ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCHOOL OF ENGINEERING AND ARCHITECTURE

Department of Computer Science and Engineering - DISI
Laurea Magistrale (Second Cycle Degree) in Computer Engineering

PROJECT WORK

for

INFORMATION SECURITY M

titolo titolo titolo titolo titolo

CANDIDATE
Giada Martina Stivala

SUPERVISOR
Chiar.ma Prof.ssa Rebecca Montanari

Academic Year 2017/2018

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Contents

Abstract **ii**

1 Introduction **1**

2 Sticky Policies in literature **3**
2.1 Hybrid Cryptosystem 4
2.2 Attribute-Based Access Control 4
2.3 Identity-Based Encryption 5
2.4 Proxy Re-Encryption 6
2.5 Existing implementations and libraries 7
2.5.1 Java Pairing Based Cryptography Library for IBE . . 7

3 A simple implementation **11**
3.1 Program flow and architecture 12
3.1.1 Android Client Implementation 13
3.1.2 Java Server Implementation 15
3.2 Structure of an XML policy 16
3.2.1 XML Validation and Parsing 16
3.3 About cryptography 17
3.3.1 X509Certificates 17
3.3.2 Message Digests 18
3.3.3 Symmetric Key Generation and Encryption 19
3.4 Performance 22

Bibliography **23**

A XSD Grammar for Sticky Policies **25**

B Java Pairing Based Cryptography Library **27**

Chapter 1

Introduction

Personal data increasingly fuel Internet applications development and spread, especially since the birth and diffusion of Big Data. Users are often unaware of the gathering, processing and storage of their data, mostly because of the obfuscation behind license agreements and the technical notions necessary to understand these processes. Data may also be moved across country borders, to be processed under different laws and regulations, with processing systems so complex to ultimately make it impossible to understand which service provider had the right to process what.

Leaving out the risks originating for accidental data disclosure by such applications, for example due to security breaches, users may wish to increase their data protection, especially in fields as medical care and financial services. More specifically, we would like a tool to selectively reduce data disclosure, preventing unwanted usage from third parties.

Current solutions include state laws about data usage and protection, business frameworks and service-level agreements, which prove to be inefficient and ineffective. Many research studies have thus advanced the suggestion of a different tool, called *Sticky Policies*, to ensure data protection and control its disclosure. *Sticky Policies* are essentially machine-readable metadata which specify the correct usage of the data they travel with [8]: through encryption, they prevent policy non-compliant use.

In this study, we focus on the mobile environment, and in particular on smartphones. A few solutions already exist, but often they fail to protect personal data with the desired granularity. Moreover, services and application run in remote cloud systems, and data is stored in distributed systems managed through complex automated procedures. Due also to the difficulty in complying with strict Data Regulation laws (e.g. GDPR), many companies are *outsourcing* this task through Single Sign-On procedures: personal data is gathered and processed by large and structured organizations, and external services rely on simple APIs for user authentication.

In general, any communication via smartphone flows through a server or a cloud and thus it would be more realistic to implement a solution which integrates with an open-source app or service adding a layer of data protection. At a high level, the data owner and sender would be able to select any condition to be verified before the recipient could access that data on the same app. In case of non-compliance, simply the data wouldn't be disclosed, while in case the access was granted, it would be possible to prevent illegal data sharing outside the app.

We would like to provide a proof-of-concept solution aimed, in particular, at Android devices. This context is exceptionally varied and quickly evolving, features which pose a limit to the extensiveness of the proposed solution. Other limitations derive from the lack of free, open-source implementations of the studied cryptographic schemes, and of open-source mobile applications.

In this project work, we start by analysing the most recent and relevant proposals about this matter, comparing different technical approaches and existing solutions. Then, we introduce a solution of our own, which tackles this issue within a limited scope of action. Finally, we present our conclusions about this work.

Chapter 2

Sticky Policies in literature

To implement *Sticky Policies* we first focus on the context for application. In a simple use-case, user Alice wants to share some personal data with user Bob through an application, which we will call *Service Provider*. We can generalize this situation for all those cases in which a service provider is fed with some personal data in order to supply the data owner with a service. To protect her privacy, Alice will specify which entities are allowed to use her data, and for which purposes through a *Sticky Policy*. The policy will be attached to the data so as to stick with them persistently, and the data will be obfuscated to prevent unauthorized access.

Most of the suggested solutions regarding this issue share a similar low-level architecture. Thus, we can recognize three basic common entities, which are essential to the development of *Sticky Policies*. They are the following:

- The *Data Owner* has a finite collection of data, which she wants to protect through fine-grained policies.

- A trusted entity which is in charge of securely storing data and generating encryption keys.

- A *Service Provider* should be considered as third party which requests data usage.

The role and implementation of the trusted entity changes according to the chosen encryption scheme and protocol.

We will now examine the main solutions available in literature, and consider their main advantages and disadvantages. It is worth saying beforehand that every solution cannot overlook the trust in the chosen third party: mainly, because once the data is decrypted it can be shared by the third party without any concern; secondly, due to the need of proving the remote hardware machines to be reliable (i.e. it always behaves the way it should, for the intended purpose). This latter issue has been dealt with through the use of Trusted Platform Modules [5].

2.1 Hybrid Cryptosystem

In this scenario, we use both asymmetric and symmetric cryptography to achieve the required *stickyness* of the policies. In a possible use-case, the service provider receives data, encrypted with a symmetric one-time-use key K, together with the policy and K, both encrypted with the public key of the *Trust Authority* and signed by the user. The service provider will interact with the Trust Authority to prove its reliability, eventually receiving the symmetric key K.

The *Trusted Authority* or *Policy Enforcement Point* always mediates the data exchange. It is a semi-trusted third party which checks the compliance of the service provider with the specified policies before releasing the symmetric keu K. A formal protocol for message exchange is suggested in [8]: `Policy, Enc(PubTA, K||h(Policy)), Sig(PrivUser, Enc(PubTA, K||h(Policy))), Enc(K, PII)`. This ensures that the policy always sticks to its data, and its integrity can be verified through a secure hashing function. Moreover, the combined usage of TA's public key and the data owner's private key ensure both confidentiality and authenticity.

This implementation relies heavily on the Public Key Infrastructure, and requires procedures for management and verification of X.509 certificates. For what concerns the PEP, it must be always reachable from the internet and, together with the Certification Authority, may be a target for attacks: first, it constitutes a *single point of failure*; additionally, if compromised, it could infect the data owner (for example through phishing attacks) or act as a man in the middle, decrypting `PII`.

In both cases, this solution proves to be computationally heavy and it does not solve the main issue of trusting the service provider not to illegitimately share data.

2.2 Attribute-Based Access Control

The Attribute-Based Access Control (ABAC) paradigm well fits our environment: by describing users and objects through a set of attributes, it allows fine-grained policy specifications for data protection. However, it requires a precise definition of the descriptive attributes and the implementation of the architecture required to process and enforce policies.

