4. Design Protocols

4.1. NodeLock License Methodology – Algorithm and Protocol

4.1.1. Client: STEP1

1. Collect node specific (hardware) details and form a message by concatenating ids, serial numbers, sequence numbers, etc.

\[ m = \text{StrFun}(h_1, h_2, h_3, h_4, \ldots, h_n) \]

Where \( h_1, h_2, h_3, \ldots, h_n \) are hardware/device specific details and \( m \) is a message of current context of hardware details, and \( \text{StrFun()} \) is a string manipulating function, which will accommodate hardware details/notations in a single message context.

2. Compute \( C_1 \) (encrypted message) from \( m \) using multi-prime rebalanced RSA.

Let \( p_1 \) be the product of \( n_1 \) randomly chosen distinct primes \( p_{11}, p_{12}, \ldots, p_{1n_1} \) and \( q_1 \) be the product of \( n_2 \) randomly chosen distinct primes \( q_{11}, q_{12}, \ldots, q_{1n_2} \).

\[ p_1 = \prod_{i=1}^{n_1} (p_{1i}) \quad \text{and} \quad q_1 = \prod_{i=1}^{n_2} (q_{1i}) \]

Let \( P_1 = (4p_1+1) \), \( Q_1 = (4q_1+1) \), and \( N_1 = P_1Q_1 \)

Compute Euler’s Totient function of \( N_1 \).

\[ \Phi(N_1) = (P_1-1)(Q_1-1) \]

Choose an integer \( e_1 \), where \( 1 < e_1 < \Phi(N_1) \), such that \( \text{GCD}(e_1, \Phi(N_1)) = 1 \)

The pair \( (N_1, e_1) \) is the public key, this shall be distributed along with the software or program for which the NodeLock license is required.

For this message \( m \in Z_{N_1} \), the cipher text is computed as

\[ C_1 = m^{e_1} \mod N_1 \]

3. Compute \( C_2 \) (encrypted message) from \( m \) using multi-prime rebalanced RSA.

Let \( p_2 \) be the product of \( n_3 \) randomly chosen distinct primes \( p_{21}, p_{22}, \ldots, p_{1n_3} \) and \( q_2 \) be the product of \( n_4 \) randomly chosen distinct primes \( q_{21}, q_{22}, \ldots, q_{2n_4} \).

\[ p_2 = \prod_{i=1}^{n_3} (p_{2i}) \quad \text{and} \quad q_2 = \prod_{i=1}^{n_4} (q_{2i}) \]

Let \( P_2 = (4p_2+1) \), \( Q_2 = (4q_2+1) \), and \( N_2 = P_2Q_2 \)
Compute Euler’s Totient function of $N_2$.

Now $\Phi(N_2) = (P_2-1)(Q_2-1)$

Choose an integer $e_2$, Where $1 < e_2 < \Phi(N_2)$, such that $\text{GCD}(e_2, \Phi(N_2)) = 1$
The pair $(N_2, e_2)$ is the public key2, this also shall be distributed along with the software or program for which the NodeLock license is required.
For this message $m \in Z_{N_2}$, the cipher text is computed as

$$C_2 = m^{e_2} \mod N_2$$

4. Create a Request file/content by combining the results of step2 and step3.
req = length of $C_1 + C_1 +$ length of $C_2 + C_2$
Here is the simple protocol to merge the contents of encrypted hardware details $C_1$ and $C_2$

4.1.2. Server: STEP2
5. After receiving the Request file at server(license generator) side, extract the contents of $C_1$ and $C_2$
req = length of $C_1 + C_1 +$ length of $C_2 + C_2$
6. Compute $d_1 = e_1^{-1} \mod \Phi(N_1)$, the private key1 is the pair $(N_1, d_1)$
For the encrypted message $C_1 \in Z_{N_1}$, the plaintext is recovered by computing $m_1 = C_1^{d_1} \mod N_1$
7. Compute $d_2 = e_2^{-1} \mod \Phi(N_2)$, the private key2 is the pair $(N_2, d_2)$
For the encrypted message $C_2 \in Z_{N_2}$, the plaintext is recovered by computing $m_2 = C_2^{d_2} \mod N_2$
8. Compare the plaintexts of $m_1$ and $m_2$ and verify whether the request file/content is valid.
   $\text{Isvalidrequestfile} = (m_1 \text{ equals to } m_2)$
   Or
   
   If $(m_1 = m_2)$ then Isvalidrequestfile = true;
   else Isvalidrequestfile = false;
   i.e., $m = m_1 = m_2$
9. Compute the license key/content/file from original message $m$.
License = $f(m)$
Here $f(m)$ is defined, such that no two hardware messages give the same license
i.e., If \( m_1 \neq m_2 \) then \( f(m_1) \neq f(m_2) \)

This is NodeLock License concept.

