2006

Progress on probabilistic encryption schemes

Kert Richardson

Progress on Probabilistic Encryption Schemes

by

Kert Richardson

kert_richardson@frontiernet.net

This Master's Project is submitted to the Faculty of the Computer Science Departement of the Rochester Institute of Technology as part of the requirements to receive the degree of

Master of Science
in
Computer Science

July 7, 2006

Stanisław P. Radziszowski, Chairman

Chris Homan, Reader

James E. Heliotis, Observer

Contents

1 Abstract **4**

2 Acknowledgments **5**

3 Overview of Probabilistic Cryptography **6**

4 Contributions to Probabilistic Cryptography **7**

5 Goldwasser-Micali Scheme **9**
- 5.1 History 9
- 5.2 Mathematical Background 9
- 5.3 Summary of Scheme 14

6 Blum-Goldwasser Scheme **14**
- 6.1 History 14
- 6.2 Mathematical Background 14
- 6.3 Summary of Scheme 20

7 Paillier Scheme **20**
- 7.1 History 20
- 7.2 Mathematical Background 20
- 7.3 Summary of Scheme 27

8 Damgård-Jurik Scheme **28**
- 8.1 History 28
- 8.2 Mathematical Background 29
- 8.3 Summary of Scheme 34

9 Design of Cryptography Software **35**
- 9.1 Using Java Cryptography Extension 36
- 9.2 Message Communication Program 36
- 9.3 RSA Implementation 38
- 9.4 Paillier Implementation 38
- 9.5 Damgård-Jurik Implementation 39
- 9.6 Second Paillier Implementation 41

10 Test Results and Recommendations **42**
- 10.1 RSA Scheme Results and Observations 43
- 10.2 Paillier Scheme Results and Observations 44
- 10.3 Damgård-Jurik Scheme Results and Observations 45
- 10.4 Second Paillier Scheme Results and Observations 47
- 10.5 Comparative Graphs 48
- 10.6 Final Recommendations 51

A Homomorphic Encryption **53**
A.1 Addition of Multiple Ciphertexts 53
A.2 Addition of Plaintext Constants to Ciphertext 54
A.3 Multiplication of Plaintext Constants by Ciphertext 54

B Chinese Remainder Theorem in Public Key Decryption **55**

1 Abstract

The purpose of this master's project is to study different probabilistic cryptography schemes. The older probabilistic schemes, Goldwasser-Micali and Blum-Goldwasser, will only be covered briefly for a historical perspective. Several new and promising schemes have appeared in the last 7 years, generating interest. I will be examining the Paillier and Damgård-Jurik schemes in depth. This report explains the mathematics behind the schemes along with their inherent benefits, while also suggesting some potential uses. Details are given on how I optimized the algorithms, with special emphasis on using the Chinese Remainder Theorem (CRT) in the Damgård-Jurik algorithm as well as the other algorithms. One of the main benefits these schemes posses is the additively homomorphic property. I explain the homomorphic properties in the description of the schemes and give an overview of these properties in Appendix A.

I create software based in the Java Cryptography Extension (JCE) that is used to do a comparative study. This includes a simple message passing program for encrypted text. I create my own implementations of Paillier, Damgård-Jurik, and a variation of Paillier's scheme as a Provider using the JCE. These implementations use the CRT along with other methods to increase performance and create optimized algorithms. The implementations are plugged into the message passing program with an implementation of RSA from another Provider. A comparative study of the timings of these three schemes is done to show which one performs better in different circumstances. Conclusions are drawn based on the results of the tests and my final opinions are stated.

2 Acknowledgments

I would first like to thank all the members of my committee for taking time from their busy schedules to help with my project. I thank Prof. Radziszowski for sticking with me for over a year 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. I thank Prof. Homan for carefully reading over all the confusing sections of my paper and giving honest and helpful comments. I also thank Professor Heliotis for being here for the final presentation.

Second, I want to thank my family for supporting me through this entire experience. I thank my wife, Joanne, for being there and encouraging me when things took longer or were harder than I anticipated. I also thank my son, Ethan, for providing endless smiles as encouragement and motivation to finish the project.

Third, I would like to thank God for giving me the ability to understand this topic and the patience to finish the project. I thank him for answering my prayers, even though they weren't always in my timing. May I always keep in mind that "The fear of the Lord is the beginning of knowledge" - Proverbs 1:7a.

Without the help of those above I do not know how I would have completed the quality project that is before you.

3 Overview of Probabilistic Cryptography

The first thing that should be understood about probabilistic encryption is what makes it unique from other types of public key cryptography. Many of the schemes that are more familiar, like RSA, are deterministic. This means that for a fixed public key the same plaintext will always be encrypted to the same ciphertext. This characteristic can cause some security concerns. One concern is that it is easy to see if the same message is sent more than once. This could be very revealing in a voting protocol when there are only two responses, yes or no. Deterministic schemes also have problems encrypting a certain subset of the message space. This is the case in RSA where 0 and 1 are always encrypted to themselves.

Probabilistic schemes avoid the problems mentioned above because they use a random number every time a message is encrypted. Assuming the number is truly random, each time a particular message m is encrypted the ciphertext will change and be indistinguishable from the last ciphertext. Probabilistic encryption is a very strong encryption that is both polynomially and semantically secure. It should also be noted that any deterministic scheme could be altered to be probabilistic by adding a random number in the encryption and decryption [10].

It was pointed out to me that several other modern schemes could also be considered probabilistic, but I have hardly found any source mention this. Both the ElGamal and Elliptic Curve schemes have the same probabilistic characteristics mentioned above. So why don't they bear the "probabilistic" nameplate? This is most likely due to the lineage they evolved from. It seems that many probabilistic schemes evolved from the original Goldwasser-Micali scheme, which first defined the idea of probabilistic cryptography. These schemes are all based on finding the discrete log with very high residuosity classes. In [14], the author mentions how these probabilistic schemes evolved from Goldwasser-Micali in varying degrees: Benaloh's scheme [3], Naccache and Stern scheme [11], Okamoto and Uchiyama scheme [12], Paillier's scheme which I discuss, and Damgård-Jurik's system [7] which I also discuss. Schemes not possessing this lineage don't have this distinction but may still have the necessary characteristics. When looking for probabilistic characteristics, keep in mind they may be found in schemes not labeled as such or by altering a deterministic scheme to be probabilistic.

At the beginning of my research into probabilistic cryptography, my assumption was that most of my work would be on the Goldwasser-Micali and Blum-Goldwasser schemes. As I dug deeper, I realized that there were very few references to them in two decades since their creation. It didn't surprise me in the case of Goldwasser-Micali because of how slow it is and its simplistic nature. It seems that the inventors were simply trying a new concept, but it seems impractical for encrypting normal amounts of text. The Blum-Goldwasser scheme seemed much more promising, as it was another public key scheme that had similar performance to RSA. The flaw with Blum-Goldwasser comes from its security, which is susceptible to chosen ciphertext attacks. At first this didn't seem debilitating to the scheme, but after further research there seems to be a very plausible way an adversary could use this attack to find the private key and decipher the message.

It would be unlikely that anyone would use a scheme with this weakness and no other advantages, when other schemes are available.

With this information I was not sure probabilistic cryptography was going to be worth studying at all. The main schemes I had seen were interesting but not really worth spending much time on. Thankfully, I found another scheme that was considered probabilistic but was very different in its nature. This scheme was called the Paillier cyptosystem and has gained a good amount of attention since it was invented in 1999. The main attraction of this scheme is not that it is probabilistic. People are more interested in the fact it is homomorphic. This means that after a text has been encrypted, simple mathematical operations can be performed on the ciphertext [13]. The result can then be decrypted showing the operations occurred correctly. This property can be very useful and has been suggested to be used in electronic voting protocols and other areas. I have seen the Paillier scheme used for its homomorphic properties in several protocols, although none have made it to real world applications yet to my knowledge. There is more about the specific homomorphic properties of these schemes in Appendix A.

After Paillier came the Damgård-Jurik scheme which is really a derivation of Paillier's scheme. It allows for flexible length encoding with the same modulus n as a benefit. Increasing the length of the encrypted text greatly increases the length of the ciphertext. It will be determined later whether this benefit is practical and allows for faster encryption and decryption. Due to their relevance and the promising future of their properties, I will be spending most of my time studying these schemes. I have created software that implements both of these schemes along with different tools to time the encryption and decryption process. The program also uses an established version of RSA that I can also run timings on. These timings should provide good data to evaluate if Paillier or Damgård-Jurik can perform comparably to RSA.

I will still be writing a history and overview of the Goldwasser-Micali and Blum-Goldwasser schemes. Their current relevance seems limited, except when using Goldwasser-Micali for bit encryption. They are both historically significant and mathematically interesting, so I will explain in detail what is behind them and how they work, as well as their shortcomings. I will also give a simple example of encryption and decryption for each. Most of my time will be spent explaining the Paillier scheme and its derivative, Damgård-Jurik. These schemes are a bit more complicated and use properties and theorems that are less well recognized. I will explain and give simple examples for both of these as well. Fortunately, the Damgård-Jurik scheme will have most of the same properties as Paillier and I will just build off from what was already described.

4 Contributions to Probabilistic Cryptography

This section outlines the contributions I made to the field of probabilistic cryptography and cryptography in general, during my master's project. All of the points outlined are unique to this project to the best of my knowledge. Since I could not find research on these points previously recorded, I conducted my own research and reported it in this paper.

- Researched and reported a complete overview of several probabilistic cryptography schemes in the same document. Two of the schemes, Goldwasser-Micali and Blum-Goldwasser, appear to be less significant at this time. Both the Paillier and Damgård-Jurik schemes are being researched more because of their homomorphic and probabilistic properties. Damgård-Jurik also has the interesting trait of flexible length encoding without increasing the key size.

- Explained the significance of the homomorphic properties of Paillier and Damgård-Jurik in Appendix A. I give a description of these useful properties and the mathematics behind how they work.

- Helped to change the policy at Sun Microsystems to allow individuals not part of a licensed software company to obtain a certificate that is necessary to create software using the JCE. Without this policy change it is impossible for an individual to create software in the JCE. This may have also been partly due to some help from a professor. They seem more lenient now and I believe they will grant the certificate to people interested in using the JCE to create cryptography tools. You will have to contact Sun to find out.

- Using the Java Cryptography Extension (JCE) I implemented both the Paillier and Damgård-Jurik cryptosystems as a provider. Both systems can be accessed using the JCE through my jar file. These schemes were built using the JCE so that they can easily be pluged into different applications requiring public key encryption. Users should note that they were implemented only for academic purposes.

- When implementing Paillier and Damgård-Jurik schemes I spent considerable time improving performance over the basic algorithms. I used hints given in the original papers along with some ideas of my own to find the algorithms with the best performance. I used the Chinese Remainder Theorem (CRT) to greatly increase speed in all algorithms, similar to methods used in most RSA implementations. I used precomputed values in my algorithms as well as making sure all algorithms were performing as few operations as possible. I found a different algorithm of decryption for the Paillier scheme that has a different performance. My final version of each cryptosystem implements the fastest algorithms I was able to develop.

- Performed controlled tests on the cryptosystems RSA, Paillier, and Damgård-Jurik. After finding the quickest algorithms for Paillier and Damgård-Jurik, I tested them along with an RSA implementation to find which one performs best. I present my results and give my opinions on the best schemes and their uses.

- Conversed with Mads J. Jurik about how to properly use the CRT when implementing the Damgård-Jurik scheme. Through our conversations I helped him realize that the exponents used could be $(p-1)$ and $(q-1)$, instead of λ when using the CRT. This is a large performance increase that he had not considered during his original work. I outlined the entire algorithm of Damgård-Jurik that uses the CRT, as I was not able to find it anywhere else.

- Developed an algorithm to use the CRT when implementing the Second Paillier scheme. Recorded the algorithm in section on implementation and used algorithm in my final implementation of Second Paillier scheme.
- Explained the significance of homomorphic properties of Paillier and Damgård-Jurik schemes in Appendix A. I give a description of these properties and their mathematics.

5 Goldwasser-Micali Scheme

5.1 History

In 1984 Shafi Goldwasser and Silvio Micali theorized the idea of probabilistic cryptography [6]. Their scheme gave the ability to encrypt the same text in many different ways without changing the modulus. This scheme was one of the first to be semantically secure, if not the first. Semantic security is defined as the ciphertext not giving any useful information about the plaintext in polynomial time, except possibly the length [20]. This scheme introduced some new ideas, that helped to inspire a new line of probabilistic schemes. In fact, the Paillier scheme seems to have emerged from this line. Although it is significant for this purpose, Goldwasser-Micali doesn't seem to have been studied much because of its slow speed and huge expansion. For these reason it is doubtful that it could be used for encrypting normal amounts of text. However, the scheme does well at bit encryption because of its ability to encrypt bits to different values. For this reason the scheme may find some practical uses.

5.2 Mathematical Background

Goldwasser-Micali encryption uses the quadratic residuosity problem as the basis for the encryption. Assuming this problem is intractable then this scheme is secure. Of course, like most encryption schemes, this has not been proven secure but seems to be a good assumption.

A quadratic residue is simply an integer in Z_n^* that for some $x \in N$ is equivalent to $x^2 \bmod n$ [15]. Since we are in mod n, the integers can be quadratic residues in several ways. Most are values that would never be squares without evaluating with the modulus.

To understand the quadratic residues, we first need to understand the concepts of the Legendre and Jacobi symbols [15]. The Legendre symbol can easily be found with the formula:

$$(\tfrac{a}{p}) \equiv a^{\frac{p-1}{2}} \bmod p, \text{ where } p \text{ is any odd prime.}$$

If the value n is not prime then one has the Jacobi symbol $(\frac{a}{n})$ (which looks just like the Legendre symbol). If the factorization of $n = p_1^{e_1} p_2^{e_2} ... p_k^{e_k}$ the Jacobi symbol can be defined with Legendre symbols as:

$$(\tfrac{a}{n}) = (\tfrac{a}{p_1})^{e_1}(\tfrac{a}{p_2})^{e_2}...(\tfrac{a}{p_k})^{e_k}.$$

The quadratic residuosity problem is defined as the following.

Definition 1 *Given integers n and $a \in J_n$, where J_n is defined as all $a \in Z_n^*$ whose Jacobi symbol is 1, find whether or not a is a quadratic residue modulo n.*

To solve the Legendre symbol the value of n needs to be factored and then the equation can be solved with the resulting Jacobi symbols. The Jacobi symbols can be solved by the above method and the solutions can be multiplied together for the final solution.

This knowledge can be used to find quadratic residues. If there are only 2 factors of n, p and q then we can easily find if $a \in Q_n$, the set all quadratic residues modulus n. If the Legendre symbol $(\frac{a}{p_x}) = 1$, where p_x is any of the prime factors, and the Jacobi symbol $(\frac{a}{n}) = 1$ then this is a quadratic residue. It is possible for the Jacobi symbol $(\frac{a}{n}) = 1$ but a not be a quadratic residue. Because of this, without knowledge of p and q there is no way to know if this a quadratic residue.

With the mathematical background of the quadratic residuosity problem, we are now able to create the keys needed for encryption, and then encrypt and decrypt a message. The following is the method used for generating the public and private keys.

