Homomorphic Encryption with Applications

Petar Stefanov

Final Year Project
B.Sc. Single Honours in Computer Science

NUI MAYNOOTH
Ollscoil na hÉireann Má Nuad

Department of Computer Science
National University of Ireland, Maynooth
Co. Kildare
Ireland

Supervisor: Dr. T. Dowling
Date: March 25, 2013

Declaration

I hereby certify that this material, which I now submit for assessment on the program of study leading to the award of B.Sc. Single Honours in Computer Science and Software Engineering, is entirely my own work and has not been taken from the work of others - save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: Date:

Acknowledgements

I would first like to thank Dr. Tom Dowling for working with me for over year a half and helping me come up with a finished project. I especially appreciate his help as I worked through many topics, trying to find the right one, and his patience in answering all my different questions. Second, I want to thank my family for supporting me through this entire experience for the past four years. Without the help of those above I do not know how I would have completed my education and the quality project that is before you.

Contents

Abstract

The purpose of this project is to study homomorphic cryptography scheme. New and promising scheme have appeared in the last 13 years, generating interest. I will be examining the Paillier Cryptosystem in depth. This is one of the few known system that preserves additive homomorphic properties. This thesis explains the mathematics behind the schemes along with their inherent benefits, while also suggesting some potential applications. It is also a Software Engineering project that utilises a framework structure in Java to enable easier development of applications of core cryptographic algorithms. In essence it allows competent Java developers to ability to include powerful cryptography with minimal impact to their development system. In particular the framework allows access to Pailler encryption with a little as one line of Java code.

Chapter 1

Introduction

1.1 Cryptography

Cryptography is a branch of mathematics. Modern cryptography is based on mathematical theory and computer science practice. Most cryptographic algorithms are designed around computational hardness assumptions, making such algorithms hard to break by any adversary. It is theoretically possible to break such a system but it is practically infeasible to do so by any known practical means.

Cryptography has been used for centuries to communicate or store secret information while keeping that information secret from third parties.
Claude Shannon build the foundations for modern cryptography in 1948 with his Information Theory. The development of digital computers made it possible to create encryption algorithms, far more complex than before. The new computer algorithms were no longer based on the simple substitution and fractioning of letters and words, but on a large number of complex operations. Until the middle of 1970, all encryption was based on symmetric-key algorithms. Both encryption and decryption performed with the same key. The disadvantage of this system was a complex key distribution system with lot of security issues.
The most important development for modern cryptography was public-key cryptogra-

phy. With public-key cryptography, two keys are used. A public key for encryption only, and a secret private key for decryption. The public key cannot be used to decrypt the information. This solved the expensive and risky problem of secret key distribution. You could make your public key available to everyone. Everyone can encrypt his message, destined to you, with your public key, but only you, with your private secret key, can decrypt the message. There is no longer a need to share a secret key! In present days public-key cryptography model is the model that drives secure E-Commerce, because of it's simple availability to large numbers of people and the achievement of confidentiality. Public-key algorithms are based on the computational complexity problem. For example Diffie-Hellman, ElGamal algorithms are based on the discrete logarithm problem and RSA is based on the problem of factorization of large numbers. Paillier algorithm is based on Decisional Composite Residuosity Assumption (stronger than factorization).

1.2 Paillier homomorphic scheme

The Paillier scheme was first published by Pascal Paillier in 1999[1]. This probabilistic scheme has generated a good amount of interest and further study since it was discovered. The main interest is centred around another property it possesses; the additive homomorphic property allows this scheme to do simple addition operations on several encrypted values and obtain the encrypted sum. The encrypted sum can later be decrypted without ever knowing the values that made up the sum. The scheme has been suggested for use in the design of voting protocols, threshold cryptosystems, watermarking, secret sharing schemes, private information retrieval and is a possible solution for improving the security around the Cloud computing.

Homomorphic encryption is a form of encryption which allows specific types of computations to be carried out on ciphertext and obtain an encrypted result which is the ciphertext of the result of operations performed on the plaintext. For instance, one person could add two or more encrypted numbers and then another person could decrypt the result, without either of them being able to find the value of the individual numbers.

The homomorphic property of this cryptosystem can be used to create secure voting systems, collision-resistant hash functions, private information retrieval schemes and enable widespread use of cloud computing by ensuring the confidentiality of processed data.

1.3 Java and Java Cryptography Architecture(JCA)

Java was chosen as the developing language for the following reasons.

- First, Java combines the properties that it is suitable for the development of large projects, and that it also enables easy and quick development (due to features such as a large standard library, garbage collection, and no pointer logic).
- Second, Java is platform-independent and can be run on multiple operating systems and platforms. Any computer program written in Java can be used in an operating system other than the one in which it was created without requiring major rework.
- Third, there exists very good cryptographic libraries implemented in Java which can be extended and integrated to any project. The Java Cryptography Architecture (JCA) is a framework for working with cryptography using the Java programming language. It forms part of the Java security API. The JCA uses a "provider"-based architecture and contains a set of APIs for various purposes, such as encryption, key generation and management, secure random number generation, certificate validation, etc.. These APIs are designed to provide an easy way for developers to integrate security into application code. Note - There are some restrictions on the type of cryptographic functionality that could be made available internationally in the JDK.

1.4 Project planning

From the beginning, a plan was set up to fullfill the necessary requirements of the project.

- Initial reading -to read through several articles and papers, to cover branch of Number theory, mathematics behind the cryptography in general.
- Develop understanding of Homomorphic encryption in particular Paillier Cryptosystem.
- Investigate applications of Paillier Cryptosystem and to decide what could be implemented to show the homomorphism the system possesses.
- Implementation - planning, design and implementation.
- Testing the system.
- Writing down the report using LaTeX.

Due to the extra work performed in the implementation step and the introduction of JCA Provider, the initial project plan had to change. See difference in Appendix B Figure B.1 and Figure B.2.

1.5 Thesis structure

The thesis is structured as follows:

Chapter 2. Overview of mathematical background behind the Paillier scheme.

Chapter 3. Paillier Cryptosystem, how it works? Key pair generation, encryption and decryption processes. Homomorphic properties of this scheme and overview of these properties.

Chapter 4. Overview of Java Cryptography Architecture (JCA) framework and its requirements. Implementation of basic Paillier algorithm. Design and implementation of JCA Provider that implement the Paillier algorithm.

Chapter 5. Implementation of E-Voting system based on Paillier scheme and application of a homomorphic cryptosystem in Cloud computing.

Chapter 6.	Testing the system and managing the quality of code.
Chapter 7.	Conclusions are drawn based on the research and the final opinions are stated.
Appendix	Uml class diagrams, Project software planning Gantt chart and applications screen shots.

Chapter 2

Mathematical Background

A detailed description of the background mathematics of this work appears in[2].

2.1 Maths behind Paillier scheme

The Paillier Cryptosystem is defined over the group $\mathbb{Z}_{n^2}^*$. In general the group structure over the set $\{0, ..., n-1\}$ with respect to multiplication modulo n define, for $n > 0$, the set

$$\mathbb{Z}_n^* \equiv \{a \in \{1, \ldots, n-1\} | gcd(a, n) = 1\};$$

$\mathbb{Z}_n^*$ consists of integers that are relatively prime to n. It can be shown that $\mathbb{Z}_n \times \mathbb{Z}_n^*$ is isomorphic to $\mathbb{Z}_{n^2}^*$, with isomorphism given by $f(a, b) = [(1+n)^a * b^n \bmod n^2]$, and it is convenient to work with $\mathbb{Z}_n \times \mathbb{Z}_n^*$ [2]. This gives us the following definition from [1].

Definition 1. A number z is said to be an n^{th} residue modulo n^2 if there exists a number $y \in \mathbb{Z}_{n^2}^*$ such that

$$z \equiv y^n \bmod n^2$$

The problem of trying to distinguish n^{th} residues from non$-n^{th}$ residues is seen as hard enough to form the basis of a cryptographic scheme. As for prime residuosity, deciding n^{th} residuosity is believed to be computationally hard [10]. There exists no polynomial time distinguisher for n^{th} residues modulo n^2 , i.e. deciding n^{th} residuosity is intractable. Currently, like RSA and others, there is no polynomial time algorithm to solve the Paillier algorithm without the private key. First there are two functions that we will discuss, since they are important in understanding how Pailliers scheme works. The first is the Euler totient function, which is generally denoted as $\phi(n)$. The function is defined [1]:

Euler Totient Function. For n the product of two primes p and q,

$$\phi(n) = (p-1)(q-1)$$

The function $\phi(n)$ is an arithmetic function that counts the number of positive integers less than or equal to n that are relatively prime to n. That is, if n is a positive integer, then $\phi(n)$ is the number of integers k in the range $1 \leq k \leq n$ for which $\gcd(n, k) = 1$. It gives the order of the multiplicative group $\mathbb{Z}_n^*$ of integers modulo n. For example let $n = 9$. Then gcd(9, 3) = gcd(9, 6) = 3 and gcd(9, 9) = 9. The other six numbers in the range $1 \leq k \leq 9$, that is, 1, 2, 4, 5, 7 and 8, are relatively prime to 9. Therefore, $\phi(9) = 6$.

The second function is the Carmichael's function, which seems much less known and is generally denoted as $\lambda(n)$ The function is defined as follows with some interesting properties [1]:

Carmichael's Function . For n the product of two primes p and q,

$$\lambda(n) = lcm(p-1, q-1)$$

Useful property of Carmichael's function is, for any

$$w \in \mathbb{Z}^*_{n^2} \begin{cases} w^\lambda &= 1 \bmod n \\ w^{\lambda n} &= 1 \bmod n^2 \end{cases}$$

In other words the smallest positive integer m such that $a^m \equiv 1 \pmod{n}$ for every integer a that is coprime to n, it defines the exponent of the multiplicative group $\mathbb{Z}^*_n$ of integers modulo n.
A very useful result is the following: $\mathbb{Z}_n \times \mathbb{Z}^*_n$ is isomorphic to $\mathbb{Z}^*_{n^2}$, with isomorphism given by $f(a, b) = [(1+n)^a * b^n \bmod n^2]$. A consequence is that a random element $y \in \mathbb{Z}^*_{n^2}$ corresponds to a random element $(a, b) \in \mathbb{Z}_n \times \mathbb{Z}^*_n$, or in other words , an element (a, b) with random $a \in \mathbb{Z}_n$ and random $b \in \mathbb{Z}^*_n$ [2]. This proposition is quite important as the encryption process discussed latter in Chapter.3 taking place in $\mathbb{Z}_n \times \mathbb{Z}^*_n$. In fact it takes place in the isomorphic group $\mathbb{Z}^*_{n^2}$.