In other terms, for any two hardware details \( m_1 \) and \( m_2 \), \( f(m_1) \neq f(m_2) \).

10. Compute \( C_3 \) (encrypted cipher text from message \( f(m) \), using multi prime re-balanced RSA.

Let \( p_3 \) be the product of \( n_5 \) randomly chosen distinct primes \( p_{31}, p_{32}, \ldots, p_{3n_5} \) and \( q_2 \) be the product of \( n_6 \) randomly chosen distinct primes \( q_{31}, q_{32}, \ldots, q_{3n_6} \).

\[
\text{i.e, } p_3 = \prod_{i=1}^{n_5} (p_{3i}) \quad \text{and} \quad q_2 = \prod_{i=1}^{n_6} (q_{3i})
\]

Let \( P_3 = (4p_3+1), Q_3 = (4q_3+1), \) and \( N_3 = P_3Q_3 \)

Compute Euler's Totient function of \( N_3 \).

\[
\text{Now } \Phi(N_3) = (P_3-1)(Q_3-1)
\]

Chose an integer \( e_3 \), Where \( 1 < e_3 < \Phi(N_3) \), such that \( \gcd(e_3, \Phi(N_3)) = 1 \)

The pair \((N_3, e_3)\) is the public key 3, this also shall be distributed along with the software or program for which the NodeLock license is required.

For this message \( m \in \mathbb{Z}_{N_3} \), the cipher text is computed as

\[
C_3 = f(m)^{e_3} \mod N_3
\]

4.1.3. Client: STEP3

11. After receiving the license key at the client (license validator) side, decode the contents of cipher text \( C_3 \).

License key = \( f(m) = C_3^{d_3} \mod N_3 \) (this is license key decoding)

12. Define function \( g \) such that \( f(m) \) identically equal to \( g(m) \).

i.e., \( f(m_1) \) and \( g(m_1) \) shall always give similar output

and for \( m_1 \) and \( m_2 \), \( f(m_1) \) identically equal to \( g(m_2) \)

13. Get the hardware details or device properties (\( h \)) and see whether license is valid

\[
\text{Isvalidlicense} = \begin{cases} 
\text{true} & \text{if } f(m) \text{ identically equal to } g(m) \\
\text{false} & \text{else}
\end{cases}
\]

14. Register the product on successful validation of NodeLock License key.
4.2. Sequence Diagrams and Workflows

A sequence diagram is a form of interaction diagram which shows objects as lifelines running down the page, with their interactions over time represented as messages drawn as arrows from the source lifeline to the target lifeline. Sequence diagrams are good at showing which objects communicate with which other objects; and what messages trigger those communications. Sequence diagrams are not intended for showing complex procedural logic.

The main purpose of a sequence diagram is to define event sequences that result in some desired outcome. The focus is less on messages themselves and more on the order in which messages occur; nevertheless, most sequence diagrams will communicate what messages are sent between a system's objects as well as the order in which they occur. The diagram conveys this information along the horizontal and vertical dimensions: the vertical dimension shows, top down, the time sequence of messages/calls as they occur, and the horizontal dimension shows, left to right, the object instances that the messages are sent to.

The first message of a sequence diagram always starts at the top and is typically located on the left side of the diagram for readability. Subsequent messages are then added to the diagram slightly lower than the previous message.

4.2.1. Key Generation

RSA involves a public key and private key. The public key can be known to everyone, it is used to encrypt messages. Messages encrypted using the public key can only be decrypted with the private key. The keys for the RSA algorithm are generated the following way:

Choose two different large random prime numbers $p$ and $q$.

Calculate $n = pq$.

$n$ is the modulus for the public key and the private keys.

Calculate the totient: $\phi(n) = (p - 1)(q - 1)$.

Choose an integer $e$ such that $1 < e < \phi(n)$, and $e$ is coprime to $\phi(n)$; i.e., $e$ and $\phi(n)$ share no factors other than 1; $\gcd(e, \phi(n)) = 1$.