Steps for Key Generation

1. Select two primes p and q that are about the same size.
2. Calculate modulus $n = pq$, the product of two primes.
3. Select $y \in Z_n$ that is also a pseudosquare mod n. A pseudosquare is a quadratic non-residue modulo n with $(\frac{a}{n}) = 1$.
4. We now have the public key (n, y) and the private key (p, q)

The key generation is mostly straightforward for this scheme and resembles other public keys schemes like RSA. The tricky part is creating the pseudosquare y. From the name, one can infer that this value will look like a square. In this case, the Jacobi symbol $(\frac{a}{n}) = 1$ is a test to see if a is quadratic residue modulo n. But just because the Jacobi symbol is 1 doesn't mean it is a quadratic residue. There is also a chance that that this value is not a quadratic residue and it is precisely these values that represent pseudosquares. There is a good trick for creating these values when n is a composite of two primes p and q [10].

Steps for Finding Pseudosquare

1. Find two quadratic non-residues, a mod p and b mod q.
2. You want to find the number y mod n (remember $n = pq$) where $y \equiv a$ mod p and also $y \equiv b$ mod q. This can be done using Chinese Remaindering.

3. We know y is a quadratic non-residue mod n. This is because a mod p is in $\bar{Q}_p$ ($\bar{Q}$ is the set of quadratic non-residues) and b mod q is in $\bar{Q}_q$, which implies they are both quadratic non-residues mod n (see fact 2.137 in [10]). Remember that $y \equiv a$ mod p and $y \equiv b$ mod q.

4. We also know by the properties of Jacobi and Legendre symbols that the Jacobi symbol $(\frac{a}{n}) = (\frac{a}{p})(\frac{b}{q}) = (-1)(-1) = 1$, since the Legendre symbol of quadratic non-residues must be -1.

5. From all the above we can conclude we have a pseudosquare, since the Legendre symbol $(\frac{a}{p}) = 1$ and the Jacobi symbol $(\frac{a}{n}) = 1$.

Now that we have the public and private keys we can see how to do encryption with Goldwasser-Micali. In our case Bob is encrypting a message m for Alice into a ciphertext c. Bob has received the public key (n, y) that was generated by Alice.

Steps for Encryption

1. Convert message m to a binary string where $m = m_1m_2m_3...m_t$ and t is the length of string and each m_i is either 0 or 1.

2. For $i = 0$ to t

   (a) Pick a random value $x \in Z_n^*$.

   (b) If $m_i = 1$ then $c_i = yx^2$ mod n and if $m_i = 0$ then $c_i = x^2$ mod n.

3. The encrypted values c_i are then placed in c where $c = c_1c_2c_3...c_t$ where again t is the size c.

It is clear from the encryption algorithm that this is an inefficient scheme and somewhat simplistic. Only one bit can be encrypted at a time and the expansion is very great. Even with these characteristics I am fascinated that with the randomness one can encrypt 0's and 1's in so many ways and yet this is considered secure. It makes you wonder if there are other easier methods to encrypt each bit, since all you need is two different outcomes. This type of scheme could be usable when doing bit encryption but would take too long for normal size encryption.

The decryption of ciphertext c of length t is as follows using the private key (p, q).

Steps for Decryption

1. For $i = 0$ to t

   (a) Find the Legendre symbol $l_i = (\frac{c_i}{p})$.

   (b) If $l_i = 1$ then $m_i = 0$, else $m_i = 1$.

2. The decrypted message is the string $m = m_1 m_2 m_3 ... m_t$.

The decryption process, as you can see, is as simple as figuring out if c_i was encrypted as quadratic residue modulo n or if it was encrypted as a pseudosquare. If c_i is encrypted as $x^2 \bmod n$ then we obviously get a quadratic residue mod n, since that is the definition of a quadratic residue. If $c_i = yx^2$ then we must have a pseudosquare. This is because the Jacobi symbol of y is $(\frac{y}{n}) = -1$ and the Jacobi symbol of x^2 is $(\frac{x^2}{n}) = 1$. Therefore we get $(-1)(1) = -1$ which implies that yx^2 is a pseudosquare mod$n$. To quickly show whether or not c_i is a quadratic residue modulo n, we use a fact stated in [10] in section 2.137, which says that if c_i is a quadratic residue modulo q or modulo p then we know it is a quadratic residue modulo n. This can be done easily, since p and q are primes, with either Legendre symbol $(\frac{c_i}{p})$ or $(\frac{c_i}{q})$.

The ability to find if a number is a quadratic residue modulo n using its factors p and q is especially useful because no one knows a method of finding quadratic residue without these factors. Since an effective way to factor large numbers is unknown, this scheme is considered secure. The security of this scheme relies on the assumption that the quadratic residuosity problem is difficult, but for now it seems to be secure. If a person were able to intercept a message, all they would see would be pseudosquares and quadratic residues modulo n. Assuming the value of n is large enough, they would not be able to get the factors p and q and would not be able to calculate the Legendre symbol.

Below is a simple example of the Goldwasser-Micali scheme. You can tell from the size of the numbers in the public and private keys that this is not realistic for a truely secure example. But the mathematics are the same and help to cement in your mind how the scheme works. The steps shown in the example below correspond to the algorithms above, so you can compare.

Key Generation

1. $p = 71, q = 61$.

2. $n = 4331$.

3. I selected several random numbers to see if they were quadratic non-residues modulo n. In the end I found the Legendre symbol $(\frac{17}{71}) \equiv 17^{(71-1)/2} \pmod{71} \equiv -1$ and $(\frac{23}{61}) \equiv 23^{(61-1)/2} \pmod{61} \equiv -1$. You can then use Chinese Remaindering to find the equivalent of both these numbers mod$n$. We know the $gcd(p, q) = 1$ since they are both prime, so we can first use the formula $k \equiv (a - b)p^{-1} \pmod{q}$. Plugging in the numbers we get $k \equiv (23 - 17)71^{-1} \pmod{61} \equiv 25 \pmod{61}$. Then we can substitute k into $b + pk \pmod{n}$ giving us $17 + 71 * 25 \equiv 1792 \pmod{4331}$. This means we can use $y = 1792$ as our pseudosquare mod n.

This gives us the public key $(n, y) = (4331, 1792)$ and the private key $(p, q) = (71, 61)$.

Encryption

We will now encrypt the message $m = 9$. This means $m = 1001$ in binary, which is a good number to encrypt since there are two 0's and two 1's to encrypt. Obviously we can encrypt much bigger numbers but we would just be encrypting the same two things over and over in a certain order. Having two examples of each should be sufficient to understand the process.

- $m_1 = 1$
  We choose $x \in Z_n^*$ so $x = 12$.
  $m_1 = 1$ so $c_1 \leftarrow yx^2 \bmod n$ which gives us
  $c_1 = 1792 * 12^2 = 2512 \bmod 4331$.

- $m_2 = 0$
  We choose random $x = 22$.
  $m_2 = 0$ so $c_2 \leftarrow x^2 \bmod n$ which gives us
  $c_2 = 22^2 = 484 \bmod 4331$.

- $m_3 = 0$
  We choose random $x = 81$.
  $m_3 = 0$ so $c_3 \leftarrow x^2 \bmod n$ which gives us
  $c_3 = 81^2 = 1378 \bmod 4331$

- $m_4 = 1$
  We choose random $x = 3001$.
  $m_4 = 1$ so $c_4 \leftarrow yx^2 \bmod n$ which gives us
  $c_4 = 1792 * 3001^2 = 2421 \bmod 4331$

This gives us the ciphertext $c = (2519, 484, 1378, 2421)$ that is sent to Alice to be decrypted.

Decryption

- $c_1 = 2519$
  We calculate the Legendre symbol $(\frac{c_1}{p}) \equiv c_1^{(p-1)/2} \bmod p$.
  This gives us $(\frac{2519}{71}) = 2519^{((71-1)/2)} \bmod 71 = -1$.
  Because we get -1, $m_1 = 1$.

- $c_2 = 484$
  The Legendre symbol $(\frac{484}{71}) \equiv 484^{((71-1/2)} \bmod 71 = 1$.
  Since we get 1, $m_2 = 0$.

- $c_3 = 1378$
  The Legendre symbol $(\frac{1378}{71}) \equiv 1378^{((71-1/2)} \bmod 71 = 1$.
  Since we get 1, $m_3 = 0$.

- $c_4 = 1238$
  The Legendre symbol $(\frac{1238}{71}) \equiv 1238^{((71-1/2)} \bmod 71 = -1$.
  Since we get -1, $m_4 = -1$.

As you can see when we concatenate m together again we get $m = 1001$ in binary or $m = 9$, which is what we started with.

5.3 Summary of Scheme

As you can see from the information provided, this scheme works and appears to be semantically secure. The problem is that the scheme has a message expansion around $\lg_2 n$ [10]. The value of n would need to be hundreds of bits long to prevent factorization and finding the private key. This means that each bit of the ciphertext would need to be expanded to just smaller than n. In an example that was reasonably secure this would make the ciphertext hundreds of times larger than the original message. Because of this fact, this scheme is not used practically but it may become useful for encrypting small amounts, such as invidual bits. The other probabilistic schemes covered later have much potential for normal encryption than this one.

6 Blum-Goldwasser Scheme

6.1 History

After the Goldwasser-Micali scheme came the Blum-Goldwasser scheme in 1985, which is based on the Blum-Blum-Shub generator [4]. The Blum-Goldwasser scheme is comparable in speed to another public key scheme, RSA. With this in mind, it does seem possible that this scheme may have potential to be studied and used in applications where RSA is currently used. Unfortunately, it is not as robust as RSA, since it has been shown to be susceptible to chosen ciphertext attacks. Because of this, the scheme has not generated much interest and has not been studied recently.

6.2 Mathematical Background

The Blum-Goldwasser scheme uses the Blum-Blum-Shub generator to create a pseudo-random bit sequence that is used for encryption. The Blum-Blum-Shub generator is considered a cryptographically secure pseudorandom bit generator under the assumption that integer factorization is intractable [15]. It provides a solid foundation encoding and decoding as you will see late in this section.

The first step to understanding Blum-Goldwasser is understanding how the Blum-Blum-Shub (BBS) generator works. The heart of this scheme is just creating a random quadratic residue x_0 and then encrypting all values with basic math and XOR's. When decrypting it is difficult to retrieve the random quadratic residue x_0. Once you have that, you just use the same math and XOR's to retrieve the plaintext.

The Blum-Blum-Shub generator generates a pseudorandom bit sequence $z_1, z_2, ..., z_l$ of length l. The following method describes it [10].

Steps for Blum-Blum-Shub generator

1. Select two distinct large primes p and q each congruent to $3 \pmod 4$.
2. Calculate modulus $n = pq$, the product of two primes.
3. Select the random seed s where $1 \leq s \leq n - 1$ such that $\gcd(s, n) = 1$.
4. Compute $x_0 \leftarrow s^2 \pmod n$.
5. For i to l compute the following:
   (a) $x_i \leftarrow x_{i-1}^2 \pmod n$.
   (b) $z_i \leftarrow$ the least significant bit of x_i (in our case we will use more than the least significant bit).
6. The sequence that is output ends up as $z_1, z_2, ..., z_l$.

As you can see from the algorithm, you select both the modulus and the seed with specific properties. Then square the seed and take its modulus, which gives a value where the least significant bit is considered random. To get more random bits, keep squaring the answer and taking the modulus. The algorithm is designed so that you can't predict the results without knowing the initial seed and modulus. In the Blum-Goldwasser scheme we will use not just the least significant bit, but several bits.

Now that we have seen the Blum-Blum-Shub generator we can see how it is incorporated into the scheme. The first step in describing Blum-Goldwasser is generating the public and private keys.

Steps for Key Generation

1. Select two large primes p and q that are about the same size and are congruent to 3 mod 4.
2. Calculate the modulus $n = pq$, the product of two primes.
3. Calculate a and b such that $ap + bq = 1$. This can be done with the extended Euclidean algorithm [15].
4. We now have the public key (n) and the private key (p, q, a, b).

With the private and public keys we can perform encryption and decryption.

Steps for Encryption

1. Let $k \leftarrow \lfloor \lg n \rfloor$ and $h \leftarrow \lfloor \lg k \rfloor$.
2. The message m is represented as a string where $m = m_1 m_2 ... m_t$ of length t. Each piece m_i is a binary string h bits long.
3. Select a seed x_0, which is also a quadratic residue modulo n. This is done by randomly choosing an integer $r \in Z_n^*$ and setting $x_0 = r^2 \bmod n$.
4. Compute the following using x_0
   for $i = 1$ to t:
   (a) Find $x_i = x_{i-1}^2 \bmod n$.
   (b) We let p_i be the h least significant bits of x_i.
   (c) Now compute $c_i = p_i \oplus m_i$, which gives us the ciphertext.
5. Compute $x_{t+1} = x_t^2 \bmod n$, which is a value that is sent with the ciphertext to help recalculate the seed x_0.
6. This gives us ciphertext $c = (c_1, c_2, ..., c_t, x_{t+1})$ which can be used with the private key for decryption. You can note the value x_{t+1} is not a typical ciphertext value, since it was not derived in any way from the plaintext, but is necessary for decryption.

The key to understanding this encryption is in step 3. As you can see x_0 is just a random quadratic residue modulo n. With the seed x_0, you repeatedly square it modulo n and take the result to get each value x_i. The masking of the text is done with the simple XOR operation. Take the bits x_i and XOR them with the bits m_i, which computes the ciphertext c_i.

The key to the decryption, as you will see next, is retrieving the value x_0. Once you get this value back, you can easily recompute each x_i, perform the XOR with c_i, and regain the original text. We are able to retrieve this value using x_{t+1}, which is sent with the ciphertext.

The steps to the decryption of ciphertext c using private key (p,q,a,b) is as follows.

Steps for Decryption

1. Compute $d_1 = (\frac{p+1}{4})^{t+1} \pmod{p-1}$.
2. Compute $d_2 = (\frac{q+1}{4})^{t+1} \pmod{q-1}$.
3. Compute $u = x_{t+1}^{d_1} \bmod p$.
4. Compute $v = x_{t+1}^{d_2} \bmod q$.
5. Compute $x_0 = vap + ubq \bmod n$.
6. Compute the following using x_0 for $i = 1$ to t:

(a) Find $x_i = x_{i-1}^2 \bmod n$.

(b) We let p_i be the h least significant bits of x_i.

(c) Now compute $m_i = p_i \oplus c_i$.

As you can see in the decryption, the majority of the work is computing the value x_0 from the private key values p and q and the value x_{t+1}.

The way we know this works starts with the fact that $x_t^{\frac{p-1}{2}} \equiv 1$. This is true because of Fermat's Little Theorem (see [16]) which states:

Theorem 1 *If p is a prime and p does not divide a, then $a^{p-1} \equiv 1 \bmod p$.*

This works in our case because x_t is a quadratic residue so we know $x_t^{\frac{1}{2}}$ exists and will work in Fermat's Little Theorem. We can rewrite our congruence as $x_t^{(\frac{1}{2})^{(p-1)}} \equiv 1$. Now that we know this, observe that:

$$x_{t+1}^{\frac{p+1}{4}} \equiv (x_t^2)^{\frac{p+1}{4}} \equiv x_t^{\frac{p-1}{2}} x_t \equiv x_t \pmod p.$$

Remember that $x_{t+1} \equiv x_t^2 \bmod p$. So the above formula gives us a way to take a quadratic residue and find the value which was squared to create it. This process can be repeated to find as many quadratic residues as needed. We know in our case this will be $(t+1)$ times to retrieve x_0:

$$x_{t+1}^{(\frac{p+1}{4})^2} \equiv x_{t-1} \bmod p$$

which leads to the following formula used in steps 1 and 3 of our algorithm:

$$u \equiv x_{t+1}^{d_1} \equiv x_{t+1}^{(\frac{p+1}{4})^{t+1}} \equiv x_0 \bmod p.$$

This can also be done using q, the other factor of n. This gives us the following formulas used in steps 2 and 4 in our algorithm:

$$v \equiv x_{t+1}^{d_2} \equiv x_{t+1}^{(\frac{q+1}{4})^{t+1}} \equiv x_0 \bmod q.$$

Both values u and v are congruent to $x_0 \bmod p$ and $\bmod q$ respectively but we are looking for $x_0 \bmod n$. This can be found using the Chinese Remainder Theorem (see [16]) and the values a and b that are generated in the key generation. Now that we have the root x_0 we can generate values x_1 to x_t and compute the plaintext from the ciphertext. Notice that step 6 in decryption is the same as step 4 in encryption, except you calculate the plaintext m_i from p_i and the ciphertext c_i.