What gives Paillier its security? The hard problem behind Paillier: Decisional Composite Residuosity Assumption - DCRA is a mathematical assumption used in cryptography. In particular, the assumption is used in the proof of the Paillier cryptosystem. Informally the DCRA states that given a composite n and an integer z, it is hard to decide whether z is a n^{th} residue modulo n^2 or not, i.e., whether there exists y such that:

$$z \equiv y^n \pmod{n^2}$$

Due to the random-self-reducibility (which is the rule that a good algorithm for the average case implies a good algorithm for the worst case, the ability to solve all instances of a problem by solving a large fraction of the instances), the validity of the DCRA only depends on the choice of n.

2.2 Homomorphic properties

A function $h : G \rightarrow H$ is a group homomorphism if whenever $a*b = c$ we have $h(a).h(b) = h(c)$. In other words, the group H in some sense has a similar algebraic structure as G and the homomorphism h preserves that.
Consider the Natural numbers $\mathbb{N}$ with addition as the operation. A function which preserves addition should have this property:$f(a+b) = f(a) + f(b)$ i.e. $f(x) = 3x$ is one such homomorphism, since $f(a+b) = 3(a+b) = 3a+3b = f(a)+f(b)$. Note that this homomorphism maps the $\mathbb{N}$ numbers back into themselves. This is not always the case. Homomorphism do not have to map between sets which have the same operations. So the homomorphism in which we are interested is :

$$(\mathbb{Z}^*_{n^2}, *) \rightarrow (\mathbb{Z}^*_n, +)$$

Formally $\forall x_1, x_2 \in \mathbb{Z}^*_{n^2}$

$$[x_1 x_2] = [x_1] + [x_2] \bmod n$$

since multiplication is the operation in the first set and the addition in the second set.

Chapter 3

Paillier Cryptosystem

The Paillier homomorphic cryptosystem is a public key cryptosystem by Paillier [1] based on the Decisional Composite Residuosity assumption (DCRA). It is homomorphic, by using public key, the product of two ciphertexts will decrypt to the sum of their corresponding plaintexts. Paillier cryptosystem is known as one of the few system that preserve additive homomorphic properties. In comparison, RSA and ElGamal are also homomorphic algorithms but they are multiplicative, which does not have the same impact which Paillier has. Also note that the Paillier cryptosystem described bellow is semantically secure against chosen-plaintext attacks (no adversary can distinguish between encryptions of different messages, even when allowed to make encryptions on its own). The scheme works as follows:

3.1 Key Pair Generation

Algorithm 1 (PaillierKeyPairGenerator) Generate pair of Public and Private keys

```
1. Calculate modulus $n$ - the product of $p$ and $q$ : two distinct large prime numbers of equal length.
2. Calculate $n^2$
3. Randomly select a generator $g \in \mathbb{Z}^*_{n^2}$ such that the order of $g$ is a multiple of $n$.
4. Calculate $\mu = (L(g^\lambda \bmod n^2))^{-1} \bmod n$ , where the function $L$ is defined as $L(u) = \frac{u-1}{n}$.
5. Generate $\lambda = \mathrm{lcm}(p-1, q-1)$ it can be computed using $\lambda = \frac{(p-1,q-1)}{gcd(p-1,q-1)}$
```

The pair keys are as follows:

- The public encryption key is (g, n).
- The private decryption key is (λ, μ).

To generate key pair the Paillier algorithm use the steps shown above. For generating the generator g there are two different ways,

- Randomly select g from a set $\mathbb{Z}^*_{n^2}$ where $gcd(\frac{g^\lambda \bmod n^2 - 1}{n}, n) = 1$.
- Select α and β randomly from a set $\mathbb{Z}^*_n$ then calculate $g = (\alpha n + 1)\beta^n \bmod n$. In this case the generator g always meets the condition above.

For this implementation was chosen to be used the first approach for generating g, as more convenience and more secure as $g \in \mathbb{Z}^*_{n^2}$. Note that in step.4 multiplicative inverse exist if and only if valid generator g was selected in previous step. There is one more key component which gives the randomness in the Paillier Cryptosystem it is used in the encryption process and is described in next section.

3.2 Encryption and Decryption

Algorithm 2 Paillier encryption and decryption

```
ENCRYPTION:
Randomly selected $r \in \mathbb{Z}^*_n$.
Plaintext $m \in \mathbb{Z}_n$.
Generator $g \in \mathbb{Z}^*_{n^2}$ such that the order of $g$ is a multiple of $n$.
Ciphertext $c = g^m r^n \bmod n^2$.
DECRYPTION:
Ciphertext $c \in \mathbb{Z}^*_{n^2}$.
Plaintext $m = L(c^\lambda \bmod n^2)\mu \bmod n$.
```

As was described in Chapter 2. the encryption $E(m, r)$ takes place in the isomorphic group $\mathbb{Z}^*_{n^2}$. The sender generates a ciphertext $c \in \mathbb{Z}^*_{n^2}$ by choosing a random $r \in \mathbb{Z}^*_n$.

For random r the Paillier algorithm allows $r \in \mathbb{Z}^*_{n^2}$ or $r \in \mathbb{Z}^*_n$. In general there is no difference whether the random $r \in \mathbb{Z}^*_{n^2}$ or random $r \in \mathbb{Z}^*_n$, since in either case the distribution of $[r^n \bmod n^2]$ is the same (as can be verified by looking at what happens in the isomorphic group $\mathbb{Z}_n \times \mathbb{Z}^*_n$). So the obvious choice for this implementation was to use $r \in \mathbb{Z}^*_n$, as the performance was much faster when the bigger n was used. For small n there are no difference at all, but dealing with large modulus n numbers, which most of the time is a fact, due to the security requirements make the computation much faster.

3.3 Properties of the Paillier Cryptosystem

A notable feature of the Paillier Cryptosystem is its homomorphic properties. The scheme takes full advantage of Carmichael's Theorem, which leads to additively homomorphic encryption function. The following properties can be described:

1. The product of two ciphertexts will decrypt to the sum of their corresponding plaintexts,

$$D(E(m_1, r_1) * E(m_2, r_2) \bmod n^2) = (m_1 + m_2) \bmod n.$$

2. The product of a ciphertext with a plaintext raising g will decrypt to the sum of the corresponding plaintexts,

$$D(E(m_1, r_1) * g^{m_2}) \bmod n^2) = (m_1 + m_2) \bmod n.$$

3. Rising a ciphertext to a constant power $k \in \mathbb{Z}_n$ results in the constant multiple of the original plaintext,

$$D(E(m)^k \bmod n^2) = km \bmod n.$$

The above homomorphic properties will be utilized by the E-Voting system discussed in Chapter.4 and particularly the property one.

Chapter 4

Developing a JCA Compliant Paillier Provider

4.1 JCA Background

A cryptographic provider is a collection of classes that implement cryptographic algorithms (signatures, ciphers and message digests). One of the classes, which will be a subclass of *java.security.Provider*, maps algorithm name to class name. The provider subclass represents the provider as a whole. A brief overview of the Java Cryptography Architecture is given in the next paragraph.

The Java Cryptography Architecture or JCA consists of two parts. The first part is the interface that defines how a developer accesses the classes that implement the cryptographic functionality. Including generic and non-cryptographic code, this first part constitutes the framework. The second part is the implementation part, holding the actual implementation of the cryptographic algorithms; such a part is commonly referred to as JCA provider or Cryptographic Service Provider (CSP). SUN offers implementations for some cryptographic algorithms as part of the Java Development Kit (JDK) [11].

The Java Cryptography Extension (JCE) implements the JCA API. Together, the JCE and the cryptographic aspects of the SDK provide a complete, platform-independent cryptography API. JCE was previously a not obligatory package (extension) to the Java

2 SDK, Standard Edition, versions 1.2.x and 1.3.x. JCE has now been integrated into the Java 2 SDK, since v 1.4.

4.2 The Java Cryptography Architecture requirements

Security, in particular cryptography, has always been a core of Java. The JCA was explicitly designed to provide and expose cryptographic operations to developers with different functionality and purposes. The architecture of the JCA consist of several broad but guiding design principles that had to be met [13].

- Algorithm independence
- Algorithm extensibility
- Implementation independence
- Implementation interoperability

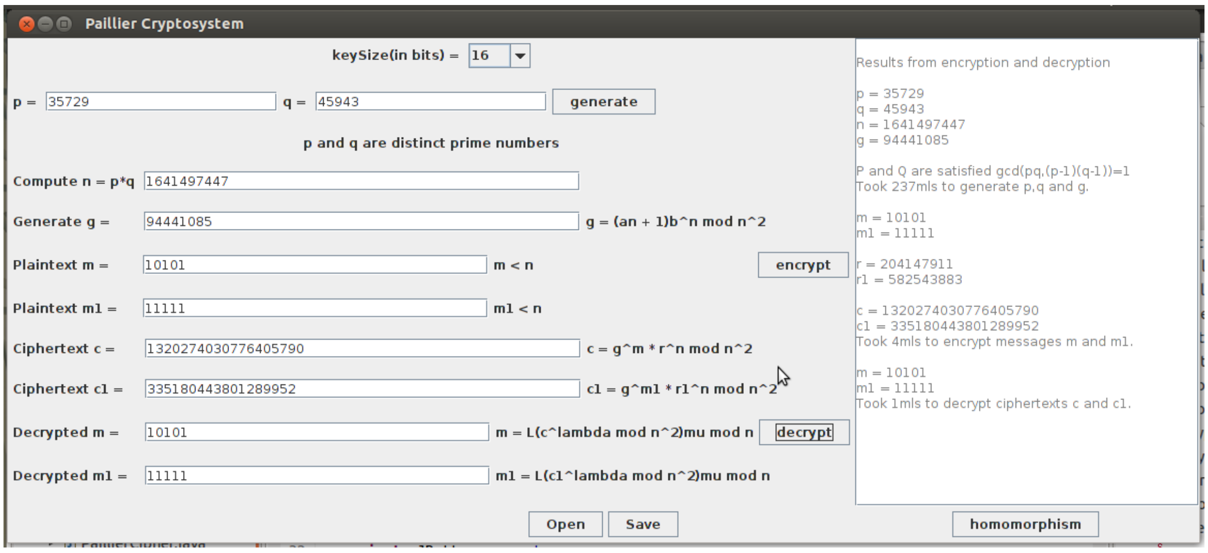
For a framework to be implementation independent it should hide the details of a provider from an application and the application should not call directly any classes of the provider. A framework should part an application and providers by sitting between them. In this fashion, an application can only see an implementation independent interface, called upward interface, of the framework. Vice versa, a framework should provide a downward API that can be implemented in many different ways. To be algorithm independent the upward interface of a framework should be abstract and only show a relationship with generic cryptographic concepts, such as Cipher, rather than concrete algorithm such as PaillierCipher. JCA supports different cryptographic operations, classified into categories. JCA refers to each category as an engine, which is simply another name for a Java class. To provide a high level of algorithm independence, each JCA engine uses the factory design pattern, where the methods are always declared static.
Algorithm extensibility simply provides a mechanism for adding new algorithms that can be classified and exposed through one of the supported engines. For this project will be