The XACML standard [11] provides a valid reference by defining not only the attributes and structure for rules, but also the components in the architecture. For the sake of simplicity, suppose we need only the *Policy Enforcement Point* from the whole XACML standard implementation, since we assume that the data owner has a client-side module to generate the required XML policies. The data owner trusts the *Policy Enforcement Point* to be both a secure storage for its personal data and to evaluate correctly the reliability and compliance of the service provider.

This solution allows a better data description in terms of granularity, and it is also more efficient once the architecture has been implemented; nonetheless, it shows the same weaknesses as the Hybrid Cryptosystem. Additionally, the effort required for architecture implementation and rule generation should not be underestimated. It is also remarkable that the mobile environment evolves quickly and is exceptionally fragmented and varied, conditions which make it more difficult to establish fixed descriptive attributes.

In this context, we can also consider Ciphertext-Policy Attribute-Based Encryption [2]. This solution does not require an intermediate entity to evaluate the policies before forwarding sensitive data to the recipient, because this assessment is embedded in the cryptographic scheme so that no unauthorized individual can decrypt it. In particular, the data owner chooses a set of attributes defining an *access tree*, the structure used at a lower level to bind attributes to the ciphertext and ensure that only individuals that satisfy the access tree can decrypt the obfuscated text.

This solution requires an available trusted entity for key generation, but does not strictly need a *Policy Enforcement Point*. It is possible to leverage on a secure storage to ease access from service providers, even though the level of trust required from this third entity is low due to data being encrypted by the data owner. As highlighted by Tang [12], a drawback of CP-ABE occurs when a private key is compromised and it is necessary to issue a new one: there is a non-negligible amount of risk related to the possibility for a potential attacker to decrypt all the ciphertexts associated with the attribute set of the compromised key.

2.3 Identity-Based Encryption

The Identity-Based Encryption (IBE) paradigm proposed by Shamir [10] can be purposely used to realize *Sticky Policies* [7].

IBE is an asymmetric cryptographic scheme which eliminates the need of the Public Key Infrastructure together with X.509 certificates, requiring only a trusted entity to generate private keys when needed. The public key should be an identifying, non-repudiable attribute, e.g. the email address: when an encrypted text is received, the recipient asks the key-generation centre to issue the corresponding private key, thus being able to decrypt the text.

In [7], Mont describes a cryptographic scheme for *Sticky Policies* based on IBE and coupled with a Trusted Platform Module. The encryption key is a XML document containing the formalization of the policy, thus reaching the desired *stickyness* for *Sticky Policies*. Any tampering or policy non-compliance will prevent the Trust Authority from generating the correct decryption key. Additionally, the Trust Authority will check the reliability of the requester before issuing the decryption key.

Finally, as shown by Shamir [9], Mont suggests to increase the security level through a *threshold scheme*, involving several Trust Authorities for key issue. In a *(k,n)* threshold scheme, the key is divided into $2k-1$ parts, and at least k pieces are necessary to encrypt or decrypt: as a disadvantage, it proves to be less efficient than a simple IBE scheme.

Tang [12] suggests a similar solution, in which the key is a string containing the identity of the recipient concatenated with some attributes or constraints, e.g. time stamps, so to make IBE finer-grained.

Both of the approaches require a new key generation every time the policy is changed, which could be inefficient in contexts as mobile devices communication; moreover, even if they do not directly use attributes in the encryption, there is still the need of a standard definition. When using Mont's solution, an XML grammar specification is necessary to avoid generating different keys for equal policies improperly written, while in Tang's case a specification should be issued to determine which attributes to use and their order.

2.4 Proxy Re-Encryption

The Proxy Re-Encryption scheme [4] can also be taken into consideration as an implementation for *Sticky Policies*, and is particularly suited to the mobile environment.

Communication and data sharing through mobile devices is supported by remote servers instead of happening point-to-point. In this context, the remote server is the Proxy, which re-encrypts data from the sender Alice's signature to the receiver Bob's. The server can be untrusted, as the scheme never produces plain text and is unidirectional and resistant to collusion attacks. To implement *Sticky Policies*, Alice sends a policy with the encrypted data: a *Policy Enforcement Point* evaluates Bob's policy compliance, and data is re-encrypted and forwarded with the re-encryption key generated by the server.

The Proxy Re-Encryption scheme can be implemented either through Identity Based Encryption or Public Key Encryption, and in this latter case provides the benefit of Alice encrypting only for the Proxy, which will then be in charge of re-encrypting with a different key for every recipient.

The Proxy Re-Encryption scheme shows some advantages with respect to the other schemes mentioned: key and policy updating are performed by informing the proxy, and no vulnerability affects previous ciphertexts encrypted with those keys or conditions thanks to the re-encryption. Disadvantages of PRE are highlighted in [12]: even though the scheme requires a lower level of security, a compromised Proxy could generate re-encryption keys for any receiver, potentially exposing personal data to any of its recipients. To address this issue, Tang presents the Type-based PRE, which introduces the notion of *data categories*, as an additional input parameter to the re-encryption key generation. With TPRE, compromise of a re-encryption key does not affect keys with a different type, and key revocation is dealt with just creating a new re-encryption key relating to a new data type (which includes the previous one).

2.5 Existing implementations and libraries

Nearly each of the possibilities considered in chapter 2 has a dedicated implementation. For Cyphertext-Policy Attribute-Based Encryption there is the `cpabe` toolkit [1], available in the C language. This library provides an encryption scheme such that each private key is associated with a set of attributes rather than with the identity of the data owner. Attributes can be provided as strings from standard input or from a file; moreover, it is possible to combine more than one attribute or rule through predefined operators as 'and', 'or', '>', '<'.

2.5.1 Java Pairing Based Cryptography Library for IBE

For Identity-Based Encryption we can rely on the Java Pairing Based Cryptography Library [3]. Private keys are generated from identities alone as well as combined with attributes describing the authorized audience, and it is possible both do encrypt and sign data. Both [3] and [1] depend on the Pairing-Based Cryptography Library [6], developed in the C language.

Listing 2.1: Java mock class for IBE implementation

```java
// Setup
AsymmetricCipherKeyPair keyPair = engine.setup(64, 3);
// KeyGen
Element[] ids = engine.map(keyPair.getPublic(), "angelo", "de caro", "unisa");
CipherParameters sk0 = engine.keyGen(keyPair, ids[0]);
CipherParameters sk01 = engine.keyGen(keyPair, ids[0], ids[1]);
CipherParameters sk012 = engine.keyGen(keyPair, ids[0], ids[1], ids[2]);
// Encryption
byte[][] ciphertext0 = engine.encaps(keyPair.getPublic(), ids[0]);
byte[][] ciphertext01 = engine.encaps(keyPair.getPublic(), ids[0], ids[1]);
byte[][] ciphertext012 = engine.encaps(keyPair.getPublic(), ids[0], ids[1], ids[2]);
// Decrypt
byte[] cleartext0 = engine.decaps(sk0, ciphertext0[1]);
System.out.println(new String(cleartext0) + "");
```

As shown in Listing 2.1, the private key is generated providing several strings in place of the data owner's identity. Different cyphertexts are produced using different attributes as a key, and the same `CipherParameters` are needed to decrypt correctly.