Here is a simple example of the Blum-Goldwasser encryption scheme. As you can tell from the size of the modulus, it would not be considered secure against attacks, but is sufficient to show how the mathematics work. The numbering of the steps below will correspond to the algorithms we described above.

Key Generation Example

1. $p = 607, q = 563$ where both p and $q = 3 \bmod 4$.
2. $n = pq = 341741$.
3. Using the Extended Euclidean algorithm to calculate $a(607) + b(563) = 1$. This gives us $a = 64$ and $b = -69$.

We get the public key $n = (341741)$ and the private key $(p, q, a, b) = (607, 563, 64, -69)$.

Encryption Example

We will now encrypt the value $m = 49827$.

1.
   - $k = \lfloor log_2 n \rfloor = \lfloor log_2 341741 \rfloor = 18$.
   - $h = \lfloor log_2 k \rfloor = \lfloor log_2 18 \rfloor = 4$.
2. We pick a random number $r = 4561 \in Z_n^*$ that we use to create our base quadratic residue. This gives us $x_0 = 4561^2 \bmod n = 298261 \bmod 341741$.
3. For $i = 1$ to 4
   (a)
   - $x_1 = x_0^2 \bmod n = 317603 \bmod 341741$.
   - The 4 least significant digits of x_1 base 2, $p_1 = 0011$.
   - This ciphertext $c_1 = 0011 \oplus 1100 = 1111$.

   (b)
   - $x_2 = x_1^2 \bmod n = 340929 \bmod 341741$.
   - The 4 least significant digits of x_2 base 2, $p_2 = 0001$.
   - This ciphertext $c_2 = 0001 \oplus 0010 = 0011$.

   (c)
   - $x_3 = x_2^2 \bmod n = 316380 \bmod 341741$.
   - The 4 least significant digits of x_3 base 2, $p_3 = 1100$.
   - This ciphertext $c_3 = 1100 \oplus 1010 = 0110$.

   (d)
   - $x_4 = x_3^2 \bmod n = 23759 \bmod 341741$.
   - The 4 least significant digits of x_4 base 2, $p_4 = 1111$.
   - This ciphertext $c_4 = 1111 \oplus 0011 = 1100$.
4. We also need to calculate $x_5 = x_4^2 \bmod n = 275690 \bmod 341741$.
5. The ciphertext that is sent is $(c_1, c_2, c_3, c_4, x_5) = (1111, 0011, 0110, 1100, 275690)$.

Decryption Example

1.
   - $d_1 = (\frac{p+1}{4})^{t+1} \pmod{p-1}$.
   - $d_1 = (\frac{608}{4})^5 \pmod{606}$.
   - $d_1 = 464$.

2.
   - $d_2 = (\frac{q+1}{4})^{t+1} \pmod{q-1}$.
   - $d_2 = (\frac{564}{4})^5 \pmod{562}$.
   - $d_2 = 483$.
3.
   - $u = x_{t+1}^{d_1} \pmod{p}$.
   - $u = 275690^{464} \pmod{607}$.
   - $u = 224$.
4.
   - $v = x_{t+1}^{d_2} \pmod{q}$.
   - $v = 275690^{483} \pmod{563}$.
   - $v = 434$.
5.
   - $x_0 = vap + ubq \pmod{n}$.
   - $x_0 = 434(64)(607) + 224(-69)(563)$.
   - $x_0 = 298261 \pmod{341741}$.
6. We can now recalculate the values x_i from x_0 and use them to decrypt the ciphertext. For $i = 1$ to 4.
   (a)
   - $x_1 = x_0^2 \bmod n = 317603 \pmod{341741}$.
   - The 4 least significant digits of x_1 base 2, $p_1 = 0011$.
   - This plaintext $m_1 = 0011 \oplus 1111 = 1100$.

   (b)
   - $x_2 = x_1^2 \bmod n = 340929 \pmod{341741}$.
   - The 4 least significant digits of x_2 base 2, $p_2 = 0001$.
   - This ciphertext $m_2 = 0001 \oplus 0011 = 0010$.

   (c)
   - $x_3 = x_2^2 \bmod n = 316380 \pmod{341741}$.
   - The 4 least significant digits of x_3 base 2, $p_3 = 1100$.
   - This ciphertext $m_3 = 1100 \oplus 0110 = 1010$.

   (d)
   - $x_4 = x_3^2 \bmod n = 23759 \pmod{341741}$.
   - The 4 least significant digits of x_4 base 2, $p_4 = 1111$.
   - This ciphertext $m_4 = 1111 \oplus 1100 = 0011$.

You can then concatenate the values of m_i together again and get $m = 1100001010100011$ base 2 or $m = 49827$. The real test of whether or not the decryption will work is if you correctly recalculate the value x_0 in step 5, which is most of the work. The rest of the work in step 6 is mostly the same as step 4 of the encryption, except you are calculating the plaintext instead of the ciphertext.

6.3 Summary of Scheme

So far from what we've seen this scheme looks pretty good. The efficiency is much better than Goldwasser-Micali, as the expansion is only a reasonably sized constant value. All the encrypted values c_i are exactly the same size as the plaintext values m_i. This only leaves the value x_{t+1} as the expansion. The speed of Blum-Goldwasser also does very well, especially compared to RSA. According to [10], it is just faster or just slower than RSA depending on the exact situation and whether or not special efficient values are chosen for RSA.

If you are looking for the catch, here it is: The security of the scheme has been shown to be vulnerable to chosen-ciphertext attacks. This may not seem that devastating if you can prevent an adversary from getting a carefully selected message encrypted with Blum-Goldwasser. Unfortunately, the practicality of this is very feasible. In [20] there is a description of several strategies for attacking this scheme by sending carefully selected values to be encrypted and evaluating the results. This leads to Wenbo's evaluation that the security notion "is hopelessly weak."[20] With this type of assessment from other cryptography experts as well, it is no wonder this scheme has also not found any practical uses. It is also generally just studied as an interesting concept without a good use.

7 Paillier Scheme

7.1 History

The Paillier scheme was first published by Pascal Paillier in 1999. This probabilistic scheme has generated a good amount of interest and further study since it was discovered. The main interest seems to be centered around another property it possesses: the 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. Because of this useful characteristic the scheme has been suggested for use in the design of voting protocols, threshold cryptosystems, watermarking, secret sharing schemes, private information retrieval, and server-aided polynomial evaluation.

7.2 Mathematical Background

The basis of the Paillier scheme is composite residuosity. In our case $n = pq$ is a composite since n is a composite of pq. This gives us the following definition from [14].

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

The problem of trying to distinguish nth residues from non-nth residues is seen as hard enough to form the basis of this scheme. This problem is known to be random self-reducible so that all of its instances are polynomially equivalent. In Wikipedia, the definition of a function that possesses random self-reducibility is [2]:

Definition 3 *A good algorithm for the average case implies a good algorithm for the worst case. Random self-reducibility is the ability to solve all instances of a problem by solving a large fraction of the instances.*

Therefore the problem is either easily solvable everywhere or not at all (random self-reducibility will be discussed more when showing how it is used in the Paillier scheme). Currently, like RSA and others, there is no polynomial time algorithm to solve the Paillier algorithm without the private key.

First there are a couple of functions that we will discuss, since they are important in understanding how Paillier's scheme works. The first is the somewhat popular Euler totient function, which is generally denoted as $\phi(n)$. The function is defined [10]:

Definition 4 *For n the product of two primes p and q, $\phi(n) = (p-1)(q-1)$.*

The second function is the Carmichael function, which seems much less known and is generally denoted as $\lambda(n)$ or just λ with the n assumed. The function is defined as follows with some interesting properties [18] [19]:

Definition 5 *For n the product of two primes p and q, $\lambda(n) = lcm(p-1, q-1)$.*
For any $w \in Z^*_{n^2}$, $w^\lambda = 1 \bmod n$ and $w^{\lambda n} = 1 \bmod n^2$.

We can now talk about the framework that will be used for encryption. It is defined as follows:

Definition 6 *For $g \in Z^*_{n^2}$ we denote the function $E_g(x, y) \rightarrow g^x y^n \bmod n^2$, where $x \in Z_n, y \in Z^*_n$.*

By observing some special cases using E_g, one can see how to create a solid scheme that always works and is hard to decipher.

The first thing to discuss is this lemma, referred to in [14].

Lemma 1 *If the order of g is a nonzero multiple of n then E_g is bijective.*

The definition of bijective tells us that when g follows the criteria of the lemma, there will be one unique solution to the function E_g when given the values for x and y. In other words given the needed values, $E_g(x, y)$ yields one solution, instead of 2 or more if g does not follow the criteria of the lemma. This lemma allows us to know that by using the proper value of g we always have a good solution. This ability to encrypt one value to exactly one solution is important in any cryptosystem. It would be impossible to decrypt a value that could come from two different encrypted vales or encrypt a value that gave no solution.

Looking at the charactertics of g according to the lemma, it says we want the order of g to be a multiple of n. Recalling the definition, we know that the order is the least positive t where $g^t = 1 \bmod n$ [15]. In our case we are working in mod n^2, so we are looking for the orders that are multiples of n, so $t = n\alpha \bmod (n^2)$ which means $g^{\alpha n} = 1 \bmod n^2$ for some α. By the properties of the Carmichael function we know that the highest order of g is $n\lambda$, which shows us that $1 \leq \alpha \leq \lambda$. The set of elements in $Z^*_{n^2}$ of order $n\alpha$ where $1 \leq \alpha \leq \lambda$ is referred to as β by Paillier and will be the set used to select g.

While the mathematics of making g any element of β works, the practicality of it may not. So far I haven't seen an efficient method for finding an arbitrary value in β. Paillier suggests choosing a g at random and then checking whether

$$\gcd(L(g^\lambda \bmod n^2), n) = 1.$$

It could take a good amount of time to find $g \in \beta$ with this method, especially since it can be hard to find these values. Fortunately we are not left with choosing any value in β, but we have a much better choice. Practically, it seems that $(1 + n) \in \beta$ works well in all circumstances and is easily computed. In fact, Paillier uses this value in his proofs. It also seems more recent sources have dropped the value of g altogether and simply replaced it with $1 + n$ (see [1]). As we will see, all good values of g are equivalent, so it makes great sense to remain with $1 + n$ which always works. The explanation above describing all values of β may be unnecessary. Rest assured that setting $g = 1 + n$ seems to always work in proofs, besides being simpler to understand and compute.

With the previous information we can define the equation that [14] uses.

Definition 7 *Assume that $g \in \beta$. For $w \in Z^*_{n^2}$, we call the nth residuosity class of w with respect to g the unique integer $x \in Z_n$ for which there exists $y \in Z^*_n$ such that $E_g(x, y) = w$.*

and

Definition 8 *If $w \in Z^*_{n^2}$, define $[w]_g$ as the smallest non-negative ineteger x such that w is expressible as $w = g^x y^n \bmod n^2$, where $g \in Z^*_{n^2}$ and $y \in Z^*_n$.*

The notation for $[w]_g$ came from Benaloh's Ph.D. on residue classes [3]. Paillier used the same notation but it a somewhat confusing manner. He calls $[w]_g$ the class of w without ever really defining what it is. I use a definition similar to the one in Benaloh's Ph.D., which is simpler to understand and never uses the term class to my knowledge. The basic meaning of $[w]_g$ implies that we are given $g \in \beta$ and $w \in Z^*_{n^2}$. There is also the random value $y \in Z^*_n$ that could be many values. With these values we are able to find x.

By finding $[w]_g$, we find the exponent of g, which is x. In the equation $w = g^x y^n \bmod n^2$ we are given w, g, y, and n. As you will see later on, we use x as the value being encrypted. Thus solving $[w]_g$ becomes the act of decryption and will give us back the original value.

Another useful observation that Paillier makes is that this problem is random self-reducible. You can recall the definition we gave earlier in this section. Basically, it refers to all instances of a problem being polynomially equivalent when a certain criteria is met

[2]. This is done by showing that when evaluating the worst case scenario of a function, it is polynomially equivalent to the average case. There still may be significant difference in time between the average and worse cases but the important factor is the polynomial time. Because the complexity of the average and worst cases are equivalent for random instances of the function, we are able to call it random self-reducible. This should give a good overview of the concept. If this concept is completely unknown to you, it may help to view the source [2] to get a full decription of random self-reducibility.

In our case the problem of computing $[w]_g$ given w is random self-reducible in two ways. In his paper Paillier proves that this problem is random self-reducible over $w \in Z^*_{n^2}$, and also over $g \in \beta$. In essence, these two observations show that the problem of finding the class is equally as hard, no matter what values of w and g are chosen from the given sets. This leads to the definition of the Composite Residuosity Class Problem from [14].

Definition 9 *We call the Composite Residuosity Class Problem the computational problem Class*$[n]$ *defined as follows: given* $w \in Z^*_{n^2}$ *and* $g \in \beta$*, compute* $[w]_g$.

It is also shown that the problem Class$[n]$ is equivalent to the problem of factoring n. This shows that there is no known polynomial time algorithm to solve Class$[n]$. It should be equivalent to the the RSA scheme as far as security goes. Since RSA seems to be one of the current standards, it bodes well for Paillier's scheme as far as security is concerned.

At this point we can start to describe a method to solve the problem Class$[n]$. Before doing this we need to describe a set of integers S_n and the function L.

$$S_n = \{u < n^2 | u = 1 \bmod n\}$$

This leads to the following function.

$$u \in S_n,\ L(u) = \tfrac{u-1}{n}$$

Now that we have this we are able to describe this Lemma from [14] which shows a method used to retrieve $[w]$ when given w. Keep in mind this is the unique number x that was descibed earlier in the base equation. You will also see that the value x is the plaintext we are retrieving.

Lemma 2 *For any* $w \in Z^*_{n^2}$*,* $L(w^\lambda \bmod n^2) = \lambda[w]_{1+n} \bmod n$.

To demonstrate this works we let $g = (1+n) \in \beta$. Since we already know that all $g \in \beta$ are equivalent, we can use this simple g to prove Class$[n]$ is always solvable. We already know that $w = (1+n)^x y^n \bmod n^2$ when we replace g with $(1+n)$. By the previous definition we are looking for the exponent of $(1+n)$ or $[w]_{1+n}$, which is called x. So putting this all together we get:

Equation 1.

$$w^\lambda = (1+n)^{x\lambda} y^{n\lambda} = (1+n)^{x\lambda} = 1 + x\lambda n \bmod n^2.$$

Now we apply this answer to $L(u)$:

$$L(1 + x\lambda n) = \lambda x = \lambda[w]_{1+n}.$$

You can see using the Carmichael function makes the base y disappear. Then we just expand what is left, taking the modulus n^2. Then applying the definitions in the lemma, we see the desired outcome.

