Journal of Theoretical and Applied Information Technology
15th September 2018. Vol.96. No 17

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

USING THE RSA AS AS AN ASYMMETRIC NON-PUBLIC KEY ENCRYPTION ALGORITHM IN THE SHAMIR THREE-PASS PROTOCOL

[1]DIAN RACHMAWATI, [2]MOHAMMAD ANDRI BUDIMAN

[1,2]Departemen Ilmu Komputer, Fakultas Ilmu Komputer dan Teknologi Informasi, Universitas Sumatera Utara, Jl. Universitas No. 9-A, Kampus USU, Medan 20155, Indonesia

E-mail: [1]dian.rachmawati@usu.ac.id, [2]mandrib@usu.ac.id

ABSTRACT

The Shamir three-pass protocol lets two parties to communicate in a secure manner without the need of exchanging any secret keys. As with many other cryptographic protocols, the Shamir three-pass protocol needs an algorithm in order to work in a proper manner. The algorithms used in the three-pass protocol should belong to the class of symmetric algorithms and follows commutative-encryption system. Our study takes an unconventional approach: instead of using a symmetric algorithm, we use RSA, an asymmetric algorithm, in the three-pass protocol. RSA is a public key crypto-system that relies its security on the difficulty of factoring a big integer into two prime numbers. However, in this study, the RSA is not used as a public key algorithm, but rather as an asymmetric non-public key encryption algorithm. This is done by setting both encryption and decryption keys to private. The complete computation of this scheme is done in Python programming language. Our study shows that this scheme works conveniently in the three-pass protocol.

Keywords: *Cryptography, Commutative-Encryption, Asymmetric Non-Public Key, RSA, Shamir Three-Pass Protocol*

1. INTRODUCTION

Cryptography can be defined as the craft and science of using mathematical techniques to communicate in a secure manner. With a symmetric cryptography concept, a message in the forms of symbols or texts is transformed into numbers. Using an encryption algorithm (or cipher) and a key (or password), the numbers are then converted into other numbers whose meanings are absurd. These meaningless numbers (or ciphertext) are then sent to the recipient via a channel (which can be a secure or an insecure channel). Meanwhile, the key is also sent to the recipient by using a different and, preferably, securer channel. The recipient then uses a decryption algorithm (which is simply the "inverse" of the encryption algorithm) and the key in order to get the real message. Thus, in order to sent a message securely, there are at least two things that should be sent to the recipient: the ciphertext and the key.

With the public key cryptography concept (as proposed by Diffie and Hellman [1]), the need to send the key to the recipient is reduced. This concept works as follows. Firstly, the recipient generates two kinds of keys: private key and public key. His public key is published in the public channel by uploading it to his key server or by other electronic means; while his private key is kept secret. Secondly, the sender of the message looks up for the recipient's public key, encrypts her message with that key, and sends the ciphertext to the recipient. Thirdly, the recipient decrypts the ciphertext with his own private keys and gets the original message back. Since it uses two different keys for encryption and decryption, the public key cryptography is often called "asymmetric cryptography" but we will show later that this is not always the case.

With the concept of public key cryptography, there is seemingly no need to send any keys. However, the public key itself still has to be published in some electronic ways, and that means everybody may know it. A savvy cryptanalyst may then use some mathematical methods to derive the private key from the public key. For example, if the public key cryptosystem being used is RSA [2] with public key $n = 113053$, then by understanding that $n = pq$ (where p and q are RSA's private keys) and using a factorization algorithm such as quadratic sieve [3], a cryptanalyst can easily derive that $113053 = 131 * 863$ and the RSA cryptosystem is compromised. Since RSA's security depends on the hardness of factoring an integer, a sensible way to

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

increase RSA security is to use a larger public key n of around 2048-bit. However, Boneh and Venkatesan [4] shows some evidences that breaking RSA may be easier than factoring if the other RSA's parameter (i.e., the RSA's exponent e) is small, even though the public key n is very large.

The Shamir three-pass protocol enables a sender to transmit a message without the need of sending, publishing, or distributing any keys. The protocol works as follows [5]. The sender encrypts the message using her own key. In the first pass, the encrypted message is sent to the recipient. Then, the recipient super-encrypts the encrypted message using his own key. In the second pass, the super-encrypted message is sent to the sender. The sender decrypts the super-encrypted message. He then sends it to the recipient in the third pass. Using his own key, the recipient decrypts it, and the original message is recovered.

