Paillier Threshold Encryption Toolbox

October 23, 2010

1 Introduction

Following a desire for secure (encrypted) multiparty computation, the University of Texas at Dallas Data Security and Privacy Lab created the `paillierp` Java packages. By using a threshold variant of the Paillier Encryption scheme as laid out in [2], we use the methods described in [1] to allow secure multiparty computation. This implementation is heavily Object Oriented, having separate objects for (1) encryption/decryption environments, (2) keys, and (3) zero knowledge proofs. As the threshold variant of Paillier is an improvement on the original scheme, the original Paillier encryption scheme is included and is a superclass of the threshold version.

1.1 Paillier's Encryption Scheme

Paillier's cryptosystem is a probabilistic encryption scheme wit a public key of an RSA modulus n. The plaintex space is $\mathbb{Z}_n$ and the ciphertext space is $\mathbb{Z}_{n^2}$. As Damgård introduces

> Paillier's Encryption scheme is a probabilistic encryption scheme based on computations in the group $\mathbb{Z}^*_{n^2}$, where n is an RSA modulus. This scheme has some very attractive properties, in that it is homomorphic, allows encryption of many bits in one operation with a constant expansion factor, and allows efficient decryption.[2]

Addition of ciphertexts is trivially easy, and multiplying ciphertexts by a constant is also simple.

1.2 Paillier's Threshold Variant

Working from Shoup's threshold version of RSA in [4], Damgård and Jurik propose in [2] a threshold version of Paillier's encryption scheme. Threshold encryption requires a pre-determined number of decryption servers to collaborate on fully decrypting a message.

Any collaboration between fewer than the specified number of decryption servers does not result in a complete decryption.

When creating keys, variables w and l are chosen, resulting in 1 public key and l private keys being generated, set with a threshold of w. It is necessary for at least w of these keys to collaborate to decrypt a message encoded with the public key.

The process for decryption involves at least w decryption servers to obtain the same ciphertext, and then apply their private key for decryption to produce a "partial decryption" or "share", each partial decryption being shared with the remaining servers. Once a server obtains at least w unique shares, that server can obtain the original plaintext.

Damgård and Jurik also proposed a generalization of the Paillier cryptosystem, allowing ciphertexts to range in $\mathbb{Z}_{n^{s+1}}$ for any $s \geq 1$. Such a feature can even allows the receiver to decide on s upon encryption and so can allow variable block lengths. In this implementation, we are more concerned with multiparty computations and so *fix s to be one* at every step.

1.3 Multiparty Computation

Cramer, Damgård, and Nielsen suggests a method of multiparty computation based on homomorphic threshold cryptosystems [1]. Their method uses such a cryptosystem to create a Boolean circuit to compute secure functions. The requirements for such computation secure against active adversary is a cryptosystem that satisfies the following properties:

1. **Addition of plaintexts** From encryptions $\overline{a}$ and $\overline{b}$ (of plaintexts a and b, respectively), it is easy to compute an encryption of $a + b$.

2. **Multiplication by a constant** From encryption $\overline{a}$ and a constant $\alpha \in \mathbb{Z}_n$, it is easy to compute a random encryption of $\alpha \cdot a$.

3. **Proving knowledge of plaintext** If a process created an encryption $\overline{a}$, it is easy to give a zero-knowledge proof that it knows a.

4. **Proving knowledge of multiplication** If a process created an encryption of $\alpha \cdot a$ and broadcasts the encryptions $\overline{\alpha}$ and $\overline{a}$, it can give a zero-knowledge proof that the decryption of that encryption is in deed the product of the decryptions of $\overline{\alpha}$ and $\overline{a}$.

5. **Threshold decryption** Given an encryption $\overline{a}$ and the corresponding public key used to encrypt it, each process could use their private key and share their partial decryptions so that each can securely compute a.

Cramer, et al. provided exactly those zero-knowledge protocols for the threshold variant of Paillier, making multiparty computation possible using Paillier.

1.4 The Packages

The intention of this suite is to provide the tools necessary to enable multiparty computation based on the threshold variant of the Paillier cryptosystem. The requirements enumerated above for such computation are all satisfied in Paillier and are provided to the user of this package.

Three packages are included in this suite, namely:

`paillierp` which includes the encryption/decryption environments.