The functions `encaps` and `decaps` provide encryption and decryption mechanisms. After practical experiments, it results that [3] is a proof-of-concept implementation and it is thus not suited for actual use. The main reasons behind this lay in the implementation of the aforementioned functions: in fact, the `encaps` function does not take any plaintext in input, but it is generated inside its body by the function `process()`. As we can see from Listing 2.2, this function calls `processBlock` supplying as input an empty byte array instead of an actual input.

Listing 2.2: Excerpt from PairingKeyEncapsulationMechanism class

```
package it.unisa.dia.gas.crypto.jpbc.kem;

import {...}

public abstract class PairingKeyEncapsulationMechanism extends
    PairingAsymmetricBlockCipher implements
    KeyEncapsulationMechanism {

     private static byte[] EMPTY = new byte[0];

         // some other functions...

         public byte[] processBlock(byte[] in) throws
    InvalidCipherTextException {
         return processBlock(in, in.length, 0);
     }

         public byte[] process() throws
    InvalidCipherTextException {
         return processBlock(EMPTY, 0, 0);
     }
}
```

It is possible to modify the source code by opening a file, or supplying a run-time byte array containing the information to encrypt, and calling `processBlock` purposely:

```
return processBlock(dataArray, 0, dataArray.length);
```

To obtain the encrypted text it is also necessary to modify the last statement in the `processBlock` function called by `process`. In fact, as shown in the documentation for class `PairingAsymmetricBlockCipher`, the function `byte[] processBlock(byte[] in, int inOff, int inLen)` takes as second and third arguments the offset and the length of data, thus requiring an invocation like the following:

```
return processBlock(in, 0, in.length);
```

The complete content of the mentioned classes is available in Appendix B.

After performing data encryption, though, the `Assert` statements to verify correct decryption fail, which leads us to think that this is only a proof-of-concept implementation. For this reason, and due to the unsuitableness of the available [1] and [6] in the chosen context for this project, we have thus decided to proceed to the implementation of Sticky Policies with a hybrid cryptosystem.

Chapter 3

A simple implementation

First, we set up the necessary entities to implement *Sticky Policies*: an Android client and a service provider. The client was developed using Android Studio and emulated through the Android Emulator with a Nexus 5X device running Android 7.0 and API level 24. The Trusted Authority was developed using Eclipse and run on Apache Tomcat 8; the communication was implemented through HTTP protocol.

The app was also tested in a Samsung tablet (SM-T230) running Android 4.4.2 and API level 19, to test performance on a real device with lower level API and operating system, together with a different screen dimension.

In this solution, *Sticky Policies* are realized via XML files and paired with personal data through the combination of symmetric and asymmetric cryptography. This approach is presented in [8], and an example XML policy was created taking as a reference the one presented in the same paper. Following this specification, we present a protocol for the communication of two Android clients, called for simplicity Alice and Bob.

Alice generates an XML policy to regulate data access, encrypting it with a symmetric key generated locally. The policy and the keys are then encrypted with the Trusted Authority's public key and signed by Alice. Purposely, Alice should obtain the public key of the Trusted Authority and also a key pair for herself: in our solution, we use self-signed X509 Certificates generated by combining the Java Cryptography Architecture and the Bouncy Castle cryptography APIs for Java. Alice thus contacts the Trusted Authority to obtain its public key, and shares her own if the Trusted Authority is not the issuer of the certificate itself.

Bob obtained Alice's encrypted data and an attached policy in clear text, together with other encrypted data to guarantee integrity, confidentiality and non-refusal of the policy and data. To decrypt Alice's data, he must follow these steps:

- Bob asks the Trust Authority to release the symmetric key, presenting the encrypted data signed by Alice.

- The Trust Authority evaluates the policy and Bob's reliability, submitting some challenges for him to complete.

- If trusted, Bob receives the symmetric key to decrypt Alice's personal data.

Data is exchanged through POST requests over a channel which is assumed to be secure.

Once the Trust Authority receives a request from the client, it checks the correct specification of the policy before proceeding to decrypt and verify the payload received. This operation is performed by a server-side parser which matches the XML file with a standard XSD grammar. In case of errors, no symmetric key is released.

The specification of the XSD grammar can be found in Appendix A.

3.1 Program flow and architecture

implementation of the protocol, insert some images add error management lifecycle considerations check manifest permissions: storage??

We start by analysing the application flow before that data is shared, given that Alice has installed an instance of the StickyPolicyApp.

The data owner can either share a simple text message (shown in Figure 3.1) or a regular file inside his phone (for example, an image, in Figure 3.2), chosen through a picker. Consequently, the underlying application generates a fitting policy depending on the data shared and on the owner itself, which is then passed on to the next `Activity` for encryption and sharing. In this solution, for the sake of simplicity, the generated policy contains mock data, except for the X509 Certificate serial number and the data type. Possible future works include adding an interactive policy generator to allow the user to define sharing constraints.

A certificate exchange is then initiated between the client and the server: this to perform both signature and encryption inside the app, and to allow the Trusted Authority to run security checks on data, when requested access by third parties. Subsequently, the policy and data are processed following the specification presented in [7]: the data is encrypted with a one-time-use symmetric key generated at the moment, and together with a digest of the policy they are first encrypted with the public key of the Trusted Authority, then signed with the data owner's private key.

It is now possible for Alice to safely share her data, sending it to Bob. Data is transferred inside the body of POST requests in JSON form. To safely and efficiently perform data serialization and deserialization we used the `com.google.code.gson` library and implemented an ad-hoc private class to hold both policy and encrypted data in a single JSON Object. For this

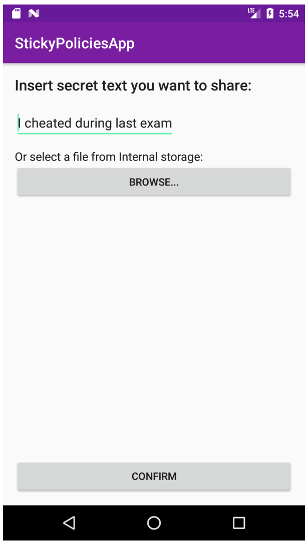

Figure 3.1: Sharing secret text

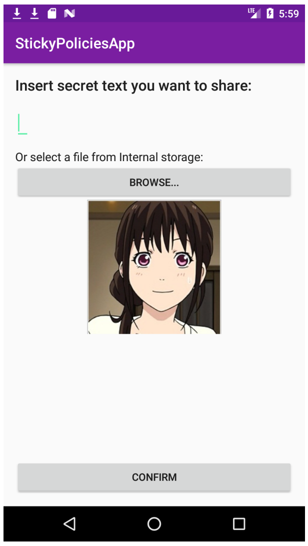

Figure 3.2: Sharing a personal image

purpose, we tested also the libraries `org.json` and `org.jabsorb`, which though proved to be unsuitable and less efficient.