When Shamir developed the the idea of three-pass protocol in 1980, the speed of data transmission and the computation speed were very slow. Therefore, just about forty years ago, securing messages with three-pass protocol which uses two encryption-decryption processes and three transmission processes was arguably less popular than securing messages with an ordinary symmetric cryptographic algorithm which uses only one encryption-decryption process and one transmission process. However, an ordinary symmetric cryptography algorithm still has its own problem: its key has to be encrypted using asymmetric cryptography algorithm and then sent to the sender via a secure channel. The problem of sending secured messages without sending their corresponding encryption keys is exactly the problem that can be solved by the three-pass protocol.

Nowadays, the speed of data transmission is very fast due to the massive developments of computing and information technology devices. Thus, the problem of sending three different encrypted messages in the three-pass protocol is minimized since the transmission time is reduced due to the advancement of technologies. As a result, the three-pass protocol may take an important role in modern information security.

In order to be implemented for practical uses, the three-pass protocol needs an encryption algorithm to encrypt and decrypt the messages. The algorithm should belong to the class of symmetric cryptography [6]. The algorithm should also follow commutative principle, so that a message can be encrypted and decrypted using two different keys in any order.

Our study takes a quite different approach. Instead of using a symmetric algorithm, we use an asymmetric algorithm in the three-pass protocol. The asymmetric algorithm is the RSA encryption algorithm. The RSA relies its security on the hardness of factoring a big integer into two large prime numbers. This paper shows that the RSA works conveniently in the three-pass protocol.

2. SHAMIR THREE-PASS PROTOCOL

The three-pass protocol was formulated by Adi Shamir in 1980. This protocol allows two parties to exchange encrypted messages without exchanging any encryption keys. Since there are no keys being sent, exchanged, published, or distributed, this protocol is also known as Shamir no-key protocol. The protocol works as the following [5]:

1. The sender and the recipient choose a symmetric cryptography encryption algorithm to be used in the three-pass protocol. The chosen algorithm should follow commutative principle [6]. By following the commutative principle, encryption and decryption using two different keys can be done in any order.
2. The sender encrypts her message, m, using her own key, K_A, resulting in a ciphertext, $c1$.
3. In the first pass, the sender sends $c1$ to the recipient.
4. The recipient super-encrypts $c1$ using his own key, K_B, resulting in another ciphertext, $c2$.
5. In the second pass, the recipient sends $c2$ to the sender.
6. The sender decrypts $c2$ using her own key, K_A, resulting in another ciphertext, $c3$.
7. In the third pass, the sender sends $c3$ to the recipient.
8. The recipient decrypts $c3$ using his own key, K_B, and therefore, he recovers the original message, m.

3. RSA ENCRYPTION SCHEME

The RSA public key cryptosystem was formulated by Ronald Rivest, Adi Shamir, and Leonard Adleman in 1978. RSA cryptosystem can be used for digital signature and encryption. RSA is

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

believed as the most widely used public key algorithm due to its simplicity.

When RSA is used as an encryption scheme (as in this work), there are three stages to follow, which are key generation, encryption, and decryption [2] [7] [8].

In the stage of key generation, the recipient does as follows:

1. Select two large and distinct prime numbers, p and q.
2. Calculate $n = pq$.
3. Calculate $\Phi(n) = (p - 1)(q - 1)$.
4. Select an integer e, so that $gcd(\Phi(n), e) = 1$ and $1 < e < \Phi(n)$.
5. Calculate d, so that $ed \equiv 1 \pmod{\Phi(n)}$.
6. The private keys are $(p, q, \Phi(n), d)$. These keys have to be kept secret.
7. The public keys are (n, e). These keys have to be published by some electronic ways so that anyone who wants to send a message to the recipient can use them in the encryption process.

In the stage of encryption, the sender does as follows:

1. Obtain the recipient's public key, (n, e).
2. Prepare the message, m.
3. Calculate the ciphertext, $c = m^e \bmod n$.
4. Send c to the recipient.

In the stage of encryption, the recipient does as follows:

1. Receive the ciphertext, c from the sender.
2. Calculate the original message, $m = c^d \bmod n$.

4. ASYMMETRIC NON-PUBLIC KEY ENCRYPTION ALGORITHM

Asymmetric cryptography is a cipher or cryptography algorithm that uses different keys for encryption and decryption. The encryption key is usually set to public while the decryption key is set to private. Therefore, it has been widely believed that the term 'asymmetric cryptography' is synonymous with the term 'public key cryptography'.

