our approach with MET. The security functions in their system are integrated as a security layer, providing agents with an infrastructure of security services. While their system relies on certificates for encryption, the idea of having the security as a layer instead of within security agents is similar.

## 2.3 Limitations of Current Solutions

Traditional methods of encryption have a few problems when being applied to a scenario such as using it in our MET-$A^3$Support system. Secret key encryption suffers from the problem of key distribution. There's no easy way to create and share secret keys between parties to send encrypted messages back and forth. The keys must be securely delivered to both parties involved in the encryption, either by using a good key establishment protocol such as Diffie-Hellman[27] or by distributing keys to each party offline.

Public key encryption does not have the key distribution problem but raises other difficulties. Key management becomes a problem because of the nature of the public key. Public keys are assigned to an individual and tied to their identity by a TAs signature. A certificate contains the public key, identifying information, and the signature. These certificates are made public in lists and are updated or disabled after their validity time period is over. Users using the encryption system must be aware of the certificates, and be able to validate the certificate of the person they wish to encrypt for. Also, since keys are generated for individuals, encryption for multiple people becomes a problem. The choice is to use a common group key. This does not allow for the flexibility of encryption for role-based access control or for including other constraints.

Identity-based encryption solves the problem of key management with public key systems, but still does not make encrypting for multiple recipients possible. The public key used in the identity-based encryption is still associated with an individual person. Encryption for multiple people would involve associating one identity with many people.

However it's still not apparent how to encrypt for multiple roles and including other constraints in the encryption to provide full access control.

Policy-based and attribute-based systems mostly come in two classes. Some are based on a threshold number of constraints being met. These are not suitable for our system as described before. We cannot have a threshold number of constraints that allows for expressing policies where one specific group of constraints needs to be met. We also want to include other aspects than simply attributes, for example time periods. Therefore we want an expressive policy to be allowed. There are also systems in which the policy is based on a combination of constraints in a logical manner that must be fulfilled according to the logical expression. This is the type of system we want to use, such as the system by Bagga and Molva.

We choose the system by Bagga and Molva to work with over another possibility, the system by Bethencourt, Sahai and Waters. The latter uses a tree structure to organize the policy. We prefer the combinations of conjunctions and disjunctions provided by Bagga and Molva since it allows for passing the policy as a string and parsing it into separate conditions. With a tree structure, it would either have to be built into the encryption system or created by the software calling the encryption functions. Another reason we want to use Bagga and Molva's work is that it builds on the identity-based encryption system for which an implementation is offered by the NUI Maynooth Crypto Group. Implementing the policy-based encryption system would become easier by reusing code. In this thesis we extend Bagga and Molva's system to include more features to make it a complete system that we can use.

## 2.4 Summary

We started this section by giving general information to help understand the main concepts in this thesis. The definitions of different cryptography and access control technologies led into a peer review of these works. Our review of public key cryptography and access control is brief because we do not use these technologies for our

project. The discussion of identity-based encryption systems offers background on how and what policy-based encryption derived from. We show that many identity-based encryption systems were proposed but focus on Boneh and Franklin's scheme. Extensions to their system led to Bagga and Molva's policy-based encryption scheme which we use in this thesis.

We discussed other policy-based encryption systems and why we chose not to use them as the starting point for our work. Some systems were based on a user being able to decrypt if they had a threshold number of credentials. Others used an access control structure in the private key instead of the public key. Bagga and Molva's scheme seemed the best choice for our work.

# Chapter 3

# 3 Policy Based Encryption

This chapter provides an overview of policy-based encryption and our motivation for using this type of encryption in our hospital system. We give details of how policy-based encryption works including details of Bagga and Molva's work. We then describe an extension to their work that would allow us to use this system.

## 3.1 Overview

Encrypting based on a policy can allow a document to be encrypted for access by multiple recipients organized under roles. A policy-based encryption scheme offers the greatest flexibility for our security needs. The approach is relatively simple and build s on the idea of encrypting under an arbitrary string. A document is encrypted under a policy, which is a combination of rules. Note that in IBE, if the public key can be an arbitrary string, then this string need not be an identity. Rather, the string may be a complete access control policy (or a hash of that policy) for the document that is to be encrypted. A user wishing to have access to a document will authenticate by logging into the system and will obtain a decryption key associated with their role fom the TA. If the user's role satisfies the requirements of the policy, their decryption key will decrypt the document.

Keeping usability in mind, it would be favorable to automate the encryption process. Staff should not be asked to enter a policy or choose from a list of policies every time they create a new document. Because of the nature of the hospital setting, we can make policies dependant on the type of document. If a document of a certain type is entered in the system, it will be encrypted based on a corresponding policy. This is possible because of the finite number of document types in a hospital setting and the access rules for these document types. For example all x-ray lab reports are available to be read by doctors.

Therefore these documents can be encrypted under a policy specifically for that type. This will also ensure that we have a relatively small number of policies to manage.

A policy look-up table (Table 3-1) can be used to check which policy to encrypt on for a given document. Each document can be labeled with the document type to know how to look up the corresponding policy. Having this data stored in a table can create a security problem. For example, if an attacker wants access to a particular document type, all they would have to do is modify the policy associated with that document type to rules that would allow them to have access to it. To ensure that the policies and policy table have not been tampered with, extra security measures are needed to protect these. One way is to add to the label of the document the hash of the system master (secret) key, the document, and the hash of the policy that should be used to encrypt it with. When encrypting the document, the label will be checked to know which policy to retrieve from the table. Once the policy is retrieved, the hash of it, the document, and the master key are then compared with the value stored along with the document's label. This is similar to the concept of secure document labeling[10]. For example, an X-ray document can be labeled as an X-ray along with the hash of the master key and policy, (TYPE=X-ray, VERIF=hash(masterKey||Policy382)). An attacker could not create a fake verification

| **Policy ID** | **Policy Content** | **Association with Document Type** |
|---|---|---|
| Policy382 | (<Doctor,role>AND<Read,action>) OR (<Administrative,role>AND<Read,action>) | X-ray |
| Policy417 | (<Doctor,role>AND<Write,action>) OR (<Nurse,role>AND<Read,action>) | Medications list |
| Policy172 | (<Doctor,role>AND<Write,action>) OR (<Nurse,role>AND<Read,action>) OR (<Administrative,role>AND<Write,action>) | Main patient record |
| Policy289 | (<Doctor,role>AND<Read,action>) | Lab report |

**Table 3-1: Policy Look-up**

value without knowing the masterKey. Also, taking a verification value from another document to apply to a different document would not verify correctly. This will ensure that no one can change the policies or policy look up table.

## 3.2 A Policy-Based Encryption System

Each policy-based encryption system follows a general model with the required algorithms. We present such a model and then give details about a specific system by Bagga and Molva. The policy-based encryption system and model we discuss are based on bilinear pairings.

### 3.2.1 Model

The policy-based encryption system will have a setup algorithm (or set of these algorithms) to create public parameters. These parameters include information about the message and ciphertext space, public keys, TAs and bilinear pairing used in the system. Each TA will be setup with its own private and public keys. A list of available TAs is kept.

The policies used in the encryption can range from a basic combination of a few roles to a more complex combination of roles and constraints (for example time periods). These combinations can be represented by different structures such as trees, where each leaf is a condition and internal nodes specify how they relate, or Boolean expressions in disjunctive or conjunctive normal form. A list of policies can be created to choose from when encrypting or in our case, a list showing document types and their related access control policy. Our system will use a finite number of policies associated with the type of document it will protect. The policy constraints will be based on employee groupings (roles), as they are already defined in the workplace, and constraints such as time periods and allowed actions.

Corresponding keys for the decryption process are created based on a user's role and are comprised of credentials. When a user requests to decrypt, they will authenticate to the TA who will provide them with the decryption keys. (Note this step can also be done as a login to the system for the duration of a session). The TA will check an assertion of each of the user's permissions and generate a credential using its own private key and the assertion. The validity of a credential can always be checked by using the TA's public key. The user receives a set of credentials representing each of their permissions. If the user's credentials satisfy the policy under which the requested document is encrypted, their decryption key (set of credentials) will decrypt the document and they will have access to it.

An encryption algorithm will encode a message or document using a given policy. The result of the encryption algorithm is a ciphertext which is illegible. A decryption algorithm will restore the original message or document when given the ciphertext and credentials that satisfy the policy used in to encrypt with. See Figure 1 for a diagram demonstrating the encryption and decryption process.

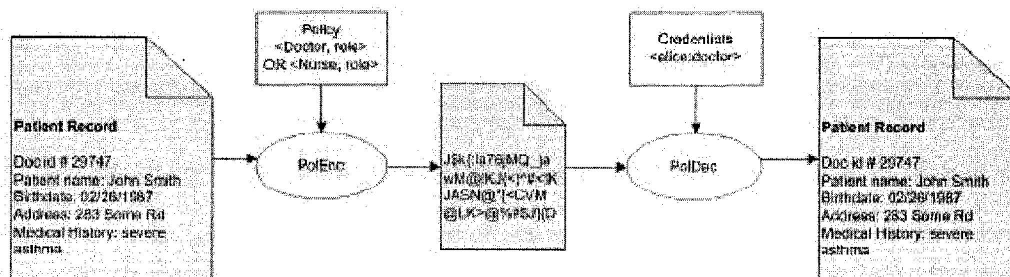


**Figure 3-1: Policy-Based Encryption**

### 3.2.2 Bagga and Molva's System

Molva & Bagga propose a policy-based encryption system described next[7].

During the setup phase, 3 setup algorithms are called to create the public parameters and initiate the TAs. The BDH setup algorithm generates parameters for the bilinear pairing

operations necessary in the encryption and decryption algorithms. The setup algorithm generates the public parameters including the hash functions, ciphertext and message space. It initiates a call to the BDH-Setup algorithm. Each TA runs the TA-Setup algorithm, generating their public and private keys. The public keys are published along with a list of TAs available.

In Bagga and Molva's system a policy is represented by a list of rules combined in conjunctive normal form or disjunctive normal form, denoted $pol_A = \wedge_{i=1}^{a}[\vee_{j=1}^{a_i}[\wedge_{k=1}^{a_{i,j}}\langle TA_{i,j,k}, A_{i,j,k}\rangle]]$. A Trusted Authority (TA) is set up to manage the policy and credential generation. Each rule in the policy, $\langle TA_{i,j,k}, A_{i,j,k}\rangle$, corresponds to a credential $A$ and the trusted authority TA which can check the validity of the credential. A user has a set of credentials, $\varsigma(R_{i,j,k}, A_{i,j,k})$, containing the TA's public key $R$ and the credential assertion $A$. The user wishing to decrypt a document authenticates to the TA (for example, by providing a username and password) to receive their decryption credentials. The TA runs the algorithm CredGen to generate the user's credentials. If the user's credentials fulfill the policy rules by having their credentials match the necessary policy credentials, the document will be decrypted.

An example of a simple policy to use in our hospital system is "A doctor or a nurse can read the lab report" and would be represented by the rules <X, Doctor:role> OR <X, Nurse:role>. This policy *pol* will specify an action *act* of reading a resource *res* which is a lab report. The lab report *res* will be encrypted using policy *pol* as follows

c = PolEnc(res, pol)

In the encryption algorithm PolEnc, each set of conjunctions of a policy, $\wedge_{k=1}^{a_{i,j}}\langle TA_{i,j,k}, A_{i,j,k}\rangle$, is assigned a mask. Each disjunction is assigned a random key value. Then, each key value is encrypted by each mask. A user satisfying one set of conjunctions will compute the corresponding mask and will be able to retrieve each key value and decrypt the entire document. Computing the mask values is the core of the encryption algorithm. The encryption algorithm is given a message m (a string of a

document converted to bytes) to be encrypted, and the policy $pol_A$ (also a string converted to bytes). First we map the conditions of each conjunction in the policy to an elliptic curve. Then we compute the mask for each set of conjunctions. Then each mask is encrypted by each random value, with the results making up the ciphertext.

A user can decrypt *c* by providing their credentials *cred = (alice:doctor)* that satisfy the policy *pol.*

res = PolDec(c, pol, cred)

Here, the person wishing to decrypt must either have an employee role of Doctor or Nurse verifiable by a Trusted Authority X. The entity in charge of user accounts and permissions would be an example of that trusted authority.

In the decryption algorithm PolDec, a set of credentials $\varsigma_{j_1,\ldots,j_a(polA)}$, each denoted $\varsigma(R_{i,j,k}, A_{i,j,k})$, the policy $pol_A$ and the ciphertext are given. In the decryption process the main computations are re-computing the masks from the credentials presented. Here, the mapping is done with the sum of the credentials and a value passed in the ciphertext. The masks values are then combined with the values in the ciphertext to reveal the random key values. If the proper key values are obtained, the plaintext is revealed.

With this format, Molva & Bagga's encryption system will work to protect the privacy of patient records. We would be able to encrypt on a set of rules controlling who can have access to the patient medical information. By including roles as the policy constraints, we can emulate the hospital access control policies for patient records. They also include a signature scheme as part of their system but we will not go into details about that, since it will not be used in our system.

The efficiency of the Bagga and Molva encryption system depends on the number of conditions that are combined to form the policy. The complexity increases linearly with the number of computations required to encrypt for each rule. For our application, we plan to have few rules combined in a very simple manner. Most policies will specify access control for a few general employee roles such as Emergency Department Doctor,

Nurse, and Administrative Staff. This will keep the overall complexity down. We will also have limited number of credentials for users, as their roles will be the credentials and we have a limited number of actions possible. They also state ways of maximizing the efficiency of encrypting by pre-computing and caching some values. Decryption is more efficient than encryption in their scheme, and reducing the encryption time would cut down on the computation time in the overall system.