Let's now consider the StickyPoliciesApp to be running on Bob's device, and to have just received a secret message from Alice. Bob forwards to the Trusted Authority the plaintext policy together with the signed key and digest, removing the encrypted personal data from the original payload. At this point, the Trusted Authority may check Bob's reliability and submit a few challenges to prove it, eventually disclosing the symmetric key to access Alice's secret message.

Both the communication and the cryptographic primitives are performed inside `AsyncTasks` to keep the main UI thread separated and responsive.

3.1.1 Android Client Implementation

The StickyPoliciesApp consists of a few `Activities`. The launcher allows both to send private data and to access received messages, but can be extended to include new functionalities, as the aforementioned policy editing.

Navigation within the activities is made through explicit `Intents`: as specified within the Android Security guidelines, implicit `Intents` may introduce security hazards as it is not known which service will respond to the intent, and the user has no control over it.

The first activity is in charge of data sharing: once the data type of the shared information is determined, it requests from the `CryptoUtils` class the data owner's certificate to retrieve its serial number, edits the sticky policy so that it matches both personal data type and certificate SN; finally, it creates a `Bundle` to pass these objects to the following activity, invoked explicitly.

The `PolicyClient` class is in charge of communication operations: it performs a certificate exchange with the Trusted Authority and it encrypts data, sending them to Bob. These operations are performed via dedicated `AsyncTask`s which are started in the `onResume()` method, so that the main UI thread is build without delay. Both tasks simply prepare data for the HTTP request, invoke a library function to handle it at a lower level and parse the results. For example, this is how the data owner's certificate is shared:

Listing 3.1: Excerpt from PolicyClient class

```
@Override
protected byte[] doInBackground(URL... urls) {
        String responseBody = null;
        URL searchUrl = urls[0];
        try {
                responseBody = NetworkUtils.
    getResponseFromHttpUrl(searchUrl, "POST", postData,
    applicationJsonContentType);
        } catch (IOException e) {
                e.printStackTrace();
                Log.d(TAG, "Error when sending encrypted POST: "
     + e.getMessage());
        }
        Gson gson = new Gson();
        return gson.fromJson(responseBody, byte[].class);
}
```

where `NetworkUtils` is a public class exposing `static` methods for network management. As for data encryption, operations are performed following the specification in [7], and an excerpt of the code is shown in Listing 3.2.

Listing 3.2: Excerpt from PolicyClient class

```
byte[] encrPiiAndIV = new byte[0];
byte[] keyAndHashEncrypted = new byte[0];
byte[] signedEncrKeyAndHash = new byte[0];
try {
        // 1) obtain policy: obtained in Bundle
        // 2) generate symmetric disposable encryption key
        byte[] encodedSymmetricKey = CryptoUtils.
    generateSymmetricRandomKey();
        byte[] initializationVec = CryptoUtils.generateSecureIV
    ();
        // 3) encrypt PII
        byte[] encryptedPii = CryptoUtils.encryptSymmetric(
    encodedSymmetricKey, initializationVec, pii);
        // 4) hash policy
        byte[] policyDigest = CryptoUtils.calculateDigest(policy
    .getBytes(Charset.forName("UTF-8")));
        // 5) append policy and digest, encrypt with PubTa
```

```java
        ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream( );
        byteArrayOutputStream.write(encodedSymmetricKey);
        byteArrayOutputStream.write(policyDigest);
        keyAndHashEncrypted = CryptoUtils.encryptAsymmetric(taPublicKey, byteArrayOutputStream.toByteArray());
        // 6) sign
        signedEncrKeyAndHash = CryptoUtils.sign(keyAndHashEncrypted);

        encrPiiAndIV = new byte[initializationVec.length + encryptedPii.length];
        System.arraycopy(encryptedPii, 0, encrPiiAndIV, 0, encryptedPii.length);
        System.arraycopy(initializationVec, 0, encrPiiAndIV, encryptedPii.length, initializationVec.length);
} catch (Exception e) {
        e.printStackTrace();
        mSearchResultsTextView.setText("Sorry, some errors happened while encrypting your data.\nDon't worry, your privacy is still safe!\nPlease try again later.");
}
```

3.1.2 Java Server Implementation

We have realized the Trusted Authority as a web server exposing two services, reachable via the corresponding servlets hosted in the servlet container Apache Tomcat.

The first one is in charge of sharing the Trusted Authority's `X509Certificate` in `PEM` format, and it also manages the data owners which want to register to the TA. Also in this case, the certificates are to be sent in `PEM` format. This state is kept server-side inside the `ServletContext`, which is bound to the servlet's life cycle rather to the single user or session.

The second service answers to data access requests: the servlet receives JSON Objects containing plaintext policies together with signed key and digest, and proceed checking the integrity of the policy, verifying the signature and decrypting the obtained data, to retrieve the symmetric key. If any of these procedures fails, the entire process is stopped and the access request is refused with an error code corresponding to the cause of the error (e.g. `403 Forbidden` in case the signature verification is not correct). In a future improved version of this app, the symmetric key could be encrypted with Bob's public key before sharing, requiring Bob to register at the Trusted Authority first. Other possible improvements include the implementation of policy-specified `actions`, as for example informing the data owner when data access is granted.

As it is, our web server shows the same weaknesses we described in the previous chapter; moreover, additional weaknesses are introduced by the communication with a potential malicious client. In other words, no check is performed on the payload (content-type, dimension, user-agent) and the client is considered to be always trusted.

3.2 Structure of an XML policy

The policy file is constructed by the data owner in order to specify which set of users can access her data. Policies are shared over the internet as strings, thus formats like XML or JSON represent the best choice in terms of interoperability and ease of use. Choosing the XML format over a new, custom implementation for Sticky Policies enables the use of well-known corroborated libraries and best practices, available due to XML widespread. Moreover, the document format is designed for data transfer and to be self-descriptive: its structure and content can easily be regulated through grammars and specifications.

Sticky Policies have been designed with a general and comprehensive structure, so as to allow broader usage than the one proposed in this solution. The main components of a Sticky Policy are the following:

- List of the Trusted Authorities in which Bob can ask for data access. They are specified by Alice and they include all the servers in which she registered her certificate.
- Details about the Data Owner, which are used server-side to retrieve the correct certificate and public key.
- One or more policy, specifying fine-grained constraints about the attached data. They may vary in target, time validity, requirements, etc.; the complete specification is available in Appendix A.

3.2.1 XML Validation and Parsing

Policies are parsed server side: this control step includes validation, i.e. checking both syntax and semantics of the document. In Java, it is possible to realize an XML parser through a `SAXParser`: this low-level implementation scans the whole document and for each start tag and end tag found fires and event, handled by a callback method to retrieve the content between the tags. While this allows to gain in efficiency and is suited also to parse large documents, it constitutes and ad-hoc tailored solution, unsuited for dynamic environments.

Document parsing is performed by a `ContentHandler`, in which callback methods are defined: we chose to implement them via the subclass `DefaultHandler` and its methods `startElement`, `characters`, `endElement`.