`paillierp.keys` which includes key objects needed for encryption and decryption.

`paillierp.zkp` which include non-interactive zero-knowledge proofs to prove knowledge of a plaintext or multiplication or partial decryption.

Most numbers here are `BigInteger` objects, capable of representing numbers up to $2^{2^{32}-1}$. This includes plaintexts, ciphertexts, and the like.

1.5 What is this document?

This document is a manual to the `paillierp` packages. While the Javadoc for the implementation is rather thorough, this will provide a quick resource for basic usage without specific mathematics.

2 `paillierp.keys` Package - Keys

In this cryptosystem, the primary step is key creation. Public keys are required for encryption, and private keys are required for decryption. Public keys are derivable from the private keys.

2.1 `paillierp.keys.KeyGen` - Key creation

In this `static` class, one can generate a single private key for the original Paillier encryption scheme, or l partial private keys for the threshold variant of the same cryptosystem.

`PaillierKey(int s, long seed)` is used to create a random `PaillierPrivateKey` of length `s`. Note that a `seed` is necessary for the randomness.

`PaillierThresholdKey(int s, int l, int w, long seed)` is used to create `l` random `PaillierPrivateThresholdKey`s of length `s`, of which `w` must collaborate to produce a decryption.

These two methods are the most necessary for random key generation. A further method to eliminate randomness is also provided.

2.2 `paillierp.keys.PaillierKey` and `paillierp.keys.PaillierPrivateKey`

As in the original version of the Paillier cryptosystem, the key is generated by choosing a RSA modulus $n = pq$ of k bits for p, q prime. Originally, an element $g \in \mathbb{Z}^*_{n^2}$ is also chosen, but we fix g to be $1 + n$ without loss of security (as given in [2]). The private key is a value d which is the least common multiple of $p - 1$ and $q - 1$.

While the choosing of these random numbers is left to `paillierp.keys.KeyGen`, the class `PaillierKey` holds the necessary values for a public key, `PaillierPrivateKey` for a private key. The constructors vary in arguments to provide flexibility. For the public key, the constructors are

`PaillierKey(BigInteger n, long seed)` Creates a new public key when given the modulus `n`.

`PaillierKey(BigInteger p, BigInteger q, long seed)` Creates a new public key from a given two odd primes `p` and `q`.

`PaillierPrivateKey(BigInteger n, BigInteger d, long seed)` Creates a new private key when given the modulus `n` and the secret value `d`.

`PaillierPrivateKey(BigInteger p, BigInteger q, BigInteger d, long seed)` Creates a new private key when given the primes `p` and `q` and the secret value `d`.

2.3 `paillierp.keys.PaillierThresholdKey` and `paillierp.keys.PaillierThresholdPrivateKey`

In the Threshold variant, the key is generated by finding four unique primes, p, q, p', q' such that $p = 2p' + 1$ and $q = 2q' + 1$. We let $n = pq$ and $m = p'q'$. We picked d to be equivalent to 0 mod m and 1 mod n.

From this, we generate a polynomial $P(x)$ with random coefficients, hiding $d = P(0)$ as the secret key and $s_i = P(i)$ for the ith partial key. These partial keys are distributed in secrecy. In addition, public verification values are generated: the value v and a value $v_i = v^{\Delta s_i} \bmod n^2$, where $\Delta = l!$.

Only n is needed for encryption, but the further public values are included in the public threshold key. A share key requires the entire set of public values as well as s_i and n. These values are necessary for each partial decryption.

`PaillierThresholdKey(BigInteger n, int l, int w, BigInteger v, BigInteger[] viarray, long seed)` Creates a new public key for the generalized Paillier threshold scheme from the given modulus `n`, for use on `l` decryption servers, `w` of which are needed to decrypt any message encrypted by using this public key.

`PaillierPrivateThresholdKey(BigInteger n, int l, int w, BigInteger v, BigInteger[] viarray, BigInteger si, int i, long seed)` Creates a new private key for the generalized Paillier threshold scheme from the given modulus `n`, for use on `l` decryption servers, `w` of which are needed to decrypt any message encrypted by using this public key. The private value `si` is for this particular key ID `i`.

Other constructors are available which include the `combineSharesConstant`. That parameter is available for efficiencey at key distribution, but it can be computed by the other values.