### 3.2.3 Key Revocation

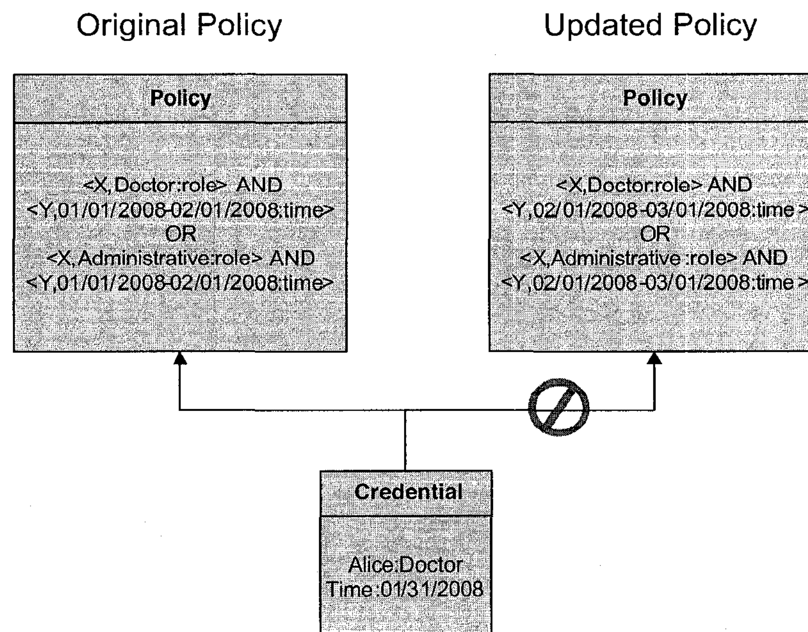


**Figure 3-2: Key Revocation by Updating Policy Time Period**

In traditional PKI we have revocation of keys, which allows for managing roles in the system and to manage compromised keys. In our system, we would have two scenarios that may require key revocation. If a staff member leaves or their role is changed, and if a key is compromised. For the first scenario, if the user's role is changed this will be reflected their account and they will receive the corresponding decryption key from the PKG. If the staff member no longer works at the hospital, their account will be disabled and they won't be able to log in, and thus won't be able to have access to the system.

Therefore, key revocation is not needed for these types of changes to accounts. However we do need revocation in the case of a key compromise. Note that a user could retrieve their key without necessarily needing to decrypt something. This is not a security threat however, since if they have the correct permissions to view a document, then they should be allowed by the system to view it, according to the access control policies.

Boneh and Franklin propose adding a time period to keys in order to handle this situation[15]. Since the policies and keys used in our system are strings, we can also associate them with a time period. Adding a time period to the policy can be done by simply adding another constraint in a conjunction with other rules. Decryption keys generated during that time period will contain the proper time to fit the policy. Updating a policy's time period will disable all keys that have been acquired before the new time period, which starts from the present time. All users will have to get a new decryption key. This ensures there is no time between revocation and disabling of keys. Users who are using the system properly will be receiving an updated key each time they make a request to have access to a document and so will not need to keep track of updating keys. In the event of key compromise, updating the policy (and updating records in batches) is sufficient to disable the key. This solution does not change the encryption system functionality and does not add much complexity. If the key compromise occurs and the policy is not updated, the keys can still be used to decrypt. This is a similar situation to a certificate-based encryption system where a compromised key has not been discovered yet, and the corresponding certificate is still listed as valid. Periodically updating the policies and tracking unusual behaviour of users in the system are good methods for preventing and noticing use of compromised keys.

If a policy is changed or updated, then all documents encrypted under that policy would have to be updated. This can be done automatically by decrypting and then encrypting under the new policy. It will not affect staff having access since they will receive the updated key when they try to get access. If they have a document open already, then the update won't affect them. If they modify and save the document, it will be encrypted

under the new available policy. Details of how this affects patient records in our MET-A$^3$Support system are given in section 5.2.5.

## 3.3 Extension

Bagga and Molva's system allows for encrypting on a policy specifying a set of conditions. This setup allows for role based access control to be handled by encryption. However we will need to add some functionality for this to be used in our system. Currently Bagga and Molva's encryption system specifies one action allowed to be taken on a decrypted resource. One action may not be enough to express our access control policies in the MET-A$^3$Support system. Traditionally, encryption systems don't specify actions allowed on the resources being decrypted, therefore we must look for an appropriate solution to build into the system.

Bagga and Molva's scheme was developed for a policy that specifies one action allowed to be taken on the resource it is protecting. The action is defined as what the user decrypting the resource is allowed to do with that resource. This action could be for example a write action, allowing the user to edit the resource. In our case, we would need for the policy to be applied to more than one action. For example if it is a doctor asking for access to a patient's record then they can read and write, but if it's a nurse then they can only read. Bagga and Molva's scheme allows for policies that are made of combinations of rules, but don't include information about the action allowed on the resource for those who fit the policy. Their scheme relies on specifying an action separately and only one per resource.

We add expressivity to their policy by allowing actions to be specified for each conjunction in the policy. Since the idea is to be able to encrypt under any arbitrary string, it seems fairly reasonable to say that we can add this information to the policy to encrypt under. A policy could therefore be represented as

pol2 = <Doctor, role> AND <Write, action>

Using default values for actions, all users would be able to perform these actions. However, a user would only be allowed to perform the action if they also satisfy the other rules in the conjunction. In the example of pol2, upon presenting a credential for the role of a doctor, a user could decrypt and have write access. The action rule will signify what permissions the user has for that particular document. This will result in treating keys for each credential in a different way, allowing default key values for the actions so that the entire conjunction is still satisfied by the user's private key.

In order to treat the action rule in the policy as a permission and not as a condition, we need to treat this term in the condition differently. Rather than simply checking that the user has the corresponding credential, we track which conjunction the action rules are associated with. Tracking the conjunction that was satisfied requires setting a paramter when the appropriate conditions are satisfied by the user's credentials during the decryption process. The action rule in that conjunction determines what the user will be allowed to do with the resource. If a conjunction was satisfied, then the user will be allowed to decrypt a resource and carry out that action on the resource. Then we can open the resource in the proper mode, such as write or read-only.

By extending their scheme to allow for more expressive policies, we can adapt this system for use in our MET-$A^3$Support project. Using simple policies that specify actions allowed on the document encrypted under that policy we strive to keep the encryption process efficient. We also have minimal number of credentials for users based on their roles in the hospital. Finally, we can automate the encryption process by specifying policies for each document type, thus allowing staff members to enter documents in the system without worrying about encryption details.

## 3.4 Summary

In the introduction section we stated certain requirements we had for an encryption and access control system. We revisit these requirements to show how Bagga and Molva's system, along with our extensions, satisfy these requirements.

1. Encryption for multiple recipients: The policy constraints can be roles which allow us to encrypt for multiple people by employee groupings. We can even have other constraints in the policy to allow for key revocation, allowed actions, and any others we see necessary. This system is designed to encrypt for multiple people without knowing them ahead of time.
2. Access control by role: In order to have access to patient information, a user must be able to decrypt the patient record. Therefore the policy-based encryption also provides access control by controlling who can decrypt. As stated before, it is based on employee roles. This leads into our next requirement,
3. One system to provide encryption and access control: The policy-based encryption protects the information with an access control policy. No additional access control or encryption technologies are necessary.
4. Ease of use: Our goal for usability was to incorporate encryption and access control without the user having to be aware of it. This is possible because policies can be created and applied based on document type, without the recipients being known ahead of time. Decryption keys can be associated with a user's session each time they log in, and applied when a request for patient information is made.

The policy-based encryption system detailed in this section will be used for our implementation as discussed in the next chapter.

# Chapter 4

# 4 Implementation

## *4.1 Overview*

We implemented the encryption system described above. We chose to use the Java language to implement it because of existing code and packages publicly available, which greatly helped speed up the implementation time. Many operations such as the hash and bilinear map operations were taken care of by using the appropriate packages. Therefore we could focus on the overall design and efficiency of the system. We also chose to follow the Java Cryptography Architecture framework, a standard framework for implementing encryption systems in Java. Our main motivation for following the JCA was for future use of the encryption system including our MET-$A^3$Support project. See Appendix A for the full code of our implementation.

The implementation was done in Java using the NetBeans Integrated Development Environment (6.0). We created 8 classes to house the functionality of the encryption system and 1 test class. In all there are 712 lnes of code including the test class. Many java packages were used to speed up the implementation including ones for elliptic curve arithmetic and large number operations. We give details of our implementation of the encryption algorithms as well as the packages we used. We then show how we organized our implementation to create a JCA Provider.

### 4.1.1 Packages Used

The policy-based encryption algorithms were implemented in Java, which proved to be a useful tool for implementing cryptographic algorithms. There are many packages available which contain many of the necessary methods for computing cryptographic operations. Some mathematical details can be abstracted by using these methods. For

example, elliptic curve arithmetic operations can be done by calling methods to do the computations (e.g.: maptopoint, multiply). We use a number of packages discussed next in order to ease the amount of implementation work required. See Table 4-1 for a complete listing of packages used.

The javax.crypto package provides the abstract CipherSpi class which we implemented to use our algorithm in PBCCipher. It is required for any Provider that wishes to create an implementation of an encryption algorithm. The standard methods allow for easily knowing how to use the cipher in an encryption system. We discuss the standards more in section 4.2. The Cipher class is used to call our instance of a PBCCipher. The getInstance method allows us to specify which cipher implementation to use. We use the ENCRYPT_MODE and DECRYPT_MODE values to set the cipher to use the encryption or decryption algorithms.

The java.Math package contains a very useful class BigInteger, offering tools for dealing with large number generation and manipulation. We use BigInteger extensively in our implementation. Both the credentials and policy computation involve using BigInteger. Also in the encryption and decryption processes we use it for creating random large numbers, and for computing the hash of the message, random number and policy, which we then combine with the public key. The master key in the system is also a BigInteger.

The java.util provides us with the data structure ArrayList. We use this for computing and storing the lists of credentials and policy constraints which make up the private and public keys. We also use this for constructing the ciphertext. In the encryption method, the ciphertext elements are stored in an ArrayList and then converted to byte array. In the decryption, the byte array is converted back to an ArrayList to access the different parts individually.

The java.security package provides many classes that were used. The Key, PublicKey and PrivateKey interfaces were extended to implement our PBCKey, PBCPublicKey, and

| Packages Used | Classes used | Reason used |
|---|---|---|
| javax.crypto | Cipher<br>CipherSpi | For creating and using the PBCCipher class |
| java.math | BigInteger | Used for operations involving large numbers, for creating keys and in encryption and decryption processes |
| java.util | ArrayList | Structure used to store policy and credential values, also used to form ciphertext in encryption |
| java.security | Key<br>PublicKey<br>PrivateKey<br>KeyPair<br>KeyPairGenerator<br>KeyPairGeneratorSpi | Key classes are used to create and use the private and public keys, and a key generator. |
| | MessageDigest | Used for all the hash operations |
| | SecureRandom | Used for creating a random number used for the master key |
| | Provider | Used for creating a provider to use our implementation |
| | AlgorithmParameters;<br>AlgorithmParameterSpec; | Used for creating the system and key parameters |
| nuim.cs.crypto.blitz | Point<br>Element | Used for AffinePoint and Element which are used to calculate both public and private keys, as well as encryption and decryption operations |
| nuim.cs.crypto.bilinear | ModifiedTatePairing<br>BilinearMap | Used for all Tate Pairing operations, using the bilinear map |
| nuim.cs.crypto.util | BitUtility | Used to transform ArrayList into bytes and vise-versa in encryption and decryption for forming and transmitting ciphertext |

**Table 4-1: Java Packages Used in Implementation**

PBCPrivateKey classes. The KeyPair class contains the standard methods getPublic() and getPrivate() to allow for easily knowing how to access the public and private keys. The KeyPairGenerator class was implemented to create instances of keys used in our encryption system. The MessageDigest provided the MD5 hash that we used in our encryption, decryption and for computing keys. The Provider class was implemented to create our own PBCProvider which allows us to use the algorithms we implemented in a standard way.

In order to save time in the implementation of the encryption algorithm, existing public code was used. The NUI Maynooth Crypto Group has made available a number of libraries with elliptic curve and bilinear pairing code[21]. We used the nuim.cs.crypto.blitz package for the field.Element and point.AffinePoint needed in the encryption, decryption, and computation of keys. The Element and AffinePoint classes are used to compute elliptic curve operations. The nuim.cs.crypto.bilinear package provides the ModifiedTatePairing which was used for the main bilinear pairings computations in the encryption and decryption methods and for computing the keys. The BilinearMap class was also used as a general map to abstract which map is needed. The nuim.cs.crypto.util package provided the BitUtility class of which we used toBytes(). This method was used in the encryption algorithm to transform the ArrayList ciphertext into bytes, and the reverse in the decryption.

### 4.1.2 Implementing the algorithms

Before the encryption and decryption methods are called, the credential values and policy values need to be computed. Computing the credential values involves computing the hash of an *assertion* and calculating the mapping onto a point of that value. Then, multiplying that by the master secret key $s$ gives the affine point *cred* of the credential.

```
byte[] raw=params.hash.digest(assertion);
AffinePoint hashvalue=params.getMap().mapToPoint(new BigInteger(1, raw));
AffinePoint cred = params.getMap().getCurve().multiply(s, hashvalue);
```

It should be noted the master key $s$ must be kept private in order for the system to remain secure.

One computation done in the encryption step can be made more efficient. The step calculating the mapping of a policy constraint with the public key can be pre-computed and stored. The hash of the policy constraint, here a string *time* containing the value of a time period, is mapped to a point on the elliptic curve. This point $A$, along with the master public key $R$, are mapped as a pair and stored as an Element.

```
A = e.mapToPoint(new BigInteger(1, hash.digest(time.getBytes())));
Element policyConstraint = e.getPair(A, R);
```

This step is the most costly of the encryption algorithm and does not rely on the plaintext. Pre-computing and storing these values upon initialization of the system significantly reduces the encryption time. Unfortunately the same can't be done in the decryption process where the calculations done with the credentials involve the ciphertext and cannot be pre-computed.