In our solution, we implemented also a `XMLErrorChecker` for error handling. This parser provides three different types of errors: fatal errors, errors and warnings, although by default only fatal errors are fired. In these cases, the parser cannot continue parsing and the method performs `System.exit(1)`, while nonfatal errors are generated when the document fails some validity constraints (e.g. invalid tag, tag not allowed...). We redefined the default behaviour by firing `SAXExceptions` also for nonfatal errors, due to the considerable importance of having a well-formed and valid document. These exceptions are then catched by the caller, at a higher level of the call stack, and handled by refusing the data disclosure request.

3.3 About cryptography

Cryptography constitutes a considerable part of the whole project. All cryptographic functions are realized as static, and accessible as if they were an external library both in the client and server. Both of them rely on the Java Cryptography Architecture together with Bouncy Castle APIs.

3.3.1 X509Certificates

The Bouncy Castle library provides functionalities to create self-signed X509 Certificates, chosen for the sake of simplicity. In a more realistic implementation, certificates should be issued by a Certification Authority, and the web server should additionally perform certificate verification (e.g. expiration and revocation). To construct the `X509Certificate`, we use 1024-bit asymmetric keys generated for the RSA algorithm, and the Bouncy Castle security provider. 2048-bit long keys could be more appropriate depending on the confidentiality of the information shared, on the mobile device and network possibilities.

Both the certificates are constructed following the same instructions. In case the Trusted Authority works also as a Certification Authority, then

```
BasicConstraints constraints = new BasicConstraints(true);
```

while for the mobile device we use a `false` parameter. Moreover, in both cases, certificates are created and handled as singleton objects, to prevent inappropriate certificate creation: the methods and fields to generate and store certificates are `private`, and their content can be retrieved through accessor methods, which also control the instance creation.

Among the Bouncy Castle library there is the LightCrypto implementation [13], available for phones or devices with limited computational power. It allows the creation of key pairs and certificates, but their interoperability with the Java Cryptography Architecture is limited so that a conventional certificate implementation is preferred.

Certificates are shared in PEM format using a `JcaPEMWriter` for serialization and a `CertificateFactory` for deserialization. This format has been preferred to ASN.1 as easier to send in a HTML request body, it being a `String` rather than a `byte[]`.

3.3.2 Message Digests

Computing message digests is an important part of this security protocol: first, it proves policy's integrity by matching the received digest with the one computed from the plaintext policy; second, it lowers the size of the data to be encrypted, giving it a fixed dimension of 32 bytes. In fact, policy files can potentially grow in size and, were the policy not to be hashed before asymmetric encryption, it could introduce a weakness within RSA's encryption method. If the overall dimension of the symmetric key and policy were greater than the dimension of the modulus for asymmetric encryption, either the dimension of the modulus would increase, or encryption would be performed after splitting the whole data into blocks of size lower than 1024 bits.

The cryptographic hash function used in this project is the SHA-256 function, and the implementation is the one provided by the class `java.security.MessageDigest`. To obtain a message digest it is necessary to follow a three-step procedure: first, an instance of `MessageDigest` is retrieved for the specific algorithm; then, it receives as input the text to hash and, finally, the `digest()` function is called.

Listing 3.3: Excerpt from CryptoUtilities class

```java
import java.security.MessageDigest;
import java.security.DigestException;
import java.security.NoSuchAlgorithmException;

        public static byte[] calculateDigest(byte[] text) throws DigestException {
            try {
                MessageDigest md = MessageDigest.getInstance(digestAlgorithm);
                md.update(text);
                return md.digest();
            } catch (NoSuchAlgorithmException cnse) {
                throw new DigestException("Couldn't make digest of partial content " + cnse.getMessage());
            }
}
```

Digests are compared with the `MessageDigest.isEqual(byte[] first, byte[] second)` function, which does a simple byte compare, as reported by the Java documentation.

Switching from SHA-256 to SHA-512 could increase efficiency in our case, thanks to the chosen device having a x86_64 architecture and being SHA-512 based on 64-bits words. Furthermore, this would not pose any problem in terms of API availability since both of them are available since API level 1+. Switching to SHA-512 would double the size of the produced digest, leaving only 64 bytes of space to encode the symmetric key. Keeping into consideration previous observations about having a maximum data size of 128 bytes, we have preferred a smaller message digest also in terms of bandwith consumption and foreseeing a possible symmetric-key dimension increase.

3.3.3 Symmetric Key Generation and Encryption

In this subsection, we will refer always to cryptographic operations happening on the Android devices, since the Trusted Authority never receives personal data.

The specification in paper [7] asks for a symmetric one-time-use key to encode personal data. Symmetric keys can be generated via a `KeyGenerator` with an algorithm-specific initialization, not providing `SecureRandom` to rely on the implementation of the highest-priority installed provider. This raises the reliability of the proposed solution, as symmetric keys are never reused and are unrelated. The available algorithms for encryption and key generation depend on the chosen security `Provider`, which is in our case the Bouncy Castle provider. Available key generation algorithms include AES for symmetric encryption, but it is only possible to use is with AES/ECB or AES/GCM/NOPADDING: since the former has known weaknesses and it is not suited to large-size files encryption, we considered the second method. A working example is shown in Listing 3.4.

Buoncy Castle also offers implementations for several Password Based Encryption methods, using AES/CBC with different key lengths, salt and padding options. We did not consider any of those algorithms as they are out of the scope of this project: encryption must be performed with a one-time-use secret, while using a fixed password would definitely reduce the security level. Moreover, PBE requires many iterations in order to process a secure derived key (around 100 000 or 200 000), which could impact on efficiency and performance.

Listing 3.4: Example code with ECB encryption on Android device

```java
private static String symmetricEncrAlgorithm = "AES";
private static int AES_KEY_SIZE = 256;

public static byte[] generateSymmetricRandomKey() {
        KeyGenerator keyGen = null;
        try {
                keyGen = KeyGenerator.getInstance(
    symmetricEncrAlgorithm);
        } catch (NoSuchAlgorithmException e) {
                e.printStackTrace();
                Log.d(TAG, "Error in generating the symmetric 
    random key: " + e.getMessage());
        }
        keyGen.init(AES_KEY_SIZE);
        SecretKey secretKey = keyGen.generateKey();
        return secretKey.getEncoded();
}
```

```java
public static byte[] encryptSymmetric(byte[] encodedSymmKey, byte[] clearText) {
        SecretKeySpec skeySpec = new SecretKeySpec(encodedSymmKey, symmetricEncrAlgorithm);
        byte[] encryptedText = new byte[0];
        try {
                Cipher cipher = Cipher.getInstance(symmetricEncrAlgorithm);
                cipher.init(Cipher.ENCRYPT_MODE, skeySpec);
                encryptedText = cipher.doFinal(clearText);
        } catch (NoSuchAlgorithmException | NoSuchPaddingException | InvalidKeyException | IllegalBlockSizeException | BadPaddingException e) {
                e.printStackTrace();
                Log.d(TAG, "Error in symmetric encryption: " + e.getMessage());
        }
        return encryptedText;
}

public static byte[] decryptSymmetric(byte[] encodedSymmKey, byte[] encryptedText) throws Exception {
        SecretKeySpec skeySpec = new SecretKeySpec(encodedSymmKey, symmetricEncrAlgorithm);
        Cipher cipher = Cipher.getInstance(symmetricEncrAlgorithm);
        cipher.init(Cipher.DECRYPT_MODE, skeySpec);
        byte[] decrypted = cipher.doFinal(encryptedText);
return decrypted;
```

Alternatively, a AES_256/CBC/PKCS7Padding algorithm is available with the AndroidOpenSSL provider: the Cipher Block Chaining algorithm is more secure than Electronic Codebook, and is suited to work in this context as a block cipher mode. Padding is needed due to unpredictable size of the policy, which may not be a multiple of the block size (128 bits).