$e$ is released as the public key exponent.

Compute $d$ to satisfy the congruence relation $de \equiv 1 \pmod{\phi(n)}$.

$d$ is kept as the private key exponent.

The public key is made of the modulus $n$ and the public (or encryption) exponent $e$. The private key is made of the modulus $n$ and the private (or decryption) exponent $d$ which must be kept secret.

For efficiency a different form of the private key can be stored:

$p$ and $q$: the primes from the key generation,

$d \mod (p - 1)$ and $d \mod (q - 1)$: often called dmp1 and dmq1.
Following diagram shows the sequence of key generation and license distribution steps in NodeLock license generation process:

\[ \frac{q}{p} \mod p \]: often called iqmp

All parts of the private key must be kept secret in this form. \( p \) and \( q \) are sensitive since they are the factors of \( n \), and allow computation of \( d \) given \( e \). If \( p \) and \( q \) are not stored in this form of the private key then they are securely deleted along with other intermediate values from key generation.

Although this form allows faster decryption and signing by using the Chinese Remainder Theorem (CRT) it is considerably less secure since it enables side channel attacks. This is a particular problem if implemented on smart cards, which benefit most from the improved efficiency. (Start with \( y = x^e \mod n \) and let the card decrypt that. So it computes \( y^e \mod p \) or \( y^d \mod q \) whose results give some value \( z \). Now, induce an error in one of the computations. Then \( \gcd(z - x, n) \) will reveal \( p \) or \( q \) [47].

Software, programming code or a Web-based service designed to generate random license keys for software developers to provide licensed users as a way of protecting their software from being used illegally (aka pirated). Some software license generators can also perform additional licensing tasks such as license key delivery, management and verification.

A license key created by a license generator is a unique series of numbers and/or letters that identifies the legal user of a software application. A license key that has been created by a license generator is provided after a software user pays for the software and agrees to the conditions of use and distribution as legally specified in the software license (also known as an End-User License Agreement, or EULA). Software license keys may be verified within the software itself or, more commonly, over the Internet via an online license verification system.

A license generator also refers to software or Web services designed to generate the language in a software license that stipulates how software can be legally used and by how many users [48].

Following diagram shows the sequence of key generation and license distribution steps in NodeLock license generation process:
Figure 4.1 Key Generation Process of NodeLock Licensing Methodology

Method: DefineSecurityParams()
Input: None
Output: Boolean status <- acknowledge
Process:

1. Boolean ack = false;
2. SecurityParamManager securityParamManager = new SecurityParamManager();
3. Int32 keyLength = 0; // 512
4. List<int> pPrimes = new List<int>(); // new Int32[] { 13, 17, 29, 31 });
5. List<int> qPrimes = new List<int>(); // new Int32[] { 11, 19, 23 });
6. Console.WriteLine("Enter keyLength");
7. String keyLengthStr = Console.ReadLine();
8. Int32.TryParse(keyLengthStr, out keyLength);
9. Console.WriteLine("Enter comma(,) seperated prime numbers for p:");
10. String[] listStrPrimes = Console.ReadLine();
11. foreach (String strPrime in listStrPrimes)
12. {
13.    int prime = 0;
14.    int32.TryParse(strPrime.Trim(), out prime);
15.    if (prime > 0 && IsPrime(prime) && !pPrimes.Contains(prime))
16.       if (prime > 0 && IsPrime(prime) && !pPrimes.Contains(prime))
NodeLock Licensing Methodologies Using RMPRSA Cryptography

```csharp
17. pPrimes.Add(prime);
18. }
19. Console.WriteLine("Enter comma(,) seperated prime numbers for q: ");
20. strPrimes = Console.ReadLine();
21. liststrPrimes = strPrimes.Split(' ,');
22. foreach (String strPrime in liststrPrimes)
23. {
24.     Int32 prime = 0;
25.     Int32.TryParse(strPrime.Trim(), out prime);
26.     if (prime > 0 && IsPrime(prime) && !qPrimes.Contains(prime))
27.         qPrimes.Add(prime);
28. }
29. ack = securityParamManager.ApplySecurityParams(keyLength, pPrimes, qPrimes, null);
31. _rebalancedRSA = new RebalancedRSA();
32. _rebalancedRSA.Ready("NodeLock
33. KeyGeneration", securityParams.PPrimes, securityParams.QPrimes, securityParams.KeyLength);
34. return ack;
```