implement the Paillier algorithm as Cipher engine. Implementation independence refers to the ability to say "I need a PaillierCipher" and immediately get the instance of a class that provides such a functionality without requiring the complicated invocation of provider specific methods. Implementation independence essentially offers the developer a choice of how to handle the presence of providers.
The Java Cryptography Architecture requires that Java security providers be code-signed (using a code-signing certificate issued by Oracle Corporation). In other hand OpenJDK does not require special handling of the JCE policy files since it's open source and therefore not export-restricted in the United States. The implementation is made by using OpenJDK and the provider will work only with OpenJDK. Next step will be to sign the provider.

4.3 Implementation

4.3.1 Basic Paillier Algorithm

The very first step was to implement the algorithm itself. For this purpose basic UI is used (*java.awt*) to visualise the process of encryption and decryption and to show the homomorphic properties that the system possess.



keySize(in bits) = 16

p = 35729 q = 45943 generate

p and q are distinct prime numbers

Compute n = p*q 1641497447

Generate g = 94441085 g = (an + 1)b^n mod n^2

Plaintext m = 10101 m < n encrypt

Plaintext m1 = 11111 m1 < n

Ciphertext c = 1320274030776405790 c = g^m * r^n mod n^2

Ciphertext c1 = 335180443801289952 c1 = g^m1 * r1^n mod n^2

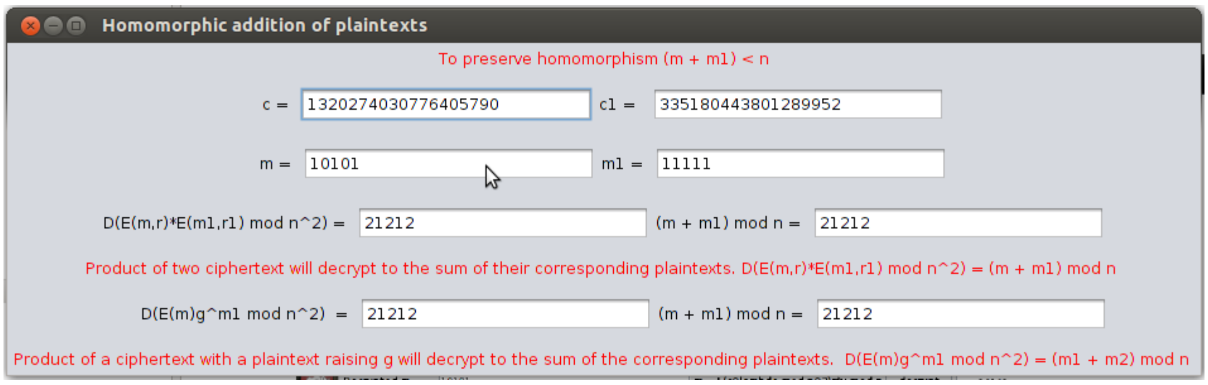
Decrypted m = 10101 m = L(c^lambda mod n^2)mu mod n decrypt

Decrypted m1 = 11111 m1 = L(c1^lambda mod n^2)mu mod n

```
Results from encryption and decryption

p = 35729
q = 45943
n = 1641497447
g = 94441085

P and Q are satisfied gcd(pq,(p-1)(q-1))=1
Took 237mls to generate p,q and g.

m = 10101
m1 = 11111

r = 204147911
r1 = 582543883

c = 1320274030776405790
c1 = 335180443801289952
Took 4mls to encrypt messages m and m1.

m = 10101
m1 = 11111
Took 1mls to decrypt ciphertexts c and c1.
```

Figure 4.1: Paillier Cryprosystem - Prototype One.

As you can see from Figure. 4.1 using the key size as an initial value of the system, p,q and g are generated. The n is computed by multiplying p and q , which is identical with RSA modulus. The user chose two messages (plaintexts), which we can encrypt and decrypt successfully using the Paillier algorithm. The next option you have is to prove homomorphism in the Paillier Cryptosystem. This can be achieved in Figure. 4.1 by clicking ”homomorphism” button.



To preserve homomorphism (m + m1) < n

c = 1320274030776405790 c1 = 335180443801289952

m = 10101 m1 = 11111

D(E(m,r)*E(m1,r1) mod n^2) = 21212 (m + m1) mod n = 21212

Product of two ciphertext will decrypt to the sum of their corresponding plaintexts. D(E(m,r)*E(m1,r1) mod n^2) = (m + m1) mod n

D(E(m)g^m1 mod n^2) = 21212 (m + m1) mod n = 21212

Product of a ciphertext with a plaintext raising g will decrypt to the sum of the corresponding plaintexts. D(E(m)g^m1 mod n^2) = (m1 + m2) mod n

Figure 4.2: Paillier Cryprosystem - Prototype One.

Figure. 4.2 produces output of performing homomorphism on the Paillier algorithm.

It shows the product of two ciphertexts will decrypt to the sum of their corresponding plaintexts, $D(E(m_1, r_1) * E(m_2, r_2) \bmod n^2) = (m_1 + m_2) \bmod n$.
Also the product of a ciphertext with a plaintext raising g will decrypt to the sum of the corresponding plaintexts, $D(E(m_1, r_1) * g^{m_2}) \bmod n^2) = (m_1 + m_2) \bmod n$.
The design of this prototype was basic. Application consist of three classes as follow: *GUIPaillier.java*, *PaillierAlgorithm.java* and *isGcdOne.java*. The implementation of the algorithm itself contains in *PaillierAlgorithm.java*. Also there is ability to save the produced results for testing purposes. See Appendix A - Figure A.1.

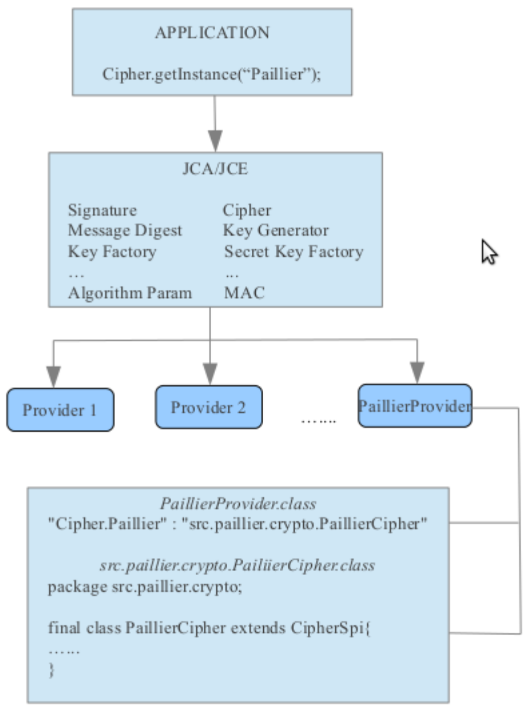
4.3.2 Paillier JCA Provider

The design of the Paillier Provider follows very strictly the requirements of JCA [11]. For the JCA Provider in general we need at least two classes - "engines" (services) to be implemented in order to get working Provider. This is minimal requirement for simple encryption and decryption. Fully fitted Provider depends on services intended to support and the strength of the algorithm itself. To show the additive homomorphic properties in the Paillier Cryptosystem was implemented the *Cipher* and *KeyPairGenerator* engines (see Class UML diagram in Appendix A- Figure A.2).

```
public class PaillierProvider extends Provider {
        private static final long serialVersionUID = -845334060612935617L;
        public PaillierProvider() {
                put("KeyPairGenerator.Paillier",
                                "src.paillier.crypto.PaillierKeyPairGenerator");
                put("Cipher.Paillier", "src.paillier.crypto.PaillierCipher");
                put("Cipher.PaillierHP",
                                "src.paillier.crypto.PaillierHomomorphicCipher");
        }
}
```

Figure 4.3: Paillier Provider.