In [14], Paillier also proves that:

$$[w]_{g_2} = [w]_{g_1}[g_2]_{g_1}^{-1} \bmod n.$$

This property allows us to take the Class of w base g_1 and the class of g_2 base g_1 and returns the class of w base g_2. Since we know how to solve the class problem we can form the following equations with $g_1 = (n + 1)$:

Equation 2.

$$\tfrac{L(w^\lambda \bmod n^2)}{L(g^\lambda \bmod n^2)} = \tfrac{\lambda[w]_{g_1}}{\lambda[g]_{g_1}} = \tfrac{[w]_{g_1}}{[g]_{g_1}} = [w]_g \bmod n.$$

The above equation does work correctly but while trying out examples of the decryption I realized these last steps are unnecessary. The denominator $L(g^\lambda \bmod n^2)$ just gives us the value $\lambda \bmod n$. Since we already know λ, calculating this is completely unnecessary. In practicality, the value $L(g^\lambda \bmod n^2)$ can be calculated before the decryption as part of the key generation, so it doesn't make the scheme much faster. Nonetheless, it is unnecessary and a simpler equation is:

Equation 3.

$$\tfrac{L(w^\lambda \bmod n^2)}{\lambda} = \tfrac{\lambda[w]_g}{\lambda} = [w]_g \bmod n.$$

With these equations complete, we have everything needed to describe the encryption and decryption. I will use my simpler equation for the decryption.

Steps for Key Generation

1. Select two large primes p and q about the same size.
2. Calculate the modulus n, the product of two primes.
3. Find a $g \in \beta$. Since $g = (1 + n)$ works and is easily calculated, this is the best choice.
4. We have the public key (n, g) and the private key (p, q).

Steps for Encryption

1. Plaintext is m where $m < n$.

2. Find a random $r \in Z_n^*$. (Some sources say $r \in Z_n$ but this will not work in the Carmichael function.)

3. Let ciphertext $c = g^m r^n \bmod n^2$.

One can see, the encryption is a simple equation but calculating it will not be quick. The exponentiation is very expensive, since the exponents are both around the same size as the modulus. Looking at the form of the equation, one will see that the decryption will be done by first removing the random part r^n, then retrieving the exponent $m \bmod n^2$.

Steps for Decryption

In my research, I found there is a newer method of decryption in later writings on Paillier's scheme [1]. The newer decryption seems to be simpler and quicker. I will describe both methods since they work in similar ways. I will refer to the original algorithm as Original Paillier or just the Paillier scheme for short and I will refer to the newer algorithm as the Second Paillier scheme.

Original Paillier Method

1. The ciphertext $c < n^2$.

2. Retrieve plaintext $m = \frac{L(c^\lambda \bmod n^2)}{\lambda} \bmod n$
(In Paillier's paper, he uses $L(g^\lambda \bmod n^2)$ as the denominator but I showed above this is unnecessary).

Second Paillier Method

1. The ciphertext $c < n^2$.

2. Calculate α, where $\alpha n = 1 \bmod \phi(n)$.

3. Let $r = c^{\alpha \bmod \phi(n)} \bmod n$.

4. Retrieve plaintext $m = L(c \cdot r^{-n} \bmod n^2)$.

For Paillier's original method of decryption we have already shown how it works. Looking at Equation 1, we can see how to extract the exponent from w using properties of the Carmichael function and the L function. All that needs to be done after Equation 1 is remove the coefficient λ, which is most easily done with Equation 3. You can also try Paillier's method in Equation 2, which requires a couple more steps.

The second method works in similar ways, but it is more like RSA decryption because it uses the Euler totient function. This method is explained in [1] but I haven't found who first discovered this decryption method. To decrypt with this method first compute:

$$\alpha n = 1 \bmod \phi(n).$$

This can be solved using the extended Euclidean algorithm [16] to find n^{-1}. With this information we can then compute:

$$r = c^{\alpha \bmod \phi(n)} \bmod n.$$

To solve the above computation, evaluate $c = (1+n)^m r^n$ in mod n. This causes $(1+n)^m = 1 \bmod n$ since n to any exponent will be 0. This leaves us with $r^{1+k\phi}$ which takes the same form as the RSA equation. In the same way as RSA, we know $r^{\phi} = 1 \bmod n$ by Euler's theorem and we are left with r. With the value of r at our disposal we can solve for the plaintext m:

$$L(c \cdot r^{-n} \bmod n^2) =$$
$$L(((1+n)^m \cdot r^n) \cdot r^{-n} \bmod n^2) =$$
$$L(1 + nm \bmod n^2) = m.$$

We can use our knowledge of r to find its inverse and then negate r from the equation. This leaves us with $(1+n)^m$ which simplifies to $(1+nm) \bmod n^2$ and all we need to do is apply the L function.

One can see, both methods eliminate the value r by forcing it to be 1. The first method uses Carmichael's function and the second Euler's totient function. After that, simply evaluate what is left mod n^2, which leaves only a few simple operations to yield the answer. We will implement both methods in the software portion of this study and evaluate which one is faster.

Key Generation Example

1. $p = 11, q = 17$.
2. $n = 187$.
3. $g = (n+1) = 188$.
4. We get the public key $(n, g) = (187, 188, 97)$ and the private key $(p, q) = (11, 17)$.

Encryption Example

1. Plaintext $m = 100$.
2. $r = 97 \in Z_n^*$.
3. Let ciphertext $c = 188^{100} \cdot 97^{187} \bmod 34969 = 26118$.

Decryption Example

Original Paillier Method

1. The ciphertext $26118 < 34969$.
2. Remember $L(u) = \frac{u-1}{n}$.

- Calculate $\lambda = \text{lcm}(17-1, 11-1) = 80$.

- $m = \frac{L(26118^{80} \bmod 34969)}{80} \bmod 187$
  (You could also calculate $L(188^{80} \bmod 34969) = 80$ for the denominator in Paillier's method).

- $m = \frac{146}{80} \bmod 187$.

- $80^{-1} = 180 \bmod 187$.

- $m = 146 \cdot 180 \bmod 187 = 100$.

Second Paillier Method

1. The ciphertext $26118 < 34969$.

2. Calculate $\phi(n) = 160$.
   $\alpha 187 = 1 \bmod 160$.
   $\alpha = 187^{-1} \bmod 160 = 83$.

3. Let $r = 26118^{83} \bmod 187 = 97$.
   We can see that this successfully retrieved our original r.

4. - Calculate $r^n = 97^{187} \bmod 34969 = 31541$.

   - Therefore $r^{-n} = 31541^{-1} \bmod 34969 = 30858$.

   - We can retrieve plaintext $m = L(26118 \cdot 30858 \bmod 34969)$.

   - $m = L(18701) = 100$.

7.3 Summary of Scheme

The Paillier system has some properties that make it appealing for different uses. Its probabilistic nature is a good feature, but the main interest in the scheme is the homomorphic property. This has spawned a good amount of interest and there seems to be a good chance we could see it used in practical applications. So far I have not seen any evidence that it is used in any commercial applications, but keep in mind it was only published in 1999. It takes many years of scrutiny before a new cryptosystem becomes practical and trusted.

I am starting to see it used in newer protocols for electronic cash, electronic voting [1] and Private Information Retrieval [9]. These protocols need the additively homomorphic properties that are not common in cryptosystems. To my understanding these things are still being researched but there is much interest and they could be gaining momentum. I still think it will be at least a couple of years, if not longer, before we see the real world applications on the market.

The efficiency is good but it will always be inherently slower than RSA. The difference is greatest in encryption where exponentiation of g^m takes much longer than any calculation in RSA encryption. This is just a longer calculation that needs to occur. The ciphertext c will always be less than n but is encrypted to the size of n^2, so the expansion is 2. This is definitely not great but could be acceptable if we find special uses for the scheme.

I have found no claims that the system is susceptible to attacks and haven't discovered any flaws myself. One can see in my description of the scheme, the mathematics contain many of the same principles as RSA, and one can decrypt the ciphertext using similar algorithms. This bodes well for the scheme as far as security is concerned, since RSA has been accepted as secure for many years. The only way to break the scheme I have seen is by factoring the modulus n, which should be set to the appropriate size to avoid this problem.

In conclusion, I do believe that Paillier's scheme has great potential, which is evidenced in all the attention it has received recently. Since Paillier will inherently be slower than schemes like RSA, its real test will be finding applications that need the homomorphic property. It is probable that if an implementation of Paillier can be close to RSA in speed, it will find its way into applications. This may take a little time, as the current implementation seems a bit slow. I will show the results from my own testing later on and compare it with other schemes.

8 Damgård-Jurik Scheme

8.1 History

The Damgård-Jurik cryptosystem [5] was first introduced just after Paillier's in 2000. Damgård-Jurik is really just a modification to Paillier's system that allows a user to increase the size of the encrypted value. Due to its nature, it is also called the Generalized Paillier system. Damgård-Jurik's scheme has also generated some interest, since it shares similar properties with Paillier's scheme. Both are probabilistic and more importantly both are homomorphic. This allows them to be mentioned as possibilities for the same applications: voting protocols, threshold cryptosystems, watermarking, secret sharing schemes, private information retrieval, and server-aided polynomial evaluation.

The question to be answered with Damgård-Jurik is whether or not the implementation will improve over Paillier's implementation, which isn't particularly quick. The big advantage of Damgård-Jurik is that one can potentially encrypt an arbitrarily large value in one try. Other schemes must divide the message into smaller pieces and before encrypting. We will see if this advantage helps the system succeed.

8.2 Mathematical Background

As we already stated, Damgård-Jurik is a variation on the Paillier scheme. Therefore, the explanation will assume that Paillier is already understood and I will refer to the previous section on Paillier instead of explaining the same concepts over.

The main difference between the two schemes is the amount of plaintext space that can be encrypted at once. In the Paillier cryptosystem we are limited to a plaintext $p < n$ and a ciphertext $c < n^2$. This gives the plaintext space of Z_n and ciphertext space of Z_{n^2}. In Damgård-Jurik it is generalized to not be limited by the modulus n. Instead it allows the plaintext space to be Z_{n^s} and a ciphertext space of $Z_{n^{s+1}}$, where theoretically s can be any integer. In essence, you are allowed to encrypt any size plaintext without dividing it into chunks by choosing the proper s.

Thinking back to the Paillier scheme, we defined the framework of the encryption as $\mathcal{E}_g(x, y) \rightarrow g^x y^n \bmod n^2$. We build off from this and generalize it to get the following framework for Damgård-Jurik [5].

Definition 10 *For $g \in Z^*_{n^{s+1}}$ and $s < p, q$ we define $\mathcal{E}_g(m, y) \rightarrow g^m y^{n^s} \bmod n^{s+1}$.*

With this equation, we get a plaintext space of Z_{n^s} and a random space of Z_n. Together we get $Z_{n^s} \times Z_n = Z_{n^{s+1}}$ which is the ciphertext space. You can see this allows us to make the ciphertext almost unlimited in size by letting s be any value less than p, q. You can also see setting $s = 1$ will give us the exact same space as Paillier. In fact, allowing $s = 1$ is the simplest case and reveals the Paillier scheme.

In [5], the authors take great care in showing that $g \in (1 + n)^j x \bmod n^{s+1}$ where j is relatively prime to n and $x \in Z^*_n$. Later on they explain that all security is equivalent regardless of the g. They recommend using $g = 1 + n$, which we saw as the standard in Paillier's scheme. It is simple to calculate this value and it tends to be easier to use. In the rest of this section we will assume $g = 1 + n$.

Damgård and Jurik show that this equation is a direct product of the multiplicative groups $G \times H$ [5]. G is the group that is cyclic of order n^s, which implies the order of G is n^s. H is defined as the group isomorphic to Z^*_n, which implies an order of $\phi(n)$ or $(p - 1)(q - 1)$. Together these give us $n^s(p - 1)(q - 1)$, which is also the size of $Z^*_{n^{s+1}}$ or $\phi(n^{s+1})$.

We can now fill out the base equation with what is known to create the equation to compute the ciphertext.

$$\begin{aligned} c &= g^m y^{n^s} \text{ where } g = (1 + n)^j x,\ y = r \in Z^*_n \\ &= (1 + n)^{mj}(x^m r^{n^s}) \end{aligned}$$

From this we can see $(1+n)^{mj} \in G$ and $(x^m r^{n^s}) \in H$. Our goal is to extract m, which is the message. As in Paillier, we will first try to eliminate the part $\in H$. We can do this by finding $d = 0 \bmod \lambda$ and $d \bmod n \in Z^*_n$ with the Chinese Remainder theorem [16]. We use d as an exponent on the ciphertext in the following way.

$$\begin{aligned} c^d &= (1+n)^{mjd}(x^m r^{n^s})^d \\ &= (1+n)^{mjd}(x^{dm} r^{n^s d}) \\ &= (1+n)^{mjd} \bmod n^{s+1}. \end{aligned}$$

You can see that x^{dm} becomes 1 because $d = k\lambda$ where k is any integer. Since we know by Carmichael's theorem that $x^\lambda \bmod n = 1$, it must be congruent $\bmod n^{s+1}$ because n^{s+1} is a multiple of n. The value $r^{n^s d} = r^{n^s(p-1)(q-1)} = 1$ since $n^s d$ is a multiple of $\phi(n)$.

What remains is $(1+n)^{mjd} \bmod n^{s+1}$. In Paillier's scheme it was easy to extract the exponent, which contains the value we are looking for, m. For Paillier, we get $(1+n)^m = 1 + nm \bmod n^2$, because all terms greater than n disappear modulus n^2. By using the function $L(u)$ described by Paillier, we are able to extract the message. The process we use in this case is similar but it takes more steps to extract m. The algorithm seems similar to the Pohlig-Hellman algorithm (see [10]), also used to calculate discrete logarithms. Ultimately the algorithm is different, since Pohlig-Hellman will not work in this case.

The first observation needed is that we can generically expand what is left of our equation using the Binomial Theorem. Using the function $L(u)$ we defined in the section on the Paillier scheme, we are able to expand to get the following.

Equation 4.

$$L((1+n)^i \bmod n^{s+1}) = (i + \binom{i}{2}n + ... + \binom{i}{s}n^{s-1}) \bmod n^s.$$

With this information, we can't simply extract $i \bmod n^s$ but we can extract $i \bmod n^1$ as we did for Paillier's scheme. This works for $s = 1$, but we need to define a method for all cases. This can be done with a neat form of induction that extracts i for each modulus one step at a time. We define each step:

$$\begin{gathered} i_1 = i \bmod n \\ i_2 = i \bmod n^2 \\ i_3 = i \bmod n^3 \\ ... \\ i_j = i \bmod n^j. \end{gathered}$$

We already know how to extract i_1 and from this step we will be able to extract i_2. From i_2 we can extract i_3 and so on. This will eventually give us i_j, no matter how big j is. To understand the process, we first need to make several observations.

Equation 5.

$$i_j = i_{j-1} + k * n^{j-1}, \text{ where } k \in Z_n.$$

You can start to see that knowledge of i_{j-1} helps us to find i_j.

After this we need to show another expansion similar to Equation 4, but this uses the more specific value i_j.

Equation 6.

$$L((1+n)^i \bmod n^{j+1}) = (i_j + \binom{i_j}{2}n + ... + \binom{i_j}{j}n^{j-1}) \bmod n^j.$$

One can then notice the following:

Equation 7.

$$\text{For } j > t > 0 \text{ one will always have } \binom{i_j}{t+1}n^t = \binom{i_{j-1}}{t+1}n^t \bmod n^j.$$

Looking closely reveals how this works. We know from Equation 5 we can replace i_j with $i_{j-1} + k * n^{j-1}$. This gives us $\binom{i_j}{t+1}n^t = \binom{i_{j-1}+k*n^{j-1}}{t+1}n^t \bmod n^j$. Keep in mind that when we calculate the combinations we can multiply each $(i_{j-1} + k * n^{j-1})$ by n^t. In the smallest case $n^t = n$, which yields the following $(i_{j-1}+k*n^{j-1})n = (i_{j-1}n+k*n^j)$. When this is evaluated mod n^j the contributions from $k * n^j$ will disappear. This observation gives us the above result we were looking for.

