An Identity Based Encryption System

Louise Owens Adam Duffy Tom Dowling

Crypto Group
Computer Science Dept.
NUI Maynooth
Co. Kildare, Ireland
e-mail: tdowling@cs.may.ie

Abstract

We describe an Identity Based Encryption (IBE) cryptosystem based on a scheme presented by Boneh and Franklin [3]. We implement the abstract mathematical concepts underlying this system. We reuse an existing Elliptic curve arithmetic API, [4] to reduce the development time of the IBE system. We present a Java Cryptographic Architecture (JCA) integrated implementation of IBE that will allow Java developers to easily take advantage of this new encryption system and thus eliminate some of the most serious practical problems associated with existing public key infrastructure systems. The JCA implementation facilitates the hiding of all the complicated mathematical details of the system and further demonstrates the utility of Java as security development language.

1 Introduction

In this paper we present an implementation of Identity Based Encryption (IBE). Traditional public key cryptosystems use very long integers, typically 1024 bits, as public keys. These systems rely on digital certificates to connect an identity like a person or a machine to a public key. The distribution, revocation, and constant checking of these certificates create many headaches for practical systems. They also slow down the process. IBE systems have the advantage that a public key is the identity, usually an arbitrary string like an email address. This removes the need for certificates and thus eliminates some of the practical problems associated with traditional systems. The paper is organized as follows. In section 2 we present a brief overview of IBE theory and the definitions of the various concepts used throughout the paper. Section 3 focuses on design issues of the IBE system. Implementation details are discussed in section 4. The embedding of our system into the JCA is discussed in section 5. Finally, section 6 will discuss conclusions and future work.

2 A Brief Overview of IBE

We present the main concepts of IBE here. A comprehensive account of IBE can be found in [3, 11]. The design of an Identity-Based Encryption system was a long-standing open problem in cryptography. The first secure and practical IBE system, was proposed by Boneh and Franklin in 2001. An IBE system can be very simply described as a public key encryption scheme in which the public key can be an arbitrary string. When Stephen sends mail to Cian at *Cian@myaddress.ie* he encrypts his message using the public key string *Cian@myaddress.ie*. There is no need for Stephen to obtain Cian's public key certificate. When Cian receives the encrypted mail he contacts a third party, called the Private Key Generator (PKG). Cian authenticates himself to the PKG in the same way he would authenticate himself to a Certification Authority (CA) and obtains his private key from the PKG. Cian can then use his private key to read his e-mail. Specifically an IBE scheme can be partitioned into four sections:

```
(1) Setup generates global system parameters and a master-key,
(2) Extract uses the master-key to generate the private key
    corresponding to an arbitrary public key string ID,
(3) Encrypt encrypts messages using the public key ID,
(4) Decrypt decrypts messages using the corresponding private key.
```

Our objectives in this paper do not cover a detailed security analysis of the system. Readers familiar with security can note that the performance of the cryptosystem is comparable to the performance of ElGamal encryption. The security of the system is based on a natural analogue of the Computational Diffie-Hellman assumption on elliptic curves.

We now present a simplified version of the IBE system. The Boneh-Franklin IBE system uses elliptic curves. The basic units for this cryptosystem are points (x, y) on an Elliptic curve, E, over a finite field, $\mathbf{F}_p$, denoted $E(\mathbf{F}_p)$, of the form

$$y^2 = x^3 + ax + b \text{ with } x, y, a, b \in \mathbf{F}_p = \{0, 1, 2, 3, \ldots, p-2, p-1\} \ .$$