3 `paillierp` Package - Encryption environments

The `paillierp` package includes two encryption/decryption environments (one for the original Paillier scheme, another for the threshold variant), and two utility classes used by both the Zero Knowledge Proofs and Keys. Each encryption environment is a class holding the key which allows encryption, decryption, the addition and multiplication of ciphertexts. We provide a class of byte utilities for later transmitting keys and proofs, and a partial decryption class for identifying partial decryptions ("shares").

3.1 `paillierp.Paillier` - Original Paillier Encryption Environment

This class provides an environment for encrypting and decrypting, setting and retrieving keys. There is a number of constructors allowing for automatically setting encryption or encryption and decryption, depending if it sees a public or private key.

The environment will store a public key (for encryption) and a private key (for decryption). To assign the public key, call `setEncryption( PaillierKey )`. To assign the private key, call `setDecryption( PaillierPrivateKey )`, or to assign both for this case, call `setEncryptionDecryption( PaillierPrivateKey )`

3.1.1 Encryption

Once a public key has been assigned to the encryption environment, whether by the constructor or by `setEncryption` or `setEncryptionDecryption`, it is possible to call `encrypt( BigInteger m )` to encode a message `m`. The resulting `BigInteger` is the encrypted message.

Other `encrypt` methods are available, both `static` and otherwise, and are listed in the online documentation.

3.1.2 Decryption

Once a private key has been assigned to the encryption environment, whether by the constructor or by `setDecryption` or `setEncryptionDecryption`, it is possible to call `decrypt( BigInteger c )` to decode a ciphertext `c`. The resulting `BigInteger` is the original message.

Just like above, other `decrypt` are available for greater flexibility. There are `static` and non-static methods, all of which are listed in the online documentation.

3.1.3 Other Utilities

As listed in Section 1.3, there are several attractive features that we would like to make available to allow some computation. In the `Paillier` encryption environment, there is no thresholding, making it impossible to provide the 5th feature, but features 1-4 are available to this regular encryption environment.

1. **Addition of Plaintexts** is provided by `add( BigInteger c1, BigInteger c2 )`. Given `c1` the encryption of a and `c2` the encryption of b, the resulting `BigInteger` is an encryption of $a + b$.
2. **Multiplication by a constant** is provided by `multiply( BigInteger c, BigInteger cons )`. Given `c` the encryption of a, the resulting `BigInteger` is an encryption of `cons` $\cdot a$.
3. **Proving knowledge of plaintext** is provided by `encryptProof( BigInteger )`. The result is a non-interactive Zero Knowledge Proof detailed in **??**.
4. **Proving knowledge of multiplication** is provided by `multiply Proof( BigInteger , BigInteger )` to produce a non-interactive Zero-Knowledge Proof as detailed in **??**.

`static` variants of the addition and multiplication methods are also available. Also available are helpful utilities for random encryptions of 1 and of zero, and a method that will randomize a ciphertext at no distortion of the original message.

3.2 `paillierp.PaillierThreshold` - Threshold Paillier Encryption Environment

The threshold encryption environment provides an object through which one can encrypt and decrypt. As in `Paillier`, there is a number of constructors to ensure maximum flexibility.

3.2.1 Encryption

Encryption with `PaillierThreshold` is just the same as with `Paillier` in Section 3.1.1.

3.2.2 Decryption

The decryption methods for `PaillierThreshold` differ from `Paillier`.

Once a private key has been assigned to the encryption environment, whetherby the custroctor or by `setDecryption` or `setEncryptionDecryption`, it is possible to call `decrypt( BigInteger c )` to decode a ciphertext `c`. The result is a `PartialDecryption`, which is **not** the original message. Rather, one must provide many `PartialDecryption` objects to `combineShares( PartialDecryption... )` to obtain a `BigInteger` which is the original message; at least `deckey.getW()` must be provided, where `deckey` is the private key.

In order to fully decrypt a ciphertext `c`, the following must happen

1. One should partially decrypt a message `c` by invoking `decrypt( c )`. What results is a `PartialDecryption`.

2. One must obtain at least `w` separate such `PartialDecryption`s generated by `w` independent `PaillierThreshold` encryption environments with unique Paillier Threshold Private Keys.