However, this opinion is proven to be untrue since there is an asymmetric algorithm called Pohlig-Hellman whose encryption and decryption keys are both set to private [9]. Therefore, Pohlig-Hellman is still an asymmetric algorithm, but it is not a public key algorithm. It can easily be figured out that some (if not all) public key cryptography algorithms can also be treated as 'asymmetric non-public key encryption algorithms' by setting both encryption and decryption keys to private [8].

5. COMMUTATIVE ENCRYPTION ALGORITHM

Not every encryption algorithm can be implemented in three-pass protocol. The three-pass protocol needs an encryption algorithm that is commutative in nature [10]. A commutative encryption algorithm allows a message to be super-encrypted using two different keys in any order. If $E(m, k)$ is an encryption function that takes a message m and a key k as the inputs and produces a ciphertext as the output, then a commutative encryption algorithm ensures that $E(E(m, k_A), k_B) = E(E(m, k_B), k_A)$.

In favor of the three-pass protocol, a commutative encryption also allows a message to be encrypted twice and decrypted twice using two different keys, and the final decryption will definitely bring back the original message. If $D(c, k)$ is a decryption function that takes ciphertext c and key k as the inputs and produces a message m as the output, then a commutative encryption algorithm ensures that $D(D(E(E(m, k_A), k_B), k_A), k_B) = m$.

A commutative algorithm usually belongs to the class of symmetric encryption algorithm. However, as mentioned earlier, a public key algorithm can also be treated as an asymmetric non-public key encryption algorithm. The difference between a symmetric encryption algorithm and an asymmetric non-public key encryption algorithm is that the former uses the same key for the encryption and decryption while the latter uses two different keys: one for the encryption and one for the decryption. Therefore, an asymmetric non-public key encryption algorithm in reality works just like a symmetric scheme with different keys being used in the encryption and the decryption. In favor of the three-pass protocol, one may use an asymmetric non-public key encryption algorithm that has a commutative property.

6. THE RSA AS A COMMUTATIVE ENCRYPTION ALGORITHM

Ambika, *et al*. [11] suggests that the RSA has a commutative property. It can be proven as follows [12].

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

The RSA encryption function is $E(m, e) = m^e \bmod n$. Let m be the message to be encrypted. Suppose that eA and eB are two different RSA encryption keys. We are now going to super-encrypt m using eA and eB. A super-encryption generally means combining two encryption algorithms in order to reinforce the security (according to [13], [14], and [15]) but in our study, a super-encryption simply means using two different keys with the same encryption algorithm (*i.e.,* RSA) without the intention to reinforce the security.

In the first case, the message m is encrypted with eA, then with eB, the result is $E(E(m, eA), eB) = E(m^{eA} \bmod n, eB) = m^{eA.eB} \bmod n$.

In the second case, the message m is encrypted with eB, then with eA, the result is $E(E(m, eB), eA) = E(m^{eB} \bmod n, eA) = m^{eB.eA} \bmod n$.

Thus, it can be concluded that $E(E(m, eA), eB) = E(E(m, eB), eA)$, so it is proven that the RSA is a commutative encryption algorithm, and, therefore, can be used as a candidate algorithm for the Shamir three-pass protocol.

7. USING THE RSA ENCRYPTION SCHEME IN THE SHAMIR THREE-PASS PROTOCOL

Since the Shamir three-pass protocol uses a symmetric algorithm, the RSA can be implemented in this protocol by setting both its encryption and decryption keys to private. The RSA algorithm is chosen since the algorithm has been widely used since 1978, its calculation is uncomplicated, and it is hard to break as long as: (1) very large prime numbers are being used for p and q; and (2) the exponent e is large enough.

Figure 1 illustrates how the RSA is used as an asymmetric non-public key algorithm to secure a message in the three-pass protocol. It can be figured out that the sender and the recipient have their own encryption and decryption keys. Unlike the typical RSA encryption scheme, the encryption keys are not published or transmitted to the other party. The encryption keys are set to private; only their respective owner has the information about their values. By default, the decryption keys are also set to private. By not transmitting any keys, the main objective of the three-pass protocol — *i.e.*, sending messages securely without the need of sending, distributing, transmitting, or publishing any keys — is preserved.