With the observations in Equations 5 and 7, i_{j-1} can be substituted for i_j in Equation 6.

Equation 8.

$$L((1+n)^i \bmod n^{j+1}) = (i_{j-1} + k * n^{j-1} + \binom{i_{j-1}}{2}n + ... + \binom{i_{j-1}}{j}n^{j-1}) \bmod n^j.$$

By eliminating all i_j from our equation we are almost at a place we can solve for i_j. Putting Equations 5-8 together creates an equation to solve for i_j with the knowledge of i_{j-1}.

$$\begin{aligned} i_j &= i_{j-1} + k * n^{j-1} \\ &= i_{j-1} + L((1+n)^i \bmod n^{j+1}) - (i_{j-1} + \binom{i_{j-1}}{2}n + ... + \binom{i_{j-1}}{j}n^{j-1}) \bmod n^j. \end{aligned}$$

This leads to our final equation for Damgård-Jurik.

Equation 9.

$$i_j = L((1+n)^i \bmod n^{j+1}) - (\binom{i_{j-1}}{2}n + ... + \binom{i_{j-1}}{j}n^{j-1}) \bmod n^j.$$

You can see in the first step, I substitute what I know $k * n^{j-1}$ equals from Equation 8. In the next step the terms i_{j-1} cancel each other out and we have our final result.

This equation only gives us knowledge of i_j from i_{j-1} and this may not be the answer we need. But one can keep repeating the algorithm over and over using induction each time. Eventually, one will reach the i_j desired. The value i_1 can always be calculated to start the induction.

An algorithm for calculating i_j for any $j \in Z_n$ is provided later in the "Steps for Decryption" subsection. You will see that while this algorithm does work, it seems to be quite expensive. There are many steps and the larger s becomes the more steps are necessary. We will see if the extra complexity is offset by the ability to encrypt larger values.

Keep in mind the value $i = jmd$ is not the final answer. We are actually looking for the message m. At this point it is easy to find m since we have knowledge of j and d. Therefore, just calculate $(jd)^{-1} \bmod n^s$ using the extending Euclidean algorithm (see [16]). Plug the answer into the equation $jmd * (jd)^{-1} = m \bmod n^s$ and the message m can be computed.

Steps for Key Generation

1. Select two large primes p and q about the same size.
2. Calculate the modulus n, the product of two primes.
3. Find $g \in Z^*_{n^{s+1}}$ such that $g = (1+n)^j x \bmod n^{s+1}$ where gcd(j,n) = 1 and $x \in H$. I recommend using $g = (1+n)$ since it is simple and works as well as any other value.
4. Choose d such that $d \bmod n \in Z^*_n$ and $d = 0 \bmod \lambda$. This can be done with the Chinese Remainder Theorem.
5. We have the public key (n, g) and the secret key (d, j).

Steps for Encryption

1. Choose $s \in Z_n$ such that one can encrypt as much as desired.
2. The plaintext is m where $m < n^s$.
3. Find a random $r \in Z^*_n$.
4. Let the ciphertext $c = g^m r^{n^s} \bmod n^{s+1}$.

Comparing with Paillier will reveal why this scheme is also known as the Generalized Paillier scheme. The Paillier algorithm is still present but the exponent n is replaced with n^s and the modulus n^2 is replaced with n^{s+1}.

Steps for Decryption

1. The ciphertext $c < n^{s+1}$.
2. Calculate $c^d = (1+n)^{jmd \bmod n^s}$.
3. Calculate jmd with the following algorithm where $a = (1+n)^{jmd}$.

```
$i := 0;$
for $j := 1$ to $s$
{
    $t_1 := L(a \bmod n^{j+1});$
    $t_2 := i;$
    for $k := 2$ to $j$
    {
        $i := i - 1;$
        $t_2 := t_2 * i \bmod n^j;$
        $t_1 := t_1 - \frac{t_2 * n^{k-1}}{k!} \bmod n^j;$
    }
    $i := t_1;$
}
```

This algorithm returns $i = jmd$.

4. Extract message $m = jmd * (jd)^{-1}$.

You can compare step 3 of this algorithm with Equation 9, since this is the same thing written out in pseudocode. You can trace through the code and see how to first calculate i_1, then continue looping until one finds i_j. Any reasonable value of s will work but one can see how much extra processing is added every time s is incremented by one.

For the following example I tried to use the same values for the key as the Paillier example above. I was then going to use the same keys to encrypt a much bigger message, even with $s = 2$. Unfortunately, I found out quickly that using n^2 creates a rather large exponent that can not be calculated easily. This forced me to use smaller values in the keys. Even with a much smaller modulus n, I was able to encrypt a much larger message.

You can see in this example, the variables p, q are even smaller than in the previous examples. The calculations get large quickly and the decryption algorithm takes a good amount of work to compute. In order to keep the example at a reasonable length, it was necessary to use numbers this artificially small. It should still serve as a good example to show how the scheme works. This may also give us a hint as to the performance of the Damgård-Jurik algorithm when implemented.

Key Generation Example

1. $p = 7, q = 5$.
2. $n = 35$.
3. $g = (n + 1) = 36$. (where $j, x = 1$).
4. $d = 9 \bmod 35$ and $d = 0 \bmod \lambda$.
   Using Chinese Remaindering we get $d = 324 \bmod n\lambda$.
5. We have the public key $(35, 36)$. The private key is $(324, 1)$.

Encryption Example

1. Choose $s = 2$ to fit our message.
2. Let plaintext $m = 1000$ where $1000 < 1225$.
3. Let the ciphertext $c = 35^{1000}23^{1225} = 17943 \bmod 42875$.

Decryption Example

1. The ciphertext $17943 < 42875$.

2. $r = 23 \in Z_n^*$.

3. Calculate $c^d = 17943^{324} = 33251 \bmod 42875$.

4. Calculate jmd with the following algorithm. We know $j = 1$ so we can disregard it. We also know $s = 2$ and $n = 35$.
   Let $a = (1+n)^{md} = 33251 \bmod n^3$.

   ```
   $i := 0$;
   for $j := 1$ to $2$
   {
       $t_1 := L(33251 \bmod 1225) = 5$;
       $t_2 := 0$;
       for $k := 2$ to $1$
       {
           - -
       }
       $i := 5$;
   for $j := 2$ to $2$
   {
       $t_1 := L(33251 \bmod 42875) = 950$;
       $t_2 := 5$;
       for $k := 2$ to $2$
       {
           $i := 4$;
           $t_2 := 5 * 4 \bmod 1225 = 20$;
           $t_1 := 950 - \frac{20*35^1}{2!} \bmod 1225 = 600$;
       }
       $i := 600$;
   }
   ```

   This algorithm shows that $i = jmd = 600$.

5. To extract the message m we need $(jd)^{-1} \bmod n^s$. Since $j = 1$ just calculate $d^{-1} \bmod 1225 = 949$, using the Extended Euclidean algorithm.
   Now extract $m = jmd * (jd)^{-1} = 600 * 949 \bmod 1225 = 1000$.

8.3 Summary of Scheme

The appeal of the Damgård-Jurik scheme is much the same as Paillier's scheme. It seems most people interested in Paillier are also interested in Damgård-Jurik, since it contains the same homomorphic property. It is being considered with Paillier for different types of applications, where the homomorphic properties can be exploited. The scheme is also relatively new, first being published in 2000. The research appears ongoing and I have also seen Damgård-Jurik's scheme used in similar protocols for electronic voting [5] and Private Information Retrieval [9]. Often they are testing the protocols against one another to see which performs better. Because of all this research, there is a possibility we could see real world applications using it in the near future.

The efficiency of Damgård-Jurik is something we will investigate later on. We can tell the expansion of the ciphertext will be smaller than Paillier with larger values of s. The expansion can be calculated as $\frac{s+1}{s}$, so when $s = 1$ we have an expansion of 2. The expansion is reduced to 1.5 when $s = 2$ and it will continue to decrease the larger s becomes. You can get the expansion factor very close to 1, but this can take very large values of s. It is unlikely that large values of s will be practical from the computational aspect.

As with Paillier, I have found no claims of people cracking the scheme. Since the schemes are based on the same principles, I believe their security is similar. It should be noted that by enlarging s we can increase the size of the plaintext without increasing security by enlarging n. Users should keep in mind the value of n for security purposes and avoid using a large value of s to keep the value of n too small. The value of n should always be large enough that it hasn't yet been factored.

Finally, I do believe that the Damgård-Jurik cryptosystem has some great potential for the same reason as Paillier's. Since they both have the same homomorphic property, there is a good chance only one will succeed. It may come down to which scheme can be implemented faster. You have seen Damgård-Jurik is much more complicated, but if it improves performance, it will be worth it. There is also a chance that this could degrade performance and doom the scheme. I will give my conclusion on which performs better in the "Testing Results and Recommendations" section.

9 Design of Cryptography Software

This section gives an overview of the program that I created to test and compare the Paillier and Damgård-Jurik cryptosystems with RSA. I will explain how the schemes were implemented and any assumptions I used. The explanations will review any techniques I used to speed up each implementation.

The first subsection will give a brief description of the Java Cryptography Extension and how I used it. The next subsection will describe the front end application that is used to manage, send, and receive encrypted messages. The following four subsections will go over the different implementations I used for each scheme. These subsections cover the schemes RSA, Paillier, Damgård-Jurik, and Second Paillier. The Second Paillier subsection describes a version of Paillier that uses a different decryption technique but can still be called the Paillier scheme.

9.1 Using Java Cryptography Extension

To make my Message Communicating program and my Cryptography schemes independent and flexible, I used the Java Cryptography Extension (JCE) as the framework for my software. The JCE contains a framework for building cryptography schemes and tools (see [8], [17]). Some of the schemes and tools are already implemented and ready for use, but one can also create their own. This allowed me to create my own encryption software, while also using an established implementation from a Provider to do my testing. The Provider is some type of group that provides a jar file with cryptography tools built in the JCE. The Classes in the Provider's jar file can be used by putting a reference to it in the java.security file or by referencing it in the code. Doing this allows one to create instances of these cryptography tools by using the specific name given to each.

For my work I used a Provider for one of the schemes and created my own Provider for the other three schemes. Because all the schemes are implemented using the JCE, it was easy to grab different instances of each scheme. The code needed little specialization to deal with the differences between schemes.

Generally speaking, a person cannot just create his own Provider. Sun specifically tells us that we must be a specialized software company to obtain a special certificate. This certificate allows one to create a jar file as a Provider that contains tools built in the JCE. Without the certificate one can compile you code, but anything derived from the JCE classes will error when run. This policy doesn't make sense to me but it seems they are loosening it. After initially denying my request, they later gave me the certificate I needed. I would imagine they will do the same for others interested in implementing cryptography in the JCE but I don't know this for sure. If you can't get this certificate you can access tools from the JCE created by other Providers, but you will not be able to create your own.

9.2 Message Communication Program

The Message Communication program is designed to be a peer-to-peer application that allows us to send encrypted messages across a network. It was built using the JCE API and RMI technology in Java. The program was designed so only two sessions could be running and communicate at once. It was created so that we could plug in about any working public key encryption scheme with very little effort. The users should see little difference between encryption types when the file gets passed if the schemes are working correctly. The big exception is the amount of time to encrypt and decrypt the file, as will be seen in the results of my experiments.

The Message Communication program is designed so the message receiver session initiates the communication with the message sender session. If this were a productional application it would need some type of authentication, but we skip this step since it is only designed for academic use. The message receiver selects the type of cryptography and any parameters needed to set up the scheme. Once it has this information, it creates the public and private keys and sends the public key to the sender along with the type of cryptography being used. The private key will only reside on the receiver, ensuring no one has a chance to obtain the decryption information. Once the type of cryptography is decided and the keys are initialized, there is no way to change these values in the receiving session. If the sending session receives a new cryptography scheme and key, it will encrypt according to these new values.

The receiver decides the directory and filename of the file that will be sent, along with the directory and filename where the file will reside on itself (the receiver). The information about the file to be sent is relayed to the sending session. Once it receives the information, it imports the file and encrypts it in chunks. The chunks are as big as possible while still allowing the entire chunk to be encrypted in one encryption. The size of the chunks is generally dependent on the size of the modulus chosen, since the plaintext can't be bigger than the modulus. Chunks of the plaintext are converted to byte arrays and sent to the cipher to be encrypted, with the cipher returning the ciphertext also as a byte array. The chunks of ciphertext are placed in an array of byte arrays, where there is one byte array for each chunk that was encrypted. The array of byte arrays is then sent to the receiver session to be decrypted.

The receiver gets the array of byte arrays with the length of the array. The decryption occurs in chunks, as each byte array is passed to the decryption cipher. Each block of decrypted ciphertext is concatenated to the previous blocks to recreate the final plaintext. Once the entire ciphertext is decrypted the plaintext is output to the file and directory specified earlier. You can view a simple diagram of the process flow in Figure 1 below.

To keep track of the amount of time used, I set both operations up to get the time stamp in milliseconds before and after the encryption and decryption. The time includes the initialization of the cipher along with operations performed on each chunk. It does not include the setup of the Message Sender, initialization of the keys, or the file transfer. The total time needed to perform these operations is displayed for both encryption and decryption.

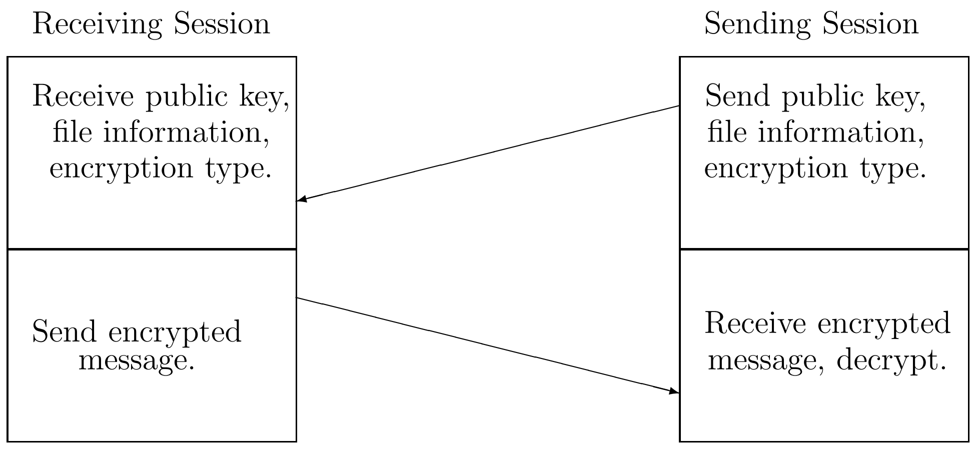

Receiving Session
Sending Session
Receive public key, file information, encryption type.
Send public key, file information, encryption type.
Send encrypted message.
Receive encrypted message, decrypt.

Figure 1. Message Passing Program.

9.3 RSA Implementation

The RSA implementation was taken from a Provider known as Cryptix. This organization has a large number of cryptography schemes implemented through the JCE, available on its website `www.Cryptix.org`. You can also get the source code for all their code, which is very helpful. It does not appear there has been much work with the Provider recently, but I found what is currently there works well.

The implementation is done using PKCS1 RSA Cryptography Standard v1.5 as noted on `www.rsasecurity.com`. This version of RSA uses padding at the beginning of encryption chunk. The padding decreases the amount of text that can be encrypted by 12 bytes, which is a small disadvantage for this scheme. The padding contains a pseudorandom string that makes the encrypted string different, even when encrypting the same value. This is one of the characteristics of a probabilistic scheme that is incorporated into a deterministic scheme.