Method: ApplySecurityParams()
Input: Int32 keyLength,
       List<Int32> pPrimes,
       List<Int32> qPrimes,
       List<Object> objParams
Output: Boolean status <- acknowledge

Process:
1. Boolean isSuccess = false;
2. if (keyLength < 1 || keyLength > 0xffff || pPrimes.Count <= 0 || qPrimes.Count <= 0)
3.     { isSuccess = SaveSecurityParams(securityParams); }
4. return isSuccess;
5. }
7. securityParams.KeyLength = keyLength;
8. securityParams.PPrimes = pPrimes;
9. securityParams.QPrimes = qPrimes;
10. securityParams.ObjParams = objParams;
11. isSuccess = SaveSecurityParams(securityParams);
12. return isSuccess;

Method: DefineLicenseDistributionSettings()
Input: None
Output: Boolean status <- acknowledge

Process:
1. Boolean ack = true;
2. try
3. {
4.     a. Console.WriteLine("Enter request file content in encrypted format: ");
5.     b. String requestContent = Console.ReadLine();
6.     c. SecurityKeyManager securityKeyManager = new SecurityKeyManager();
7.     d. RebalancedRSA rebalancedRSA = securityKeyManager.GetSecuritySettings();
8.     e. Console.WriteLine("License Distribution Details : "+ rebalancedRSA.ToString());
4. }
5. On catch ()
NodeLock Licensing Methodologies Using RMPRSA Cryptography

6. {
   a. ack = false;
7. }
8. return ack;

Method: GetSecuritySettings()
Input: None
Output: Boolean status <- acknowledge
Process:
1. _rebalancedRSA= new RebalancedRSA();
2. _rebalancedRSA.Ready("NodeLock
   KeyGeneration",securityParams.PPrimes,securityParams.QPrimes,securityParams.KeyLength);
3. return _rebalancedRSA;

4.2.2. License Distribution

Allowing an individual or group to use a piece of software. Nearly all applications are licensed rather than sold. There are a variety of different types of software licenses. Some are based on the number machines on which the licensed program can run whereas others are based on the number of users that can use the program. Most personal computer software licenses allow you to run the program on only one machine and to make copies of the software only for backup purposes. Some licenses also allow you to run the program on different computers as long as you don't use the copies simultaneously [49].

License registration tools and services provide the means for presenting a software developer's software licenses (also known as an End-User License Agreement, or EULA) to an end user as well as facilitating the transfer of required registration information and consent to the license. Successfully registering a software application often activates the software for legal use by the end user or enterprise. License registration is often used in conjunction with generation and license management tools to prevent the developer's software from being copied, shared or otherwise illegally used (aka pirated) by non-licensed users [50].

Software license management refers to the software tools or processes used by an organization to control and document where and how the company's software products are able to run in order to enforce and ensure compliance with software licenses (also known as an End-User License Agreement, or EULA).

License management tools are also employed by software vendors to monitor and ensure compliance with the developer's software license. These tools help prevent the developer's software from being copied, shared or otherwise illegally used (pirated) by non-licensed users. License management on the software vendor's side may also involve creating trial, personal, corporate and/or subscription-based license keys via a license generator; software product activation; product deactivation; and similar tasks.

Software licensing technology enables software developers to protect their intellectual property by facilitating the process of creating, issuing and managing software licenses [51]. Licensing technology typically includes license key generation, license registration and license
management tools to prevent the developer's software from being copied, shared or otherwise illegally used (aka pirated) by non-licensed users.

**Figure 4.2 License Generation Process of NodeLock Licensing Methodology**

**Method:** GenerateLicense()

**Input:** None

**Output:** Boolean status <- acknowledge

**Process:**
1. Boolean ack = true;
2. try
3. {
4.     Console.WriteLine("Enter Request file content in encrypted format:");
5.     String requestContent = Console.ReadLine();
6.     LicenseKeyDistributor licenseKeyDistributor = new LicenseKeyDistributor();
7.     RebalancedRSA rebalancedRSA = licenseKeyDistributor.GetKeyDetails();
8.     Utilities utilities = new Utilities();
9.     String hardwareDetails = rebalancedRSA.Decrypt(requestContent);
10.    String licenseContent = utilities.GetLicenseContent(requestContent);
11.    String encryptedLicense = rebalancedRSA.Encrypt(licenseContent);
12.    Console.WriteLine("Encrypted License Content : " + encryptedLicense);
13. }
14. catch (Exception ex)
15. {
16.     ack = false;
17. }
18. return ack;