We define abstract concepts of addition, $P + Q$, and scalar multiplication by an integer, αQ, on the points of $E(\mathbf{F}_p)$. We also define a special point at infinity, ∞. These definitions make $E(\mathbf{F}_p)$ suitable for cryptographic purposes. Mathematically we make $E(\mathbf{F}_p)$ a finite abelian group. Details of how these concepts are implemented appear in [2, 9]. The *order* of a point P is defined to be the smallest integer n such that $nP = \infty$. It is worth noting that it is a very difficult problem to find Q given αQ. We use the symbol $\oplus$ to denote bitwise exclusive or, XOR.
The inverse of an integer, a, in the finite field $\mathbf{F}_p$ is denoted by a^{-1} and defined by $a * a^{-1} = 1 \bmod p$. The concept of division in finite fields is equivalent to multiplication by an inverse i.e. $a \div b \bmod p \equiv a * b^{-1} \bmod p$. The IBE system makes heavy use of these and other operations defined on $E(\mathbf{F}_p)$. Fundamental to the system is the concept of a bilinear mapping. An example of such a mapping is the Weil pairing, $\hat{e}$, that takes two points on $E(\mathbf{F}_p)$, outputs an integer, and has the property that

$$\hat{e}(xP, yQ) = \hat{e}(P, Q)^{xy} \text{ for any points } P, Q \text{ and for any integers } x, y.$$

The master secret is some integer s known only to the private key generator. This number must be kept secret at all times. Let P be an arbitrary point on $E(\mathbf{F}_p)$. Let the message to be encrypted be M. The identity string is embedded onto a point Q on the curve. Q is the public key. The corresponding private key is the point sQ. To encrypt, the sender picks a random k and computes $g = \hat{e}(Q, sP)^k$. The sender then computes $M \oplus H(g)$, where $H(g)$ is the hash of g and sends the pair $(A, B) = (kP, M \oplus H(g))$ to the receiver. The receiver can then use s and kP to compute g since

$$\hat{e}(sQ, kP) = \hat{e}(Q, sP)^k = g$$

By computing $B \oplus H(g)$ the message is retrieved since

$$B \oplus H(g) = M \oplus H(g) \oplus H(g) = M$$

An adversary sees only P, sP, kP and Q, and it is difficult to compute g with just this information.

3 Design of IBE

We give an overview of some of the design issues involved in the IBE system. In order to minimise the implementation work it was decided to take advantage of an existing elliptic curve cryptosystem implementation, [4]. All the elliptic curve arithmetic can be handled by this API. A convenient algorithm for embedding strings onto points on elliptic curves appears in [3]. This algorithm requires a careful choice of elliptic curve, namely a super-singular curve of the form $y^2 = x^3 + 1$. The algorithm also requires a hashing algorithm. Implementations of common hashing algorithms are available in the *java.security* package. The underlying finite field choice is also important as the Weil pairing on $E(\mathbf{F}_p)$ requires points of particular orders to operate on. Specifically the Weil pairing requires a primes p such that $p = lq - 1$. These primes are generated

in the constructor using Gordon's algorithm [10]. The only user input into this process is to choose the bit length of q. The main focus of the design was to implement the Weil pairing. The algorithm proposed in [3] was chosen. The details are mathematically involved and are not discussed here. Note that the IBE system also allows the Tate pairing to be used as the bilinear mapping. Tate requires some technical changes but the property of bilinearity works the same as Weil.

4 Implementation of IBE

In this section we give implementation details of the IBE system. Extensive use of elliptic curve arithmetic is made throughout the IBE system. An elliptic curve arithmetic package is provided by *nuim.cs.crypto.ellipticcurve* [4]. We make particular use of two classes in this package, *AffinePoint* and *EllipticCurve* . Points on an elliptic curve are represented as *AffinePoint* objects while the elliptic curve itself is an *EllipticCurve* object. The *EllipticCurve* class contains the methods to add and scalar multiply *AffinePoint* objects. We set up an elliptic curve object as follows,

```
EllipticCurve c = new EllipticCurve(xCoeff,constCoeff,fieldSize);
```

We can retrieve the point at infinity, add and scalar multiply as follows,

```
AffinePoint infinity = c.pointAtInfinity();
AffinePoint P1plusP2 = c.add(P1,P2);
AffinePoint pubkey   = c.multiply(secretNumber,pointQ);
```

Hashing algorithms are required for the IBE process. Java provides us with a neat way to implement these algorithms by utilizing the *MessageDigest* class of *java.security*. To implement an $MD5$ hashing algorithm to hash an arbitrary length string we do the following,

```
MessageDigest mdHash = MessageDigest.getInstance( "MD5" );
mdHash.update(StringToBytes);   // Feed the Byte Array containing the string to the hash
byte[] raw = mdhash.digest();   // Compute the hash
BigInteger hashvalue=new BigInteger(raw);  //Convert to BigInteger for later embedding
```