To implement this solution we need to provide an initialization vector to be XORed with the first block of plaintext: to obtain it we rely on the Java class `SecureRandom`, and we create it with the same size of block in CBC. In this case, when creating a symmetric key we just need to specify `"AES"` as input parameter, while for the encryption and decryption methods we add a initialization vector:

Listing 3.5: Example code with CBC encryption on Android device

```java
private static String symmetricEncrAlgorithm = "AES/CBC/PKCS5Padding";

public static byte[] encryptSymmetric(byte[] encodedSymmKey, byte[] clearText) {
        SecretKeySpec skeySpec = new SecretKeySpec(encodedSymmKey, symmetricEncrAlgorithm);
        byte[] encryptedText = new byte[0];
        try {
```

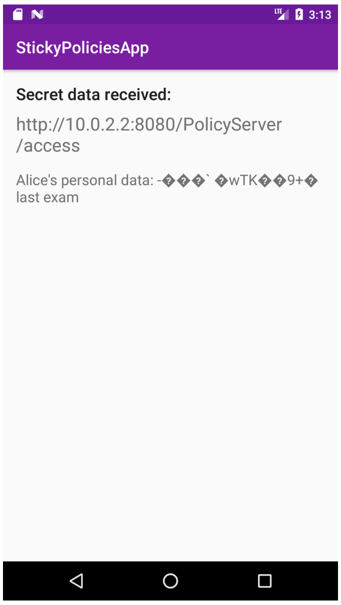

Figure 3.3: Decrypted text in Bob's device.

```
                Cipher cipher = Cipher.getInstance(
    symmetricEncrAlgorithm, "AndroidOpenSSL");
                SecureRandom secureRandom = new SecureRandom();
                byte[] iv = new byte[cipher.getBlockSize()];
                secureRandom.nextBytes(iv);
                cipher.init(Cipher.ENCRYPT_MODE, skeySpec, new
    IvParameterSpec(iv));
                encryptedText = cipher.doFinal(clearText);
        } catch (InvalidAlgorithmParameterException |
    NoSuchAlgorithmException | NoSuchPaddingException |
    InvalidKeyException | IllegalBlockSizeException |
    BadPaddingException | NoSuchProviderException e) {
                e.printStackTrace();
                Log.d(TAG, "Error in symmetric encryption: " + e
    .getMessage());
                throw new SecurityException(e.getMessage());
        }
        return encryptedText;
}
```

From the code in Listing 3.5 descends immediately that it is impossible to share data with this encryption algorithm, due to the need of initialization vectors. It is useful to observe the result of Bob's decryption of Alice's secret data "I cheated during last exam", as shown in Figure 3.3

In the decrypted text, the first part of the text is wrong, while the second part can be understood. This is because if the IV is not shared or is incorrect, the receiver will still be able to understand the message. Consequently, to use CBC it should be necessary to share also the initialization vector together with the secret key to encrypt the whole message properly. In other words, this introduces another weakness in the message protocol, because a potential attacker would only need to get hold of the key to encrypt information, instead of both key and initialization vector.

symmetric key generation, symmetric key algorithm for encryption

sign, verify, encrypt and decrypt with asymmetric keys

performance of the whole process, padding, salt, random generation in android

3.4 Performance

Bibliography

[1] John Bethencourt, Amit Sahai, and Brent Waters. Ciphertext-policy attribute-based encryption toolkit. URL: `http://hms.isi.jhu.edu/acsc/cpabe/`.

[2] John Bethencourt, Amit Sahai, and Brent Waters. Ciphertext-policy attribute-based encryption. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 321–334. IEEE, 2007.

[3] Angelo De Caro and Vincenzo Iovino. jpbc: Java pairing based cryptography. In *Proceedings of the 16th IEEE Symposium on Computers and Communications, ISCC 2011*, pages 850–855, Kerkyra, Corfu, Greece, June 28 - July 1, 2011.

[4] Matthew Green and Giuseppe Ateniese. Identity-based proxy re-encryption. In *Applied Cryptography and Network Security*, pages 288–306. Springer, 2007.

[5] ISO/IEC. Iso/iec 11889-1:2009 information technology – trusted platform module, 2009.

[6] Ben Lynn. The pairing-based cryptography library, 2013. URL: `https://crypto.stanford.edu/pbc/`.

[7] Marco Casassa Mont, Siani Pearson, and Pete Bramhall. Towards accountable management of identity and privacy: Sticky policies and enforceable tracing services. In *Database and Expert Systems Applications, 2003. Proceedings. 14th International Workshop on*, pages 377–382. IEEE, 2003. URL: `http://www.hpl.hp.com/techreports/2003/HPL-2003-49.pdf?jumpid=reg_R1002_USEN`.

[8] Siani Pearson and Marco Casassa-Mont. Sticky policies: an approach for managing privacy across multiple parties. *Computer*, 44(9):60–68, 2011.

[9] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979. URL: `http://doi.acm.org/10.1145/359168.359176`.

[10] Adi Shamir. Identity-based cryptosystems and signature schemes. In George Robert Blakley and David Chaum, editors, *Advances in Cryptology*, pages 47–53, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg.

[11] OASIS Standard. Extensible access control markup language (xacml) version 2.0, 2005.

[12] Qiang Tang. *On Using Encryption Techniques to Enhance Sticky Policies Enforcement*. Number WoTUG-31/TR-CTIT-08-64 in CTIT Technical Report Series. Centre for Telematics and Information Technology (CTIT), Netherlands, 2008.

[13] Gert Van Ham. Lightweight cryptographic library, 2003. URL: `http://jcetaglib.sourceforge.net/lightcrypto`.