**Method:** GetPrivateKeyDetails()
**Input:** None  
**Output:** Boolean status <- acknowledge

**Process:**
1. RebalancedRSA_rebalancedRSA = null;
2. SecurityKeyManager securityKeyManager = new SecurityKeyManager();
3. _rebalancedRSA = securityKeyManager.GetSecuritySettings();
4. return _rebalancedRSA.GetPrivateKey();

**Method:** GetPublicKeyDetails()
**Input:** None  
**Output:** Boolean status <- acknowledge

**Process:**
1. RebalancedRSA_rebalancedRSA = null;
2. SecurityKeyManager securityKeyManager = new SecurityKeyManager();
3. _rebalancedRSA = securityKeyManager.GetSecuritySettings();
4. return _rebalancedRSA.GetPublicKey();

### 4.2.3. NodeLock software preparation and installation

![Software Preparation and Installation Diagram](image)

**Figure 4.3 Software Preparation and Installation of NodeLock Licensing Methodology**

**Method:** GetPublicKeyDetails()
**Input:** None  
**Output:** Boolean status <- acknowledge

**Process:**
1. RebalancedRSA_rebalancedRSA = null;
2. SecurityKeyManager securityKeyManager = new SecurityKeyManager();
3. _rebalancedRSA = securityKeyManager.GetSecuritySettings();
4. return _rebalancedRSA.GetPublicKey();

**Method: GetKeyDetails()**

Input: None

Output: Boolean status <- acknowledge

Process:

1. RebalancedRSA _rebalancedRSA = null;
2. SecurityKeyManager securityKeyManager = new SecurityKeyManager();
3. _rebalancedRSA = securityKeyManager.GetSecuritySettings();
4. return _rebalancedRSA;

**Method: GetNodeLockSetup()**

Input: None

Output: String fileName <- setup name

Process:

1. String fileName = String.Empty;
2. RebalancedRSA _rebalancedRSA = null;
3. SecurityKeyManager securityKeyManager = new SecurityKeyManager();
4. _rebalancedRSA = securityKeyManager.GetSecuritySettings();
5. StorageManager storageManager = new StorageManager();
6. String fileName = "SampleSoftwareSetup.msi";
7. storageManager.CreateLicenseEnabledSetup(fileName, _rebalancedRSA.GetLicenseDetails());
8. return fileName;

**Method: InstallSoftware()**

Input: None

Output: Boolean status <- acknowledge

Process:

1. Boolean ack = true;
2. try
3. {
4. String fileName = GetNodeLockSetup();
5. Utilities utilities = new Utilities();
6. utilities.InstallSoftware(fileName);
7. }
8. catch (Exception ex)
9. {
10. ack = false;
11. }
12. return ack;
4.2.4. Certificate generation & software activation

**Method**: GetPrivateKeyDetails()
**Input**: None
**Output**: Boolean status <- acknowledge

**Process**:
1. RebalancedRSA._rebalancedRSA = null;
2. SecurityKeyManager securityKeyManager = new SecurityKeyManager();
3. _rebalancedRSA = securityKeyManager.GetSecuritySettings();
4. return _rebalancedRSA.GetPrivateKey();

**Method**: GetPublicKeyDetails()
**Input**: None
**Output**: Boolean status <- acknowledge

**Process**:
1. RebalancedRSA._rebalancedRSA = null;
2. SecurityKeyManager securityKeyManager = new SecurityKeyManager();
3. _rebalancedRSA = securityKeyManager.GetSecuritySettings();
4. return _rebalancedRSA.GetPublickey();

**Method**: InstallNodeLockLicense()
**Input**: None

---

**Figure 4.4 Certificate Generation and Software Activation of NodeLock Licensing Methodology**

`sd Generate nodelock license`
Output: Boolean status <- acknowledge

Process:

1. Boolean ack = true;
2. try
3. {
4. String fileName = GetNodeLockSetup();
5. Utilities utilities = new Utilities();
6. utilities.InstallNodeLockLicense(fileName);
7. }
8. catch (Exception ex)
9. {
10. ack = false;
11. }
12. return ack;