The Provider class is used to register implementations of security services that Paillier will support as part of the JDK Security API or one of its extensions. See Figure 4.3.
The application API methods in each engine class are routed to the provider's implementations through classes that implement the corresponding Service Provider Interface (SPI). In other words for each engine class, there is a corresponding abstract SPI class which defines the methods that each cryptographic service provider's algorithm must implement. The name of each SPI class is the same as that of the corresponding engine class, followed by Spi. For example, the Cipher engine class provides access to the functionality of a cipher algorithm. The actual provider implementation is supplied in a subclass of CipherSpi. Applications call the engine class API methods, which in turn call the SPI methods in the actual implementation, in this case PaillierCipher.
Each SPI class is abstract. To supply the implementation of a particular type of service for a specific algorithm, a provider must subclass the corresponding SPI class and provide implementations for all the abstract methods, in some cases the implementation of these methods could be empty, but they must present in your class. Also you can add your own functionality as well, by implementing additional methods which are not part in the SPI class by default.
For each engine class in the API, implementation instances are requested and instantiated by calling the getInstance() factory method in the engine class. A factory method is a static method that returns an instance of a class. The engine classes use the framework provider selection mechanism described above to obtain the actual backing implementation (SPI), and then creates the actual engine object. Each instance of the engine class encapsulates (as a private object) the instance of the corresponding SPI class, known as the SPI object. All API methods of an API object are declared final and their implementations invoke the corresponding SPI methods of the encapsulated SPI object [11].
To make this more clearer, review the following code and illustration:


APPLICATION
Cipher.getInstance("Paillier");
JCA/JCE
Signature
Message Digest
Key Factory
...
Algorithm Param
Cipher
Key Generator
Secret Key Factory
...
MAC
Provider 1
Provider 2
....
PaillierProvider
PaillierProvider.class
"Cipher.Paillier" : "src.paillier.crypto.PaillierCipher"
src.paillier.crypto.PailiierCipher.class
package src.paillier.crypto;
final class PaillierCipher extends CipherSpi{
......
}

Here an application wants an "Paillier" Cipher instance, and does not care which provider is used. The application calls the getInstance() factory methods of the Cipher engine class, which in turn asks the JCA framework to find the first provider instance that supports "Paillier". The framework consults each installed provider, and obtains the provider's instance of the Provider class. The framework searches each provider, finally finding a suitable entry in PaillierProvider. This database entry points to the implementation class *src.paillier.crypto.PaillierCipher* which extends CipherSpi, and is thus suitable for use by the Cipher engine class. An instance of *src.paillier.crypto.PaillierCipher* is created, and is encapsulated in a newly-created instance of javax.crypto.Cipher, which is returned to the application. When the application now does the init() operation on the Cipher instance, the Cipher engine class routes the request into the corresponding engineInit() backing method in the src.paillier.crypto.PaillierCipher class.
For this project two versions of *Cipher* class were implemented. The first one *PaillierCipher* supports encryption and decryption of different-sized chunks of data. The class contains the logic that breaks down the input data into block-sized chunks for encrypting or de-

crypting. The size of the chunks depends on the key size initialized for the cipher. After that the encryption or decryption is carried out on each chunk and the result is stored in internal buffer. When the process is completed the all chunks are concatenated together. The second one, *PaillierHomomorphicCipher* supports only decimal numbers and the input should be strictly smaller than the key, otherwise exception is thrown. Use of this class is only for E-Voting application purposes, discussed in next chapter. The *Cipher* class is most complex and important class in this JCA Provider. Both classes share the same methods with only difference in ability for buffering the data when is needed and calculating the input and output size, as mentioned above. The *Cipher* class has a *update*() and *doFinal*() methods that are used to feed data into the cipher and gather the results. These methods end up calling *engineUpdate*() and *engineDoFinal*() in *CipherSpi*. This implementation is a subclass of *CipherSpi*.
The *KeyPairGenerator* class is used to generate pair of public and private keys. There are two ways to generate a key pair: in an algorithm-independent manner, and in an algorithm-specific manner. The only difference between the two is the initialization of the object. For this project was chosen to be used the first approach. The second way to generate a keys requires implementation of *AlgorithmParametersClasses* which was out of scope and not necessary to do on this occasion (see Chapter 7.). Key pair generation object is obtained by using *KeyPairGenerator.getInstance*("Paillier") static factory method. Key pair generator share the concepts of a key size and a source of randomness. The key size is interpreted differently for different algorithms, in the case of Paillier algorithm, the key size corresponds to the length of the modulus n represented in bits.

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance("Paillier");
kpg.initialize(512);
KeyPair keyPair = kpg.generateKeyPair();
PublicKey pubKey = keyPair.getPublic();
PrivateKey privKey = keyPair.getPrivate();
```

Figure 4.4: KeyPairGenerator-static factory method.

For each engine class in the API, a implementation instance is requested and instantiated by calling one of the getInstance methods on the engine class, specifying the name of the desired algorithm whose implementation is desired. The initializations and obtaining the public and private keys is shown in Figure 4.4.

The snippet code below shows the randomness in the Paillier Cryptosystem. Its generates random $r \in \mathbb{Z}_n^*$ for each separate encryption step using the same modulus n. Paillier cryptosystem allows the generated r to differ every time, such that the same plaintext encrypted several times will produce different ciphertext. This functionality is provided outside of the *PaillierKeyPairGenerator*, class responsible for generating the key pair and its called from the *PaillierCipher* class when encryption is performed. By definition r is part of the Public key, but it is generated separately from the Public key (discussed in Chapter.3). The n and g, the other two components of the Public key are generated once and remain the same for the entire process, but r is vary for each encryption. In comparisons with other algorithms here we provide flexibility for using random r generator outside of the *KeyPairGenerator* class. See Figure 4.5.

```
protected BigInteger generateRandomRinZn(BigInteger n) {
        BigInteger r;
        int strength = 0;
        strength = n.bitLength();
        // generate r random integer in Z*_n
        do {
                r = new BigInteger(strength , SECURE_RANDOM);
        } while (r.compareTo(n) >= 0 || r.gcd(n).intValue() != 1);
        return r;
}
```

Figure 4.5: generateRandomRinZn method.

To show the homomorphism in the Paillier Cryptosystem by using the JCA Provider, it needs to be implemented several more interfaces and engine classes(see Chapter 7.). The most important one in terms of use of homomorphic properties is $KeyFactory$, as it breaks down a key into its discrete parts. The E-Voting application, described in next chapter needs to have an access to modulus n, to perform the homomorphism and random r, for demonstrating purposes. Due to a time constrain it was used a short cut solution by overriding $toString()$ method within the $PaillierKey$ class to gain this access. It is temporary solution and in future release this method should be removed. The $PaillierKey$ class implements the Key interface which is the top-level interface for all keys: private and public. The method from Figure 4.5 is implemented within the same class as well for more convenience.

Chapter 5

Applications of Homomorphic Properties

5.1 Application to E-Voting

In general E-Voting is a tool for making the electoral process more efficient and to increase trust in its management. Well implemented, E-Voting solutions can increase the security of the ballot and speed up the processing of the results. The main application of homomorphic capability of Paillier will be an E-Voting system using Paillier Cryptosystem and the implemented JCA PaillierProvider. The additive homomorphic property will directly return the tally, this is the biggest advantage of the Paillier Cryptosystem. The system works as follows:
Given the option to chose two new members for college committee, voter has the choice whether to chose two, one or no candidates. Each vote have to be presented in numeric form and encrypt separately, using the same Paillier public key, but different random r, such that the same votes will produced different ciphertexts. In other words; ensures voter privacy. The numerical base b is chosen, such that the base is greater than the number of voters. This application is using base b to be 10, as the application is only for demonstrating purposes and has only 8 voters.

APPLICATION PARAMETERS
1. Choose b to be base 10. $b > N_v$
2. $N_v = 8$,Number of voters.
3. $N_c = 4$,Number of candidates.
4. To each candidate is assigned decimal representation: Candidate 1 $=10^0$, Candidate 2$=10^1$, Candidate 3$=10^2$, Candidate 4$=10^3$.
5. $m_{max} = 10^2 + 10^3 = 1100$. The maximum vote message that can ever happen to be encrypted.
6. $T_{max} = N_v * m_{max} = 8 * 1100 = 8800$. The maximum possible tally.
7. To be able to encrypt T_{max}, modulus n,must be bigger than T_{max}.

Figure 5.1: Application parameters.

The intention is to use $D(E(m_1, r_1) * E(m_2, r_2) \bmod n^2) = (m_1 + m_2) \bmod n$. The implementation did not restrict the choice of each voter about the number of candidates they can select, as the application aims to show the homomorphism in Paillier Cryptosystem in general and to outline as more as possible different situations.

Each voter choice is calculated as a sum of all selected candidates by this voter. For example if the voter had selected Candidate 3 and Candidate 4 the vote message to be encrypt will be the sum of the decimal representation for Candidate 3 and Candidate 4 (see Figure 5.1-4); i.e. $m = 10^2 + 10^3 = 1100$.

After calculating the sum of each voter choice the system encrypt separately each vote message, $E(m_i, r_i) = c_i = g^{m_i} {r_i}^n \bmod n^2$. Where c_i is the ciphertext for each voter choice. Next step is to multiply all encrypted messages modulus n^2;

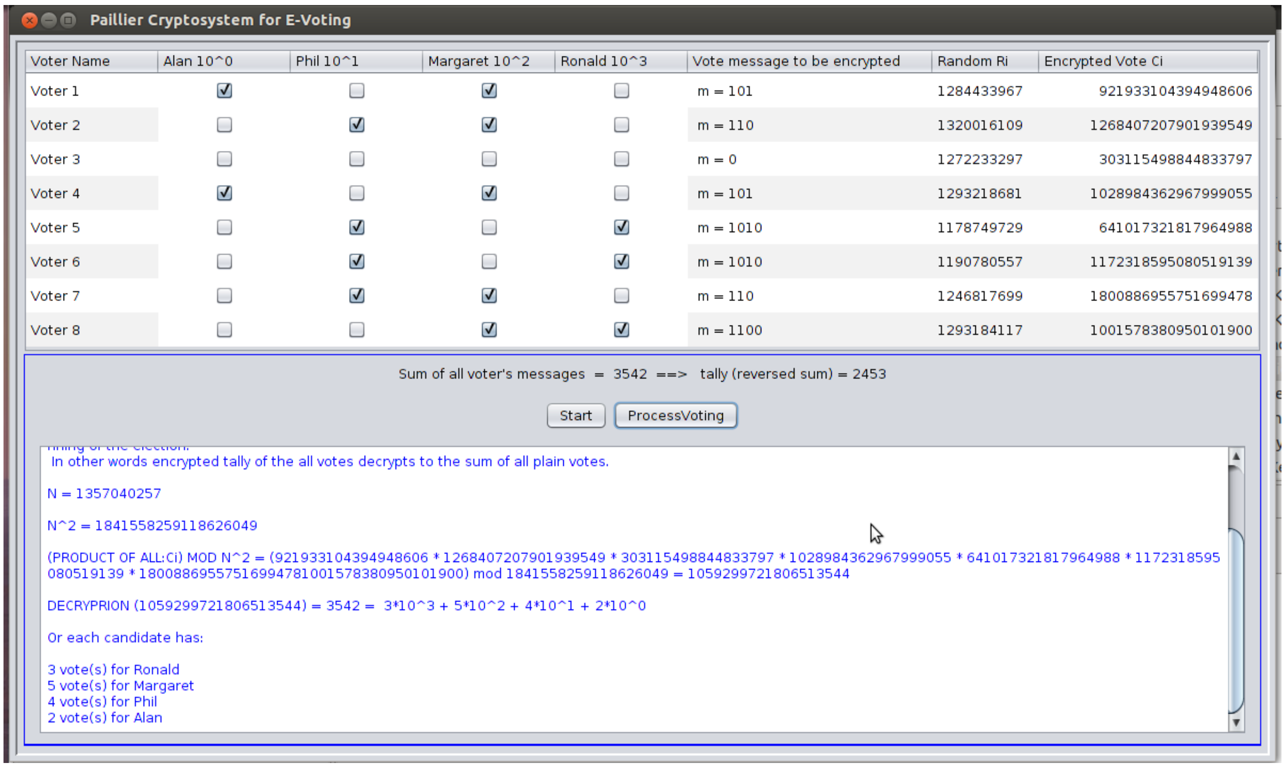
$$T = \prod_{i=1}^{N_v} c_i \bmod n^2$$

The above result T - tally, the product of all encrypted messages $\bmod n^2$ contains all information what we needed after election process is completed, after each voter made his choice. The result now can be stored or transmitted via internet until authorities are ready to revile the results from the election. Note because base ten is used, the simple tally is the reversed sum of all voters messages after calculation the sum of each individual voter is done.

To obtain the election results need to decrypt the product of all encrypted messages $\bmod n^2$ - T, described above by using the Paillier private key;

$$m = L(T^{\lambda} \bmod n^2)\mu \bmod n$$

The result after decryption is the simple tally. In other words encrypted tally of the all votes decrypts to the sum of all plain votes, $D(T) = \sum_{i=1}^{N_v} m_i \bmod n$. In general decrypted tally needs to be converted to a number with the base chosen at the beginning of the election, but here the system using base ten, so there is no conversion needed. To visualise the process see Figure 5.2.



Voter Name	Alan 10^0	Phil 10^1	Margaret 10^2	Ronald 10^3	Vote message to be encrypted	Random Ri	Encrypted Vote Ci
Voter 1	✓		✓		m = 101	1284433967	921933104394948606
Voter 2		✓	✓		m = 110	1320016109	1268407207901939549
Voter 3					m = 0	1272233297	303115498844833797
Voter 4	✓		✓		m = 101	1293218681	1028984362967999055
Voter 5		✓		✓	m = 1010	1178749729	641017321817964988
Voter 6		✓		✓	m = 1010	1190780557	1172318595080519139
Voter 7		✓	✓		m = 110	1246817699	1800886955751699478
Voter 8			✓	✓	m = 1100	1293184117	1001578380950101900

Sum of all voter's messages = 3542 ==> tally (reversed sum) = 2453

