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

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

Here goes some title

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 recipi-

ents. 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).

Chapter 3

Getting one's hands dirty

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.

3.1 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', '>', '<'.

3.1.1 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 3.1: Java mock class for IBE implementation

```
1 // Setup
2 AsymmetricCipherKeyPair keyPair = engine.setup(64, 3);
3 // KeyGen
4 Element[] ids = engine.map(keyPair.getPublic(), "angelo", "de_
    caro", "unisa");
```

```
5  CipherParameters sk0 = engine.keyGen(keyPair, ids[0]);
6  CipherParameters sk01 = engine.keyGen(keyPair, ids[0], ids[1]);
7  CipherParameters sk012 = engine.keyGen(keyPair, ids[0], ids[1],
       ids[2]);
8  // Encryption
9  byte[][] ciphertext0 = engine.encaps(keyPair.getPublic(), ids
       [0]);
10 byte[][] ciphertext01 = engine.encaps(keyPair.getPublic(), ids
       [0], ids[1]);
11 byte[][] ciphertext012 = engine.encaps(keyPair.getPublic(), ids
       [0], ids[1], ids[2]);
12 // Decrypt
13 byte[] cleartext0 = engine.decaps(sk0, ciphertext0[1]);
14 System.out.println(new String(cleartext0) + "");
```

As shown in Listing 3.1, 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 3.2, this function calls `processBlock` supplying as input an empty byte array instead of an actual input.

Listing 3.2: Excerpt from PairingKeyEncapsulationMechanism class

```
1  package it.unisa.dia.gas.crypto.jpbc.kem;
2
3  import {...}
4
5  public abstract class PairingKeyEncapsulationMechanism extends
       PairingAsymmetricBlockCipher implements
       KeyEncapsulationMechanism {
6
7      private static byte[] EMPTY = new byte[0];
8
9          // some other functions...
10
11         public byte[] processBlock(byte[] in) throws
       InvalidCipherTextException {
12         return processBlock(in, in.length, 0);
13     }
14
15         public byte[] process() throws
       InvalidCipherTextException {
16         return processBlock(EMPTY, 0, 0);
17     }
18 }
```

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.

3.2 Realizing a Hybrid Cryptosystem

In this scenario, *Sticky Policies* are realized via XML files and paired through the combination of symmetric and asymmetric cryptography with some personal data. 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. For this purpose, 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. Purposely, Alice 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.

To decrypt Alice's data, Bob has to follow this steps:

- 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.

- 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 sent 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.

3.2.1 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.

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.