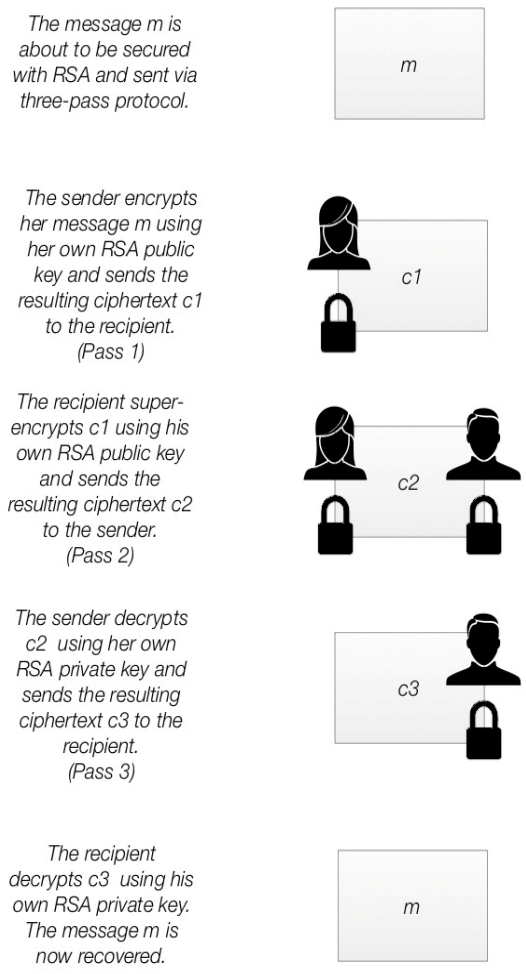

Figure 1 Using the RSA in the three-pass protocol

The whole scheme works as follows (the signs * and ^ denote multiplication and exponentiation, respectively):

1. The sender generates the private keys for her own use as follows.
   (a) Generate two very large random prime numbers, p_{sender} and q_{sender}. (This can be done with a primality test algorithm, such as Agrawal-Kayal-Saxena algorithm [16]).
   (b) Calculate $n_{sender} = p_{sender} * q_{sender}$.
   (c) Calculate $\Phi_{sender}(n_{sender}) = (p_{sender} - 1)(q_{sender} - 1)$.
   (d) Select an integer e_{sender}, so that $gcd(\Phi_{sender}(n_{sender}), e_{sender}) = 1$ and $1 < e_{sender} < \Phi_{sender}(n_{sender})$.
   (e) Calculate d_{sender}, so that $e_{sender} * d_{sender} \equiv 1 \pmod{\Phi_{sender}(n_{sender})}$.

(f) All the parameters (p_{sender}, q_{sender}, n_{sender}, $\Phi_{sender}(n_{sender})$, e_{sender}, d_{sender}) are set to private.

2. The recipient generates the private keys for his own use as follows.
   (a) Generate two very large random prime numbers, $p_{recipient}$ and $q_{recipient}$.
   (b) Calculate $n_{recipient} = p_{recipient} * q_{recipient}$.
   (c) Calculate $\Phi(n_{recipient}) = (p_{recipient} - 1)(q_{recipient} - 1)$.
   (d) Select an integer $e_{recipient}$, so that $gcd(\Phi(n_{recipient}), e_{recipient}) = 1$ and $1 < e_{recipient} < \Phi(n_{recipient})$.
   (e) Calculate $d_{recipient}$, so that $e_{recipient} * d_{recipient} \equiv 1 \pmod{\Phi(n_{recipient})}$.
   (f) All the parameters ($p_{recipient}$, $q_{recipient}$, $n_{recipient}$, $\Phi(n_{recipient})$, $e_{recipient}$, $d_{recipient}$) are set to private.
3. The sender encrypts her message, m by calculating the ciphertext, $c_1 = m \wedge e_{sender} \bmod n_{sender}$.
4. As the first pass, the sender sends c_1 to the recipient.
5. The recipient super-encrypts c_1 by calculating $c_2 = c_1 \wedge e_{recipient} \bmod n_{recipient}$.
6. As the second pass, the recipient sends c_2 to the sender.
7. The sender decrypts c_2 by calculating $c_3 = c_2 \wedge d_{sender} \bmod n_{sender}$.
8. As the third pass, the sender sends c_3 to the recipient.
9. The recipient decrypts c_3 by calculating $m = c_3 \wedge d_{recipient} \bmod n_{recipient}$. The m, which is the original message, has been recovered.