```
In other words encrypted tally of the all votes decrypts to the sum of all plain votes.

N = 1357040257

N^2 = 1841558259118626049

(PRODUCT OF ALL:Ci) MOD N^2 = (921933104394948606 * 1268407207901939549 * 303115498844833797 * 1028984362967999055 * 641017321817964988 * 1172318595
080519139 * 18008869557516994781001578380950101900) mod 1841558259118626049 = 1059299721806513544

DECRYPRION (1059299721806513544) = 3542 =  3*10^3 + 5*10^2 + 4*10^1 + 2*10^0

Or each candidate has:

3 vote(s) for Ronald
5 vote(s) for Margaret
4 vote(s) for Phil
2 vote(s) for Alan
```

Figure 5.2: Paillier E-Voting application - screen shot.

The implementation of the E-Voting application described above taking full advantage of JCA framework by using the implemented JCA PaillierProvider described in Chapter 4. Instead implementing the Paillier algorithm within the application the JCA Paillier-Provider is used. The provider is added as an external JAR to the project build path and made available to the application with one line of Java code (see Figure 5.3).

```
import src.paillier.crypto.PaillierProvider;
...
Security.addProvider(new PaillierProvider());
```

Figure 5.3: Adding JCA PaillierProvider.

The detailed UML class diagram of E-Voting application (see Appendix A.3) gives an outline of the structure of the code. The application consists of three classes as follows, *GUI.java*, *Candidates.java*, *PaillierCryptosystemUtil.java.* Swing components are used for creating graphical user interfaces (GUIs) for the application. The implementation contains in *GUI* class. The *Candidates* class was implemented as a Java Enumeration, the choice was made based on the advantages provided by Java Enumeration. In context of the application each candidate has a name and unique decimal representation. So you can not assign anything else other than predefined values and adding new constants is easy and you can add new constants without breaking existing code. Within the *GUI* class is made an instance of a *PaillierCryptosystemUtil* class, which is responsible for initialising the key size and generating public and private keys, also the encryption and decryption. Detailed Java docs explains the use of each method within the classes. They are generated and provided with the source code.

5.2 Extension to Cloud Computing Security

Cloud computing is a promising computing model that enables on-demand network access to a shared computing resources. Cloud computing offers a group of services, including

Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS). The storage in the Cloud is an important service of cloud computing, which allows data owners to move data from their local computing systems to the Cloud. More and more data owners are starting to choose to host their data in the Cloud. In a cloud environment the customer cannot rely on the security and confidentiality of the remote resource. A solution is to enables a customer to generate a program that can be executed by a third party, without revealing the underlying algorithm or the processed data. The problem is that while data can be sent to and from a cloud provider's data center in encrypted form, the servers that power a cloud can't do any work on it that way. Now Gentry, an IBM researcher, has shown that it is possible to analyze data without decrypting it[7]. This helps to secure applications and data in a Cloud environment.
The key is to encrypt the data in such a way that performing a mathematical operation on the encrypted information and then decrypting the result produces the same answer as performing an analogous operation on the unencrypted data. The correspondence between the operations on unencrypted data and the operations to be performed on encrypted data is known as a homomorphism.
Craig Gentry is the first person who create a fully homomorphic Cryptosystem contains both, additive and multiplicative properties [7].

Chapter 6

Software testing

6.1 Software validation and verification

Are we building the right product? The JCA PaillierProvider meets the application's needs. To show the additive homomorphic properties preserved by Paillier Cryptosystem were implemented the *Cipher* and *KeyPairGenerator* engines as part of the provider. These engine classes provide needed functionality to satisfy the requirements for showing the homomorphic properties that the scheme possess.

Are we building the product right? JCA PaillierProvider was build according to the requirements and design specifications. It follows very strictly the requirements of Java Cryptography Architecture Reference Guide for Java Platform Standard Edition 6 issued by Oracle. Evident of that is that the integrated provider works with the E-Voting application.