3. Combine the at least `w` partial decryptions by calling `combineShares(` share$_1$`,` share$_2$`, ...,` share$_w$`)`. The result is a `BigInteger` which is the original message.

3.2.3 Other Utilities

As above in 3.1.3, `PaillierThreshold` implements the full set of features needed for secure multiparty computation.

1. **Addition of Plaintexts** is provided by `add( BigInteger c1, BigInteger c2 )`. Given `c1` the encryption of a and `c2` the encryption of b, the resulting `BigInteger` is an encryption of $a + b$.

2. **Multiplication by a constant** is provided by `multiply( BigInteger c, BigInteger cons )`. Given `c` the encryption of a, the resulting `BigInteger` is an encryption of `cons` $\cdot a$.

3. **Proving knowledge of plaintext** is provided by `encryptProof( BigInteger )`. The result is a non-interactive Zero Knowledge Proof detailed in Section 4.

4. **Proving knowledge of multiplication** is provided by `multiply Proof( BigInteger , BigInteger )` to produce a non-interactive Zero-Knowledge Proof as detailed in Section 4.

5. **Threshold decryption** is provided by `decrypt` and `combineShares` as detailed above in 3.2.2. Further, **proving knowledge of partial decryption** is provided by `decryptProof( BigInteger )`.

3.3 `paillierp.ByteUtils` - Byte Manipulation Utilities

`ByteUtils` is a bundle of `static`, byte manipulation methods for the `toByteArray` methods as listed in Section 5.

3.4 `paillierp.PartialDecryption` - Product of Threshold Decryption

This is simply a wrapper of a `BigInteger` and an ID number for the purposes of decryption with `PaillierThreshold` as detailed in Section 3.2.2.

4 `paillierp.zkp` Package - Non-interactive Zero Knowledge Proofs

For secure multiparty computation, Cramer, et al. in [1] requires a cryptosystem to have five capabilities as listed on p. 4 in [1] (reproduced in this documentation at Section 1.3). To ensure complete security in these computations, even against malicious attacks, they have required two proofs of computation, viz. a proof that one knows the plaintext for a ciphertext and a proof that one knows a constant and has multiplied a ciphertext by that constant. A further proof has been developed to provide proof that one has indeed partially decrypted a ciphertext.

Zero Knowledge Proofs is a method for one party to prove the veracity of a mathematical statements, without revealing particular numbers of that statement. Standardly, a zero knowledge proof is an interactive protocol for proving the veracity of the statement, where an input is given from without. On the other hand, a *non-interactive* proof is one which automatically chooses the input by something just as random as the possible input from without: the automatic input is the hash of the proving variables.

Each of the original values are required in the constructors. That is to say, for example, `EncryptionZKP` requires knowledge of the plaintext and the private key before generating a zero knowledge proof. *Nothing of sensitive data will be saved.* But for the paranoid, methods and constructors for saving these objects as `byte` arrays are available.

Each object generates the new ciphertext or partial decryption retrievable by `getValue()`, and the method `verify()` recomputes the hash to provide veracity of the values. Also, `verifyKey( Paillier(Threshold)Key )` is available to ensure all public values used in the proof correspond with the given key.

EncryptionZKP This provides knowledge that one knows the plaintext of a given encryption. This protocol is described on page 40 of [1].

MultiplicationZKP This provides knowledge that one has multiplied a ciphertext by a constant which only the multiplier knows. The protocol is from page 40 of [1].

DecryptionZKP This provides knowledge that one has indeed partially decrypted a ciphertext. The method is found on pages 16-17 of [3].

5 Transferring information

Transferring of both public and private keys, and of the `PartialDecryption` wrapper and of each zero knowledge proof is made easy in this package. Not only are they `serializable`, but they also are given a `toByteArray()` method to specifically encode the object as a `byte` array for communication. Each of the above objects are also given a constructor which takes as input the byte array as specifically engineered by that object's `toByteArray()`.