8. EXPERIMENTS AND DISCUSSIONS

Consider a scenario that a sender wants to send a character *'M'* to a recipient by using the RSA encryption scheme in three-pass protocol. She then looks into the ASCII table, and finds out that the corresponding number for the letter *'M'* is *77*. So, she lets the message $m = 77$.

A. Key Generation (Sender)

The sender generates two large prime numbers, p_{sender} and q_{sender}.

$$p_{sender} = 8738227201932281947$$
$$q_{sender} = 16588444808609115061$$

She then calculates n_{sender}.

$$n_{sender} = p_{sender} * q_{sender} = 8738227201932281947 * 16588444808609115061 = 144953599664340515826441122412016103767$$

Next, she calculates $\Phi(n_{sender})$.

$$\Phi(n_{sender}) = (p_{sender} - 1)(q_{sender} - 1) = (8738227201932281947 - 1) * (16588444808609115061 - 1) = 144953599664340515801114450401474706760$$

She selects an integer e_{sender}, so that $gcd(\Phi(n_{sender}), e_{sender}) = 1$ and $1 < e_{sender} < \Phi(n_{sender})$.

$$e_{sender} = 428648674228166083835088547848365250 49$$

She calculates d_{sender}, so that $e_{sender} * d_{sender} \equiv 1 \pmod{\Phi(n_{sender})}$.

$$d_{sender} = 823402472941652219594147997581599764 49$$

Since here RSA is treated as an asymmetric non-public key encryption algorithm, all of the above values are kept private by the sender.

B. Key Generation (Recipient)

The recipient generates two large prime numbers, $p_{recipient}$ and $q_{recipient}$.

$$p_{recipient} = 14646989093166241543$$
$$q_{recipient} = 15865985506086455321$$

He then calculates $n_{recipient}$.

$$n_{recipient} = p_{recipient} * q_{recipient} = 14646989093166241543 * 15865985506086455321 = 232388916659981982113466262076963600303$$

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Next, he calculates $\Phi(n_{recipient})$.

$$\Phi(n_{recipient}) = (p_{recipient} - 1)(q_{recipient} - 1) = (14646989093166241543 - 1) * (15865985506086455321 - 1) = 232388916659981982082953287477710903440$$

He selects an integer $e_{recipient}$, so that $gcd(\Phi(n_{recipient}), e_{recipient}) = 1$ and $1 < e_{recipient} < \Phi(n_{recipient})$.

$$e_{recipient} = 845202719331570061977092138862460556 57$$

He calculates $d_{recipient}$, so that $e_{recipient} * d_{recipient} \equiv 1 \pmod{\Phi(n_{recipient})}$.

$$d_{recipient} = 189992303824978400795252857905616448393$$

Because RSA is used as an asymmetric non-public key encryption algorithm, all of the above values are also kept private by the recipient.

C. Encryption Stage (Sender)

The sender encrypts her message, m into the ciphertext, $c_1 = m \text{ ^ } e_{sender} \bmod n_{sender}$.

$$c_1 = m \text{ ^ } e_{sender} \bmod n_{sender} = 77 \text{ ^ } 428648674228166083835088547848365250 49 \bmod 1449535996643405158264411224120161037 67 = 434076761036427120121828284883053347 14$$

In the first pass, the sender sends c_1 to the recipient.

D. Super-Encryption Stage (Recipient)

The recipient super-encrypts c_1 by calculating $c_2 = c_1 \text{ ^ } e_{recipient} \bmod n_{recipient}$.

$$c_2 = c_1 \text{ ^ } e_{recipient} \bmod n_{recipient} = 434076761036427120121828284883053347 14 \text{ ^ } 845202719331570061977092138862460556 57 \bmod 2323889166599819821134662620769636003 03 = 718490443386443492953194387829084440 96$$

In the second pass, the recipient sends c_2 to the sender.

E. Decryption Stage (Sender)

The sender decrypts c_2 by calculating $c_3 = c_2 \text{ ^ } d_{sender} \bmod n_{sender}$.

$$c_3 = c_2 \text{ ^ } d_{sender} \bmod n_{sender} = 718490443386443492953194387829084440 96 \text{ ^ } 823402472941652219594147997581599764 49 \bmod 1449535996643405158264411224120161037 67 = 434076761036427120121828284883053347 14$$

In the third pass, the sender sends c_3 to the recipient.