Install and configure the provider. There are two ways to install provider on any system.

- Statically: To statically add the provider, edit the java.security file found in the lib/security directory underneath the JDK installation directory in your machine.

You need to add the following line:

$$Security.provider.number = src.paillier.crypto.PaillierProvider$$

Where "number" indicates the order of the provider in your configuration file and the order that the providers is searched by the JVM.

- Dynamically: If you do not want to modify the java.security file, you can add the provider dynamically. The provider should be added as an external JAR to the project build path and make available to the application with following code:

$$import\ src.paillier.crypto.PaillierProvider;$$

$$...$$

$$Security.addProvider(newPaillierProvider());$$

Both ways of using the JCA PaillierProvider were tested and both of them are working correctly. Note - This provider will work only with OpenJDK. The JCA requires that Java security providers be code-signed (using a code-signing certificate issued by Oracle Corporation). If you run the Oracle or Sun Java JDK you wont be able to use this provider, since the provider is not registered. OpenJDK does not require special handling of the JCE policy files since it's open source and therefore not export-restricted in the United States.

System testing The best way to test the provider is to write a small program that attempts to find the provider and test if the algorithm is available. For this purpose was implemented a test program $TestJCAProvider$. With help of this program system integration testing activities were performed during the development of the provider. Agile testing approach was employed for testing the provider, as most of the test effort was on-going during the development process. Some unexpected results and problems occurred during the testing, which helps me to improve my code and find the right solution. Several test cases were established and they were performed repetitively during this implementation (see the table bellow).

Test	Test Description	Expected Results	Result
Test 1	Pass message bigger than the key length.	1.'PaillierHP' cipher, should throw exception; 2.'Paillier' cipher, system should works	1.-passed 2.-passed
Test 2	Pass message as a text .	1.'PaillierHP' cipher, should throw exception; 2.'Paillier' cipher, system should works	1.-passed 2.-passed
Test 3	Pass message as a decimal representation	1.'PaillierHP' cipher, system should works; 2.'Paillier' cipher, system should works	1.-passed 2.-passed
Test 4	Initialise the KeyPairGenerator with key size less than 8	Key size is set to default-64 bits	passed
Test 5	Initialise the KeyPairGenerator with key size greater than 3096	Key size is set to default-64 bits	passed

Figure 6.1: Test cases.

Further test cases available on request.

6.2 Quality of code

For testing quality of my code I integrated PMD to my Eclipse IDE for analyse efficiency and quality of my code. It finds unused variables, empty catch blocks, unnecessary object creation, and so forth. This is simply programming mistake detector. PMD scans Java source code and looks for potential problems like:

- Possible bugs - empty try/catch/finally/switch statements.
- Dead code - unused local variables, parameters and private methods. Suboptimal code - wasteful String/StringBuffer usage.
- Overcomplicated expressions - unnecessary if statements, for loops that could be while loops.
- Duplicate code - copied/pasted code means copied/pasted bugs.

After scan completion it generates reports which are placed in the project directory , based on the reports I improved my code. Great tool, help me a lot to develop my skills as a programmer, to avoid making silly mistakes and to start writing clean and readable code.

Chapter 7

Conclusions and Further work

7.1 Conclusions

In the beginning implementation of JCA Compliant Provider was not in my plan , but after being advised from my supervisor I just took it. The main achievement of this project became the implementation of JCA Paillier Provider. I gained very good understanding of Java Cryptography Architecture and how this framework works. Now any application that wants to use Paillier algorithm could use the implemented Provider by adding it as an external library. It was shown clearly the use of the additive homomorphism in the Paillier algorithm for E-Voting applications and advantages in terms of privacy and security. The additive homomorphic scheme was discussed and the possibility for enhanced Cloud computing security was mentioned.

7.2 Further work

The Paillier Cryptosystem is not as fast as RSA in either encryption or decryption, but it is fast enough that it could be considered for uses that RSA doesnt work for. The Paillier