Method: GetNodeLockLicenseWithPrivateKey ()
Input: None
Output: ObjLicenseWithPrivateKey

Process:

1. RebalancedRSA _rebalancedRSA = null;
2. SecurityKeyManager securityKeyManager = new SecurityKeyManager();
3. _rebalancedRSA = securityKeyManager.GetSecuritySettings();
4. PrivateKey privateKey = _rebalancedRSA.GetPrivateKey();
5. LicenseCertificateGenerator licenseCertificateGenerator = new LicenseCertificateGenerator();
6. Obj = licenseCertificateGenerator.LicenseWithPrivateKey();
7. Return Obj;

4.2.5. Additional Libraries and Utilities Used

Method: SetupRSAKeys() 
Input: None
Output: ObjkeyDetails

Process:

1. s_keys_are_setup = false;
2. if (String.IsNullOrEmpty(pString)) return false;
3. if (String.IsNullOrEmpty(qString)) return false;
4. P = new BigInteger(pString, 10);
5. Q = new BigInteger(qString, 10);
6. N = P * Q;
7. R = (P - 1) * (Q - 1) [27];
8. E = R.genCoPrime(_bitCount, new Random());
9. // choose x and y that satisfy e*x - r*y = 1
10. X = 1;
11. Y = 1;
12. while (((E * X) - (R * Y)) != 1)
10. { 
11. X++; 
12. Y = ((E * X) - 1) / R; 
13. if (X > N) 
14. { 
15. // choose a different e 
16. E = R.genCoPrime(_bitCount, new Random()); 
17. 
18. // restart search for suitable x and y 
19. X = 1; 
20. Y = 1; 
}
21. s_keys_are_setup = true; 
22. return s_keys_are_setup;

Method: RSAEncrypt() 
Input: plainText 
Output: Obj <-> cyperText 
Process: 
1. String ciphervalue = String.Empty; 
2. foreach (Char ch in message) 
3. { 
3.1.1.String str = Encrypt_1(ch.ToString()); 
3.1.2.ciphervalue += str.Length + str; 
4. } 
5. return ciphervalue;

Method: RSADecrypt() 
Input: cyperText 
Output: Obj <-> plainText 
Process: 
1. String planText = String.Empty; 
2. Int32 i = 0; 
3. while (i < ciphervalue.Length) 
4. { 
4.1.1.String len = ciphervalue[i].ToString(); 
4.1.2.Int32 l = Int32.Parse(len); 
4.1.3.String ct = ciphervalue.Substring(i + 1, l); 
4.1.4.planText += Decrypt_1(ct); 
4.1.5.i += (l + 1); 
5. } 
6. return planText;

Method: RSAReady() 
Input: NA 
Output: Obj <-> generatedKeys 
Process:
Utilities u = new Utilities();
_bitCount = bitCount;
// List<int> primeList1 = u.GetPrimesFromConsole(strTitle, "p Series");
// List<int> primeList2 = u.GetPrimesFromConsole(strTitle, "q Series");
s_ps.Clear();
s_qs.Clear();

String strDetails = String.Empty;
String str = String.Empty;
Int32 p = 1, q = 1;
Boolean isFirst = true;
foreach (Int32 pi in primeList1)
{
  p *= pi;
  if (!isFirst) str += "+";
  str += pi.ToString();
  isFirst = false;
}
strDetails += Environment.NewLine + "p (" + s_ps.Count.ToString() + ") Given Primes) = " + str + " = " + p.ToString();
str = String.Empty;
isFirst = true;
foreach (Int32 qi in primeList2)
{
  q *= qi;
  if (!isFirst) str += "+";
  str += qi.ToString();
  isFirst = false;
}
strDetails += Environment.NewLine + "q (" + s_qs.Count.ToString() + ") Given Primes) = " + str + " = " + q.ToString();
Int32 p4 = 4 * p + 1, q4 = 4 * q + 1;

strDetails += Environment.NewLine + "Q = (4*" + q.ToString() + "+1) = " + q4.ToString();
strDetails += Environment.NewLine + "Performing P, Q Adjustments (if not primes)...";
p4 = u.GetNextPrime(p4);
q4 = u.GetNextPrime(q4);
SetupKeys(p4.ToString(), q4.ToString());
strDetails += Environment.NewLine + "P = " + s_p + Environment.NewLine + "Q = " + s_q + Environment.NewLine + "N = (P*Q) = " + s_n + Environment.NewLine + "Phi = (P-1) *(Q-1) = " + s_r;

strDetails += Environment.NewLine + "Public Key = (E,N) = (" + s_e + "+"," + s_n + ")";
strDetails += Environment.NewLine + "Private Key = (D,N) = (" + s_d + "+"," + s_n + ")";

return strDetails;