The scheme is implemented using the Chinese Remainder Theorem (CRT) to increase the speed of decryption. Using the CRT in RSA has mostly become a standard , since it is relatively easy to do and can increase decryption speed up to 4 times. We are also able to use CRT in other schemes and achieve similar results as will be seen later. You can view an explanation on how CRT is used in public key cryptography in Appendix B.

9.4 Paillier Implementation

The first Paillier implementation is generally taken straight from [14]. The encryption and decryption algorithms I used were taken from section 4 of his paper and were outlined earlier in section 6.2. I did use the suggestions in section 7 to increase efficiency [14]. One very helpful suggestion is to use Chinese Remainder Theorem for decryption. To do this, perform the operations mod p and mod q and then bring them together mod n when finished. Paillier describes this in detail in this section and one can read about the general advantages of the CRT approach in Appendix B.

The other ways I increase performance include choosing specific values or precomputing values in key generation or the initialization of the scheme. These steps only run once at the beginning, so the values are only calculated once instead of every time a chunk is encrypted or decrypted. All the values in both keys are precomputed, as well as the following ways I increased performance.

1. Precompute the value r^n only once in initialization for each message passed. This may not be possible for all protocols since a different r may be needed to increase security. If two chunks of the same message were exactly the same, then the ciphertext would be the same. This would be a rare situation when passing normal messages. It would be interesting to study if keeping the value of r allows for any easy way to crack the ciphertext.

2. Set $g = (1 + n)$. This is the simplest value and there seems to be no benefit of calculating something more complicated.

3. Precompute n^2, which is p^2 and q^2 when using CRT.

4. Precompute $L(g^\lambda \bmod n^2)^{-1} \bmod n$ which needs to be $L(g^\lambda \bmod p^2)^{-1} \bmod p$ and $L(g^\lambda \bmod q^2)^{-1} \bmod q$ when using CRT.

In my explanation on Paillier, I pointed out that calculating $L(g^\lambda \bmod n^2)^{-1} \bmod n$ is unnecessary since you are just finding the value λ. Using the CRT I could only get decryption to work using this value when calculating mod p and mod q. This is the only way I found it works, although I may have been doing something wrong. I assume there is a good explanation for this but I didn't research it further because it doesn't affect the timings much. The value is quickly precomputed only once either way, so the timings are virtually the same.

9.5 Damgård-Jurik Implementation

For Damgård-Jurik, I used the Ph.D. thesis of Jurik [7] as the main source for my algorithms. I used the encryption scheme described in section 2.2. I was also able to use some of the suggestions from section 2.3.2.to optimize my calculations, the biggest of which was using the CRT.

The explanation on how to use the CRT is quite brief and not complete. A lot of the calculations are similar to Paillier's use of the CRT, but there are a few differences that aren't intuitive. I actually became very frustrated trying to implement this due to the missing information, so I e-mailed Jurik for an more detail. I will give a full description of the algorithm we discussed, since I don't believe it is available anywhere else. These are the steps needed to use the CRT for Damgård-Jurik.

1. First we can reduce the ciphertext *ciph* size by evaluating it with each modulus: $ciph_p = ciph \bmod p^{s+1}$ and $ciph_q = ciph \bmod q^{s+1}$.

2. Then we remove the random part by calculating $d_p = ciph_p^{p-1} \bmod p^{s+1}$ and $d_q = ciph_q^{q-1} \bmod q^{s+1}$.

3. Now we need to find the discrete log by using the dLog(a) function described. The function is slightly different than before since we must perform it modulus p and q. The L function is different for each value: $L_p(a) = (\frac{a-1 \bmod p^{s+1}}{p}) * q^{-1} \bmod p^s = mes'_p$ and $L_q(a) = (\frac{a-1 \bmod q^{s+1}}{q}) * p^{-1} \bmod q^s = mes'_q$ (these are described in the paper). With the new L functions one can compute dLog(a) for each value d_p and d_q and replacing the modulus n with p or q respectively. However, you do not replace n when computing n^{k-1}, which was the stumbling block I had. These calculations give us mes'_p and mes'_q.

4. Now that we used the discrete log function to retrieve the exponent, we can remove $(p-1)$ and $(q-1)$ added to the exponent earlier. We do this with $mes_p = mes'_p * (p-1)^{-1} \bmod p^s$ and $mes_q = mes'_q * (q-1)^{-1} \bmod q^s$. This gives us the message in both moduli, p^s and p^s.

5. The final step is using Chinese Remaindering to take our two values and create the congruent value modulus n. We will call the function $\text{CRT}(mes_p, p^s, mes_q, q^s) = mes_n$. This is the message in modulus n, the answer we are looking for.

The other ways I increased performance are similar to Paillier, by choosing specific values or precomputing values in key generation or the initialization of the scheme. Both run once at the beginning, so these values are only calculated once, instead of every time a chunk is encrypted or decrypted. All the values in both keys are precomputed and the following methods to increase performance as well.

1. Precompute the value r^{n^s} only once in the initialization for each message passed. The argument for or against this is the same as for Paillier. It may not work well in all situations but seems to work well for normal message passing. It definitely speeds up encryption by only calculating once.

2. Choose value $g = (1 + n)$. Even though Damgård-Jurik describes a much more complicated $g = (1 + n)^j x \bmod n^{s+1}$, they still admit there is no reason to use this more complicated version.

3. Choose value $d = \lambda$. I didn't find any reason for using any other higher value and λ is the simplest.

4. Precompute n^s, which is p^s and q^s when using CRT. Also precalculate n^{s+1}, which is p^{s+1} and q^{s+1} using the CRT.

5. Precompute the value $(jd)^{-1} \bmod n^s$ $(jd = \lambda)$ used to remove jd after we calculate the exponent. When using CRT this is $(p - 1)^{-1} \bmod p^s$ and $(q - 1)^{-1} \bmod q^s$

6. There are several values in the L function that can be precomputed and passed in. The first is n^j where $1 \leq j \leq (s + 1)$, which is p^j and q^j where $1 \leq j \leq (s + 1)$ when using the CRT. You will still need n^j as well when using CRT. All these values will need to be created in arrays to access them easily. The other key value is $(k!)^{-1} \bmod n^j$ where $2 \leq k \leq s$ and $1 \leq j \leq s$. This will require a two-dimensional array to store. I found that one does not need to calculate $(k!)$ for p^j and q^j since it is later evaluated with those moduli and precomputing mod n^j worked fine.

7. When using the CRT, there are some special values that can be precomputed for the L function. These are $p^{-1} \bmod q^s$ and $q^{-1} \bmod p^s$. These are not needed when not using CRT.

8. In the discrete log function, one can precompute the value $L(a \bmod n^{s+1})$ before entering the loop. You can then evaluate this mod n^j each iteration, which is the same as evaluating $L(a \bmod n^{j+1})$ each time. Of course, when using CRT, one can change out modulus n with either p or q and do the same calculations. This also works with the special L function mentioned above for CRT calculations.

All these optimizations do add some complexity to the algorithms but I believe they are worth it. I recommend when implementing your own version of Damgård-Jurik to first get your code working without CRT, then change it to use CRT. You can also view my source code for any specific questions on what I did. I commented out all the code I used originally before implementing CRT algorithms, so one can still see what worked without the CRT. The final code uses the CRT, so one can also see how that was done.

9.6 Second Paillier Implementation

I found the Second Paillier method in several places, but there was no official source of where it originated. The best reference for it is [1]. Since the encryption is exactly the same as the original Paillier, I used the same logic for encrypting. Decryption is a little different and so the optimizations are different too.

I used CRT again but it is less effective here because of other optimizations one can see. Again, I did some precomputing of values and chose specific values to increase efficiency. All this was done in the key generation or initialization, once per message. The following is a list of these optimizations.

1. Precompute the value r^n only once in the initialization for each message passed. This may still be a problem the same way it was for the Original Paillier scheme, but it helps efficiency much more using this algorithm. This helps in encryption by only calculating r^n once but you will see later it can also help in decryption.
2. Choose value $g = (1 + n)$. This is the simplest value and there seems to be no benefit of calculating something more complicated.
3. Precalculate the value α. When using the CRT one needs $\alpha_p = n^{-1} \bmod (p-1)$ and $\alpha_q = n^{-1} \bmod (q-1)$.
4. When doing decryption, only calculate r^{-n} once per message. Since this value does not change, this can be exploited in my algorithm. Because this is the major part of decryption, the operation is much quicker.
5. Precompute n^2, which is p^2 and q^2 when using CRT.
6. When using the CRT there are some special values that can be precomputed for the L function, as there were for Damgård-Jurik. These are $p^{-1} \bmod q^2$ and $q^{-1} \bmod p^2$. These are not needed when not using CRT.

The method for using the CRT is similar to what we have already seen before.

1. The first step is to find $\alpha_p = n^{-1} \bmod (p-1)$ and $\alpha_q = n^{-1} \bmod (q-1)$.
2. Then use ciphertext c to find $d_p = c^{\alpha_p} \bmod p^2$ and $d_q = c^{\alpha_q} \bmod q^2$.
3. Next we use the modified L function to calculate $L_p(a) = ((a - 1 \bmod p^2)/p) * q^{-1} \bmod p = m_p$ and $L_q(a) = ((a - 1 \bmod q^2)/q) * p^{-1} \bmod q = m_q$.

4. Finally, we can find the congruence mod n by doing the function CRT(m_p, p, m_q, q) to give us our decrypted message.

Unlike the other two schemes I used the CRT for, I don't know if it helped the Second Paillier scheme. Even so, it does not make decryption slower and it was interesting to see it worked, yet without much improvement.

10 Test Results and Recommendations

In this section, I will outline all the tests I did on each cryptosystem I implemented, while giving my observations. I will also list all of the timings I recorded from running each scheme with different parameters. I will conclude with my final recommendations.

The tests I performed were basically the same on each scheme. I created 5 files called file1.txt - file5.txt, each one bigger than the last. The sizes are: 1024, 2048, 4096, 8192, and 16384 bytes. I performed encryption on each of these files with different size keys. The size of the keys were: 1024, 2048, and 4096 bits. I tried using bigger keys and bigger files, but the timings took too long and data was not showing anything significant. The timings should show how each scheme performs with different size keys on different amounts of data.

One thing I wanted to note before I move into the tests is that I didn't take into account the time to set up the keys. For the smaller key sizes this did not matter much, as the keys were generated in a matter of a few seconds. For keys 4096 bits and larger, the key generation took minutes and the larger key increased time exponentially. This would seem impractical to me in the real world, but it did not get noted in my other tests.

10.1 RSA Scheme Results and Observations

Time in milliseconds, E = Encryption, D = Decryption, C = Chunks

RSA	1024	2048	4096
File1.txt - 1k	E - 15	E - 15	E - 47
	D - 203	D - 657	D - 2578
	C - 9	C - 5	C - 3
File2.txt - 2k	E - 32	E - 32	E - 62
	D - 312	D - 1093	D - 4594
	C - 18	C - 9	C - 5
File3.txt - 4k	E - 46	E - 63	E - 110
	D - 625	D - 2031	D - 8094
	C - 36	C - 17	C - 9
File4.txt - 8k	E - 94	E - 125	E - 219
	D - 1250	D - 4063	D - 15859
	C - 72	C - 34	C - 17
File5.txt - 16k	E - 187	E - 281	E - 468
	D - 2500	D - 8969	D - 29922
	C - 143	C - 68	C - 33

Table 1. RSA Timings.

As I mentioned in my description of the implementation, the Provider for RSA is a professional organization, so I expected it to work well and perform fast. I was not disappointed, since the RSA times were overall the fastest of all the schemes. I expected this since the calculations for RSA are much simpler and they take less operations. It is unlikely any of the other cryptosystems studied here will be able to surpass RSA in a straight speed competition. So, if you are looking for a fast public key cryptosystem, I still think RSA as the best of the bunch.

Specifically the encryption was the fastest by far, beating all others by at least 100 times. For decryption it was also fast, but not always the fastest, as it got beat by the Second Paillier cryptosystem. The encryption was so much faster any other scheme that RSA was always faster with its overall time.

10.2 Paillier Scheme Results and Observations

Time in milliseconds, E = Encryption, D = Decryption, C = Chunks

Paillier	1024	2048	4096
File1.txt - 1k	E - 1750	E - 6687	E - 27031
	D - 563	D - 2235	D - 10421
	C - 9	C - 5	C - 3
File2.txt - 2k	E - 3750	E - 14562	E - 53922
	D - 1078	D - 4156	D - 17031
	C - 17	C - 9	C - 5
File3.txt - 4k	E - 7532	E - 26938	E - 114078
	D - 2016	D - 7703	D - 30453
	C - 33	C - 17	C - 9
File4.txt - 8k	E - 14797	E - 53406	E - 206110
	D - 3938	D - 14562	D - 56782
	C - 65	C - 33	C - 17
File5.txt - 16k	E - 29656	E - 111719	E - 422891
	D - 7860	D - 29094	D - 117765
	C - 130	C - 65	C - 33

Table 2. Paillier Timings.

Paillier seemed to perform admirably against the standard RSA. It was not as fast in either encryption or decryption, but it seems fast enough that it could be considered for uses that RSA doesn't work for. 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. The decryption was much closer but the Second Paillier scheme was even faster in decryption. It seems that either this algorithm or the Second Paillier scheme could be used for an application that needs homomorphic encryption.

Specifically, the encryption was anywhere from 100 to almost 1000 times slower depending on how big the file and key became. The decryption was much better performing at a range of 2-3 times slower when the key was smaller. The performance difference was greater when the key got bigger, but these size keys probably aren't practical at this time. The CRT was a big help reducing decryption by 3 to 4 times. Without it, the decryption times were generally higher than encryption.

10.3 Damgård-Jurik Scheme Results and Observations

Time in milliseconds, E = Encryption, D = Decryption, C = Chunks

DJ, s = 2	1024	2048	4096
File1.txt - 1k	E - 3953	E - 15719	E - 58609
	D - 718	D - 3079	D - 14875
	C - 5	C - 3	C - 2
File2.txt - 2k	E - 8141	E - 31015	E - 116328
	D - 1281	D - 5156	D - 23828
	C - 9	C - 5	C - 3
File3.txt - 4k	E - 15750	E - 60547	E - 243469
	D - 2250	D - 9188	D - 39562
	C - 17	C - 9	C - 5
File4.txt - 8k	E - 31969	E - 120297	E - 461391
	D - 4797	D - 17141	D - 66219
	C - 33	C - 17	C - 9
File5.txt - 16k	E - 63235	E - 241000	E - 901516
	D - 8782	D - 33141	D - 125000
	C - 65	C - 33	C - 17

Table 3. Damgård-Jurik Timings, s = 2.

Time in milliseconds, E = Encryption, D = Decryption, C = Chunks

DJ, s = 3	1024	2048	4096
File1.txt - 1k	E - 6703	E - 28390	E - 103516
	D - 703	D - 3422	D - 13547
	C - 3	C - 2	C - 1
File2.txt - 2k	E - 13390	E - 54360	E - 201109
	D - 1437	D - 5609	D - 26875
	C - 6	C - 3	C - 2
File3.txt - 4k	E - 26766	E - 111141	E - 424703
	D - 2641	D - 11312	D - 41906
	C - 11	C - 6	C - 3
File4.txt - 8k	E - 53641	E - 218953	E - 816641
	D - 5312	D - 19344	D - 80672
	C - 22	C - 11	C - 6
File5.txt - 16k	E - 106656	E - 411515	E - 1598079
	D - 9875	D - 38281	D - 147484
	C - 43	C - 22	C - 11

Table 4. Damgård-Jurik Timings, s = 3.