6 Example

We close with an example of the `paillierp` package, particularly `PaillierThreshold`.

```java
package testingPaillier;

import paillierp.Paillier;
import paillierp.PaillierThreshold;

import java.math.BigInteger;
import java.util.Random;

import paillierp.key.KeyGen;
import paillierp.key.PaillierPrivateThresholdKey;
import paillierp.zkp.DecryptionZKP;

public class Testing {
  public static void main(String[] args) {

    System.out.println("Create new keypairs.");
    Random rnd = new Random();
    PaillierPrivateThresholdKey[] keys =
      KeyGen.PaillierThresholdKey(128, 6, 3, rnd.nextLong());
    System.out.println("Six keys are generated, with a threshold of 3.");

    System.out.println("Six people use their keys: p1, p2, p3, p4, p5, p6")
        ;
    PaillierThreshold p1 = new PaillierThreshold(keys[0]);
    PaillierThreshold p2 = new PaillierThreshold(keys[1]);
    PaillierThreshold p3 = new PaillierThreshold(keys[2]);
    PaillierThreshold p4 = new PaillierThreshold(keys[3]);
    PaillierThreshold p5 = new PaillierThreshold(keys[4]);
    PaillierThreshold p6 = new PaillierThreshold(keys[5]);

    System.out.println("Alice is given the public key.");
    Paillier alice = new Paillier(keys[0].getPublicKey());
```

```
// Alice encrypts a message
BigInteger msg = BigInteger.valueOf(135819283);
BigInteger Emsg = alice.encrypt(msg);
System.out.println("Alice encrypts the message "+msg+" and sends "+
    Emsg+" to everyone.");
// Alice sends Emsg to everyone

System.out.println("p1 receives the message and tries to decrypt all
   alone:");
BigInteger p1decrypt = p1.decryptOnly(Emsg);
if (p1decrypt.equals(msg)) {
  System.out.println("p1 succeeds decrypting the message all alone.");
} else {
  System.out.println("p1 fails decrypting the message all alone. :(");
}

System.out.println("p2 and p3 receive the message and " +
    "create a partial decryptions.");
DecryptionZKP p2share = p2.decryptProof(Emsg);
DecryptionZKP p3share = p3.decryptProof(Emsg);
// p2 sends the partial decryption to p3
// p3 sends the partial decryption to p2

System.out.println("p2 receives the partial p3's partial decryption " +
    "and attempts to decrypt the whole message using its own " +
    "share twice");
try {
BigInteger p2decrypt = p2.combineShares(p2share, p3share, p2share);
  if (p2decrypt.equals(msg)) {
    System.out.println("p2 succeeds decrypting the message with p3.");
  } else {
    System.out.println("p2 fails decrypting the message with p3. :(");
  }
} catch (IllegalArgumentException e) {
  System.out.println("p2 fails decrypting and throws an error");
}

System.out.println("p4, p5, p6 receive Alice's original message and " +
    "create partial decryptions.");
DecryptionZKP p4share = p4.decryptProof(Emsg);
DecryptionZKP p5share = p5.decryptProof(Emsg);
DecryptionZKP p6share = p6.decryptProof(Emsg);
// p4, p5, and p6 share each of their partial decryptions with each
   other

System.out.println("p4 receives and combines each partial decryption" +
    " to decrypt whole message:");
BigInteger p4decrypt = p4.combineShares(p4share, p5share, p6share);
if (p4decrypt.equals(msg)) {
      System.out.println("p4 succeeds decrypting the message with p5 and p6
          .");
    } else {
      System.out.println("p4 fails decrypting the message with p5 and p6.
          :(");
    }
  }
}
```

References

[1] Cramer, R., Damgård, I., and Nielsen, J. B. Multiparty computation from threshold homomorphic encryption. In *EUROCRYPT '01: Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques* (London, UK, 2001), Springer-Verlag, pp. 280–299.

[2] Damgård, I., and Jurik, M. A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In *PKC '01: Proceedings of the 4th International Workshop on Practice and Theory in Public Key Cryptography* (London, UK, 2001), Springer-Verlag, pp. 119–136.

[3] Damgård, I., Jurik, M., and Nielsen, J. B. A generalization of Paillier's public-key system with applications to electronic voting. `http://www.brics.dk/~ivan/GenPaillierfinaljour.ps`, 2003. Manuscript.

[4] Shoup, V. Practical threshold signatures. In *EUROCRYPT'00: Proceedings of the 19th international conference on Theory and application of cryptographic techniques* (Berlin, Heidelberg, 2000), Springer-Verlag, pp. 207–220.