Appendix A

XSD Grammar for Sticky Policies

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <xs:element name="stickyPolicy" type="stickyPolicyType"
    />

        <xs:complexType name="stickyPolicyType">
                <xs:sequence>
                        <xs:element name="trustedAuthority" type
    ="xs:string" maxOccurs="unbounded" minOccurs="1" />
                        <xs:element name="owner" type="ownerType
    " maxOccurs="1" minOccurs="1" />
                        <xs:element name="policy" type="
    policyType" maxOccurs="unbounded" minOccurs="1" />
        </xs:sequence>
        </xs:complexType>

        <xs:complexType name="policyType">
                <xs:sequence>
                        <xs:element name="target" type="
    xs:string" maxOccurs="unbounded" minOccurs="1" />
                        <xs:element name="dataType" type="
    dataType" maxOccurs="unbounded" minOccurs="1" />
                        <xs:element name="validity" type="
    dateType" maxOccurs="1" minOccurs="1" />
                        <xs:element name="constraint" type="
    xs:string" maxOccurs="unbounded" minOccurs="1" />
                        <xs:element name="action" type="
    xs:string" maxOccurs="unbounded" minOccurs="1" />
                </xs:sequence>
        </xs:complexType>

        <xs:complexType name="ownerType">
                <xs:sequence>
                        <xs:element name="referenceName" type="
    xs:string" maxOccurs="1" minOccurs="1"  />
```

```
                        <xs:element name="ownersDetails" type="xs:string" maxOccurs="unbounded" minOccurs="1"  />
                        <xs:element name="certificateSerialNumber" type="xs:unsignedLong" maxOccurs="1" minOccurs="1" />
                </xs:sequence>
        </xs:complexType>

        <xs:simpleType name="dataType" >
                <xs:restriction base="xs:string">
                        <xs:enumeration value="picture" />
                        <xs:enumeration value="text" />
                        <xs:enumeration value="video" />
                        <xs:enumeration value="audio" />
                        <xs:enumeration value="position" />
                </xs:restriction>
        </xs:simpleType>

        <xs:complexType name="dateType">
                <xs:sequence>
                        <xs:element name="day" type="dayType" />
                        <xs:element name="month" type="monthType" />
                        <xs:element name="year" type="yearType" />
                </xs:sequence>
        </xs:complexType>

        <xs:simpleType name="dayType" >
                <xs:restriction base="xs:positiveInteger">
                        <xs:minInclusive value="1" />
                        <xs:maxInclusive value="31" />
                </xs:restriction>
        </xs:simpleType>

        <xs:simpleType name="monthType" >
                <xs:restriction base="xs:positiveInteger">
                        <xs:minInclusive value="1" />
                        <xs:maxInclusive value="12" />
                </xs:restriction>
        </xs:simpleType>

        <xs:simpleType name="yearType" >
                        <xs:restriction base="xs:positiveInteger">
                        <xs:minInclusive value="2017" />
                        <xs:maxInclusive value="2027" />
                        </xs:restriction>
        </xs:simpleType>
</xs:schema>
```

Appendix B

Java Pairing Based Cryptography Library

Here are reported the complete classes from [3], mentioned in chapter 3.

In this class we have a `main` function to show a sample usage of IBE cryptography.

Listing B.1: AHIBEDIP10.class

```java
package it.unisa.dia.gas.crypto.jpbc.fe.ibe.dip10;

import it.unisa.dia.gas.crypto.jpbc.fe.ibe.dip10.engines.
    AHIBEDIP10KEMEngine;
import it.unisa.dia.gas.crypto.jpbc.fe.ibe.dip10.generators.
    AHIBEDIP10KeyPairGenerator;
import it.unisa.dia.gas.crypto.jpbc.fe.ibe.dip10.generators.
    AHIBEDIP10SecretKeyGenerator;
import it.unisa.dia.gas.crypto.jpbc.fe.ibe.dip10.params.*;
import it.unisa.dia.gas.crypto.kem.KeyEncapsulationMechanism;
import it.unisa.dia.gas.jpbc.Element;
import it.unisa.dia.gas.jpbc.Pairing;
import it.unisa.dia.gas.plaf.jpbc.pairing.PairingFactory;
import org.bouncycastle.crypto.AsymmetricCipherKeyPair;
import org.bouncycastle.crypto.CipherParameters;
import org.bouncycastle.crypto.InvalidCipherTextException;

import java.util.Arrays;

import static org.junit.Assert.*;


/**
 * @author Angelo De Caro (jpbclib@gmail.com)
 */
public class AHIBEDIP10 {


    public AHIBEDIP10() {
```

```java
    }


    public AsymmetricCipherKeyPair setup(int bitLength, int length) {
        AHIBEDIP10KeyPairGenerator setup = new AHIBEDIP10KeyPairGenerator();
        setup.init(new AHIBEDIP10KeyPairGenerationParameters(bitLength, length));

        return setup.generateKeyPair();
    }

    public Element[] map(CipherParameters publicKey, String... ids) {
        Pairing pairing = PairingFactory.getPairing(((AHIBEDIP10PublicKeyParameters) publicKey).getParameters());

        Element[] elements = new Element[ids.length];
        for (int i = 0; i < elements.length; i++) {
            byte[] id = ids[i].getBytes();
            elements[i] = pairing.getZr().newElementFromHash(id, 0, id.length);
        }
        return elements;
    }


    public CipherParameters keyGen(AsymmetricCipherKeyPair masterKey, Element... ids) {
        AHIBEDIP10SecretKeyGenerator generator = new AHIBEDIP10SecretKeyGenerator();
        generator.init(new AHIBEDIP10SecretKeyGenerationParameters(
                (AHIBEDIP10MasterSecretKeyParameters) masterKey.getPrivate(),
                (AHIBEDIP10PublicKeyParameters) masterKey.getPublic(),
                ids
        ));

        return generator.generateKey();
    }

    public CipherParameters delegate(AsymmetricCipherKeyPair masterKey, CipherParameters secretKey, Element id) {
        AHIBEDIP10SecretKeyGenerator generator = new AHIBEDIP10SecretKeyGenerator();
        generator.init(new AHIBEDIP10DelegateGenerationParameters(
                (AHIBEDIP10PublicKeyParameters) masterKey.getPublic(),
                (AHIBEDIP10SecretKeyParameters) secretKey,
                id
```

```
        ));

        return generator.generateKey();
    }

    public byte[][] encaps(CipherParameters publicKey, Element
    ... ids) {
        try {
            KeyEncapsulationMechanism kem = new
    AHIBEDIP10KEMEngine();
            kem.init(true, new AHIBEDIP10EncryptionParameters((
    AHIBEDIP10PublicKeyParameters) publicKey, ids));

            byte[] ciphertext = kem.process();

            assertNotNull(ciphertext);
            assertNotSame(0, ciphertext.length);

            byte[] key = Arrays.copyOfRange(ciphertext, 0, kem.
    getKeyBlockSize());
            byte[] ct = Arrays.copyOfRange(ciphertext, kem.
    getKeyBlockSize(), ciphertext.length);

            return new byte[][]{key, ct};
        } catch (InvalidCipherTextException e) {
            e.printStackTrace();
            fail(e.getMessage());
        }
        return null;
    }

    public byte[] decaps(CipherParameters secretKey, byte[]
    cipherText) {
        try {
            KeyEncapsulationMechanism kem = new
    AHIBEDIP10KEMEngine();

            kem.init(false, secretKey);
            byte[] key = kem.processBlock(cipherText);

            assertNotNull(key);
            assertNotSame(0, key.length);

            return key;
        } catch (InvalidCipherTextException e) {
            e.printStackTrace();
            fail(e.getMessage());
        }

        return null;
    }


    public static void main(String[] args) {
```

```java
        AHIBEDIP10 engine = new AHIBEDIP10();