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"  />
```

```
26                          <xs:element name="ownersDetails" type="
    xs:string" maxOccurs="unbounded" minOccurs="1"  />
27                  </xs:sequence>
28          </xs:complexType>
29
30          <xs:simpleType name="dataType" >
31                  <xs:restriction base="xs:string">
32                          <xs:enumeration value="picture" />
33                          <xs:enumeration value="text" />
34                          <xs:enumeration value="video" />
35                          <xs:enumeration value="audio" />
36                          <xs:enumeration value="position" />
37                  </xs:restriction>
38          </xs:simpleType>
39
40          <xs:complexType name="dateType">
41                  <xs:sequence>
42                          <xs:element name="day" type="dayType" />
43                          <xs:element name="month" type="monthType
    " />
44                          <xs:element name="year" type="yearType"
    />
45                  </xs:sequence>
46          </xs:complexType>
47
48          <xs:simpleType name="dayType" >
49                  <xs:restriction base="xs:positiveInteger">
50                          <xs:minInclusive value="1" />
51                          <xs:maxInclusive value="31" />
52                  </xs:restriction>
53          </xs:simpleType>
54
55          <xs:simpleType name="monthType" >
56                  <xs:restriction base="xs:positiveInteger">
57                          <xs:minInclusive value="1" />
58                          <xs:maxInclusive value="12" />
59                  </xs:restriction>
60          </xs:simpleType>
61
62          <xs:simpleType name="yearType" >
63                          <xs:restriction base="xs:positiveInteger
    ">
64                          <xs:minInclusive value="2017" />
65                          <xs:maxInclusive value="2027" />
66                          </xs:restriction>
67          </xs:simpleType>
68  </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

```
1  package it.unisa.dia.gas.crypto.jpbc.fe.ibe.dip10;
2
3  import it.unisa.dia.gas.crypto.jpbc.fe.ibe.dip10.engines.
       AHIBEDIP10KEMEngine;
4  import it.unisa.dia.gas.crypto.jpbc.fe.ibe.dip10.generators.
       AHIBEDIP10KeyPairGenerator;
5  import it.unisa.dia.gas.crypto.jpbc.fe.ibe.dip10.generators.
       AHIBEDIP10SecretKeyGenerator;
6  import it.unisa.dia.gas.crypto.jpbc.fe.ibe.dip10.params.*;
7  import it.unisa.dia.gas.crypto.kem.KeyEncapsulationMechanism;
8  import it.unisa.dia.gas.jpbc.Element;
9  import it.unisa.dia.gas.jpbc.Pairing;
10 import it.unisa.dia.gas.plaf.jpbc.pairing.PairingFactory;
11 import org.bouncycastle.crypto.AsymmetricCipherKeyPair;
12 import org.bouncycastle.crypto.CipherParameters;
13 import org.bouncycastle.crypto.InvalidCipherTextException;
14
15 import java.util.Arrays;
16
17 import static org.junit.Assert.*;
18
19
20 /**
21  * @author Angelo De Caro (jpbclib@gmail.com)
22  */
23 public class AHIBEDIP10 {
24
25
26     public AHIBEDIP10() {
```

```
27      }
28
29
30      public AsymmetricCipherKeyPair setup(int bitLength, int
        length) {
31          AHIBEDIP10KeyPairGenerator setup = new
        AHIBEDIP10KeyPairGenerator();
32          setup.init(new AHIBEDIP10KeyPairGenerationParameters(
        bitLength, length));
33
34          return setup.generateKeyPair();
35      }
36
37      public Element[] map(CipherParameters publicKey, String...
        ids) {
38          Pairing pairing = PairingFactory.getPairing(((
        AHIBEDIP10PublicKeyParameters) publicKey).getParameters());
39
40          Element[] elements = new Element[ids.length];
41          for (int i = 0; i < elements.length; i++) {
42              byte[] id = ids[i].getBytes();
43              elements[i] = pairing.getZr().newElementFromHash(id,
         0, id.length);
44          }
45          return elements;
46      }
47
48
49      public CipherParameters keyGen(AsymmetricCipherKeyPair
        masterKey, Element... ids) {
50          AHIBEDIP10SecretKeyGenerator generator = new
        AHIBEDIP10SecretKeyGenerator();
51          generator.init(new
        AHIBEDIP10SecretKeyGenerationParameters(
52                  (AHIBEDIP10MasterSecretKeyParameters) masterKey.
        getPrivate(),
53                  (AHIBEDIP10PublicKeyParameters) masterKey.
        getPublic(),
54                  ids
55          ));
56
57          return generator.generateKey();
58      }
59
60      public CipherParameters delegate(AsymmetricCipherKeyPair
        masterKey, CipherParameters secretKey, Element id) {
61          AHIBEDIP10SecretKeyGenerator generator = new