Picking suitable field sizes involves producing large primes. We take advantage of the *BigInteger* class in *java.math* to provide these primes. The following segment generates a number of length *bitlength* with a probability of $2^{-certainty}$ of not being prime,

```
Random rnd = new Random()
s = new BigInteger(bitLength,certainty,rnd);
```

Division over finite fields occurs repeatedly in the algorithms computing the Weil pairing. As mentioned before this requires the calculation of an inverse. We use the *modInverse* method of *BigInteger* to compute these inverses. The following code segment computes the inverse mod q of a *BigInteger* b,

```
BigInteger binverse = b.modInverse(q);
```

Testing of the system was aided by computational examples in [12]. This source proved particularly useful when testing point order calculation algorithms and Weil pairing components of the system. A Java Archive (JAR) file of the elliptic curve arithmetic package and related documentation is available from the authors.

5 Embedding into JCA

In this section we discuss the integration of our IBE system into the Java Cryptographic Architecture, JCA. Firstly we give an overview of JCA. A comprehensive treatment of JCA appears in [8]. JCA is a

framework that specifies design patterns for designing cryptographic concepts and algorithms. For example any mathematical algorithm that performs encryption is called a Cipher. The JCA architecture separates concepts from their implementations. These concepts are encapsulated by classes in the `java.security` and `javax.crypto` packages. These classes are called concept classes. For example the concept of a Cipher is represented by the `javax.crypto.Cipher` concept class. JCA relies heavily on the factory method design pattern to supply instances of its concept classes. A factory method is basically a special kind of static method that returns an instance of a class. A thorough discussion of the factory method design pattern appears in [7]. The idea here is that a concept class is asked for an instance that implements a particular algorithm. This is accomplished using a `getInstance` factory method. The following code fragment demonstrates the process by producing an instance of the Cipher concept class that uses the Rinjdael algorithm:

```
Cipher cryptobjt = Cipher.getInstance("Rinjdael");
```

One major advantage of this set up is that to change the cryptographic algorithm used you need only change the argument in the `getInstance` method. We still have not explained how to implement the algorithms called via these factory methods. This implementation is accomplished by security providers. These providers are at the root of JCA. A provider is a collection of algorithm classes headed up by a `java.security.Provider` object. This object keeps a list of all algorithms supported by the security provider. If we wish to use our IBE cryptosystem in this way we must follow the JCA specification and design a security provider to contain our algorithm.

The IBE cryptosystem is provided by the `IbeProvider` class (located in `nuim.cs.crypto.ibe` package) and is created as follows;

```
Provider provider = new IbeProvider();
```

Our IBE cryptosystem contains system parameters that specify the key size, the type of hashing algorithm - MD5, SHA, etc. - to be used and the bilinear mapping - Tate or Weil Pairing - central to the IBE cryptosystem. Before using IBE we may specify these parameters;

```
// specify key size of 128 bits
int keysize = 128;
// use Weil Pairing for the bilinear mapping
BilinearMap map = new WeilPairing();
// use MD5 hashing algorithm
MessageDigest hash = MessageDigest.getInstance( "MD5" );
AlgorithmParameterSpec parameters =
    new IbeSystemParameters( keysize, map, hash );
```

The system parameters may be changed to increase the key size, use alternative bilinear mappings such as Tate Pairing and use other hashing algorithms such as SHA. The key size may not be less than 128 bits in keeping with the advice of [6] to "*make* (software) *secure by default.*" For ease of use we also provide a default constructor that uses the settings above;

```
AlgorithmParameterSpec parameters = new IbeSystemParameters();
```

The parameters for the key pair generation are the same as the system parameters but include the identity string being used, for example, *cian@myaddress.ie*. The class `IbeKeyParameters` is a subclass of `IbeSystemParameters` so only one object need be created and used for both key pair generation and encryption and decryption.

```
// keysize, map and hash values as set previously
// specify the identity
String identity = "Cian@myaddress.ie"
// create the parameters for key pair generation
AlgorithmParameterSpec parameters =
    new IbeKeyParameters( identity, keysize, map, hash );
```

All other parameters required for the IBE cryptosystem can be generated internally reducing the need for user interaction.