F. Final Decryption Stage (Recipient)

The recipient decrypts c_3 by calculating $m = c_3 \text{ ^ } d_{recipient} \bmod n_{recipient}$.

$$m = c_3 \text{ ^ } d_{recipient} \bmod n_{recipient} = 434076761036427120121828284883053347 14 \text{ ^ } 189992303824978400795252857905616448393 \bmod 2323889166599819821134662620769636003 03 = 77$$

Finally, the recipient has recovered the original message $m = 77$. He looks into the ASCII table, and finds the corresponding character for number *77* is *'M'*, which is the original character the sender wants him to read.

9. THE PYTHON CODES

The whole computation of our scheme is done in Python programming language. The development environment is Pythonista and the operating system is iOS 11.2.5 which runs in A10X Fushion chip with 64-bit architecture. The complete listing of the codes is provided as follows.

```
#title: The RSA Cryptosystem in
Shamir Three-Pass Protocol
#author: Mohammad Andri Budiman &
Dian Rachmawati
#version: 4.7
```

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

```
#date: Nov 12th 2017
#time: 07:00

import math, random, sys

sys.setrecursionlimit(10000)

class RSA(object):

    def __init__(self):
        maxi = pow(2, 64) # 2^72
        self.p, self.q =
self.getPrivateKeys(maxi)
        self.n = self.p * self.q
        self.totient = (self.p - 1)
* (self.q - 1)
        self.e =
self.compute_e(self.totient)
        self.d =
self.compute_d(self.e,
self.totient)

    def isCoprime(self, p, q):
        if self.gcd(p, q) == 1:
            return True
        return False

    def gcd(self, m, n):
        if n == 0:
            return m
        return self.gcd(n, m % n)

    def rnd(self, min, max):
        return random.randint(min,
max)

    def compute_e(self, totient):
        e = self.rnd(2, totient -
1)
        while not self.isCoprime(e,
totient):
            e = self.rnd(2, totient
- 1)
        return e

    def compute_d(self, e,
totient):
        d, _ = self.extended_gcd(e,
totient)
        return d % totient

    def doEncrypt(self, plaintext):
        print "\nEncryption"
        self.plaintext = plaintext
        print "plaintext = ",
self.plaintext
        length =
len(self.plaintext)
        plainchar =
self.text2char(self.plaintext)
        plainnum =
self.char2num(plainchar)
        ciphernum =
self.encrypt(plainnum, self.e,
self.n)

        for i in range(length):
            print
repr(plainchar[i]).ljust(7),
'=>\t',
repr(plainnum[i]).ljust(7),
'=>\t',
repr(ciphernum[i]).ljust(7)
        return  ciphernum

    def doDecrypt(self, ciphernum):
        print "\nDecryption"
        m = self.decrypt(ciphernum,
self.p, self.q, self.d)

        for i in
range(len(ciphernum)):
            print
repr(ciphernum[i]).ljust(7),
'=>\t', repr(m[i]).ljust(7),
'=>\t', repr(chr(m[i])).ljust(7)
        return m

    def doBoth(self, plaintext):
        ciphernum =
self.doEncrypt(plaintext)
        m =
self.doDecrypt(ciphernum)

    def doTPP(self, message):
        plaintext = message
        print "plaintext =",
plaintext
        print

        alice = RSA()
        bob = RSA()

        print "alice.p =", alice.p
        print "alice.q =", alice.q
        print "alice.n =", alice.n
        print "alice.totient =",
alice.totient
        print "alice.e =", alice.e
        print "alice.d =", alice.d
        print
```