Time in milliseconds, E = Encryption, D = Decryption, C = Chunks

DJ, s = 4	1024	2048	4096
File1.txt - 1k	E - 10812	E - 40172	E - 171125
	D - 1188	D - 6000	D - 21860
	C - 3	C - 2	C - 1
File2.txt - 2k	E - 21781	E - 82282	E - 337578
	D - 1859	D - 8250	D - 44531
	C - 5	C - 3	C - 2
File3.txt - 4k	E - 45219	E - 169625	E - 638875
	D - 3578	D - 14282	D - 65468
	C - 9	C - 5	C - 3
File4.txt - 8k	E - 88094	E - 343781	E - 1362546
	D - 6515	D - 26781	D - 118469
	C - 17	C - 9	C - 5
File5.txt - 16k	E - 179968	E - 665875	E - 2672078
	D - 12203	D - 48438	D - 206672
	C - 33	C - 17	C - 9

Table 5. Damgård-Jurik Timings, s = 4.

For Damgård-Jurik I did 3 sets of tests to see results for bigger values of s. I did tests with $s = 2, 3, 4$ but did not do $s = 1$ since this is Paillier's scheme. I did not do any tests above $s = 4$ because it seems clear that the higher values of s do not help the system perform better. The idea that one could adjust s to encrypt any size value in one chunk is good in theory, but I found it to be slower in my testing. In some cases Damgård-Jurik algorithm was close to Paillier, but overall it was significantly slower. It seems that doubling, tripling, or more, the size of the exponent, makes the calculation too slow to make up for the extra values encrypted. I would recommend using Paillier over Damgård-Jurik, since it is slower and has no advantages over Paillier.

Specifically, Damgård-Jurik was about 2 times slower in encryption but only marginally slower in decryption when $s = 2$. Of course, each time s increased, the encryption time was twice as slow. Decryption again was only marginally slower as s increased, but these problems seem too hard to overcome and make the scheme outperform Paillier. I believe the scheme would need to outperform Paillier to find a practical use.

10.4 Second Paillier Scheme Results and Observations

Time in milliseconds, E = Encryption, D = Decryption, C = Chunks

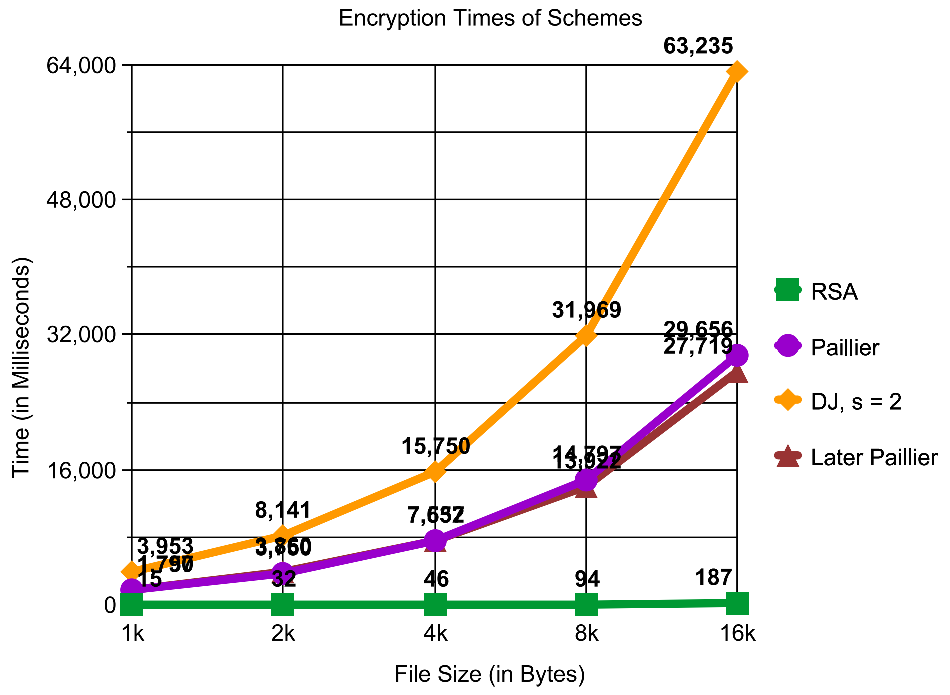
Second Paillier	1024	2048	4096
File1.txt - 1k	E - 1797	E - 6891	E - 25641
	D - 156	D - 1000	D - 7781
	C - 9	C - 5	C - 3
File2.txt - 2k	E - 3860	E - 13781	E - 51813
	D - 172	D - 1000	D - 7547
	C - 17	C - 9	C - 5
File3.txt - 4k	E - 7657	E - 27844	E - 103515
	D - 172	D - 1047	D - 7688
	C - 33	C - 17	C - 9
File4.txt - 8k	E - 13922	E - 54734	E - 206828
	D - 203	D - 1125	D - 7906
	C - 65	C - 33	C - 17
File5.txt - 16k	E - 27719	E - 108343	E - 413000
	D - 250	D - 1172	D - 8360
	C - 130	C - 65	C - 33

Table 2. Second Paillier Timings.

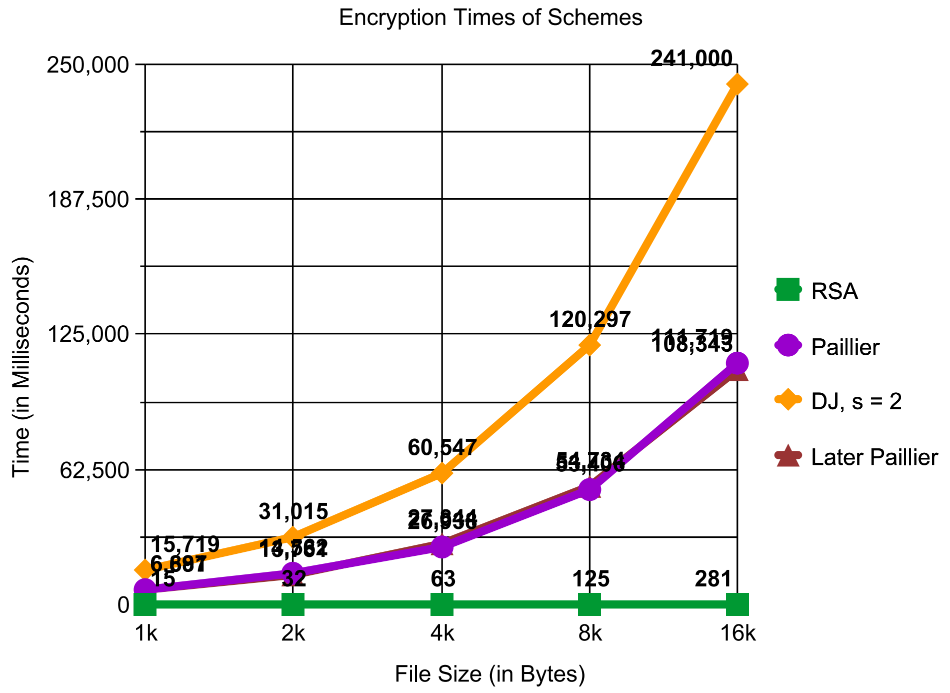
This second version of Paillier was the fastest performing of all the schemes I studied. It wasn't as fast as RSA overall, but did have some great times for decryption. It was always faster than RSA for decryption and got comparably faster the greater the size of the private key. The encryption performed was the same as the original Paillier scheme, since it is exactly the same algorithm, at anywhere from 100 to 1000 times slower than RSA. If some way is found to increase encryption time, I think it will compete close to the speed of RSA, especially for larger data sizes. Overall, RSA was still much faster because of encryption but I would recommend using Second Paillier for probabilistic homomorphic encryption scheme since the encryption is fastest.

Interestingly, the decryption for this scheme was only slightly slower as the file size got much bigger. For this reason the decryption was much faster for bigger keys and files than any other scheme. Since encryption is the same as the Original Paillier system, any improvement in the original encryption will also speed up this scheme.

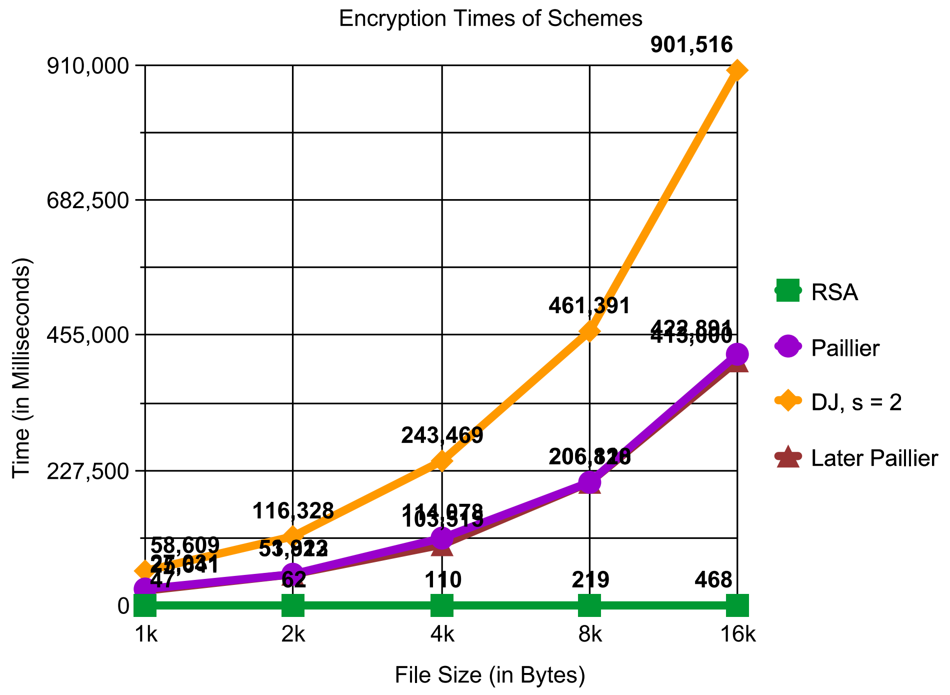
10.5 Comparative Graphs



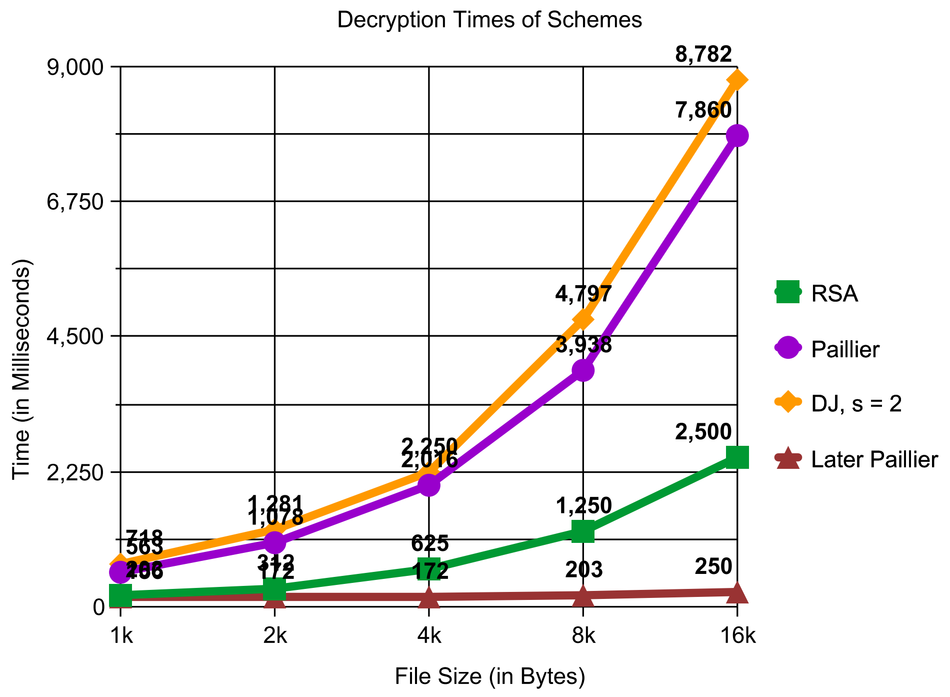
Graph 1. Probabilistic Encryption with 1024 Bit Keys



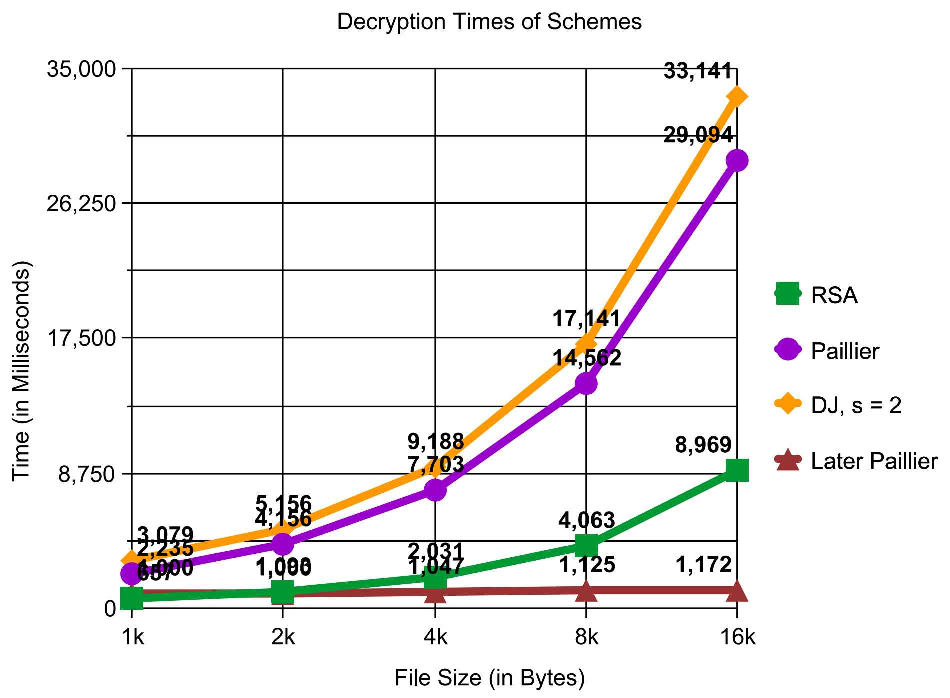
Graph 2. Probabilistic Encryption with 2048 Bit Keys



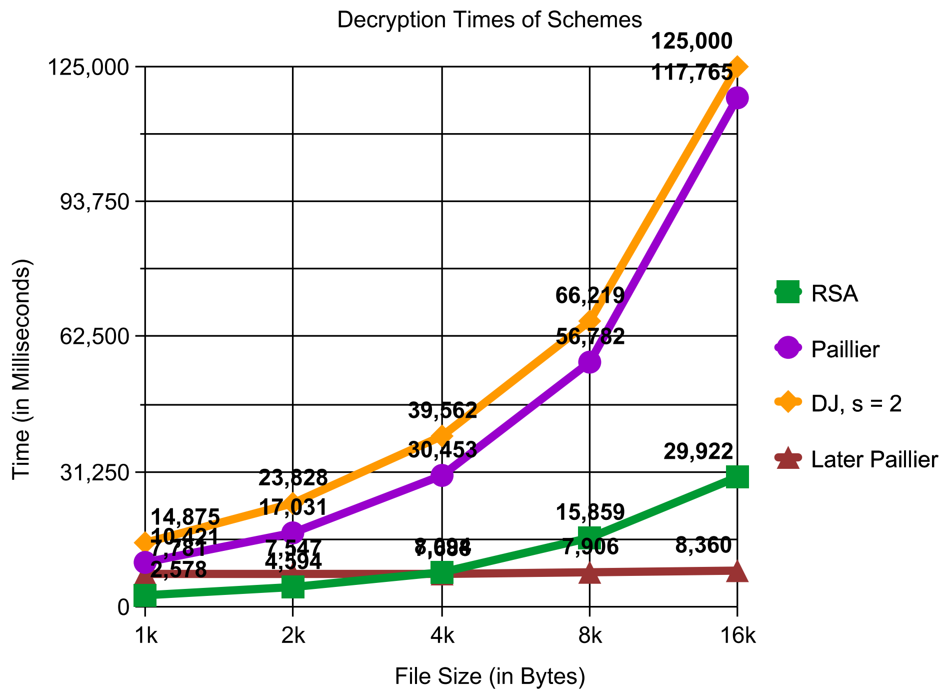
Graph 3. Probabilistic Encryption with 4096 Bit Keys



Graph 4. Probabilistic Encryption with 1024 Bit Keys



Graph 5. Probabilistic Encryption with 2048 Bit Keys



Graph 6. Probabilistic Encryption with 4096 Bit Keys

From the six graphs above one can compare the timings I found for both encryption and decryption for three different key sizes. It should be easy to compare the results and see the patterns that emerge. I did not graph Damgård-Jurik for s = 3 and 4 because the s = 2 was always the fastest and I didn't feel the graph would add much.