We use the SecureRandom() method of the java.security package to create random key values used in the encryption.

```
BigInteger trandom = new BigInteger(params.numBits, new SecureRandom());
```

We then calculate a value $U$ using $t$, a combination of all the *trandom* values, the policy *polA*, the message to encrypt $m$, and the master public key $p$.

```
BigInteger r = new BigInteger(1, params.hash.digest(concatenateByte(m, t,
polA.getFullPol())));
AffinePoint U = e.getCurve().multiply( r, p );
```

$U$ is used in the ciphertext to determine if the correct credentials were presented. If the U calculated from the ciphertext does not match $U$ sent as a parameter, then the plaintext is not revealed.

Calculating the masks $g$ which will encrypt each *trandom* key is done as follows using the policy constraints (*polA.getPol(i,j,k)*).

```
for (int i = 0; i<polA.a; i++){
```

```
for(int j=0; j<polA.ai; j++){
    for(int k=0; k<polA.aij; k++){
        g[10]= g[j-1].multiply(polA.getPol(i,j,k));
    g[10]=g[10].modPow(r);
```

*g[10]* is one mask which is then combined with the corresponding *trandom* value to give $v$, one piece of the ciphertext. The ciphertext calculated contains $U$, each $v$ value, and $w$ which is a combination of the plaintext message $m$ and a hash of $t$.

In the decryption method, the credential AffinePoint values *cred.getCred()* are added together and mapped with $U$ sent as a parameter in the ciphertext. We make use here of the Modified Tate Pairing map computation add(AffinePoint, AffinePoint).

```
AffinePoint sumCred=params.getMap().getCurve().add(sumCred, cred.getCred());
g[9]=e.getPair(sumCred,U);
```

We then retrieve the key values from $v$ given in the ciphertext and the equivalent of the mask computed in the encryption step, which is the hash of the value $g$ concatenated with the indices.

```
t = xorArray(v, params.hash2.digest(concatenateByte(g[9].toByteArray(), i, j)) );
```

If the proper key values are obtained, then the plaintext will be revealed by taking the hash with $w$, a value sent in the ciphertext.

```
byte[] m = xorArray(w, params.hash3.digest(t));
```

$U$ is computed from $m$ and the policy $pol_A$, and if the $U$ computed matches the $U$ given in the ciphertext, then the message $m$ is returned.

## 4.2 Java Cryptography Architecture

The Java Cryptography Architecture (JCA) is a framework for creating and using cryptographic methods in Java[41]. In JCA, security providers are created to use security functions of different algorithms. Implementing our policy-based encryption to fit the specifications of the JCA will allow for easily integrating the security features into our CDSS. The encryption algorithm can then be used without having to know the details of its implementation. Integrating our encryption system with our hospital system is made to

Key

Provider

PBCKey
String getAlgorithm()
byte[] getEncoded()
String getFormat()

PBCProvider
static final String PBC
PBCProvider()

«PrivateKey»

PBCPrivateKey
ArrayList credList
PBCPrivateKey(ArrayList)
ArrayList getCred()

PBCPublicKey
byte[] policy
ArrayList computedPolicy
int size
int a
int ai
int aij
PBCPublicKey(byte[], ArrayList)
byte[] getFullPol()
Element getPol(int, int, int)

«PublicKey»

PBCCipher
PBCSystemParameters params
int opmode
Key key
PBCPublicKey publicKey
PBCPrivateKey privateKey
SecureRandom secureRandom
*engineInit( int, Key, SecureRandom)*
engineInit( int, Key, SecureRandom)
engineInit( int, Key, SecureRandom)

PBCSystemParameters
BilinearMap e
AffinePoint randomGeneratorP
AffinePoint R
MessageDigest hash()
PBCSystemParameters(BilinearMap, MessageDigest, BigInteger)
AffinePoint getP()
BilinearMap getMap()
AffinePoint getR()
MessageDigest getHash()
void setHash(MessageDigest)

CipherSpi

KeyPairGeneratorSpi

«AlgorithmParameterSpec»

PBCKeyPairGenerator
PBCKeyParameters params
void initialize(AlgorithmParameterSpec)
void initialize(PBCKeyParameters)
void initialize(int, SecureRandom)
KeyPair generateKeyPair()

PBCKeyParameters
byte[] policy
ArrayList computedPolicy
ArrayList credList
PBCKeyParameters(MessageDigest, AffinePoint)
PBCKeyParameters( MessageDigest, AffinePoint, BilinearMap, BigInteger, String )
byte[] getPublicKey()
ArrayList getPrivateKey()
ArrayList getComputedPol()
byte[] concatenateByte(byte[], byte[])

**Figure 4-1: UML Class Diagram for PBCProvider**

be fairly easy by using JCA. The code to use the encryption is simplified and follows a standard protocol. In our implementation, we create a PBCProvider to include the implementation of the policy-based encryption algorithm.

Creating and using the PBCProvider is very similar to the instructions given for the identity-based encryption provider IbeProvider by the NUI Maynooth Crypto Group group in [50]. A nice feature of the JCA is that the instructions to use different encryption algorithms are very similar. The same steps must be taken to create the provider, parameters, generate keys and use the cipher encryption and decryption functions. The differences lie in choosing different parameters and public key values. In our implementation a policy is used for the public key instead of an identity string.

In order to follow the JCA architecture, we implemented interfaces in Java of the required classes. These include the Provider, Cipher, AlgorithmParameterSpec, KeyPairGenerator, KeyPair, PrivateKey and Public Key classes. See Figure 4-1 for the complete class diagram. In order to use the policy based encryption, a PBCProvider must be created and initiated with the proper parameters. The first step to create the PBCProvider is to create a new provider and system parameters. The system parameters include the choice of hash function we will use, the Modified Tate pairing "map", and the system master (secret) key "mkey".

```
Provider provider = new PBCProvider();
AlgorithmParameterSpec params = new PBCParameters(map, hash, mKey);
```

The key parameters are the hash, system public key "pub", the Modified Tate pairing "map", the system master key "mKey", and the credential assertion values "creds". The policy is created in the code for the key parameters constructor. The reason for this is discussed in section 4.3.1. The credentials can be passed as a string which will be parsed by the code in the key parameters constructor. Here we use an example of a set of credentials "Doctor, Emergency" stating a user has the employee role of a Physician in the Emergency Department and can decrypt resources protected by policies requiring these credentials.

String simplePolicy = "Doctor, Emergency";

AlgorithmParameterSpec keyParams = new PBCKeyParameters(hash, pub, map, mKey, creds);

The private and public keys can then be generated using a KeyPairGenerator.

Keys used in JCA compliant encryption algorithms are usually generated in pairs. This would mean a private key and a public key. For our system, each request to encrypt a resource will require the policy to encrypt under, and each request to decrypt a resource will require the policy it is encrypted under and the user's credentials. Therefore, for each request for encryption or decryption there will be a 'key pair', the policy encrypting the document and the user's credentials associated with the session.

The cipher can be created using Cipher.getInstance specifying the PBCProvider as a parameter. In order to use the encryption the cipher is first initialized and then passed a message to be encrypted. The message can be any string or values transformed into bytes. Here we create a cipher to use the *PBCProvider* we have created. We set the *cipher* to encryption mode and pass the parameters and public key (the policy) to encrypt on. Then we pass a message as a block of bytes *plaintext* we wish to encrypt to the *cipher.doFinal* method. The result is the encrypted message stored in *ciphertext*.

```
Cipher cipher = Cipher.getInstance(PBCProvider.PBC, provider);
cipher.init(Cipher.ENCRYPT_MODE, publicKey, params);
byte[] ciphertext = cipher.doFinal(plaintext);
```

Similarly, the cipher can be initialized for decryption mode and passed a block of ciphertext bytes to be decrypted.

## 4.3 Issues and Solutions

We include some discussion of difficulties faced during the implementation period. These obstacles and our solutions to them determined the final implementation details. We describe some problems faced and our decisions in solving them.

### 4.3.1 Policy and Credential Computation

One of the biggest difficulties was figuring out how to store and use the policies and credentials in practice. Since policies used by the encryption algorithm are written in a combination of conjunctive and disjunctive normal form, entering the policies as strings and then parsing them into separate conditions or rules seems like the best idea. In order to parse the policy, the levels of ANDs and ORs must be represented properly in order to keep the original meaning of the policy. This proved difficult to do without a parser built to handle the structure of the policy. For now, the policy is created in the PBCKeyParameters class, built from individual conditions to form the policy. It is not taking a string as a parameter and parsing it. Future work on the system would include building a parser for handling the policy creation. For managing the computed policies, we use lists to represent the disjunctive and conjunctive normal forms. The policy list contains elements representing the first level of rules ANDed together. Each element is itself a list containing elements ORed together. And finally each of those elements is a list of conditions ANDed together. This seemed to work better than using arrays due to the natural levels that can be created by making lists of lists, and the easily accessible elements by using get(int index) method. The credential assertions are easier to parse since there is no structure needed, a simple list suffices. Each credential assertion is processed separately. The results are stored as a list of byte arrays so that we can access credentials individually. For credentials, storing as a list follows the design of the algorithms in which the credentials are presented as a list.

Creating keys is done by generating a key pair which includes a public key and a private key. In other encryption systems this is normal since each public key has only one private key corresponding to it. In policy-based encryption, the policy is the public key and can have many distinct private keys associated with it. The idea of a key pair in policy-based encryption may not seem so necessary. In our system, we think of the key pair as being the keys needed for that request. So if a user is requesting to decrypt a document, that user's private key will be the one chosen to be generated in that key pair. Now, if encryption is the only process being done, which does not require the private key, then

the private key need not be generated at all for that request. In this case we can simply compute and retrieve the public key without generating a key pair.

```
PublicKey publicKey = keyParam.getPublicKey();
```

This leaves the problem of where to compute the policy and credentials. Since each condition and assertion has to be computed before using the keys, this must be computed when creating the keys. This can take place in the PBCKeyPairGenerator or PBCKeyParameters classes. Since we want to be able to retrieve the public key without computing the private key, we cannot do this in the PBCKeyPairGenerator class. Therefore the PBCKeyParameters class must handle the computations. When creating the new key parameters, there is a choice of constructors to use to pass the parameters and the policy, or the parameters and the credentials. When creating only the public key, we initialize the parameters with the hash function, the system public key pub, and the bilinear map.

```
PBCKeyParameters keyParam = new PBCKeyParameters(hash, pub, map);
```

When creating the key pair, we initialize the parameters with the credential assertions (as strings) and the same parameters as above.

```
PBCKeyParameters keyParam = new PBCKeyParameters(hash, pub, map, mKey,
creds);
```

Allowing this flexibility in computing the keys improves the speed of the encryption system, since these computations are costly.

### 4.3.2 Message Blocks and Ciphertext Blocks

One choice we had to make was the length of the message blocks that are processed in the encryption method. Usually in encryption systems you specify the block size to encrypt on. Ciphertext lengths correspond to the processed message. In our system, we chose for now to keep a varying block length. The random numbers $t$ in the encryption method are created based on the size of the message passed. The doFinal method will accept a message of any length and treat it as one block. If it is desired that a message being encrypted be split into blocks, the main method calling the encryption can

determine the length of the block to be encrypted. This will allow us to encrypt patient records properly in our MET-A$^3$Support system, where we plan to encrypt each field in a patient record separately. Each field can be treated as one block.

## 4.4 Summary

This section included the details of implementing the policy-based encryption system described in section 3. We gave details of the java implementation including the classes used and created in order to build a functional encryption system. We transformed the algorithms into code. We also discussed how the policies and credentials are created and managed. Problems and difficulties are discussed to show the mode of thinking while working at creating an encryption system. We showed how to create a provider to contain the implementation. In the next section we give details on how we use this provider to protect data in a real software system. Having the code organized into a provider allows us to easily incorporate it into another system.

# Chapter 5

# 5 MET-A3Support System

The MET-A$^3$Support system will provide physicians with interactive evidence-based decision support in emergency room settings. A physician using the system will input signs, symptoms and test results, and receive recommendations about possible patient management. The system will use information from patient medical records and present these records to the physician. The MET Research Program has worked on researching and developing this system. Our goal is to provide security and privacy technologies for this project to protect the patient medical information being used and transferred within this system.

In order to protect the patient medical information we need to determine:

- What information to protect
- Where to provide encryption and decryption
- How to incorporate security without disrupting the main features of the system which are to provide doctors with information

This chapter provides details about the work I have done to integrate the policy-based encryption with the MET-A$^3$Support system as well as some general information about the system architecture and communication of agents in the multi-agent system.

## *5.1 Privacy of Patient Medical Information*

Before we discuss the details of how we use the policy-based encryption, we define privacy of patient records. We state that protecting the privacy of a patient record means

protecting any data on that medical record which can be linked to an individual. In Canada, the Personal Information Protection and Electronic Documents Act (PIPEDA) defines personal health information as information about an identifiable individual if the information relates to the physical or mental health of the individual including information about health services provided to the individual[52]. Similarly in the United States, title II of the Health Insurance Portability and Accountability Act (HIPAA 1996) defines Personal Health Information as any information about health status, provision of health care, or payment for health care that can be linked to an individual[33]. Both PIPEDA and HIPAA stress that health information is only considered personal health information if it can be linked to an individual. Therefore, information contained in a patient record that cannot be linked to the patient is not required to be protected.

We need to protect the personal health information in patient records that are used by our MET-A$^3$Support system. We will encrypt identifying information and any other information not needed for computations by the agents in our system using policy-based encryption. Required data such as signs, symptoms and test results (for example a blood pressure reading) will be left in the clear to be used by the clinical decision support software. The allowed plaintext data can be decided on ahead of time to ensure patient identifying information is kept private. In some cases, non-identifying information can be linked to an individual if data can be tracked over time or combined with other data. For example, a birth date alone does not indentify an individual, but a birth date and last name might. In [61] Sweeney describes an algorithm for removing identifying information from database entries. Methods such as these can allow for us to leave some information that our system needs decrypted. This allows for the multi-agent CDSS to complete tasks without having to encrypt and decrypt multiple times which would end up slowing down the performance of the system. We will be protecting the privacy of the patient records by encrypting sensitive information and providing access control with the policy-based encryption.

## 5.2 Integrating Policy-Based Encryption into a Multi-Agent System

In order to integrate the encryption with the multi-agent system we look at the system architecture and where encryption is needed.