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

```
        print "bob.p =", bob.p
        print "bob.q =", bob.q
        print "bob.n =", bob.n
        print "bob.totient =",
bob.totient
        print "bob.e =", bob.e
        print "bob.d =", bob.d
        print
        print

        length = len(plaintext)
        plainchar =
self.text2char(plaintext)
        plainnum =
self.char2num(plainchar)

        print
"********************************
***************"
        print "Process 1
(Encryption by Alice): plaintext
=> c1"
        print
"********************************
***************"
        print
        c1 = self.encrypt(plainnum,
bob.e, bob.n)

    self.printFirstEncrypt(plainnum
, c1)
        print
        print

        print
"********************************
***************"
        print "Process 2
(Encryption by Bob): c1 => c2"
        print
"********************************
***************"
        print
        c2 = self.encrypt(c1,
alice.e, alice.n)
        self.printEncrypt(c1, c2)
        print
        print

        print
"********************************
***************"
        print "Process 3
(Decryption by Alice): c2 => c3"
        print
"********************************
***************"
        print
        c3 = self.decrypt(c2,
alice.p, alice.q, alice.d)
        self.printDecrypt(c2, c3)
        print
        print

        print
"********************************
***************"
        print "Process 4
(Decryption by Bob): c3 =>
plaintext"
        print
"********************************
***************"
        print
        pt = self.decrypt(c3,
bob.p, bob.q, bob.d)
        self.printFinalDecrypt(c3,
pt)
        print
        print

    def printEncrypt(self,
plainnum, ciphernum):
        for i in
range(len(plainnum)):
            print
repr(plainnum[i]).ljust(7),
'=>\t',
repr(ciphernum[i]).ljust(7)

    def printDecrypt(self,
ciphernum, plainnum):
        for i in
range(len(ciphernum)):
            print
repr(ciphernum[i]).ljust(7),
'=>\t', repr(plainnum[i]).ljust(7)

    def printFirstEncrypt(self,
plainnum, ciphernum):
        plainchar =
self.num2char(plainnum)
        for i in
range(len(plainnum)):
            print
repr(plainchar[i]).ljust(7),
'=>\t',
repr(plainnum[i]).ljust(7),
'=>\t',
repr(ciphernum[i]).ljust(7)
```

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

```
    def printFinalDecrypt(self,
ciphernum, plainnum):
        for i in
range(len(ciphernum)):
            print
repr(ciphernum[i]).ljust(7),
'=>\t',
repr(plainnum[i]).ljust(7),
'=>\t',
repr(chr(plainnum[i])).ljust(7)

    def rnd(self, mini, maxi): #get
random number between mini and
maxi (and including mini and maxi)
        return random.randint(mini,
maxi)

    def AKS(self, n):
        z = random.randint(2, n -
2)
        return True if pow(1 + z,
n, n) == (1 + pow(z, n, n)) % n
else False

    def getRandomPrime(self, mini,
maxi):
        p = self.rnd(mini, maxi) //
2 * 2 + 1 # make sure it's odd
        while not self.AKS(p):
            p = self.rnd(mini,
maxi) // 2 * 2 + 1
        return p

    def getPrivateKeys(self, maxi):
        p = 1
        q = 1
        while(p * q < maxi or p ==
q):
            p =
self.getRandomPrime(1, maxi)
            q =
self.getRandomPrime(p + 1, maxi)
        return p, q

    def dec2bin(self, d):
        binary = []
        while d != 0:
            binary.append(d % 2)
            d = d // 2
        binary.reverse()
        return binary

    def bin2dec(self, b):
        b.reverse()
        d = 0
        k = 0
        for i in b:
            d = d + i * pow(2, k)
            k = k + 1
        return d

    def bin2str(self, b):
        s = ""
        for bin in b:
            s = s + str(bin)
        return s

    def extended_gcd(self, a, b):
        if b == 0:
            return 1, 0
        k = a / b
        r = a % b
        m, n = self.extended_gcd(b,
r)
        return n, m - k * n

    def text2char(self, text):
        char = {}
        for i in range(len(text)):
            char[i] = text[i]
        return char

    def char2num(self, char):
        num = {}
        for i in range(len(char)):
            num[i] = ord(char[i])
        return num

    def num2char(self, num):
        char = {}
        for i in range(len(num)):
            char[i] = chr(num[i])
        return char
```