Method: IsPrime ()
Input: NA
Output: Obj ← isPrime
Process:

1. bool isPrime = true;
2. if (number == 2 || number == 3) return true;
3. if(number%2 == 0 || number %3 ==0) return false;
4. for(Int32 i =5;i<number;i+=2)
5. { 
   5.1.1. if (number % i == 0) 
   5.1.2. { 
   5.1.2.1. isPrime = false;
   5.1.2.2. break;
   5.1.3. }
   5.1.3. }
6. }
7. return isPrime;

Method: GetPrimes ()
Input: countRequired
Output: Obj ← List of primeNumbers
Process:

1. List<Int32> primesList = new List<Int32>();
2. primesList.Add(3);
3. for (Int32 n = 5; primesList.Count < count; n += 2)
4. { 
   4.1.1. if(IsPrime(n))
   4.1.1.1. primesList.Add(n);
   5. }
6. return primesList;

Method: GetNextPrime ()
Input: number
Output: Obj ← next prime number
Process:

1. while (IsPrime(number) == false)
2. { 
   2.1.1. number++;
3. }
4. return number;

Method: GetRandomNumber ()
Input: range
Output: Obj ← random number
Process:
1. Int32 ran = 0;
2. String guid = Guid.NewGuid().ToString();
3. foreach (char ch in guid)
4. {
   (a) ran += ch;
5. }
6. return ran % range