### 5.2.1 System Architecture

MET-$A^3$Support is designed as a multi agent system where each agent handles a specific task. Each agent is responsible for one task and is either proactive (setting out to accomplish a specific task) or reactive (providing a service when called upon). Agents provide evaluation suggestions using decision models discovered from data, treatment suggestions using clinical practice guidelines, clinical evidence from a medical library, and patient records from the Hospital Information System (HIS). For more information on the functionality of the system, see [45].

The implementation of the MET-$A^3$Support system is in Java using the Java Agent Development Framework (JADE) compliant with FIPA specifications. The system uses patient records, which are received from the HIS as HL7 messages[34]. Communication with the HIS is controlled by the HIS synchronizer which passes records to the system to be stored. The central repository for the MET-$A^3$Support system is implemented as the Blackboard. It lies between task agents and information database agents. The Blackboard reacts when patient data is added to the Blackboard or modified, actions initiated by system agents. The Encounter Supporter agent handles requests from the user for patient information and decision support. Other agents handle the decision support functionality and interact with the encounter supporter. The general architecture of the MET-$A^3$Support system is shown in Figure 5-1. Physicians will use a portable tablet PC connected to a wireless network – the Encounter Supporter agent will be installed on a portable device and the remaining agents will be hosted on the server.

The encryption and decryption services will be needed by the following agents in our system:

- HIS Synchronizer agent: This agent requires both encryption and decryption services to encrypt new patient information coming into the system and decrypting outgoing information (which will then be transformed into the HIS format).
- Encounter Supporter agent: This agent requires decryption services to present the patient information on the user device for the physician.

We will use JADE Services to incorporate the security functions into our system.

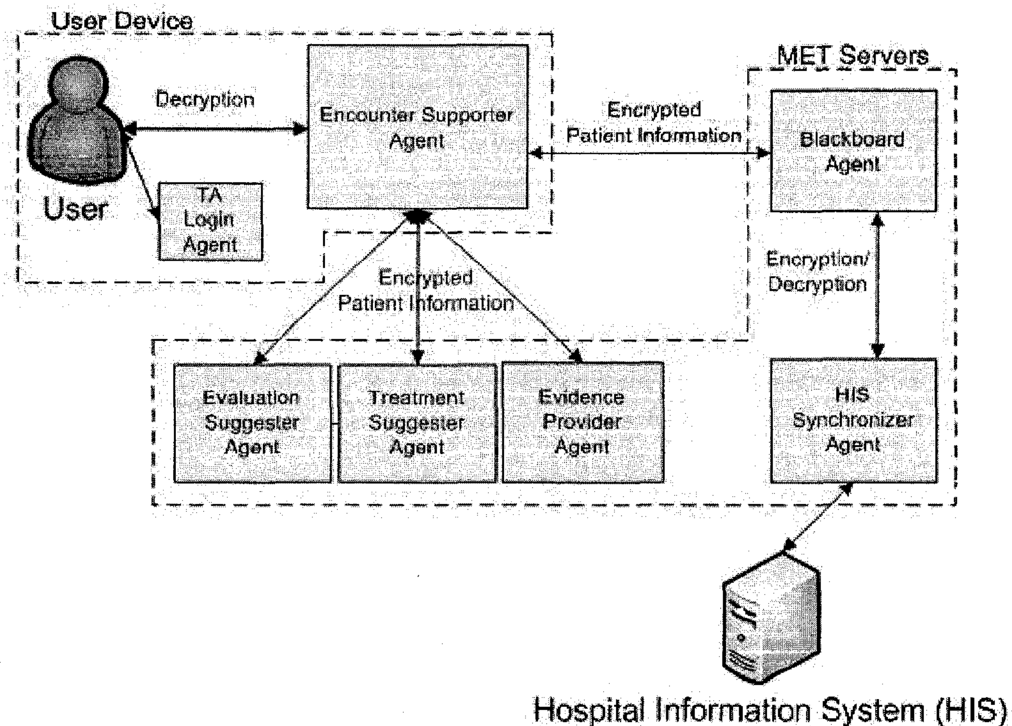


**Figure 5-1: MET-A$^3$Support Architecture**

When integrating security into this system we focus on the Blackboard since records kept here must be encrypted. There will have to be some fields available in plaintext for access by the agents. Some agents need information about the document in order to know which documents to pull or write to. The agents which need access to more information, such as

the content of a document, will also need to be able to decrypt or encrypt depending on the permissions of the user invoking that agent on their PC. Information being stored on the Blackboard is stored on the application servers and must be kept as securely as possible.

### 5.2.2 Organization of Security

Our original idea for incorporating security into the MET-A$^3$Support system was to have security agents. Having agents in charge of encryption tasks fit the current multi-agent model. We could have an encryption agent responsible for encryption and decryption as well as the TA agent responsible for authentication and credential creation. We could also have a policy agent if necessary, although simply having a lookup table of available policies would be sufficient for the encryption agents to get the information they need to perform the operations. The security agents will be needed on both the client and server machines since we will need encryption services for patient information entering the system (on the server side) and patient information being displayed (on the client device).

Using security agents requires sending data to the encryption agents to be encrypted. This is one extra 'leap' with unprotected data that is unnecessary. It would be better to encrypt patient information before the agent sends it in a message, especially if it will be sent over the network from one device to another. Also, looking at the system architecture layout, we will need security agents at different points in the system. We will need security agents to handle incoming patient records for the HIS Synchronizer as well as agents to handle the Encounter Supporter decryption needs for presenting information to the physician. Instead of using agents, we will use JADE Services, built into the infrastructure, to handle the encryption tasks. This will allow each agent to have access to encryption services as needed and where needed. All messages will be encrypted before being sent and so communication between agents will be secured.

With the security integrated within the JADE infrastructure, client side agents and server agents will all have access to security functions. A set of encryption related services

would be available to agents which could invoke functions when needed. This gives our system more flexibility with where encryption and decryption can take place, avoiding any information from being transmitted on the network unencrypted. It also allows for an easier organizational view of the MET-$A^3$Support system architecture. A log-in or TA agent will still be responsible for authentication and providing the user's session with their decryption key including credentials. Once these are associated with a user's session then that will be made available for encryption services on the network until the session has ended due to log out or period of inactivity. We will have to ensure that agents will all have access to encryption services at all times. While slight delays may be acceptable for encrypting or decrypting data to be passed to agents, it would not be acceptable if this information is needed in an emergency situation. Allowing multiple agents to use encryption services at the same time is necessary.

### 5.2.3 Protecting Patient Information

Patient records are organized as a collection of fields, each of which has a value (for example, a patient name field will have value "John Smith"). We use the entity-attribute-value model to represent this model[48]. In our case the entity is the patient record. The record will be a collection of values and each value will be linked to a certain attribute. Since each record has values organized separately, these values can be encrypted separately. Therefore records can be encrypted on a per-field basis. This method allows for records to be updated without decrypting the contents. One field in a record can be written over with a new encrypted value on the blackboard. Each record will also contain an identifier value for tracking purposes. This number is simply an assigned number while the record is in the system and not a patient identifying number. We will be leaving some values of the patient record in clear text. Any patient identifying information and information which is not needed for computations by the agents will be encrypted. This will allow for patient information to be transmitted on the network between agents without having to encrypt and decrypt each time. The values used by the agents for computation will be available.

The patient records on the blackboard are of patients who are currently in the emergency department. Updating policies will only affect these patient records, which are currently being used by the system. These patient records will be decrypted and re-encrypted under the new policy. New patient records entering the system will be encrypted with the new policy. If the updating happens when a user is using the system, they will provide their login information to receive the corresponding updated credentials. The update procedure can be timed so as to cause the least amount of disruption to users (when not many patient records are currently in the system).

### 5.2.4 Agent Communication and Encryption

The agents send messages formatted according to the Agent Communication Language (ACL) to communicate[9]. If patient information is exchanged it is contained in the content part of the message. The content can be represented as either string or bytes. When a patient record is sent in a message, it is sent as one string. Agents receiving patient information in a message can parse the message content to reconstruct the patient record.

We will be creating a JADE Service to handle the encryption functions called EncryptionService. The EncryptionService will encrypt or decrypt patient information sent in a message, depending on what functionality is requested. Agents can directly access features provided by the Service using a Service Helper, here called EncryptionHelper. The direct call to the service would allow us to encrypt the content of the message before formulating the ACLMessage, allowing us to encrypt the values of a patient record separately. When decrypting the content of the message, the encrypted values will be decrypted and all values will be used recreate the original patient record format. Agents will need only to know when to call an encryption service, not the details of how the encryption works. This provides some level of abstraction with the encryption functionality. The Evaluation Suggester, Treatment Suggester, and Evidence Provider agents need not worry about encryption or decryption since the information they require will be in the clear. The HIS Synchronizer and Encounter Supporter agents will use the

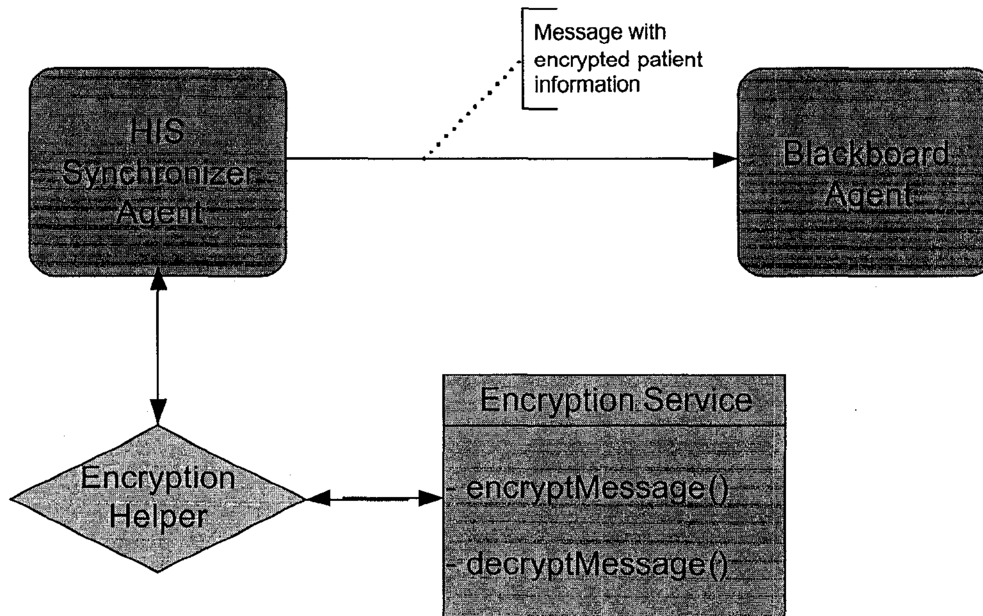


**Figure 5-2: Agent Using Encryption Service**

encryption services appropriately to encrypt or decrypt patient information by retrieving the EncryptionHelper and using the functions encryptMessage() or decryptMessage().

Figure 5-2 shows the how HIS Synchronizer agent will use the Encryption Helper to encrypt a message to be sent to the Blackboard Agent. Upon system initialization, the EncryptionService will be initialized by creating the PBCProvider as shown in section 3.2. The EncryptionService encryptMessage() method can parse message content and create a cipher to encrypt each part of the patient record.

Each call to cipher.doFinal will encrypt one block of bytes (plaintext) containing a field from the patient record. The result will be the encrypted value (ciphetext) of that field.

```
Cipher cipher = Cipher.getInstance(PBCProvider.PBC, provider);
cipher.init(Cipher.ENCRYPT_MODE, publicKey, params);
byte[] ciphertext = cipher.doFinal(plaintext);
```

Each patient record field will be encrypted separately, and these values can then be concatenated together and sent as an encrypted message. The EncryptionService

decryptMessage() mtehod can parse the encrypted message and decrypt each value needed.

### 5.2.5 Policies and Credentials

In MET-A$^3$Support, there are a finite number of policies to be used, which determine access control for patient records. The policies are determined by the hospital to specify the privacy and access control rules for patient records. These policies are not patient-dependent. The policies are not unique to each patient's record. Instead the policies are specific to the type of document being protected. For example, a patient's medical record consists of multiple documents. One document could be a lab report, which has an access control policy associated with it, regardless of which patient it refers to. The association of privacy policies to document types is because of the nature of the emergency room, where multiple nurses and doctors may care for the same patient over the course of their stay. With this setup, we can issue policies for encryption on an automatic basis without user interaction. The Encryption Service will apply the proper policy (chosen from a pre-determined list) for the document being encrypted.

Credentials are generated based on the user's accepted login and the permissions associated with their account. The entity in charge of account permissions (and login) acts as the TA for the encryption system and stores the credentials for each user. The Encryption Service will request credentials associated with the user account for the current user session from the TA. Having the decryption credentials associated with the user accounts means the user does not need to be aware of the encryption process. The user logs into the system as usual and proceeds with their normal tasks.

Updating policies affects only the patient records currently in the system. These can be decrypted and re-encrypted under the new policy. If there are multiple documents of the same type to be re-encrypted, they will be done in a batch. This will ensure that the initialization of the cipher only has to be done once to initialize keys since they will use the same policies for encryption. Then, the decryption and encryption of the documents

can be done efficiently. Credentials are generated per session, when a user log into the system. If the policy is updated and a user's credentials no longer work, they will be asked to log in again to a new session. This will happen especially when the time period of a policy changes and the credentials have an older time stamp. If a policy is updated and the credentials still work, then there is no need to ask the user to log in again. Updates to credentials such as changing definitions of roles or an employee's permissions will affect the user the next time they log in to the system when they generated new credentials. Having the updates reflected when a user logs in to a new session keeps the user from having to manage updates or be aware of encryption tasks.

## 5.3 Workflow

A standard scenario of how a user will proceed to complete tasks using the MET-$A^3$Support system is as follows.