To use the IBE cryptosystem a `KeyGenerator` must be created and initialized (using the system parameters) and a private and public key generated as follows

```
// create the IBE key generator
KeyPairGenerator kpg =
    KeyPairGenerator.getInstance( IbeProvider.IBE, provider );
// initialise the key generator
kpg.initialize( parameters, new SecureRandom() );
// generate the key pair
KeyPair keyPair = kpg.generateKeyPair();
// get the private key from the key pair
PrivateKey privateKey = keyPair.getPrivate();
// get the public key from the key pair
PublicKey publicKey = keyPair.getPublic();
```

Once the keys have been generated then the cipher may be initialized and used to encrypt or decrypt messages as appropriate

```
// create the IBE cipher
Cipher cipher = Cipher.getInstance( IbeProvider.IBE, provider );

// initialise the cipher using the system parameters
// and prepare it for encrytion
cipher.init( Cipher.ENCRYPT_MODE, publicKey, parameters,
    new SecureRandom() );
// encrypt the plaintext into ciphertext
byte ciphertext[] = cipher.doFinal( plaintext );

// initialise the cipher using the system parameters
// and prepare it for decryption
cipher.init( Cipher.DECRYPT_MODE, privateKey, parameters,
    new SecureRandom() );
// decrypt the ciphertext into plaintext
byte plaintext[] = cipher.doFinal( ciphertext );
```

It is worth noting that any users intending to develop providers compliant with the JCA must obtain a certificate from Sun Microsystems Corp. in order to be accepted by the JCA.

6 Conclusions and future work

We have presented a Java IBE system. We have demonstrated the suitability of Java as a language for developing easy to use and easy to integrate security products. We have significantly reduced development time by reusing an existing elliptic curve arithmetic package. Future work will involve timings of the various candidate algorithms for the different IBE components to determine the optimal performance. A good account of potential speedups appear in [1]. We also hope to extend our basic IBE implementation to include many other IBE based applications like public key revocation systems, IBE signature schemes and Key-escrow to mention a few.
We also intend to extend our system into other areas of mathematics and security. In particular the Weil pairing component and the underlying elliptic curve arithmetic can be used as cryptanalytic tools to attack certain elliptic curve cryptosystems. Details of the attack appear in [2]. Note the code is freely available from our crypto group website www.crypto.cs.may.ie.

References

[1] Barreto, P., Kim, H.Y., Lynn, B., and Scott, M. *Efficient Algorithms for Pairing-Based Cryptosystems* Advances in Cryptology – Crypto'2002, Lecture Notes in Computer Science 2442, Springer-Verlag (2002), pp. 354–368.

[2] Blake, I., Seroussi, G., and Smart, N. *Elliptic Curves in Cryptography* Cambridge University Press, 1st edition, 1999.

[3] Boneh, D. and Franklin, M. *Identity Based Encryption from the Weil pairing* SIAM J. of Computing, Vol. 32, No. 3, pp. 586-615, 2003.

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

[5] Crandall, R., and Pomerance, C. *Prime Numbers, A Computational Perspective* Springer, 1st edition, 2001.

[6] Ferguson, N. and Schneier, B *Practical Cryptography* Wiley, 2003

[7] Gamma, E., Helm, R., Johnson, R., and Vlissides J. *Design Patterns, Elements of Reusable Object Oriented Software* Addison Wesley, 1st edition, 1994.

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

[9] Koblitz, N. *A Course in Number Theory and Cryptography* Springer, 2nd edition, 1994.

[10] Menezes, A et al. *Handbook of Applied Cryptography* CRC, 1997.

[11] Shamir, A. *Identity-based cryptosystems and signature schemes* Advances in Cryptology-Crypto '84, Lecture Notes in Computer Science, Vol 196, Springer-Verlag, pp 47-53, 1984.

[12] Washington, L. *Elliptic Curves, number theory and cryptography*
Chapman and Hall/CRC, 2003