Method Summary - BigInteger

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BigInteger</td>
<td>abs()</td>
</tr>
<tr>
<td></td>
<td>Returns a BigInteger whose value is the absolute value of this BigInteger.</td>
</tr>
<tr>
<td>BigInteger</td>
<td>add(BigInteger val)</td>
</tr>
<tr>
<td></td>
<td>Returns a BigInteger whose value is (this + val).</td>
</tr>
<tr>
<td>BigInteger</td>
<td>and(BigInteger val)</td>
</tr>
<tr>
<td></td>
<td>Returns a BigInteger whose value is (this &amp; val).</td>
</tr>
<tr>
<td>BigInteger</td>
<td>andNot(BigInteger val)</td>
</tr>
<tr>
<td></td>
<td>Returns a BigInteger whose value is (this &amp; ~val).</td>
</tr>
<tr>
<td>int</td>
<td>bitCount()</td>
</tr>
<tr>
<td></td>
<td>Returns the number of bits in the two's complement representation of this</td>
</tr>
<tr>
<td></td>
<td>BigInteger that differ from its sign bit.</td>
</tr>
<tr>
<td>int</td>
<td>bitLength()</td>
</tr>
<tr>
<td></td>
<td>Returns the number of bits in the minimal two's-complement</td>
</tr>
<tr>
<td></td>
<td>representation of this BigInteger, excluding a sign bit.</td>
</tr>
<tr>
<td>BigInteger</td>
<td>clearBit(int n)</td>
</tr>
<tr>
<td></td>
<td>Returns a BigInteger whose value is equivalent to this BigInteger with the</td>
</tr>
<tr>
<td></td>
<td>designated bit cleared.</td>
</tr>
<tr>
<td>int</td>
<td>compareTo(BigInteger val)</td>
</tr>
<tr>
<td></td>
<td>Compares this BigInteger with the specified BigInteger.</td>
</tr>
<tr>
<td>BigInteger</td>
<td>divide(BigInteger val)</td>
</tr>
<tr>
<td></td>
<td>Returns a BigInteger whose value is (this / val).</td>
</tr>
<tr>
<td>BigInteger[]</td>
<td>divideAndRemainder(BigInteger val)</td>
</tr>
<tr>
<td></td>
<td>Returns an array of two BigIntegers containing (this / val) followed by (</td>
</tr>
<tr>
<td></td>
<td>(this % val).</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>doubleValue()</code></td>
<td>Converts this BigInteger to a double.</td>
</tr>
<tr>
<td><code>equals(Object x)</code></td>
<td>Compares this BigInteger with the specified Object for equality.</td>
</tr>
<tr>
<td><code>flipBit(int n)</code></td>
<td>Returns a BigInteger whose value is equivalent to this BigInteger with the designated bit flipped.</td>
</tr>
<tr>
<td><code>floatValue()</code></td>
<td>Converts this BigInteger to a float.</td>
</tr>
<tr>
<td><code>gcd(BigInteger val)</code></td>
<td>Returns a BigInteger whose value is the greatest common divisor of abs(this) and abs(val).</td>
</tr>
<tr>
<td><code>getLowestSetBit()</code></td>
<td>Returns the index of the rightmost (lowest-order) one bit in this BigInteger (the number of zero bits to the right of the rightmost one bit).</td>
</tr>
<tr>
<td><code>hashCode()</code></td>
<td>Returns the hash code for this BigInteger.</td>
</tr>
<tr>
<td><code>intValue()</code></td>
<td>Converts this BigInteger to an int.</td>
</tr>
<tr>
<td><code>isProbablePrime(int certainty)</code></td>
<td>Returns true if this BigInteger is probably prime, false if it's definitely composite.</td>
</tr>
<tr>
<td><code>longValue()</code></td>
<td>Converts this BigInteger to a long.</td>
</tr>
<tr>
<td><code>max(BigInteger val)</code></td>
<td>Returns the maximum of this BigInteger and val.</td>
</tr>
<tr>
<td><code>min(BigInteger val)</code></td>
<td>Returns the minimum of this BigInteger and val.</td>
</tr>
<tr>
<td><code>mod(BigInteger m)</code></td>
<td></td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>Returns a BigInteger whose value is (this mod m).</td>
<td></td>
</tr>
<tr>
<td>modInverse(BigInteger m)</td>
<td>Returns a BigInteger whose value is (this-1 mod m).</td>
</tr>
<tr>
<td>modPow(BigInteger exponent, BigInteger m)</td>
<td>Returns a BigInteger whose value is (this exponent mod m).</td>
</tr>
<tr>
<td>multiply(BigInteger val)</td>
<td>Returns a BigInteger whose value is (this * val).</td>
</tr>
<tr>
<td>negate()</td>
<td>Returns a BigInteger whose value is (-this).</td>
</tr>
<tr>
<td>nextProbablePrime()</td>
<td>Returns the first integer greater than this BigInteger that is probably prime.</td>
</tr>
<tr>
<td>not()</td>
<td>Returns a BigInteger whose value is (~this).</td>
</tr>
<tr>
<td>or(BigInteger val)</td>
<td>Returns a BigInteger whose value is (this</td>
</tr>
<tr>
<td>pow(int exponent)</td>
<td>Returns a BigInteger whose value is (this exponent).</td>
</tr>
<tr>
<td>probablePrime(int bitLength, Random rnd)</td>
<td>Returns a positive BigInteger that is probably prime, with the specified bitLength.</td>
</tr>
<tr>
<td>remainder(BigInteger val)</td>
<td>Returns a BigInteger whose value is (this % val).</td>
</tr>
<tr>
<td>setBit(int n)</td>
<td>Returns a BigInteger whose value is equivalent to this BigInteger with the designated bit set.</td>
</tr>
<tr>
<td>shiftLeft(int n)</td>
<td>Returns a BigInteger whose value is (this &lt;&lt; n).</td>
</tr>
<tr>
<td>Method Type</td>
<td>Method Name</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>BigInteger</td>
<td>shiftRight(int n)</td>
</tr>
<tr>
<td>int</td>
<td>signum()</td>
</tr>
<tr>
<td>BigInteger</td>
<td>subtract(BigInteger val)</td>
</tr>
<tr>
<td>boolean</td>
<td>testBit(int n)</td>
</tr>
<tr>
<td>byte[]</td>
<td>toByteArray()</td>
</tr>
<tr>
<td>String</td>
<td>toString()</td>
</tr>
<tr>
<td>String</td>
<td>toString(int radix)</td>
</tr>
<tr>
<td>static BigInteger</td>
<td>valueOf(long val)</td>
</tr>
<tr>
<td>BigInteger</td>
<td>xor(BigInteger val)</td>
</tr>
</tbody>
</table>

*Table 2 Methodology Summary of BigInteger*