1. The physician logs in by providing their user name and password, and by scanning a barcode on their employee badge (see section 6.1.1).
2. The Accounts agent/TA will verify the login information. If correct then the Accounts agent will generate credentials and associate those credentials with a session id.
3. The physician can then input symptoms in order to get some decision support for diagnosing their new patient.
4. The Encounter Supporter agent will send a message requesting the patient record from the Blackboard agent.
5. The Blackboard agent will send the request to the HIS Synchronizer agent for the new patient's record to be brought into the system.
6. The HIS Synchronizer agent will pull patient record from the HIS. It will call the EncryptionService to encrypt the necessary fields in the record with the appropriate policy and send the partially encrypted record as a message to the Blackboard agent

7. The Blackboard agent will send the patient record to the Encounter Supporter agent.
8. The Encounter Supporter agent will send the input symptoms and patient records as a message to the Evaluation Suggester agent.
9. The Evaluation Suggester agent will process the symptoms and patient record and return the results including the full patient record in a message to the Encounter Supporter agent.
10. The Encounter Supporter agent will then display results from that message to user including patient record. This requires using the EncyrptionService to decrypt the patient record. The decryption method will use the credentials associated with the physician's current session.
11. The Physician can then accept the diagnosis information or input new symptoms to receive new information.

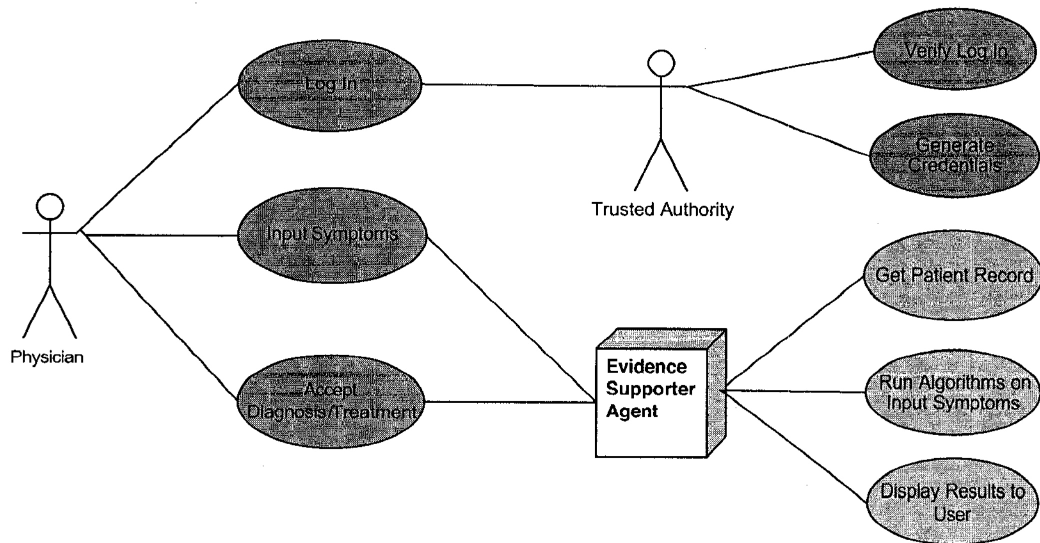


**Figure 5-3: Task Diagram**

## *5.4 Security in the MET-A3Support System*

We need more security features than just the encryption and access control provided by the policy-based encryption. Proper authentication of users logging into the system is

important to maintain the security of the system. Protecting the data from unauthorized modification requires more than just access control. We should also include integrity mechanisms to be able to track if any changes are made. Also we should have audit logs to make sure the system is being used properly. Finally, we have an extra security issue that is unique to our project, for which we implement automatic file purging.

### 5.4.1 Authentication Mechanism

The security of the encryption system relies on strong authentication. The system is only as secure as its ability to prevent unauthorized access. Currently in the hospital, the standard username and password method is used without any restrictions on password choice. We wanted to explore authentication options that would be both secure and usable so as not to change the user experience too much.

The username and password combination is the most popular authentication method. There are many reasons why this is not a secure method[1]. Users will choose very easy to remember passwords that are also easy to break. If we imposed restrictions on the password choice to force users to have stronger passwords, they would do what they could to make logging in easier, by writing them down for instance. We have already had feedback from doctors who would not be happy having password restrictions being put in place and encouraged us to look at other options. Passwords are subject to social engineering attacks which could be easy to manipulate in an emergency setting. If a malicious person claims there's an emergency and asks for a staff member's password, the staff member may be more likely to divulge information.

A two-factor authentication would be desirable to provide improved security. Our motivation for finding a proper two-factor authentication mechanism lies in working with what we have. One factor will be the username and password system without restrictions on the password. The second factor could be a biometric fingerprint or RFID tag. The tablet PC is equipped with both a fingerprint reader and RFID reader. We discuss both and what we chose for the second factor in our authentication scheme.

One of the most common biometrics in use for authentication is the fingerprint. Using the fingerprint readers has an advantage that users don't need to 'remember' to bring a token with them to work. However some usability problems may make this less desirable to use. In order to provide your fingerprint for verification, a user must place their finger in a certain position. Factors such as placement, heat, cold, and perspiration can all affect how accurate the system is[42]. We want our users to be able to log in every time because the setting is the emergency room of a real hospital.

Another issue with fingerprint biometrics is that not everyone can give the fingerprint to enroll. Fingerprints damaged by injuries could be a problem. We want to make sure everyone can use the system, including current employees and future employees. In the healthcare setting many employees may be wearing hygienic gloves which would not allow them to use the fingerprint reader without first removing them. This is a substantial usability problem in our setting. When discussing the fingerprint option with doctors who will be using the system, they seemed reluctant to use fingerprints in the authentication stage and wanted something easier.

The RFID reader offers an alternative to the fingerprint reader. Doctors carry employee badges which can be equipped with a barcode. To sign in a doctor swipes their card (or, if it is a proximity reader, simply has their badge somewhere on their person) and provides their login password. Everyone can be given a badge and it will always be accepted. They don't have to have a high entropy password which may be hard to remember. They may be less likely to simply tell someone their password to let them have access, since they would also have to provide their card. However this option is still available if the doctor wishes to delegate his tasks to another employee for a period of time. This provides for a flexible authentication system compared to using a fingerprint which cannot be delegated. We decided to go ahead with using the doctor's badges and the RFID reader as the second factor in our authentication system.

Unlike the fingerprint method, a doctor can forget their employee badge. If they borrow someone else's pass or get a temporary one, this will not work with the reader since it will be a different pass. We could manage to have temporary badges available for the day which can be associated with their account. One way around this is to use a common backup method of asking the user a series of pre-answered questions. If the user answers correctly and provides their usual login information along with it, then the user can log in for the day. Proper tracking of these events is important for being able to notice and document malicious behavior.

### 5.4.2 Audit Logs

An important privacy requirement and a feature our system must ultimately include is an audit log. Audit logs allow for tracking user's activity on the system. If by any chance someone is misusing the system, then a record of that activity must be available. For example if a doctor is accessing large amounts of patient records that clearly aren't all their patients, this would be worth investigating. It could be that a malicious user has gotten access to that account and is stealing patient information. It would be a good addition to our system to add logging functionality of all access and changes to patient records. Specifically we want to track when a user logs on and off, which records they request, and which records they make changes to.

### 5.4.3 Integrity Mechanism

Another privacy aspect is not only protecting the data from unauthorized access but also from unauthorized modification. Ensuring proper access control for read/write permissions is essential. However we also want to include some sort of integrity mechanism to be able to check that no changes have been done maliciously. For example if someone changes a record stored in the database we need to be able to check this.

A Message Authentication Code (MAC) applied to a record would provide us with a way of checking for any changes made[38].We could do this on an individual record basis,

creating a MAC of the entire record, storing the MAC value elsewhere on the system, and comparing the stored MAC with a freshly-computed MAC for every record read event.

### 5.4.4 File Purging

Consider the scenario where a doctor has their tablet PC which they bring home everyday. At the end of their shift, they may have seen a number of patients and have their data saved on the PC. The doctor goes home and someone in their household uses the PC for other purposes. It could be connected to the Internet at this point and anyone with access to the PC could have access to the sensitive records.

We need a system of preventing records from leaving the hospital which is facilitated by the use of the tablet PCs. A doctor may not notice or remember that he still has files on his PC when leaving the hospital. It's much more obvious with paper-based records, where a doctor has to actually carry them out or knowingly put them in a bag. If it becomes a habit of taking the tablet PC home, then the doctor may not remember to erase patient data each time. This means we need to know when a tablet PC is being taken out of the hospital so that we can erase those files that haven't been deleted.

One simple thing to do is to purge all files on shutdown. However sometimes PCs are not shut down properly especially in the case of a tablet PC which may lose battery power. Therefore it would be better to implement on system startup. Each time the laptop is booted, the files are purged. Since doctors wouldn't be turning it off and on all day, they shouldn't lose any records until they are done their work. We don't need to implement this on hibernate states, so that if a doctor leaves his PC for a while with no activity then he'll still have the information he needs on it.

If a tablet PC leaves the hospital while still turned on, we still need to purge the files. In this case we can't rely simply on the files being purged when the PC is turned off or rebooted. The connection to the wireless network can be used as an event trigger to purge all files. When the tablet PC is taken out of range and no longer has a connection with the

network, the patient data files will be erased. A doctor who is in the wireless network area doing his work will not lose any data. It would only affect tablet PCs that are taken far enough away from the hospital department with the network connection available.

## 5.5 Summary

We described the MET-A$^3$Support multi-agent clinical decision support system. The central temporary storage is handled by the Blackboard agent. Patient records coming into the system at the HIS Synchronizer agent are then stored on the blackboard. Three main functionality agents on the server do computations involving the patient record information. Patient records are presented to the physician on a wireless client device by the Encounter Supporter agent.

We added the encryption system to protect the parts of patient records that contained personal identifying information. We use JADE Service and Service Helper to integrate our encryption algorithms within the agent communication. Agents that need to encrypt or decrypt messages will use the Service Helper which will handle the encryption tasks. The encryption will take place when messages enter the system at the HIS Synchronizer agent. Policies used in the encryption will be automatically applied based on the type of information being encrypted that is specified by the agent. Decryption will take place at the client device to show the physician the patient information. The encounter supporter agent will handle the request. Users will be given a session when they log in that has their decryption credentials associated with it, so there is no need for them to authenticate themselves at each request to view patient records.

# Chapter 6

# 6 Conclusion

We give a summary of the contributions made in this thesis. We also talk about future research that would benefit the work on both the encryption system and the MET-$A^3$Support system.

## *6.1 Contributions*

We started off by looking at available encryption systems to fit our security requirements for the MET-$A^3$Support system. We wanted an encryption system that allowed for access control by encrypting on roles and other constraints in the form of a policy. This gives both access control and encryption in one package. We would be able to encrypt a resource without knowing who the exact recipients are, since the rules under the encryption guide who can decrypt it without specifying individuals. Policy-based encryption offers the best solution for our security needs. We also wanted to be able to automate and abstract the encryption tasks. We do this by associating the policy with document types, so that when a document is saved, a policy can be applied and have it encrypted without user action. Credentials are associated with a user's session and generated on a successful login.

We found that Bagga and Molva's policy-based encryption system closely fit our requirements with an extension to their work. We add some expressivity to the policy to allow for specifying the action allowed once a resource is acquired. We add the information in the policy and track the user's request to open a document in the proper access mode.

We offer a complete working Java implementation of the encryption system. We implemented the algorithms discussed in section 3 using some existing code for elliptic

curve arithmetic and bilinear operations. We gave details for how to manage and create keys. We added some steps for efficiency by doing calculations involving the policy before the encryption algorithm is called. This allows for faster (multiple) encryptions after the initialization has been done. We also show how to only compute the public key when encryptions are needed, to save time from having to compute the private key which is only needed for decryptions. After implementing the algorithms we created the PBCProvider to use the algorithms. Having our system follow this framework makes it easy to incorporate into other software systems.

Our final contribution involved a real world example of using the policy-based encryption system by integrating it with an existing hospital system. We demonstrated the need for security technologies to protect sensitive data according to privacy legislation. We show to protect patient identifying information using the policy-based encryption. Details were provided about the MET-$A^3$Support system architecture and how the encryption fits into place. We showed that it was easy to integrate the encryption Provider with the existing system because of the JCA framework. Encryption and decryption are automated and the details abstracted so that users don't need to do extra work.

## *6.2 Future Work*

The policy-based encryption system works properly as is, and the implementation can be used in systems as we have shown with the MET-$A^3$Support system. However, there are always improvements to be made and more work to be done. Some improvements include using Security Mediators for the key revocation instead of time stamps and using formal languages to write the policies. Testing is also something that must be done to encourage adoption of this technology.

### 6.2.1 SEMs

SEcurity Mediators or SEMs can be used as another solution to key revocation[49]. Instead of relying on a time period of validity for a key, SEMs can revoke individual keys at any time. In an encryption system with SEMs, the user's private key is separated into two parts. One part is kept secret by the user and the other part is kept by the SEM. When a user wants to decrypt something, they send the ciphertext to the SEM, which returns a partially decrypted result. The user can then use their half of the private key to complete the decryption. Having the user cooperate with the SEM for each decryption request means that at any time the SEM could deny service to the user, which would mean the user cannot decrypt. This is the equivalent of key revocation, since the user can no longer decrypt. This can be done in any situation such as a user having privilege changes. Using this form of key revocation has the advantage that it can be done on an individual basis, instead of using a time period which will affect all users with a key generated in the old time period. We have already a TA available that will process requests for keys once per session, and the same could be done with an SEM.

### 6.2.2 Testing

The first testing to be done should be to verify the correctness of the implemented encryption algorithms. We used java code from the NUIM group and should test it to make sure that it is in fact correct. One way to test the encryption algorithm is to use a known example of its use and compare values at every step. Then we can see that our computations are working correctly.