scheme could be used for an application that needs homomorphic encryption. I think the main thing that needs to be worked on is encryption, as it is much slower and there are very few optimizations for it. Damgard and Jurik, suggested optimisations of the Paillier Cryptosystem which could be considered for improving the algorithm itself and speed up the encryption process [3]. This requires further research and additional work. I notice that the decryption was much better performing at a range of two times slower when the key was smaller. The performance difference was greater when the key got bigger. As suggestion for further work is that the Chinese Remainder Theorem (CRT) could be a big help for pre computation of the private key components to reduce decryption process. My observations are that the decryption time were generally higher than encryption.
In terms of the JCA Provider it is necessary to be added ability for storing and managing the keys and precomputed keys data. The JCA provides extensible architecture to store and manage the keys. In general Key objects and key specifications *KeySpecs* are two different representations of key data. *KeySpec* engine class needs to be implemented to provide functionality to group and type safety for all key specifications. Cipher use key objects to initialize the encryption algorithm, but keys may need to be converted into a more portable format for transmission or storage. We need to provide ability for Key object to be translated into a key specification, a level of transparency will be achieved. This transparency will allow us to "look into" the key and at its meta-data. We will achieve this by implementing *KeyFactory* class engine which is designed to perform conversions between opaque cryptographic Keys and key specifications and vice versa.
And finally the implemented JCA Provider need to be code-signed. For this purpose application need to be sent to Oracle to register the provider.
In overall I want to state that I am very proud of what work I have done on the project and how much I learned from it. I can say for sure that the experience I gained working on this project will help me throughout my entire career as a software engineer.

Appendix A

Implementation

A.1 Basic Paillier algorithm - Prototype One

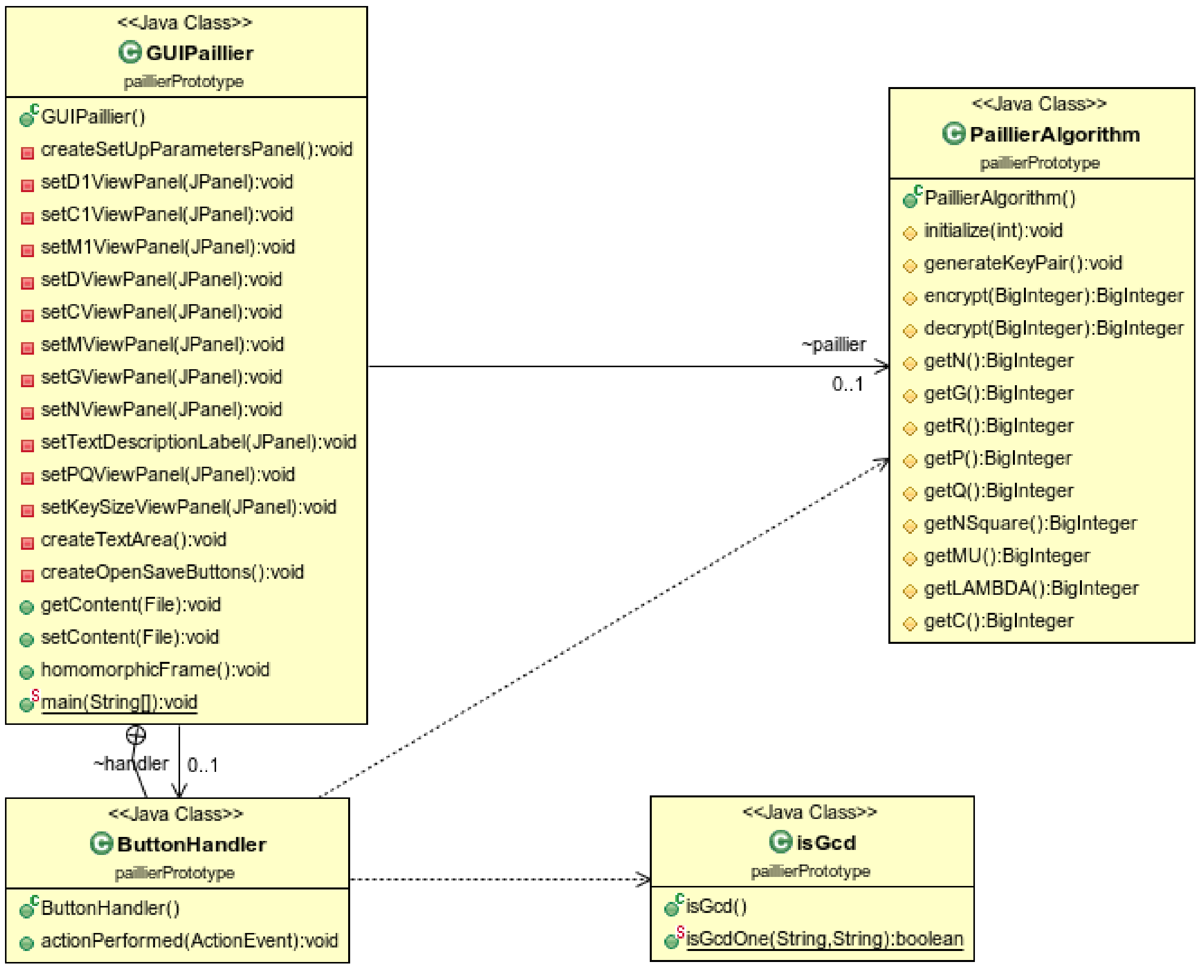
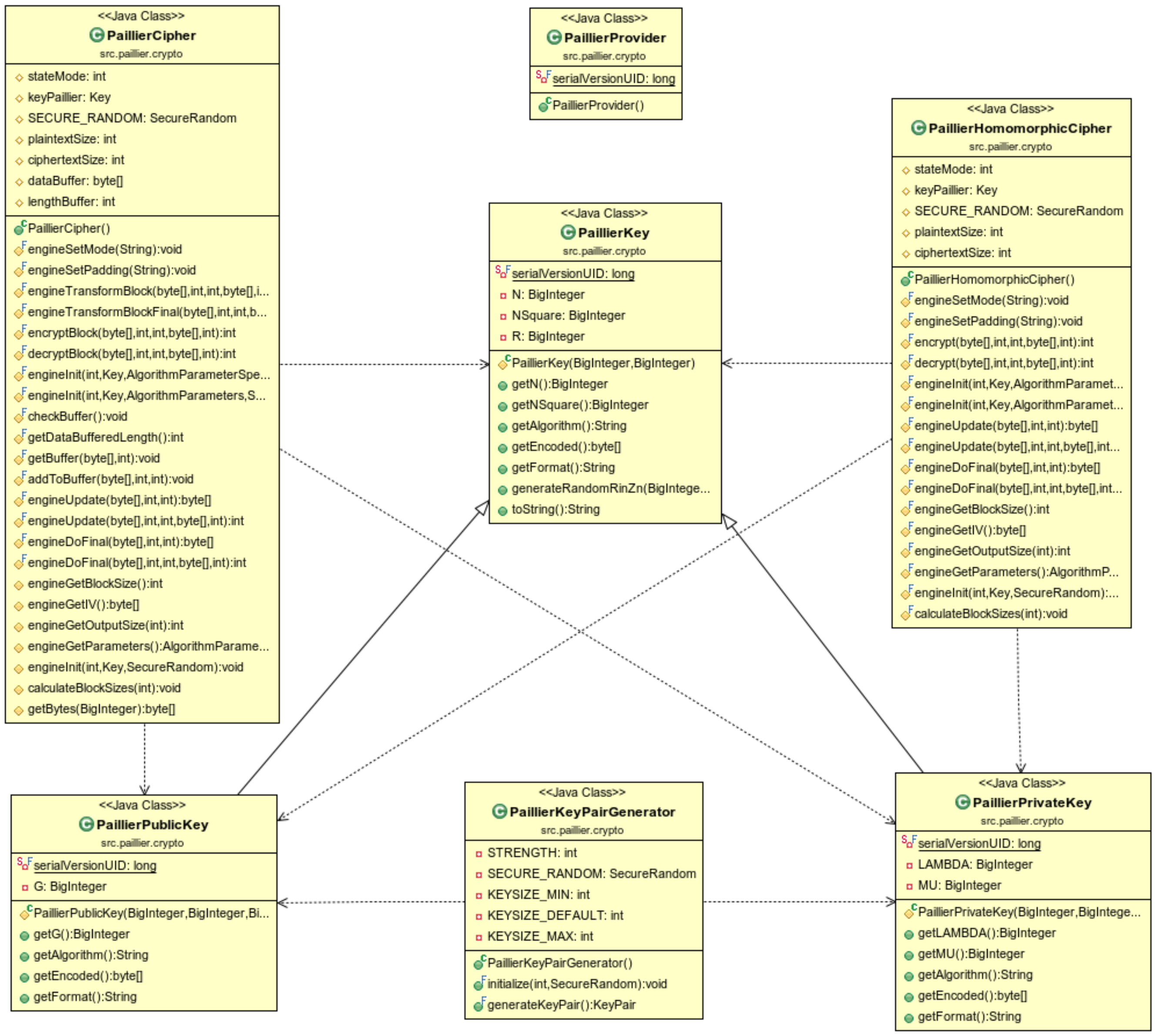


Figure A.1: Paillier Prototype One - Class diagram.

Figure A.1 shows the UML-Class diagram of Paillier Cryprosystem - Prototype One in details, included the relationships between the methods.

A.2 JCA PaillierProvider

PaillierProvider class UML diagram is shown below with the relationship and dependencies beteewn the classes.



<<Java Class>>
PaillierCipher
src.paillier.crypto
stateMode: int
keyPaillier: Key
SECURE_RANDOM: SecureRandom
plaintextSize: int
ciphertextSize: int
dataBuffer: byte[]
lengthBuffer: int
PaillierCipher()
engineSetMode(String):void
engineSetPadding(String):void
engineTransformBlock(byte[],int,int,byte[],i...
engineTransformBlockFinal(byte[],int,int,b...
encryptBlock(byte[],int,int,byte[],int):int
decryptBlock(byte[],int,int,byte[],int):int
engineInit(int,Key,AlgorithmParameterSpe...
engineInit(int,Key,AlgorithmParameters,S...
checkBuffer():void
getDataBufferedLength():int
getBuffer(byte[],int):void
addToBuffer(byte[],int,int):void
engineUpdate(byte[],int,int):byte[]
engineUpdate(byte[],int,int,byte[],int):int
engineDoFinal(byte[],int,int):byte[]
engineDoFinal(byte[],int,int,byte[],int):int
engineGetBlockSize():int
engineGetIV():byte[]
engineGetOutputSize(int):int
engineGetParameters():AlgorithmParame...
engineInit(int,Key,SecureRandom):void
calculateBlockSizes(int):void
getBytes(BigInteger):byte[]
<<Java Class>>
PaillierProvider
src.paillier.crypto
serialVersionUID: long
PaillierProvider()
<<Java Class>>
PaillierHomomorphicCipher
src.paillier.crypto
stateMode: int
keyPaillier: Key
SECURE_RANDOM: SecureRandom
plaintextSize: int
ciphertextSize: int
PaillierHomomorphicCipher()
engineSetMode(String):void
engineSetPadding(String):void
encrypt(byte[],int,int,byte[],int):int
decrypt(byte[],int,int,byte[],int):int
engineInit(int,Key,AlgorithmParamet...
engineInit(int,Key,AlgorithmParamet...
engineUpdate(byte[],int,int):byte[]
engineUpdate(byte[],int,int,byte[],int...
engineDoFinal(byte[],int,int):byte[]
engineDoFinal(byte[],int,int,byte[],int...
engineGetBlockSize():int
engineGetIV():byte[]
engineGetOutputSize(int):int
engineGetParameters():AlgorithmP...
engineInit(int,Key,SecureRandom):...
calculateBlockSizes(int):void
<<Java Class>>
PaillierKey
src.paillier.crypto
serialVersionUID: long
N: BigInteger
NSquare: BigInteger
R: BigInteger
PaillierKey(BigInteger,BigInteger)
getN():BigInteger
getNSquare():BigInteger
getAlgorithm():String
getEncoded():byte[]
getFormat():String