        // Setup
        AsymmetricCipherKeyPair keyPair = engine.setup(64, 3);

        // KeyGen
        Element[] ids = engine.map(keyPair.getPublic(), "angelo", "de caro", "unisa");

        CipherParameters sk0 = engine.keyGen(keyPair, ids[0]);
        CipherParameters sk01 = engine.keyGen(keyPair, ids[0], ids[1]);
        CipherParameters sk012 = engine.keyGen(keyPair, ids[0], ids[1], ids[2]);

        CipherParameters sk1 = engine.keyGen(keyPair, ids[1]);
        CipherParameters sk10 = engine.keyGen(keyPair, ids[1], ids[0]);
        CipherParameters sk021 = engine.keyGen(keyPair, ids[0], ids[2], ids[1]);

        // Encryption/Decryption
        byte[][] ciphertext0 = engine.encaps(keyPair.getPublic(), ids[0]);
        byte[][] ciphertext01 = engine.encaps(keyPair.getPublic(), ids[0], ids[1]);
        byte[][] ciphertext012 = engine.encaps(keyPair.getPublic(), ids[0], ids[1], ids[2]);

        // Decrypt
        assertEquals(true, Arrays.equals(ciphertext0[0], engine.decaps(sk0, ciphertext0[1])));
        assertEquals(true, Arrays.equals(ciphertext01[0], engine.decaps(sk01, ciphertext01[1])));
        assertEquals(true, Arrays.equals(ciphertext012[0], engine.decaps(sk012, ciphertext012[1])));

        assertEquals(false, Arrays.equals(ciphertext0[0], engine.decaps(sk1, ciphertext0[1])));
        assertEquals(false, Arrays.equals(ciphertext01[0], engine.decaps(sk10, ciphertext01[1])));
        assertEquals(false, Arrays.equals(ciphertext012[0], engine.decaps(sk021, ciphertext012[1])));

        // Delegate/Decrypt
        assertEquals(true, Arrays.equals(ciphertext01[0], engine.decaps(engine.delegate(keyPair, sk0, ids[1]), ciphertext01[1])));
        assertEquals(true, Arrays.equals(ciphertext012[0], engine.decaps(engine.delegate(keyPair, sk01, ids[2]), ciphertext012[1])));
        assertEquals(true, Arrays.equals(ciphertext012[0], engine.decaps(engine.delegate(keyPair, engine.delegate(keyPair, sk0, ids[1]), ids[2]), ciphertext012[1])));
```

```
        assertEquals(false, Arrays.equals(ciphertext01[0],
    engine.decaps(engine.delegate(keyPair, sk0, ids[0]),
    ciphertext01[1])));
        assertEquals(false, Arrays.equals(ciphertext012[0],
    engine.decaps(engine.delegate(keyPair, sk01, ids[1]),
    ciphertext012[1])));
        assertEquals(false, Arrays.equals(ciphertext012[0],
    engine.decaps(engine.delegate(keyPair, engine.delegate(
    keyPair, sk0, ids[2]), ids[1]), ciphertext012[1])));
    }


}
```

This class shows which methods are called when encrypting and decrypting.

Listing B.2: PairingKeyEncapsulationMechanism.class

```
package it.unisa.dia.gas.crypto.jpbc.kem;

import it.unisa.dia.gas.crypto.jpbc.cipher.
    PairingAsymmetricBlockCipher;
import it.unisa.dia.gas.crypto.kem.KeyEncapsulationMechanism;
import org.bouncycastle.crypto.InvalidCipherTextException;

/**
 * @author Angelo De Caro (jpbclib@gmail.com)
 */
public abstract class PairingKeyEncapsulationMechanism extends
    PairingAsymmetricBlockCipher implements
    KeyEncapsulationMechanism {

    private static byte[] EMPTY = new byte[0];


    protected int keyBytes = 0;

    public int getKeyBlockSize() {
        return keyBytes;
    }

    public int getInputBlockSize() {
        if (forEncryption)
            return 0;

        return outBytes - keyBytes;
    }

    public int getOutputBlockSize() {
        if (forEncryption)
            return outBytes;

        return keyBytes;
```

```java
    }


    public byte[] processBlock(byte[] in) throws InvalidCipherTextException {
        return processBlock(in, in.length, 0);
    }

    public byte[] process() throws InvalidCipherTextException {
        return processBlock(EMPTY, 0, 0);
    }

}
```

Listing B.3: Excerpt from PairingAsymmetricBlockCipher class

```java
package it.unisa.dia.gas.crypto.jpbc.cipher;

import it.unisa.dia.gas.jpbc.Pairing;
import org.bouncycastle.crypto.AsymmetricBlockCipher;
import org.bouncycastle.crypto.CipherParameters;
import org.bouncycastle.crypto.DataLengthException;
import org.bouncycastle.crypto.InvalidCipherTextException;
import org.bouncycastle.crypto.params.ParametersWithRandom;

/**
 * @author Angelo De Caro (jpbclib@gmail.com)
 */
public abstract class PairingAsymmetricBlockCipher implements AsymmetricBlockCipher {

    protected CipherParameters key;
    protected boolean forEncryption;

    protected int inBytes = 0;
    protected int outBytes = 0;

    protected Pairing pairing;


    /**
     * Return the maximum size for an input block to this engine.
     *
     * @return maximum size for an input block.
     */
    public int getInputBlockSize() {
        if (forEncryption) {
            return inBytes;
        }

        return outBytes;
    }

```

```java
    /**
     * Return the maximum size for an output block to this engine.
     *
     * @return maximum size for an output block.
     */
    public int getOutputBlockSize() {
        if (forEncryption) {
            return outBytes;
        }

        return inBytes;
    }

    /**
     * initialise the cipher engine.
     *
     * @param forEncryption true if we are encrypting, false otherwise.
     * @param param         the necessary cipher key parameters.
     */
    public void init(boolean forEncryption, CipherParameters param) {
        if (param instanceof ParametersWithRandom) {
            ParametersWithRandom p = (ParametersWithRandom) param;

            this.key = p.getParameters();
        } else {
            this.key = param;
        }

        this.forEncryption = forEncryption;

        initialize();
    }

    /**
     * Process a single block using the basic cipher algorithm.
     *
     * @param in    the input array.
     * @param inOff the offset into the input buffer where the data starts.
     * @param inLen the length of the data to be processed.
     * @return the result of the cipher process.
     * @throws org.bouncycastle.crypto.DataLengthException the input block is too large.
     */
    public byte[] processBlock(byte[] in, int inOff, int inLen) throws InvalidCipherTextException {
        if (key == null)
            throw new IllegalStateException("Engine not initialized");

```

```java
        int maxLength = getInputBlockSize();

        if (inLen < maxLength) {
                System.out.println("inLen: " +inLen + "; maxLength: " + maxLength);
            throw new DataLengthException("Input too small for the cipher.");
        }
        return process(in, inOff, inLen);
    }


    public abstract void initialize();

    public abstract byte[] process(byte[] in, int inOff, int inLen) throws InvalidCipherTextException;

}
```