Usability testing and performance testing are both necessary future work for our system. We want to perform usability testing of the system to ensure that user would be able to use the system properly to provide maximum security possible. If the system is not usable, users may try to work around it, for example by having one person log into the system on a tablet PC and then keeping that session open even when other users want to use it. Usability is one of the main reasons for many other encryption systems not being popular in use. We would want to test both the policy-based encryption system on its own

and the MET-A$^3$Support system using the encryption. To test the policy-based encryption a simple user interface would be constructed. We would have the users attempt to do both encryption and decryption. For encrypting, we want to test how easy it would be for a user to choose a policy and encrypt a document with it. To decrypt a document we would have the user provide a username and password to see the results.

We also want to test how usable the system is when integrated into the MET-A$^3$Support system. Users should be able to log into a session and make multiple requests for patient records. Since the encryption does not require tasks by the user, it may seem unnecessary to test for usability. However with the encryption requiring time and causing some delay in receiving/viewing patient records, it is worthwhile to test how users react to and understand the processes taking place. We also want to test usability when updates are done to the policies or credentials to see if this affects usability, especially if users have to log in again.

Performance testing is also necessary to determine how viable a system like this would be for use in a system such as the MET-A$^3$Support system. This type of testing would focus on how long certain operations (such as encryption) take on given platforms and environments. Initial testing gives a general idea of how long each main process in the encryption system takes. Table 6-1 shows the timing for key initialization (including computing the policy), encryption, and decryption with two different sized policies. The

| | **6 Conditions in policy** | **8 Conditions in policy** |
|---|---|---|
| **Key initialization time** | 53 seconds | 70 seconds |
| **Encryption time** | 1 seconds | 1 seconds |
| **Decryption time** | 8 seconds | 9 seconds |

**Table 6-1: Initial Performance Testing of Encryption System**

timing efficiency is affected primarily by the number of conditions in the policy, close to these sizes. For a policy with 6 conditions, the total time for initialization of key and encryption is less that one minute. For a patient record with multiple fields to encrypt, the encryption process (after initialization) requires only 1 second. Therefore encrypting a

patient record would take approximately 1 minute with initialization time. If multiple patient records of the same type are to be encrypted (and use the same policy) then it will take only a few seconds more, since initialization will not have to be repeated. Decryption for presenting the patient information on the client device would only take 8 seconds. The preliminary performance testing was completed on a 1.61GHz HP laptop with 1GB RAM. These times should be appropriate for our hospital system which is designed to play a supportive role by providing evidence support for diagnosis and treatment information. More testing should be done to determine if a policy-based encryption system such as this would be viable in a more urgent software system requiring smaller delays.

The preliminary testing shows that the timing can be acceptable for our hospital system. However we want to do a complete formal testing. We would again want to test the performance of the policy-based encryption system on its own, and as well when integrated with the MET-$A^3$Support system. We want to test the individual operations such as the initialization, computing policies, computing credentials, encryption and decryption. We also want to test the computations we have done to increase efficiency. For this we should test the creation of policies and the encryption without the efficient step of pre-computing the policy conditions and then again with the efficient step included. We should also test the creation of both keys at the beginning versus individually when we only need a public key created for encryption. We should do the testing on the devices to be used in the hospital trial period which are portable tablet PCs.

### 6.2.3 Policies and Credentials

There is much work left to be done with the policies and credentials. For now, the policies and credentials are created and computed in the PBCKeyParameters class. For the policy, the conditions are separate strings that are processed and organized together into lists. The strings are not input by the user, but built into the class function. This is because in order to parse the full policy string, a parser must be constructed to take the Boolean expression and separate the conditions while keeping the proper structure. A

parser can be made to allow for policies to be input by an administrative user or even read in from a file. The same can be done for credentials, which can be stored with the user's account information and read in when needed.

Currently we have no formal definitions of the policies and credentials, they are simply strings. The policies are structured into a Boolean expression of disjunctive and conjunctive normal form, and are parsed into individual conditions. There is no formal document containing the policy or any other features that may be useful to have kept with it. Languages such as eXtensible Access Control Markup Language (XACML) allow for writing access control policies in a formal structure[31]. The policy-based encryption does require the policy to be in a Boolean structure for processing, however it can be parsed from a string or document written in any way, as long as the Boolean structure is understood and translated when the policy needs to be computed. Having a standard structure for writing the policies used in the encryption system would benefit any organization using the system to allow them to follow standard procedures for writing and using policies.

## 6.3 Lessons Learned

During implementation of the encryption methods we found that Java was the best language to use. Java has many useful packages for developing encryption systems. With built in hash functions and large number manipulation, many of the complex operations are possible with just one call. Also in our case, we found other public implementations with code that we could use, including the code for the bilinear pairing. By reusing some code and the built in Java packages, it was possible to focus on the overall design of the system instead of spending all the effort on the individual operations. Also by using the JCA framework our implementation would be relatively easy to integrate with another system, as we have shown with our hospital system. Having implemented the encryption system in Java also allows for using it across many platforms and devices.

We learned many lessons in applying security technologies to in a health care setting. Some of the best security solutions for use elsewhere may not necessarily be fit for use in hospital systems. For example, fingerprint biometrics is a well-studied authentication technology that solves the problem of forgetting a password or hardware token. However, in a hospital setting, a simple part of a doctor's task, wearing sanitary gloves, creates problems in using the technology. There are unique requirements like these for a health care setting that will ultimately determine whether a new technology will be adopted. With hospitals using more electronic systems and electronic patient medical records, security technologies will be required. However, security is not the first priority in a health care setting, where caring for the patient's health needs come first. In order to incorporate security into hospital systems, every scenario has to be discussed with the users while designing such a system to ensure usability and acceptance. Reducing the number of tasks required by the user and the complexity of these tasks will also lead to better acceptance.

# Bibliography


[1] A. Adams and M. Sasse, Users are not the enemy, In Communications of the ACM, pages 40-46, 1999.

[2] Array Networks Inc. SSL VPN vs IPSec VPN, Jan. 2003, accessed on November 15, 2008.

[3] G. Ateniese and B. Medeiros, Identity-based Chameleon Hash and Applications, In Proceedings of Financial Cryptography, FC2004, LNCS 3110, pages 164-180, Springer-Verlag, 2004.

[4] J. Baek, J. Newmarch, R. Safavi-Naini and W. Susilo, A Survey of Identity-Based Cryptography, In Proceedings of the Australian UNIX and Open Systems User Group, AUUG 2004, pages 95-102, Victoria, Australia, 2004.

[5] J. Baek and Y. Zheng, Identity-Based Threshold Decryption, In Proceedings of Public Key Cryptography, PKC 2004, LNCS 2947, pages 262-276, Springer-Verlag, 2004.

[6] W. Bagga and R. Molva, Collusion-Free Policy-Based Encryption, In Proceedings of Information Security Conference, ISC 2006, pages 233-245, 2006.

[7] W. Bagga and R. Molva, Policy-based cryptography and applications. In Proceedings of Financial Cryptography and Data Security, FC 2005, LNCS 3470, pages 72–87, Springer-Verlag, 2005.

[8] W. Bagga, R. Molva and S. Crosta, Policy-Based Encryption Schemes From Bilinear Pairings, Proceedings of the 2006 ACM Symposium on Information, computer and communications security, ASIACCS 2006, March 2006.

[9] F. Bellifemine, G. Caire and D.Greenwood, Developing Multi-Agent Systems with JADE, Wiley Series in Agent Technology, JohnWiley & Sons, New York, NY, USA, 2007.

[10] Berson, William, Secure document and method and apparatus for producing and authenticating same, United States, Pitney Bowes Inc. (Stamford, CT), Patent #5388158, 1995. http://www.freepatentsonline.com/5388158.html, accessed September 15, 2008.

[11] J. Bethencourt, A. Sahai and B. Waters. Ciphertext-Policy Attribute-Based Encryption, Proceedings of the 2007 IEEE Symposium on Security and Privacy,

IEEE Computer Society, Washington, DC, USA, pages 321-334, 2007.

[12] A. Bittau, M. Handley and J. Lackey, The Final Nail in WEP's Coffin, The 2006 IEEE Symposium on Security and Privacy, pages 386-400, 2006.

[13] A. Boldyreva, Efficient Threshold Signatures, Multisignatures and Blind Signatures Based on the Gap-Diffie-Hellman-group Signature Scheme, In Proceedings of Public Key Cryptography, PKC 2003, LNCS 2567, pages 31-46, Springer-Verlag, 2003.

[14] D. Boneh and X. Boyen, Efficient Selective-ID Secure Identity Based Encryption Without Random Oracles, Advances in Cryptology - Proceedings of EUROCRYPT 2004, LNCS 3027, pages 223-238, Springer-Verlag, 2004.

[15] D. Boneh and M. Franklin, Identity-Based Encryption from the Weil Pairing, In Proceedings of CRYPTO 2001, pages 213-229, Springer-Verlag, 2001.

[16] D. Boneh, C. Gentry and M. Hamburg, Space-Efficient Identity Based EncryptionWithout Pairings, Proceedings of the 48th Annual IEEE Symposium on Foundations of Computer Science, pages 647-657, IEEE Computer Society, 2007.

[17] D. Boneh, C. Gentry, B. Lynn and H. Shacham, Aggregate and Verifiably Encrypted Signatures, from Bilinear Maps, Advances in Cryptology - Proceedings of EUROCRYPT 2001, LNCS 2656, pages 416-432, Springer-Verlag, 2003.

[18] D. Boneh, B. Lynn, and H. Shacham, Short Signatures from the Weil Pairing, Advances in Cryptology - Proceedings of ASIACRYPT 2001, LNCS 2248, pages 566-582, Springer-Verlag, 2001.

[19] X. Boyen, Multipurpose Identity-Based Signcryption: A Swiss Army Knife for Identity-Based Cryptography, Advances in Cryptology - Proceedings of CRYPTO 2003, LNCS 2729, pages 382-398, Springer-Verlag, 2003.

[20] R. Bradshaw, J. Holt and K. Seamons, Concealing complex policies with hidden credentials, Cryptology ePrint Archive, Report 2004/109, 2004.

[21] A. Burnett, K. Winters and T. Dowling, A Java Implementation of an Elliptic Curve Cryptosystem, Recent advances in Java Technology, Computer Science press, Trinity College, 2002.

[22] J. Cha and J. Cheon, An Identity-Based Signature from Diffie-Hellman Groups, In Proceedings of Public Key Cryptography, PKC 2003, LNCS 2567, pages 18-30, Springer-Verlag, 2003.

[23] M. Chase, Multi-authority attribute-based encryption, The Fourth Theory of Cryptography Conference, TCC 2007, LNCS 4392, pages 515-534, Springer, 2007.

[24] L. Chen, K. Harrison, D. Soldera and N. P. Smart, Applications of Multiple Trust Authorities in Pairing Based Cryptosystems, Proceedings of InfraSec 2002, LNCS 2437, pages 260-275, Springer-Verlag, 2002.

[25] C. Cocks, An Identity Based Encryption Scheme Based on Quadratic Residues, Cryptography and Coding - Institute of Mathematics and Its Applications International Conference on Cryptography and Coding, Proceedings of IMA 2001, LNCS 2260, pages 360-363, SpringerVerlag, 2001.

[26] Canadian Standards Association, 2008, http://www.csa.ca/standards/privacy/code/Default.asp?language=english, accessed on September 15, 2008.

[27] W. Diffie and M. E. Hellman, New directions in cryptography, IEEE Trans. Info. Theory IT-22 No.6, pages 644-654, November 1976.

[28] Directive 95/46/EC of the European Parliament and of the Council of 24 October 1995 on the protection of individuals with regard to the processing of personal data and on the free movement of such data 1995. http://www.cdt.org/privacy/eudirective/EU_Directive_.html, accessed on September 15, 2008.

[29] D. Ferraiolo, J. Cugini and R. Kuhn, Role-based access control (RBAC): Features and motivations, In Proceedings of the 11th Annual Conference on Computer Security Applications, pages 241-248, 1995.

[30] C. Gentry and A. Silverberg, Hierarchical ID-Based Cryptography, Proceedings of ASI-ACRYPT 2002, LNCS 2501, pages 548-566, Springer-Verlag, 2002.

[31] S. Godik and T. Moses, eXtensible Access Control Markup Language, XACML Version 1.0, Oasis Standard, 2003.

[32] 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, pages 89-98, 2006.

[33] Health Insurance Portability and Accountability Act of 1996 (HIPAA 1996) http://aspe.hhs.gov/admnsimp/pl104191.htm, accessed on September 15, 2008.

[34] Health Level 7, http://www.hl7.org/, 2008, accessed on September 15, 2008.

[35] F. Hess, Efficient Identity Based Signature Schemes Based on Pairings, In Proceedings of Selected Areas in Cryptography, SAC 2002, LNCS 2595, pages 310-324, Springer-Verlag, 2002.

[36] J. Holt, R. Bradshaw, K. E. Seamons and H. Orman, Hidden credentials, In Proceedings of the 2003 ACM Workshop on Privacy in the Electronic Society, ACM Press, 2003.

[37] J. Horwitz and B. Lynn. Toward Hierarchical Identity-Based Encryption. In Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques: Advances in Cryptology, pages 466-481, 2002.

[38] R. R. Jueneman, S. M. Matyas and C. H. Meyer, Message Authentication, IEEE Communications Magazine, pages 29-40, 1985.

[39] A. Kapadia, P. Tsang and S. W. Smith. Attribute-Based Publishing with Hidden Credentials and Hidden Policies. In 14th Annual Network and Distributed System Security Symposium, NDSS 2007, pages 179-192, 2007.

[40] S. Khanvilkar and A. Khokhar, Virtual Private Networks: An Overview with Performance Evaluation, IEEE Communications Magazine, pages 146-154, October 2004.

[41] J. Knudsen, Java Cryptography, O'Reilly, 1998.

[42] G. Lassmann, Some results on robustness, security and usability of biometric systems, In Proceedings of the 2002 IEEE International Conference on Multimedia and Expo, ICME 2002, pages 577-579, 2002.

[43] B. Libert and J. Quisquater, New Identity-Based Signcryption Schemes Based on Pairings, IEEE Information Theory Workshop, 2003. (See also Cryptology ePrint Archive, Report 2003/023).