generateRandomRinZn(BigIntege...
toString():String
<<Java Class>>
PaillierPublicKey
src.paillier.crypto
serialVersionUID: long
G: BigInteger
PaillierPublicKey(BigInteger,BigInteger,Bi...
getG():BigInteger
getAlgorithm():String
getEncoded():byte[]
getFormat():String
<<Java Class>>
PaillierKeyPairGenerator
src.paillier.crypto
STRENGTH: int
SECURE_RANDOM: SecureRandom
KEYSIZE_MIN: int
KEYSIZE_DEFAULT: int
KEYSIZE_MAX: int
PaillierKeyPairGenerator()
initialize(int,SecureRandom):void
generateKeyPair():KeyPair
<<Java Class>>
PaillierPrivateKey
src.paillier.crypto
serialVersionUID: long
LAMBDA: BigInteger
MU: BigInteger
PaillierPrivateKey(BigInteger,BigIntege...
getLAMBDA():BigInteger
getMU():BigInteger
getAlgorithm():String
getEncoded():byte[]
getFormat():String

Figure A.2: Paillier Provider - Class diagram.

A.3 Homomorphic application

The UML class diagram of E-Voting Application is shown below.

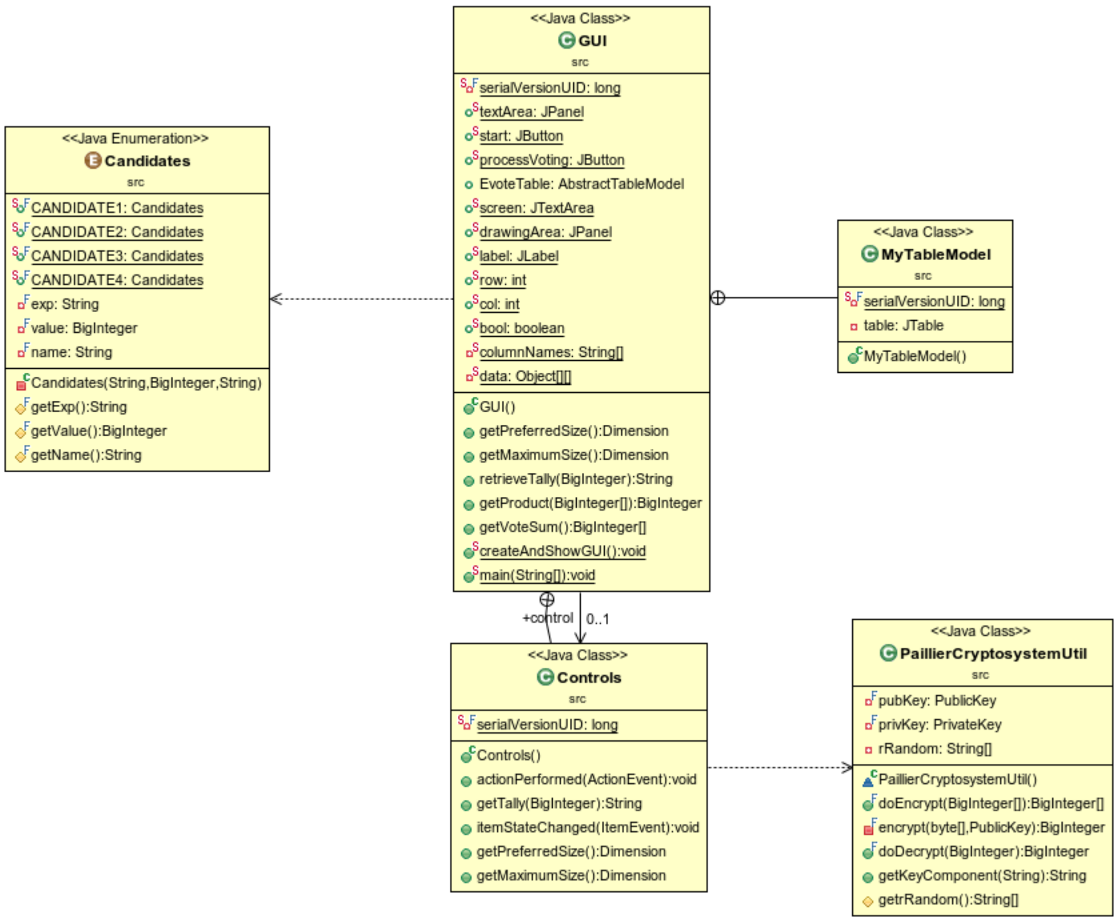


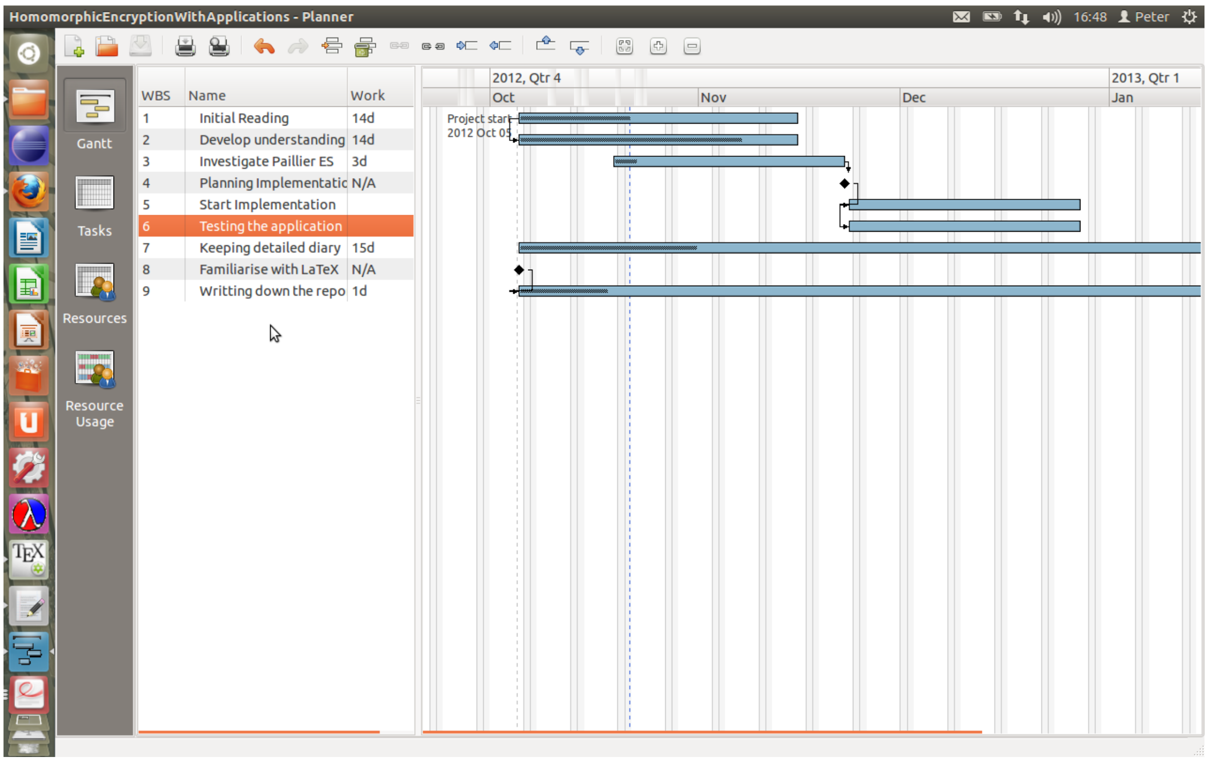
Figure A.3: Paillier E-Voting application - class diagram.

All source code has supported generated Java docs, each package contains a README.txt file as well.

Appendix B

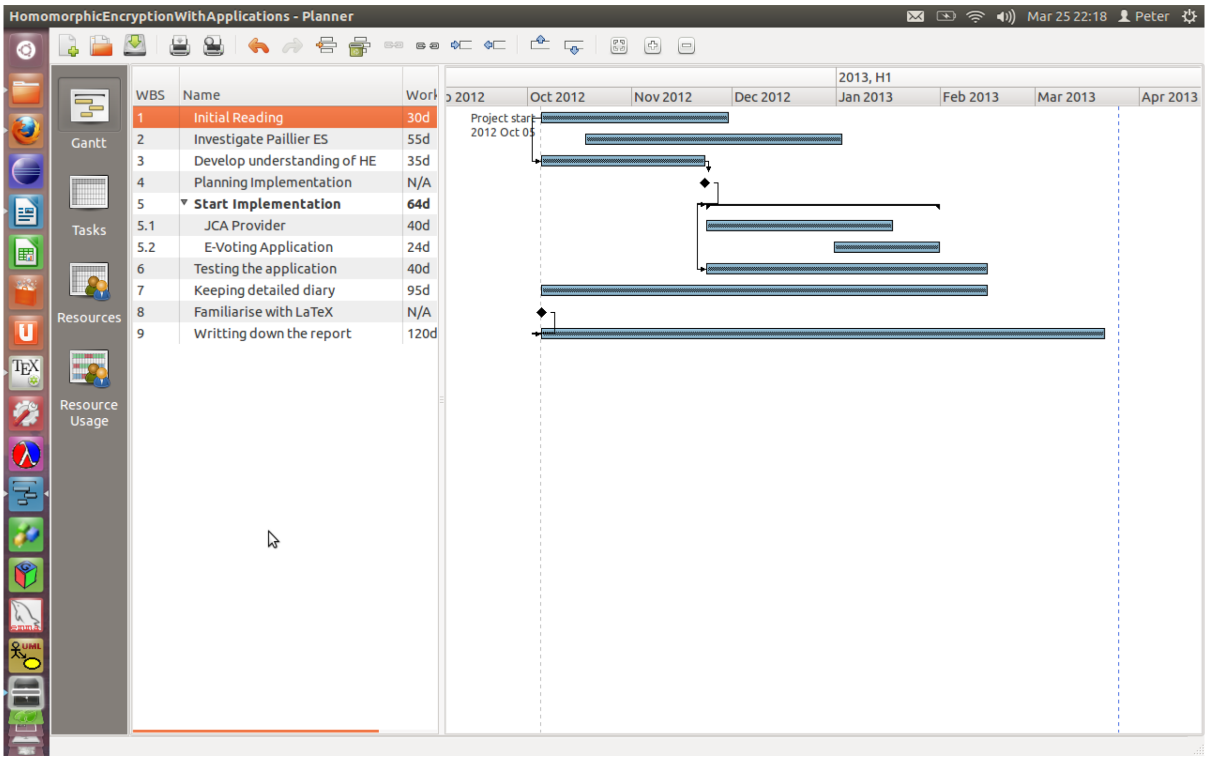
Software Project Management Plan

Difference between the initial and final project planning, due to the extra work performed in the implementation step and the introduction of JCA Provider.


HomomorphicEncryptionWithApplications - Planner

WBS	Name	Work
1	Initial Reading	14d
2	Develop understanding	14d
3	Investigate Paillier ES	3d
4	Planning Implementatic	N/A
5	Start Implementation	
6	Testing the application	
7	Keeping detailed diary	15d
8	Familiarise with LaTeX	N/A
9	Writting down the repo	1d

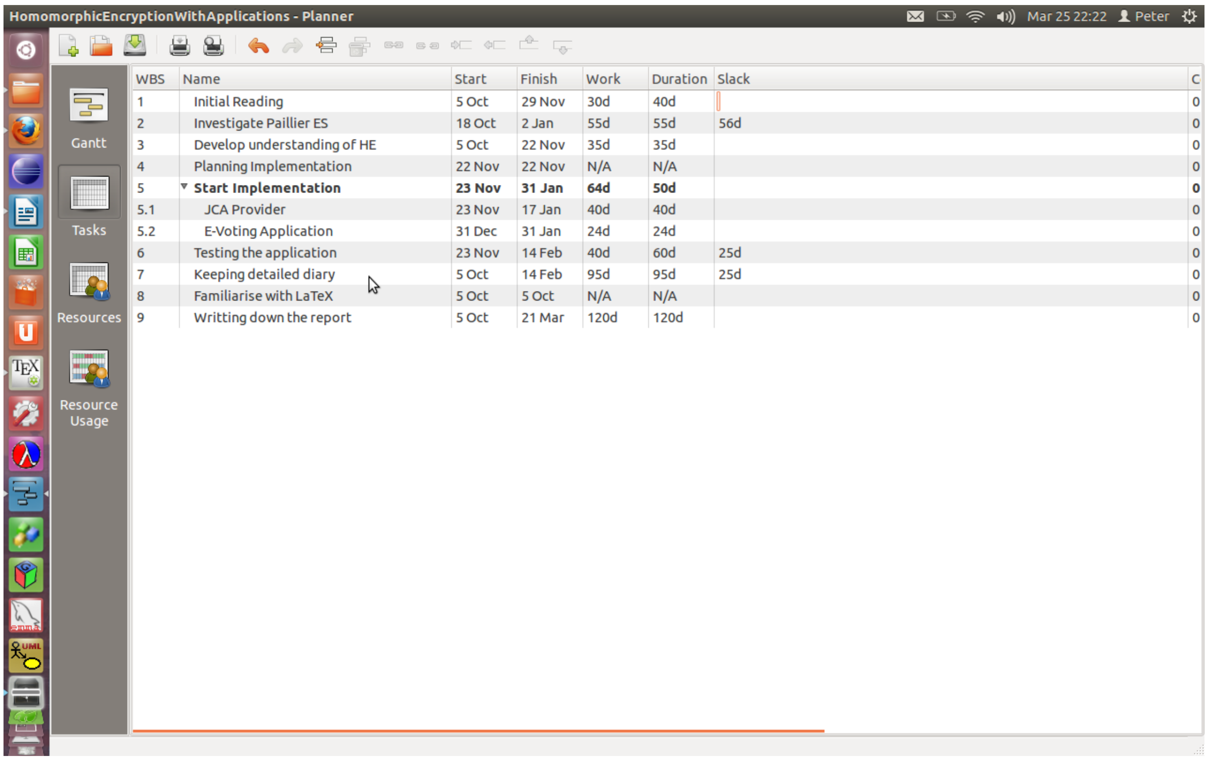
Figure B.1: Initial Gantt chart.


HomomorphicEncryptionWithApplications - Planner

WBS	Name	Work
1	Initial Reading	30d
2	Investigate Paillier ES	55d
3	Develop understanding of HE	35d
4	Planning Implementation	N/A
5	Start Implementation	64d
5.1	JCA Provider	40d
5.2	E-Voting Application	24d
6	Testing the application	40d
7	Keeping detailed diary	95d
8	Familiarise with LaTeX	N/A
9	Writting down the report	120d

Figure B.2: Final Gantt chart.

Task activities based on the final project plan shown below. Note - detailed diary is available on request.



WBS	Name	Start	Finish	Work	Duration	Slack	C
1	Initial Reading	5 Oct	29 Nov	30d	40d		0
2	Investigate Paillier ES	18 Oct	2 Jan	55d	55d	56d	0
3	Develop understanding of HE	5 Oct	22 Nov	35d	35d		0
4	Planning Implementation	22 Nov	22 Nov	N/A	N/A		0
5	**Start Implementation**	**23 Nov**	**31 Jan**	**64d**	**50d**		**0**
5.1	JCA Provider	23 Nov	17 Jan	40d	40d		0
5.2	E-Voting Application	31 Dec	31 Jan	24d	24d		0
6	Testing the application	23 Nov	14 Feb	40d	60d	25d	0
7	Keeping detailed diary	5 Oct	14 Feb	95d	95d	25d	0
8	Familiarise with LaTeX	5 Oct	5 Oct	N/A	N/A		0
9	Writting down the report	5 Oct	21 Mar	120d	120d		0

Figure B.3: TaskActivities.

Bibliography

[1] Paillier,Pascal., *Public-Key Cryptosystems Based on Composite Degree Residuosity Classes*, Published in J. Stern, Ed., Advances in Cryptology EUROCRYPT 99, vol. 1592 of Lecture Notes in Computer Science, pp. 223−238, Springer-Verlag, 1999.

[2] Katz,Jonathan.,& Lindell,Yehunda. *Introduction to Modern Cryptography:principals and protocols*, Chapman& Hall/CRC, 2008.

[3] Damgard,I. & Jurik, M.,J.,*A Generalisation, a Simplification and some Applications of Pailliers Probabilistic Public-Key System*, BRICS DS−00−45, http://www.brics.dk, 2001.

[4] Jurik, M.,J., *Extension to the Paillier Cryptosystem with Applications to Cryptographical Protocols*, BRICS DS−03−9, http://www.brics.dk, 2003.

[5] Koblitz, N., *A Course in Number Theory and Cryptography*, Springer, Second edition, 1994.

[6] Garrett, P., *Making,Breaking Codes: An Introduction to Cryptography*, Prentice−Hall, 2001.

[7] Gentry, Craig, *A Fully Homorphic Encryption Scheme*, Stanford University, PhD Dissertation, 2009.

[8] Menezes,A., van Oorschot,P., & Vanstone,S.,*Handbook of Applied Cryptography(Discrete Mathematics and Its Applications)*, CRC Press, 1996.

[9] Knudsen,J., *Java Cryptography*, O'Reilly& Associates, 1998.

[10] Naccache,D., & Stern J., *A New Public-Key Cryptosystem Based on Higher Residues*, LNCS 1403, Advances in Cryptology, Proceedings of Eurocrypt98, Springer−Verlag, p. 308−318, 1998.

[11] Oracle & its affiliates., *Java Cryptography Architecture − JCA Reference Guide*, http://docs.oracle.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html

[12] Choinyambuu, Sansar., *Homomorphic taylling with Paillier Cryptosystem*, E-Voting seminar, HSR Hochschule fr Technik Rapperswil, 12/6/2009

[13] Weiss, Jason., *Java Cryptography Extensions - Practical guide for programmers*, Elservier Inc, Morgan Kaufmann Publisher, 2004.