From all the graphs it is easy to see that RSA is always the quickest or near the quickest. Paillier is always a little quicker than Damgård-Jurik for both encryption and decryption. The Later Paillier scheme is virtually the same as the Original Paillier scheme for encryption because they use the same algorithm. For decryption the Later Paillier method is faster than any other scheme. The most important graphs are encryption and decryption where the key is 1024 bits, since this is the most common keysize at this time.

10.6 Final Recommendations

This concludes my results. My final recommendation for a probabilistic homomorphic cryptosystem is to use the Paillier system with the second method of decryption. It is the fastest for encryption and decryption and has all the same benefits. The security seems as good as any of the schemes, so there is no reason to go with another. Certainly, all these schemes are fairly new, so there could be another variation in the works that could trump all of these. It will be interesting to see where the study of these cryptosystems goes in the next few years.

I believe my idea to reuse the same random value r should work without making it less secure. As I mentioned before, it would only be a problem if several chunks of the message were exactly the same. This should not happen often and it could be overcome by sending the chunks in different messages. I am curious to see if anyone finds a security flaw with using this r in the implementation. If it is found unpractical to use this value, it would greatly change the results of my tests. It would increase the encryption time significantly and also greatly increase the decryption time of the Second Paillier scheme. Of course, the test would need to be rerun to get the exact numbers.

As for the private key size, the recommendation on RSA Laboratories is 1024 bits for today. This works faster than the larger keys and has not yet been factored effectively. Doubling the length of the private key generally increases the encryption and decryption time by 4 times, even though the encryption chunks are twice as large. Of course, the security is increased much more than this, but it doesn't seem necessary at this time. The exception to this is RSA encryption that only increases time by about 2 times.

The different file sizes generally increase encryption time at a very steady rate. When the file size doubles it takes twice as long to encrypt the file with twice as many chunks for the same size key, which shouldn't surprise anyone. The big exception to this was the Second Paillier scheme which only had a nominal increase in time when the file doubled in size. In fact the decryption time did not even double with a file 16 times bigger. The increase in time was very linear and was similar when the key size doubled or quadrupled. This is obviously because computing r, the most complicated calculation, is only needed once, no matter how long the data is.

I think my results show that RSA will never be surpassed by one of these probabilistic schemes as far as speed and security are concerned. RSA has been tested for over 30 years and has proven to be a secure and reliable system, whereas these schemes are still very young and I don't believe they are proven completely secure yet. RSA will most likely always be intrinsically faster than the schemes I studied. Fortunately probabilistic, homomorphic schemes shouldn't need to directly compete with RSA to be useful. Their uses will be specific to their unique properties.

So, after all my research , do I think these cryptosystems have a chance to be practically used in the future? I definitely think they have great potential in the future and may even be used right now. The additively homomorphic properties have a lot of practical uses that are being studied. It will just be a matter of time before these are used in a real world application. If the encryption time can be increased, then there will be even more interest and uses for this type of cryptography. I am very excited to see what applications that use these cryptosystems will emerge over the next few years.

A Homomorphic Encryption

Both the Paillier and Damgård-Jurik cryptosystems have several homomorphic properties that are very useful. These properties are: addition of multiple ciphertexts, the addition of a plaintext constant to a ciphertext, and the multiplication of plaintext constant by a ciphertext [1]. We will explain the mathematics behind each of these. You will need to understand the mathematics of how each cryptosystem works to understand this appendix.

A.1 Addition of Multiple Ciphertexts

For Paillier's scheme, I will show how to take two messages m_1 and m_2, first encrypt them with different random values r_1 and r_2, then multiply them together, and finally see the result is additively homomorphic.

$$\begin{aligned}
&E_g(m_1, r_1) = g^{m_1} * r_1^n \bmod n^2 \\
&E_g(m_2, r_2) = g^{m_2} * r_2^n \bmod n^2 \\
&E_g(m_1, r_1) * E_g(m_2, r_2) = (g^{m_1} * r_1^n)(g^{m_2} * r_2^n) \bmod n^2 \\
&\qquad\qquad = g^{m_1+m_2}(r_1 * r_2)^n \bmod n^2 \\
&\qquad\qquad = E_g(m_1 + m_2, r_1 * r_2)
\end{aligned}$$

You can see above that when the exponents of g are combined together with the same g they exponents are added together. This shows how multiplication of the ciphertexts is equivalent to adding the plaintexts. We know that both random values belong to Z_n^*, so this implies $r_1 * r_2 \in Z_n^*$ (see 2.125 in [10]). Although the random value changed, it will still disappear with the exponent λ because it is still in Z_n^*. This will leave us with $g^{m_1+m_2}$ which can be solved the same as before. This shows us how the homomorphic addition works.

The mathematics to show additive homomorphism for Damgård-Jurik are similar.

$$\begin{aligned}
&\mathcal{E}_g(m_1, r_1) = g^{m_1} * r_1^{n^s} \bmod n^{s+1} \\
&\mathcal{E}_g(m_2, r_2) = g^{m_2} * r_2^{n^s} \bmod n^{s+1} \\
&\mathcal{E}_g(m_1, r_1) * \mathcal{E}_g(m_2, r_2) = (g^{m_1 j} x^{m_1} * r_1^{n^s})(g^{m_2 j} x^{m_2} * r_2^{n^s}) \bmod n^{s+1} \\
&\qquad\qquad = (g^{(m_1+m_2)j}(x^{m_1 m_2})(r_1 * r_2)^{n^s} \bmod n^{s+1} \\
&\qquad\qquad = \mathcal{E}_g(m_1 + m_2, r_1 * r_2)
\end{aligned}$$

Virtually the same thing happens here as did in Paillier for the values r and m. The extra value $x \in Z_n^*$ will now become $x^{m_1 m_2}$ when doing homomorphic addition. This is still no problem in decryption because the exponent of x doesn't matter. The decryption still evaluates $x^{c\lambda} = 1$, where c is any positive integer coefficient. The exponent can be anything that is a multiple of λ. This shows that Damgård-Jurik's scheme is additively homomorphic, the same as Paillier's.

A.2 Addition of Plaintext Constants to Ciphertext

We show how to add a constant c to an encrypted value and that it is like adding the constant to the ciphertext before encryption. The process for Paillier follows.

$$\begin{aligned} E_g(m,r) * g^c &= g^m * r^n * g^c \bmod n^2 \\ &= g^{(m+c)} * r^n \bmod n^2 \\ &= E_g(m+c,r) \end{aligned}$$

In Paillier, the multiplication to a power c with the base g is the same as addition. Because of the characteristics of grouping exponents of the same base g, c gets added to m. The value $m+c$ is then retrieved in decryption.

Again, Damgård-Jurik is very similar and the explanation is basically the same.

$$\begin{aligned} \mathcal{E}_g(m,r) * g^c &= g^m * r^{n^s} * g^c \bmod n^{s+1} \\ &= g^{(m+c)} * r^{n^s} \bmod n^{s+1} \\ &= \mathcal{E}_g(m+c,r) \end{aligned}$$

A.3 Multiplication of Plaintext Constants by Ciphertext

Finally, we show that we can multiply a constant c by an encrypted value and it is the same as multiplying the constant by the ciphertext before encryption. The process for Paillier follows.

$$\begin{aligned} E_g(m,r)^c &= (g^m * r^n)^c \bmod n^2 \\ &= g^{(m*c)} * r^{(n*c)} \bmod n^2 \\ &= E_g(m*c, r^c) \end{aligned}$$

In the Paillier scheme, if you take the ciphertext to a power c this is the same as multiplying $m*c$. Because of the characteristics of grouping exponents the exponent c is multiplied by each of the current exponents in E_g. The value r^c will still disappear since it will be in Z_n^*. This works because $r \in Z_n^*$ and any two values $r_1 * r_2$ will always stay in Z_n^* (see 2.125 in [10]). This leaves us with the exponent of g as $m*c$, which can be extracted in decryption.

The mathematics and explanation for Damgård-Jurik will also be very similar.

$$\begin{aligned} \mathcal{E}_g(m,r)^c &= (g^m * r^{n^s})^c \bmod n^{s+1} \\ &= g^{(m*c)} * r^{(n^s*c)} \bmod n^{s+1} \\ &= \mathcal{E}_g(m*c, r^c) \end{aligned}$$

B Chinese Remainder Theorem in Public Key Decryption

The Chinese Remainder Theorem (CRT) is given as follows in [16].

Theorem 2 *Suppose gcd(m, n) = 1. Given a and b, there exists exactly one solution* $x \bmod mn$ *to the simultaneous congruences*

$$x \equiv a \bmod m,\ x \equiv b \bmod n.$$

The CRT can be very useful in our schemes because our modulus $n = pq$ and p and q are prime. Therefore, it is possible to perform two separate decryption operations on a ciphertext, one mod p and one mod q. We can then use the CRT to find an answer to both of these decryptions mod n. This answer is the same as if we had just performed one operation mod n.

When I first saw this technique used to calculate the decryption, I was skeptical that it would improve performance much. In order to use the CRT, one must perform encryption twice and then use the CRT to get the final answer in modulus n. After implementing the changes and looking at the difference in the operations, I realized that I was very wrong.

The reason why this approach is so effective is that we are able to use the the exponents $p-1$ and $q-1$. In RSA and Paillier the exponent is $\phi = (p-1)(q-1)$ or $\lambda = \frac{(p-1)(q-1)}{2}$ respectively. This means that the exponent used with the CRT is roughly the square root of what is used without the CRT. A small example of this is when p = 97, q = 103, and n = 9991. If we have a base b, then can either calculate $b^{96} \bmod 97$ and $b^{100} \bmod 101$ or $b^{9792} \bmod 9991$. You can see it is much quicker to evaluate the 2 statements that use the factors of n instead of the one statement because the exponents are so much smaller. In real world examples where the modulus n is 1024 bits, the difference is even more apparent. Of course, one could use modular exponentiation to calculate these quicker, but even so it is about 8 times faster to calculate with a factor p than with n. Since we have two of these calculations with the CRT, we get 4 a times improvement. The convergence of the two numbers to mod n is generally negligible in time, so one can expect up to a 4 times increase in speed. There are other operations going on, so one can expect 4 to be an upper bound for time improvement. But even with everything else, I never had an improvement less than 2 times and it was generally over 3 times better. Using the CRT is definitely worth the effort if possible when doing decryption.

So the question may arise, "Can we use the CRT for encryption?" It seems possible with knowledge of p and q but that is precisely the problem. We can't share p or q with the person encrypting, since these values are our private key. If we share them this is no longer a public key scheme and we lose the main concept of what we are doing. It seems very unlikely some type of operation using the CRT without knowledge of p or q is possible, so we will say this idea is not plausible.

References

[1] Anonymous, Paillier Cryptosystem, `http://en.wikipedia.org/wiki/Paillier_cryptosystem`, (2006).

[2] Anonymous, Random Self-reducibility, `http://en.wikipedia.org/wiki/Random_Self-reducibility`, (2006).

[3] Josh Daniel Cohen Benaloh, Verifiable Secret-Ballot Elections, *Doctoral Dissertation, Yale University*, (1996).

[4] Manuel Blum, Shafi Goldwasser, An Efficient Probabilistic Public-Key Encryption Scheme which Hides All Partial Information, *Advances in Cryptology-Proceedings of CRYPTO 84 (LNCS 196)*, (1985) 289-299.

[5] Ivan Damgård and Mads J. Jurik, A Generalisation, a Simplification and Some Applications of Paillier's Probabilistic Public-Key System, *PKC 2001, LNCS series*, volume 1992 (2001) 119-136.

[6] Shafi Goldwasser, Silvio Micali, Probabilistic Encryption, *Journal of Computer and System Sciences, 28*, (1984) 270-299.

[7] Mads J. Jurik, Extensions to the Paillier Cryptosystem with Applications to Cryptological Protocols, *BRICS Disertation Series*, (August 2003), `http://www.brics.dk/DS/03/9/`.

[8] Jonathan Knudsen, Java Cryptography, *O'Reilly & Associates, Java Series*, (1998).

[9] Laura Lincoln, Symmetric Private Information Retrieval via Additive Homomorphic Probabilistic Encryption, `http://www.cs.rit.edu/~lbl6598/thesis/Lincoln_Full_Document.pdf`, (2006).

[10] Alfred Menezes, Paul van Oorschot, and Scott Vanstone, *Handbook of Applied Cryptography*, (1996).

[11] David Naccache and Julien Stern, A New Public-Key Cryptosystem, *Advances in Cryptography, Proceedings of Eurocyrpt '97, Springer-Verlag*, (1997) 27-36.

[12] Tatsuaki Okamoto and Shigenori Uchiyama, A New Public-Key Cryptosystem as secure as Factoring, *Advances in Cryptography, Proceedings of Eurocyrpt '98, Springer-Verlag*, (1998) 308-318.

[13] Prof. Rafail Ostrosky, Lecture 8 of Foundations of Cryptography, `http://www.cs.ucla.edu/~rafail/TEACHING/WINTER-2005/L8/L8.pdf`, (2005).

[14] Pascal Paillier, Public-Key Cryptosystems Based on Composite Degree Residuosity Classes, *Advances in Cryptography - EUROCRYPT '99*, Springer Verlag LNCS series (1999) 223-238.

[15] Douglas R. Stinson, Cryptography Theory and Practice, *CRC Press LLC* (1995).

[16] Wade Trappe and Lawrence C. Washington, Introduction to Cryptography with Coding Theory, *Prentace Hall* (2002).

[17] Jason Weiss, Java Cryptograghy Extensions, Practical Guide for Programmers, *Morgan Kaufmann Publishers*, (2004).

[18] Eric W. Weisstein, Carmichael's Theorem, *MathWorld–A Wolfram Web Resource*, (1999-2006), `http://mathworld.wolfram.com/CarmichaelsTheorem.html`.

[19] Eric W. Weisstein, Carmichael's Function, *MathWorld–A Wolfram Web Resource*, (1999-2006), `http://mathworld.wolfram.com/CarmichaelFunction.html`.

[20] Mao Wenbo, On Probabilistic Encryption and Semantic Security, *Modern Cryptography, Theory & Practice*, (2004) 472-497.