        AHIBEDIP10SecretKeyGenerator();
62          generator.init(new
        AHIBEDIP10DelegateGenerationParameters(
63                  (AHIBEDIP10PublicKeyParameters) masterKey.
        getPublic(),
64                  (AHIBEDIP10SecretKeyParameters) secretKey,
65                  id
```

```
66          ));
67
68          return generator.generateKey();
69      }
70
71      public byte[][][] encaps(CipherParameters publicKey, Element
    ... ids) {
72          try {
73              KeyEncapsulationMechanism kem = new
    AHIBEDIP10KEMEngine();
74              kem.init(true, new AHIBEDIP10EncryptionParameters((
    AHIBEDIP10PublicKeyParameters) publicKey, ids));
75
76              byte[] ciphertext = kem.process();
77
78              assertNotNull(ciphertext);
79              assertNotSame(0, ciphertext.length);
80
81              byte[] key = Arrays.copyOfRange(ciphertext, 0, kem.
    getKeyBlockSize());
82              byte[] ct = Arrays.copyOfRange(ciphertext, kem.
    getKeyBlockSize(), ciphertext.length);
83
84              return new byte[][][]{key, ct};
85          } catch (InvalidCipherTextException e) {
86              e.printStackTrace();
87              fail(e.getMessage());
88          }
89          return null;
90      }
91
92      public byte[] decaps(CipherParameters secretKey, byte[]
    cipherText) {
93          try {
94              KeyEncapsulationMechanism kem = new
    AHIBEDIP10KEMEngine();
95
96              kem.init(false, secretKey);
97              byte[] key = kem.processBlock(cipherText);
98
99              assertNotNull(key);
100             assertNotSame(0, key.length);
101
102             return key;
103         } catch (InvalidCipherTextException e) {
104             e.printStackTrace();
105             fail(e.getMessage());
106         }
107
108         return null;
109     }
110
111
112     public static void main(String[] args) {
```

```
113         AHIBEDIP10 engine = new AHIBEDIP10();
114 
115         // Setup
116         AsymmetricCipherKeyPair keyPair = engine.setup(64, 3);
117 
118         // KeyGen
119         Element[] ids = engine.map(keyPair.getPublic(), "angelo"
    , "de_caro", "unisa");
120 
121         CipherParameters sk0 = engine.keyGen(keyPair, ids[0]);
122         CipherParameters sk01 = engine.keyGen(keyPair, ids[0],
    ids[1]);
123         CipherParameters sk012 = engine.keyGen(keyPair, ids[0],
    ids[1], ids[2]);
124 
125         CipherParameters sk1 = engine.keyGen(keyPair, ids[1]);
126         CipherParameters sk10 = engine.keyGen(keyPair, ids[1],
    ids[0]);
127         CipherParameters sk021 = engine.keyGen(keyPair, ids[0],
    ids[2], ids[1]);
128 
129         // Encryption/Decryption
130         byte[][] ciphertext0 = engine.encaps(keyPair.getPublic()
    , ids[0]);
131         byte[][] ciphertext01 = engine.encaps(keyPair.getPublic
    (), ids[0], ids[1]);
132         byte[][] ciphertext012 = engine.encaps(keyPair.getPublic
    (), ids[0], ids[1], ids[2]);
133 
134         // Decrypt
135         assertEquals(true, Arrays.equals(ciphertext0[0], engine.
    decaps(sk0, ciphertext0[1])));
136         assertEquals(true, Arrays.equals(ciphertext01[0], engine
    .decaps(sk01, ciphertext01[1])));
137         assertEquals(true, Arrays.equals(ciphertext012[0],
    engine.decaps(sk012, ciphertext012[1])));
138 
139         assertEquals(false, Arrays.equals(ciphertext0[0], engine
    .decaps(sk1, ciphertext0[1])));
140         assertEquals(false, Arrays.equals(ciphertext01[0],
    engine.decaps(sk10, ciphertext01[1])));
141         assertEquals(false, Arrays.equals(ciphertext012[0],
    engine.decaps(sk021, ciphertext012[1])));
142 
143         // Delegate/Decrypt
144         assertEquals(true, Arrays.equals(ciphertext01[0], engine
    .decaps(engine.delegate(keyPair, sk0, ids[1]), ciphertext01
    [1])));
145         assertEquals(true, Arrays.equals(ciphertext012[0],
    engine.decaps(engine.delegate(keyPair, sk01, ids[2]),
    ciphertext012[1])));
146         assertEquals(true, Arrays.equals(ciphertext012[0],
    engine.decaps(engine.delegate(keyPair, engine.delegate(
    keyPair, sk0, ids[1]), ids[2]), ciphertext012[1])));
```

```
147
148         assertEquals(false, Arrays.equals(ciphertext01[0],
    engine.decaps(engine.delegate(keyPair, sk0, ids[0]),
    ciphertext01[1])));
149         assertEquals(false, Arrays.equals(ciphertext012[0],
    engine.decaps(engine.delegate(keyPair, sk01, ids[1]),
    ciphertext012[1])));
150         assertEquals(false, Arrays.equals(ciphertext012[0],
    engine.decaps(engine.delegate(keyPair, engine.delegate(
    keyPair, sk0, ids[2]), ids[1]), ciphertext012[1])));
151     }
152
153
154 }
```