[44] J. Malone-Lee, Identity-Based Signcryption, IACR ePrint Archive, Report 2002/098. http://eprint.iacr.org/

[45] W. Michalowski, R. Slowinski, S. Wilk, K. Farion, J. Pike and S. Rubin, Design and Development of a Mobile System for Supporting Emergency Triage, Methods of Information in Medicine, vol. 44, no. 1, pages 14-24, 2005.

[46] M. C. Mont, P. Bramhall, C. R. Dalton and K. Harrison, A Flexible Role-based Secure Messaging Service: Exploiting IBE Technology in a Health Care Trial, Hewlett-Packard Laboratories, technical report HPL-2003-21, 2003.

[47] Motion Computing, Motion C5, 2007, http://www.motioncomputing.com/, accessed on September 15, 2008.

[48] P. M. Nadkarni, L. Marenco, R. Chen, E. Skoufos, G. Shepherd and P. L. Miller, Organization of Heterogeneous Scientific Data Using the EAV/CR Representation, Journal of the American Medical Informatics Association, vol. 6, pages 478-493, 1999.

[49] D. Nali, A. Miri and C. Adams, Efficient Revocation of Dynamic Security Privileges in Hierarchically Structured Communities, Proceedings of the Second Annual Conference on Privacy, Security and Trust, 2004.

[50] L. Owens, A. Duffy and T. Dowling, An Identity Based Encryption system, In Proceedings of the 3rd international symposium on Principles and practice of programming in Java, ACM International Conference Proceeding Series, vol. 91, pages 154-159, Springer-Verlag, 2004.

[51] Personal Health Information Protection Act (PHIPA 2004), http://www.health.gov.on.ca/english/public/legislation/bill_31/personal_info.html, accessed on September 15, 2008.

[52] Personal Information Protection and Electronic Documents Act (PIPEDA 2000), http://www.privcom.gc.ca/legislation/02_06_01_01_e.asp

[53] M. Pirretti, P. Traynor, P. McDaniel, and B. Waters, Secure Attribute-Based Systems, In Proceedings of the 13th ACM conference on Computer and communications security, CCS 2006, pages 99-112, 2006.

[54] A. Sahai and B. Waters, Fuzzy Identity Based Encryption, IACR ePrint Archive, Report 2004/086, http://eprint.iacr.org/.

[55] Secure Computing, SafeWord and HIPAA, 2008, http://www.securecomputing.com/index.cfm?skey=672, accessed on September 15, 2008.

[56] A. Shamir, How to Share a Secret, Communications of the ACM, Vol. 22, 1979, pages 612-613.

[57] A. Shamir, Identity-based cryptosystems and signature schemes, In Proceedings of CRYPTO 84 on Advances in cryptology, pages 47-53. Springer-Verlag New York, Inc., 1985.

[58] Shamus Software, http://www.shamus.ie, accessed on September 15, 2008.

[59] N. Smart. Access control using pairing based cryptography, In Proceedings CT-RSA 2003, LNCS 2612, pages 111-121, Springer-Verlag, April 2003.

[60] Stanford Applied Crypto Group, IBE Secure E-mail, http://crypto.stanford.edu/ibe/, accessed on September 15, 2008.

[61] L. Sweeney, Maintaining Anonymity When Sharing Medical Data, the Datafly System. MIT Artificial Intelligence Laboratory Working Paper. Cambridge: AIWP-WP344, 1997.

[62] K. Sycara, J. A. Giampapa, B. K. Langley, M. Paolucci, The RETSINA MAS, a Case Study, Software Engineering for LargeScale Multi-Agent Systems: Research Issues and Practical Applications, LNCS 2603, Garcia et al., ed., pages. 232-250, Springer-Verlag, 2003.

[63] E. Tews and M. Beck, Practical Attacks Against WEP and WPA, http://dl.aircrack-ng.org/breakingwepandwpa.pdf, accesssed November 19, 2008.

[64] Voltage, http://www.voltage.com, accessed on September 15, 2008.

[65] F. Zhang and K. Kim, ID-based Blind Signature and Ring Signature from Pairings, Advances in Cryptology, Proceedings of ASIACRYPT 2002, LNCS 2501, pages 533-547, Springer-Verlag, 2002.

[66] F. Zhang, R. Safavi-Naini and W. Susilo, An Efficient Signature Scheme from Bilinear Pairings and Its Applications, In Proceedings of Public Key Cryptography, PKC 2004, LNCS 2947, pages 262-276, Springer-Verlag, 2004.

# Appendix A

## Glossary of Terms

**Ciphertext** – A random looking string produced by encrypting a plaintext

**Ciphertext space** – Set of ciphertexts an encryption algorithm could produce

**Collusion** – When users work together (typically to achieve something malicious)

**Credentials** – Permissions granted to a user

**Cryptographic hash function** – Function that produces a fixed length string from the input string, the input cannot be guessed from the result

**Decryption** – the process of transforming a ciphertext back into a plaintext by means of a decryption algorithm and secret key

**Encryption** – the process of transforming a plaintext into a ciphertext by means of an encryption algorithm and a key

**Identity-based cryptography** – A public key cryptography system where the public key is a string representing the identity of the user

**Message space** – Set of messages allowed as plaintext for the encryption algorithm (usually described as strings of a certain length)

**Plaintext** – A message usually in the form of a string that will be encrypted

**Policy** – A string representing a set of access control rules

**Policy-based cryptography** – A public key cryptography system where the public key is a policy and the private key consists of credentials

**Public key** – A key value that is publicly known, mathematically related to a corresponding private key

**Public key cryptography (asymmetric cryptography)** – An encryption system that uses two different keys, one public and one private, to encrypt and decrypt messages

**Private key** – A key value that must be kept secret, mathematically related to a corresponding public key

**Private key cryptography (symmetric cryptography)** – An encryption system that uses one key, kept secret, to encrypt and decrypt messages

**Resource** – A document, message, or other data that is to be protected by encryption

**Role-based access control** – method for limiting access only to authorized users

**Security Mediator** – A trusted authority which keeps a portion, or all, of each user's private key and cooperates with users in the decryption process

**Trusted Authority** – An entity in charge of generation and distribution of keys

**User** – A person using a software system

# Appendix B

# Java Code for PBCProvider

## *PBCCipher.java*

```
package PBC;

import javax.crypto.Cipher;
import javax.crypto.CipherSpi;
import java.math.BigInteger;
import java.security.Key;
import java.security.SecureRandom;
import java.security.AlgorithmParameters;
import java.security.spec.AlgorithmParameterSpec;
import java.util.ArrayList;

import nuim.cs.crypto.bilinear.*;
import nuim.cs.crypto.blitz.point.AffinePoint;
import nuim.cs.crypto.blitz.field.Element;
import nuim.cs.crypto.util.BitUtility;
/**
 *
 * @author Kathryn Garson
 * August 17, 2008
 *
 * class PBCCipher
 *  contains the code implementing the encryption and decryption algorithms
 *
 */
public class PBCCipher extends CipherSpi{
   PBCSystemParameters params;
   int opmode;
   Key key;
   PBCPublicKey publicKey;
   PBCPrivateKey privateKey;
   SecureRandom secureRandom;

   //Not used, use engineInit(int,Key,AlgorithmParameterSpec,SecureRandom) instead
   protected void engineInit( int opmode, Key key, SecureRandom secureRandom ) {

   }
   //Not used, use engineInit(int,Key,AlgorithmParameterSpec,SecureRandom) instead
```

```
protected void engineInit( int opmode, Key key,
   AlgorithmParameters algorithmParameters, SecureRandom secureRandom ) {

}

//Initializes the cipher to encryption or decryption mode with the proper key
protected void engineInit( int opmode, Key key, AlgorithmParameterSpec
algorithmParameterSpec, SecureRandom secureRandom ) {
   this.opmode=opmode;
   if(opmode==Cipher.ENCRYPT_MODE){
      this.publicKey=(PBCPublicKey) key;
   }
   if(opmode==Cipher.DECRYPT_MODE){
      this.privateKey=(PBCPrivateKey)key;
   }
   this.params=(PBCSystemParameters)algorithmParameterSpec;
   this.secureRandom = secureRandom;
}

//Not used, use instead engineDoFinal(byte[], int, int)
public int engineDoFinal( byte[] input, int inputOffset, int inputLen, byte[] output, int
outputOffset ) {
   return 0;
}

//Main method that calls encryptBlock and decryptBlock
public byte[] engineDoFinal( byte[] input, int inputOffset, int inputLen ) {
   if( opmode == Cipher.ENCRYPT_MODE ) {
      return(encryptBlock(input));
   }
   else if(opmode == Cipher.DECRYPT_MODE){
      return(decryptBlock(input));
   }
   return null;
}

//Not used, use instead engineDoFinal(byte[], int, int)
public byte[] engineDoFinal(byte[] input){
   return null;
}

//Not used
public int engineUpdate( byte[] input, int inputOffset, int inputLen, byte[] output, int
outputOffset ) {
   return 0;
}
```

```
//Not used
public byte[] engineUpdate( byte input[], int inputOffset, int inputLen ) {
   return null;
}

//Main encryption algorithm, called from engineDoFinal(byte,int,int)
public byte[] encryptBlock(byte[] m){
   //pick random ti {0,1}^n for i = 1,...,a
   BigInteger trandom = null;
   byte[] t = null;
   for(int i=0; i<publicKey.a; i++){
      do {
         trandom = new BigInteger(m.length*8,2,new SecureRandom() );
      }
      while( trandom.compareTo( BigInteger.ZERO ) <= 0 );

      //compute t = XOR of all ti
      if(i==0)
         t=trandom.toByteArray();
      else
         t=xorArray(trandom.toByteArray(), t);
   }

   //compute r = hash0(m||t||pol)
   byte[] temp = concatenateByte(m, t);
   temp = concatenateByte(temp, publicKey.getFullPol());
   byte[] temp2 = params.hash0.digest(temp);
   //create biginteger from resulting hash
   BigInteger r = new BigInteger(1, temp2);

   //get p
   AffinePoint p = params.getP();
   //get map
   BilinearMap map = params.getMap();
   //compute U=r.p
   AffinePoint u = map.getCurve().multiply( r, p );

    Element[] g = new Element[publicKey.size];
    Element temporary;
    byte[] one = new byte[1];
    byte[] two = new byte[1];
    byte[] tempg;

   //start ciphertext
   ArrayList list = new ArrayList();
```

```
  list.add(u);

    //For i = 1; for j = 1, 2, k=1
  for (int i = 0; i<publicKey.a; i++){
    for(int j=0; j<publicKey.ai; j++){
      for(int k=0; k<publicKey.aij; k++){

        //compute gi,j = product[e(Ri,j,k, Hash0(Ai,j,k))]
        if (k>0){
          temporary = g[j].multiply(publicKey.getPol(i,j,k));
        }
        else{
          temporary= publicKey.getPol(i,j,k);
        }
        g[j]=temporary;
      }

    //compute vi,j = ti XOR Hash2(gi,j^r||i||j)
    g[j]=g[j].modPow(r);
    one[0] = (byte)(i+1);
    two[0]=(byte)(j+1);
    tempg = concatenateByte(g[j].toByteArray(), one);
    tempg = concatenateByte(tempg, two);
    byte[] v = xorArray(t, params.hash0.digest(tempg));
    list.add(v);
    }
  }

  byte[] w = xorArray(m, params.hash0.digest(t));

  //set ciphertext c = (U, [vi,1, vi,2, ..., vi,ai] 1<i<a, w)
  list.add(w);
  byte[] c =  BitUtility.toBytes( list );
  return c;
}

//Main decryption algorithm called from engineDoFinal(byte[],int,int)
public byte[] decryptBlock(byte[] c){
  //decompose ciphertext into (u, [v1, v2], w)
  ArrayList arrayList = (ArrayList) BitUtility.fromBytes( c );
  //u is the first object in the list
  AffinePoint u = (AffinePoint) arrayList.get( 0 );
 //w is the last object in the list
  byte[] w = (byte[]) arrayList.get(arrayList.size()-1);

  BilinearMap map = params.getMap();
```

```
Element[] g = new Element[publicKey.ai];

 AffinePoint cred;
 byte[] t = null;
 AffinePoint sumCred = null;
 byte[] v=null;
for(int i=0; i<publicKey.ai; i++){
   //there is ai number of v's in the list
   v = (byte[]) arrayList.get(i+1);
   for(int j=0; j<privateKey.getCred().size(); j++){
      cred=(AffinePoint)privateKey.getCred().get(j);
      if(j==0)
            sumCred=cred;
      else
         sumCred=params.getMap().getCurve().add(sumCred, cred);

   }
      g[i]=map.getPair(sumCred,u);

      //compute ti = vi,ji xor Hash2(gi,ji||i||ji)
      byte[] one = new byte[1];
      one[0] = (byte)publicKey.a;
      byte[] two = new byte[1];
      two[0] = (byte)(i+1);

      byte[] tempg = concatenateByte(g[i].toByteArray(), one);
      tempg = concatenateByte(tempg, two);

      t = xorArray(v, params.hash0.digest(tempg) );

      //compute m=w XOR Hash3(XOR all ti, i=1 to a)
      byte[] m = xorArray(w, params.hash0.digest(t));

      //compute U = Hash0(m||xor ti||polA).P
       byte[] tempH = concatenateByte(m, t);
      tempH=concatenateByte(tempH, publicKey.getFullPol());

      //get p
      AffinePoint p = params.getP();
      BigInteger tempH2 =new BigInteger(params.hash0.digest(tempH));
      AffinePoint u2 = map.getCurve().multiply(tempH2, p);
      //System.out.println("U2 in decrypt: "+u2.toString());
      if (u.equals(u2)){
         return m;
      }
```

```
    }
    return null;
}

protected AlgorithmParameters engineGetParameters() {
    return( null );
}
protected byte[] engineGetIV() {
    return( null );
}
protected int engineGetOutputSize( int param ) {
    return( -1 );
}
protected void engineSetMode( String str ) {
}

protected void engineSetPadding( String str ){
}
protected int engineGetBlockSize() {
    return( -1 );
}
public byte[] concatenateByte(byte[] a, byte[] b){
    byte[] r = new byte[a.length + b.length];
        System.arraycopy(a, 0, r, 0, a.length);
        System.arraycopy(b, 0, r, a.length, b.length);
        return r;
}
public byte [] xorArray(byte [] a, byte [] b) {
    int len = (b.length > a.length) ? b.length : a.length ;
    byte res[] = new byte[len];
    for( int i=0; i<len ; i++){
        if( i < b.length ){
            if( i < a.length)
                res[i] =(byte)(b[i] ^ a[i]);
            else
                res[i] =(byte)(b[i] ^ 0);
        }
        else
            res[i] = (byte) (0 ^ a[i]);
        }
    return res;
}

//used for debugging display purposes only
public static String byteToString(byte[] bytearray){
    String s="";
```

```
      for(int i=0; i<bytearray.length; i++){
        s+=( char )bytearray[ i ];
      }
        return s;
    }
}
```