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

Listing B.2: PairingKeyEncapsulationMechanism.class

```
1  package it.unisa.dia.gas.crypto.jpbc.kem;
2
3  import it.unisa.dia.gas.crypto.jpbc.cipher.
       PairingAsymmetricBlockCipher;
4  import it.unisa.dia.gas.crypto.kem.KeyEncapsulationMechanism;
5  import org.bouncycastle.crypto.InvalidCipherTextException;
6
7  /**
8   * @author Angelo De Caro (jpbclib@gmail.com)
9   */
10 public abstract class PairingKeyEncapsulationMechanism extends
       PairingAsymmetricBlockCipher implements
       KeyEncapsulationMechanism {
11
12     private static byte[] EMPTY = new byte[0];
13
14
15     protected int keyBytes = 0;
16
17     public int getKeyBlockSize() {
18         return keyBytes;
19     }
20
21     public int getInputBlockSize() {
22         if (forEncryption)
23             return 0;
24
25         return outBytes - keyBytes;
26     }
27
28     public int getOutputBlockSize() {
29         if (forEncryption)
30             return outBytes;
31
32         return keyBytes;
```

```
33      }
34
35
36      public byte[] processBlock(byte[] in) throws
        InvalidCipherTextException {
37          return processBlock(in, in.length, 0);
38      }
39
40      public byte[] process() throws InvalidCipherTextException {
41          return processBlock(EMPTY, 0, 0);
42      }
43
44  }
```

Listing B.3: Excerpt from PairingAsymmetricBlockCipher class

```
1   package it.unisa.dia.gas.crypto.jpbc.cipher;
2
3   import it.unisa.dia.gas.jpbc.Pairing;
4   import org.bouncycastle.crypto.AsymmetricBlockCipher;
5   import org.bouncycastle.crypto.CipherParameters;
6   import org.bouncycastle.crypto.DataLengthException;
7   import org.bouncycastle.crypto.InvalidCipherTextException;
8   import org.bouncycastle.crypto.params.ParametersWithRandom;
9
10  /**
11   * @author Angelo De Caro (jpbclib@gmail.com)
12   */
13  public abstract class PairingAsymmetricBlockCipher implements
        AsymmetricBlockCipher {
14
15      protected CipherParameters key;
16      protected boolean forEncryption;
17
18      protected int inBytes = 0;
19      protected int outBytes = 0;
20
21      protected Pairing pairing;
22
23
24      /**
25       * Return the maximum size for an input block to this engine
        .
26       *
27       * @return maximum size for an input block.
28       */
29      public int getInputBlockSize() {
30          if (forEncryption) {
31              return inBytes;
32          }
33
34          return outBytes;
35      }
36
```

```
37      /**
38       * Return the maximum size for an output block to this
    engine.
39       *
40       * @return maximum size for an output block.
41       */
42      public int getOutputBlockSize() {
43          if (forEncryption) {
44              return outBytes;
45          }
46
47          return inBytes;
48      }
49
50      /**
51       * initialise the cipher engine.
52       *
53       * @param forEncryption true if we are encrypting, false
    otherwise.
54       * @param param         the necessary cipher key parameters.
55       */
56      public void init(boolean forEncryption, CipherParameters
    param) {
57          if (param instanceof ParametersWithRandom) {
58              ParametersWithRandom p = (ParametersWithRandom)
    param;
59
60              this.key = p.getParameters();
61          } else {
62              this.key = param;
63          }
64
65          this.forEncryption = forEncryption;
66
67          initialize();
68      }
69
70      /**
71       * Process a single block using the basic cipher algorithm.
72       *
73       * @param in    the input array.
74       * @param inOff the offset into the input buffer where the
    data starts.
75       * @param inLen the length of the data to be processed.
76       * @return the result of the cipher process.
77       * @throws org.bouncycastle.crypto.DataLengthException the
    input block is too large.
78       */
79      public byte[] processBlock(byte[] in, int inOff, int inLen)
    throws InvalidCipherTextException {
80          if (key == null)
81              throw new IllegalStateException("Engine not
    initialized");
82
```

```
83          int maxLength = getInputBlockSize();
84
85          if (inLen < maxLength) {
86                  System.out.println("inLen: " +inLen + "; maxLength: " + maxLength);
87              throw new DataLengthException("Input too small for the cipher.");
88          }
89          return process(in, inOff, inLen);
90      }
91
92
93      public abstract void initialize();
94
95      public abstract byte[] process(byte[] in, int inOff, int inLen) throws InvalidCipherTextException;
96
97  }
```