## *PBCKey.java*

```
package PBC;

import java.security.Key;
/**
 *
 * @author Kathryn Garson
 * August 17, 2008
 *
 * class PBCKey
 *  Used as the main Key class for our encryption system, implemented in
 *  PBCPrivateKey and PBCPublicKey
 *
 */
public class PBCKey implements Key{

   public String getAlgorithm() {
      return( new String( "Policy Based Encryption" ) );
   }

   public byte[] getEncoded() {
      return( null );
   }

   public String getFormat() {
      return( null );
   }
}
```

### *PBCKeyPairGenerator.java*

```
package PBC;
import java.security.KeyPair;
import java.security.KeyPairGeneratorSpi;
import java.security.spec.AlgorithmParameterSpec;
import java.security.SecureRandom;
import java.util.ArrayList;
/**
 *
 * @author Kathryn Garson
 * August 17, 2008
 *
 * class PBCKeyPairGenerator
 *  generates and returns a key pair containing the public and private key
 *
 */
public class PBCKeyPairGenerator extends KeyPairGeneratorSpi{
   PBCKeyParameters params;
   public void initialize( AlgorithmParameterSpec params ) {
      if( params == null ) {
         throw new NullPointerException( "params cannot be null" );
      }

      this.params = (PBCKeyParameters) params;
   }
   public void initialize(PBCKeyParameters params){
      this.params=params;
   }
   public void initialize( int keysize, SecureRandom random ) {

   }
   public KeyPair generateKeyPair() {
      ArrayList computed = params.getComputedPol();
      byte[] pol = params.getPublicKey();
      ArrayList credList = params.getPrivateKey();
      PBCPublicKey publicKey = new PBCPublicKey(pol, computed);
      PBCPrivateKey privateKey = new PBCPrivateKey(credList);
      return (new KeyPair( publicKey, privateKey));
   }
}
```

## *PBCKeyParameters.java*

```
package PBC;

import java.math.BigInteger;
import java.security.MessageDigest;
import java.security.spec.AlgorithmParameterSpec;
import java.util.ArrayList;
import nuim.cs.crypto.bilinear.*;
import nuim.cs.crypto.blitz.point.*;
import nuim.cs.crypto.blitz.field.Element;
/**
 *
 * @author Kathryn Garson
 * August 17, 2008
 *
 * class PBCKeyParameters
 *  creates policy and credentials
 *
 */
public class PBCKeyParameters implements AlgorithmParameterSpec{
   byte[] policy;
   ArrayList computedPolicy;
   ArrayList credList;

   //Computes public key (policy)
   public PBCKeyParameters(MessageDigest hash, AffinePoint R, BilinearMap e) {

      String doc = new String("Doctor");
      String time = new String("Emergency");
       this.policy = concatenateByte(doc.getBytes(), time.getBytes());

      ArrayList computedFirstAnd = new ArrayList();
      ArrayList temp = new ArrayList();
      this.computedPolicy = new ArrayList();

      byte[] tempA = hash.digest(doc.getBytes());
      AffinePoint A = e.mapToPoint(new BigInteger(1, tempA));
      Element temporary = e.getPair(A, R);
      computedFirstAnd.add(temporary);

      tempA = hash.digest(time.getBytes());
      A = e.mapToPoint(new BigInteger(1, tempA));
      temporary = e.getPair(A, R);
      computedFirstAnd.add(temporary);
```

```
    temp.add(computedFirstAnd);
    this.computedPolicy.add(temp);
  }

  //Computes private key (credentials) and calls first constructor to compute public key
   public PBCKeyParameters( MessageDigest hash,
     AffinePoint R, BilinearMap e, BigInteger masterKey, String assertions )
   {
    this(hash, R, e);

    String[] splitAssertions = assertions.split(",");
    AffinePoint cred;

    this.credList = new ArrayList();
    byte[] raw;
    AffinePoint hashvalue;

    for(int i=0; i<splitAssertions.length; i++){
        raw=hash.digest(splitAssertions[i].getBytes());
        hashvalue=e.mapToPoint(new BigInteger(1, raw));
        cred = e.getCurve().multiply(masterKey, hashvalue);
        this.credList.add(cred);
    }

  }

//Returns the public key (policy) as a byte array
public byte[] getPublicKey(){
      return policy;
  }

//Returns the public key (policy) as an arrayList with each condition pre-computed
public ArrayList getComputedPol(){
   return computedPolicy;
}

//Returns the private key (credentials) in an arrayList
public ArrayList getPrivateKey(){
      return credList;
  }

//Helper function to concatenate byte arrays
public byte[] concatenateByte(byte[] a, byte[] b){
      byte[] r = new byte[a.length + b.length];
         System.arraycopy(a, 0, r, 0, a.length);
```

```
        System.arraycopy(b, 0, r, a.length, b.length);
        return r;
    }
}
```

***PBCPrivateKey.java***

```
package PBC;

import java.security.PrivateKey;
import java.util.ArrayList;
/**
 *
 * @author Kathryn Garson
 * August 17, 2008
 *
 * class PBCPrivateKey
 *  contains the private key credentials in an arraylist
 *
 */
public class PBCPrivateKey extends PBCKey implements PrivateKey{
   ArrayList credList;
   public PBCPrivateKey(){
   }
   public PBCPrivateKey(ArrayList credList){
     this.credList=credList;
   }
   public ArrayList getCred(){
      return credList;
   }
}
```

## *PBCProvider.java*

```
package PBC;

import java.security.Provider;
/**
 *
 * @author Kathryn Garson
 * August 17, 2008
 *
 * class Provider
 *  creates the Provider "PBCProvider" for our implementation
 *
 */
public class PBCProvider extends Provider{
   public static final String PBC = new String( "PBC" );

   public PBCProvider() {
      super( "PBCProvider", 1.0, "Policy-Based Encryption System" );

      clear();
      put( "KeyPairGenerator." + PBC, PBCKeyPairGenerator.class.getName() );
      put( "Cipher." + PBC, PBCCipher.class.getName() );
   }
}
```

## *PBCPublicKey.java*

```
package PBC;
import java.security.PublicKey;
import java.util.ArrayList;

import nuim.cs.crypto.blitz.field.Element;

/**
 *
 * @author Kathryn Garson
 * August 17, 2008
 *
 * class PBCPublicKey
 *  contains the policy in byte array form and pre-computed form stored as an arrayList
 *
 */
public class PBCPublicKey extends PBCKey implements PublicKey  {
   byte[] policy;
   ArrayList computedPolicy;
   int size;
   int a;
   int ai;
   int aij;

   public PBCPublicKey(byte[] policy, ArrayList computedPolicy){
       this.policy=policy;
       this.computedPolicy=computedPolicy;
       this.a=computedPolicy.size();
       ArrayList temp = (ArrayList) computedPolicy.get(0);
       this.ai=temp.size();
       temp = (ArrayList) temp.get(0);
       this.size=Math.max(temp.size(),this.ai);
       this.aij=temp.size();
   }

   public byte[] getFullPol(){
      return policy;
   }
   public Element getPol(int i, int j, int k){
      ArrayList andOrList = (ArrayList) computedPolicy.get(0);
      ArrayList andList = (ArrayList) andOrList.get(j);

      Element policyConstraint = (Element) andList.get(k);
```

```
        return policyConstraint;
    }
}
```

### *PBCSystemParameters.java*

```
package PBC;

import java.math.BigInteger;
import java.security.MessageDigest;
import java.security.spec.AlgorithmParameterSpec;

import nuim.cs.crypto.bilinear.*;
import nuim.cs.crypto.blitz.point.*;
/**
 *
 * @author Kathryn Garson
 * August 17, 2008
 *
 * class PBCSystemParameters
 *  computes the system parameters
 *
 */
public class PBCSystemParameters implements AlgorithmParameterSpec{
   BilinearMap e; //map
   AffinePoint randomGeneratorP; //P
   AffinePoint R; //public key
   MessageDigest hash0;

   public PBCSystemParameters(MessageDigest hash, BilinearMap e, BigInteger
masterKeyS){
      this.e = e;
      //chose a random P generator
      randomGeneratorP = e.getCurve().randomPoint();
      //create the public key
      R = e.getCurve().multiply( masterKeyS, randomGeneratorP );
      if(hash == null) {
         throw new NullPointerException( "Hash cannot be null" );
      }
      this.hash0=hash;
   }

   //Return the random generator P
   public AffinePoint getP(){
      return randomGeneratorP;
   }
   //Return the bilinear map
   public  BilinearMap getMap(){
      return e;
```

```
    }
    //Return the public key R
    public AffinePoint getR(){
        return R;
    }
    //Return the hash function used
    public MessageDigest getHash(){
        return hash0;
    }
    //Set the hash function
    public void setHash(MessageDigest hash){
        if(hash == null) {
            throw new NullPointerException( "Hash cannot be null" );
        }
        this.hash0=hash;
    }
}
```

### *Test.java*

```
package PBC;

import javax.crypto.Cipher;
import java.math.BigInteger;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.security.KeyPair;
import java.security.PublicKey;
import java.security.PrivateKey;
import java.security.SecureRandom;
import java.security.Provider;
import java.security.Security;
import javax.crypto.NoSuchPaddingException;
import java.security.InvalidAlgorithmParameterException;
import javax.crypto.BadPaddingException;
import java.security.InvalidKeyException;
import javax.crypto.IllegalBlockSizeException;
import nuim.cs.crypto.bilinear.*;
/**
 *
 * @author Kathryn Garson
 * August 17, 2008
 *
 * class Test
 *  demonstrates how to use the encryption methods
 *
 */
public class Test {

   public static void main(String[] args) {

      //Create the provider
      Provider pbc = new PBCProvider();
      Security.addProvider(pbc);

      //Create hash, map and masterkey
      MessageDigest hash = null;
      try {
         hash = MessageDigest.getInstance( "MD5" );
      }
      catch( NoSuchAlgorithmException nsae ) {

      }
```

```
        BilinearMap map= new ModifiedTatePairing();
        BigInteger masterKey = new BigInteger( map.getQ().bitLength() - 1, new
SecureRandom() );

        //Create the system parameters
        PBCSystemParameters parameters = new PBCSystemParameters(hash, map,
masterKey);

        //Create the key parameters
        String creds = "Doctor,Emergency";
        PBCKeyParameters keyParams = new
PBCKeyParameters(parameters.getHash(),parameters.getR(), map, masterKey, creds );
        System.out.println("Key parameters generated: policies and credentials created");

        //Create and initialize the keypairgenerator
        PBCKeyPairGenerator kpg = new PBCKeyPairGenerator();
        kpg.initialize( keyParams );
        System.out.println("key pair generator created and initialized");

        //Generate key pairs
        KeyPair keys = kpg.generateKeyPair();
        PublicKey policy=keys.getPublic();
        PrivateKey credList = keys.getPrivate();
        System.out.println("Keys generated");

        //Get the cipher instance for PBCProvider
        Cipher cipher = null;
        try {
          cipher=cipher.getInstance(PBCProvider.PBC);
        }catch(NoSuchAlgorithmException e1){
          System.out.println("No such algorithm excepteion: "+e1);
        }catch( NoSuchPaddingException e2 ) {
          System.out.println("No Such Padding Exception: "+e2);
        }

        //Initialize the cipher for encryption
        try{
          cipher.init(Cipher.ENCRYPT_MODE,policy,parameters,new SecureRandom());
        }catch(InvalidKeyException e3){
          System.out.println("Invalid Key Exception: "+e3);
        }catch(InvalidAlgorithmParameterException e4 ){
          System.out.println("Invalid Algorithm Parameter Exception: "+e4);
        }

        //Encrypt a message and print the result
        String message = "eighjfaskdiuhewajbajebr";
```

```
        System.out.println("Message to encrypt: "+message);
        byte[] ciphertext=null;
        try{
           ciphertext =cipher.doFinal(message.getBytes());
        }catch( IllegalBlockSizeException e5 ) {
           System.out.println("Illegal Block Size Exception: "+e5);
        }
        catch( BadPaddingException e6 ) {
           System.out.println("Bad Padding Exception: "+e6);
        }
        System.out.println("ciphertext: "+ciphertext);

        //Initliaze the cipher for decryption
        try{
           cipher.init(Cipher.DECRYPT_MODE, credList, parameters, new
SecureRandom());
        }catch(InvalidKeyException e7){
           System.out.println("Invalid Key Exception: "+e7);
        }catch(InvalidAlgorithmParameterException e8){
           System.out.println("Invalid Algorithm Parameter Exception: "+e8);
        }

        //Decrypt the ciphertext and print out the result
        byte[] plaintext =null;
        try{
           plaintext=cipher.doFinal(ciphertext);
        }catch( IllegalBlockSizeException e9) {
           System.out.println("Illegal Block Size Exception: "+e9);
        }
        catch( BadPaddingException e10) {
           System.out.println("Bad Padding Exception: "+e10);
        }
        System.out.println("Plaintext recovered: "+byteToString(plaintext));

    }//end main

    //Helper method to print out message contained in a byte array
    public static String byteToString(byte[] bytearray){
        String s="";
        for(int i=0; i<bytearray.length; i++){
          s+=( char )bytearray[ i ];
        }
          return s;
    }
}
```