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

```
    def encrypt(self, plainnum, e,
n):
        ciphernum = {}
        for i in
range(len(plainnum)):
            b =
self.dec2bin(plainnum[i])
            p = self.bin2dec(b)
            ciphernum[i] = pow(p,
e, n)
        return ciphernum

    def inverse(self, m, n):
        a, b = self.extended_gcd(m,
n)
        return a % n

    def decrypt(self, ciphernum, p,
q, d):
        plainnum = {}
        n = p * q
        for i in
range(len(ciphernum)):
            plainnum[i] =
pow(ciphernum[i], d, n)
            print "plainnum[", i,
"] = ", plainnum[i]
        print
        return plainnum

andri = RSA()
andri.doTPP("ANDRI")
```

```
plaintext = ANDRI

alice.p = 16926617606706357841
alice.q = 17717671050190411471
alice.n = 299900242747984544327973660111557194111
alice.totient = 299900242747984544293329371454660424800
alice.e = 158180809970004010408483149405134281031
alice.d = 121568915837723616609601407234103522871

bob.p = 6917062912748087083
bob.q = 9090026819936891993
bob.n = 628762873920709094334484193911294264l9
bob.totient = 628762873920709094174413296584444447344
bob.e = 40535715859441766014242702583412300041
bob.d = 520926563635076274613882685436502963 93

************************************************
Process 1 (Encryption by Alice): plaintext => c1
************************************************
'A'     =>   65      =>   222172138463700480577939372849149351 23L
'N'     =>   78      =>   294250096720696372467706822365308964 56L
'D'     =>   68      =>   554594155689010075962396177631595352 35L
'R'     =>   82      =>   616952512392080121098031822228567317 54L
'I'     =>   73      =>   216466281898111689941705766505565966 06L

************************************************
Process 2 (Encryption by Bob): c1 => c2
************************************************
222172138463700480577939372849149351 23L => 146696024461763708706009006734565758287L
294250096720696372467706822365308964 56L => 175067399971642338014978738060960038409L
554594155689010075962396177631595352 35L => 105544840793164683955577520030278657548L
616952512392080121098031822228567317 54L => 211800147689278895295838832287166588195L
216466281898111689941705766505565966 06L => 124239748849015308540791843584863893131L

************************************************
Process 3 (Decryption by Alice): c2 => c3
************************************************
146696024461763708706009006734565758287L => 222172138463700480577939372849149351 23L
175067399971642338014978738060960038409L => 294250096720696372467706822365308964 56L
105544840793164683955577520030278657548L => 554594155689010075962396177631595352 35L
211800147689278895295838832287166588195L => 616952512392080121098031822228567317 54L
124239748849015308540791843584863893131L => 216466281898111689941705766505565966 06L

************************************************
Process 4 (Decryption by Bob): c3 => plaintext
************************************************
222172138463700480577939372849149351 23L => 65L     => 'A'
294250096720696372467706822365308964 56L => 78L     => 'N'
554594155689010075962396177631595352 35L => 68L     => 'D'
616952512392080121098031822228567317 54L => 82L     => 'R'
216466281898111689941705766505565966 06L => 73L     => 'I'
```

Figure 2 Experimenting the scheme in Pythonista development environment

In Figure 2, we show the experimental result of using the RSA as an asymmetric non-public key encryption algorithm in the Shamir three-pass protocol in Pythonista development environment. In this scenario, Alice (the sender) would like to securely send her message (plaintext) "ANDRI" to Bob (the recipient). Both parties generate their own encryption and decryption keys. In Process 1, Alice encrypts the plaintext into c_1 using her public key and sends it to Bob. In Process 2, Bob super-encrypts c_1 into c_2 using his public key, and sends it to Alice. In Process 3, Alice decrypts c_2 with her private key, resulting in c_3; c_3 is then sent to Bob. In Process 4, Bob decrypts c_3 with his own private key, resulting in the plaintext "ANDRI" which Alice wants Bob to read in the first place. Since the plaintext is thoroughly recovered, one may deduce that the experiment has been finished successfully.

10. CONCLUSIONS

This work shows that the RSA can be used as an encryption algorithm for the Shamir three-pass protocol since the RSA has a commutative property that is essential to the three-pass protocol. The stages and the computations being carried out in each stage in order to use the RSA encryption in the Shamir three-pass protocol have been shown thoroughly. The Python source code for the whole system has also been provided.

It should be noted that the RSA encryption used in this work does not follow the conventional public key encryption where the encryption keys *(n, e)* are set to public, but here the these encryption keys (and also the decryption keys) are set to private and the one who does the encryption is the one who generates the keys him/herself. Thus, here we have treated the RSA as an asymmetric non-public key encryption algorithm. An asymmetric non-public key encryption algorithm is an encryption algorithm

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

that uses different keys for encryption and decryption but none of the keys are set to public.

In the final decryption stage, it can be seen that the original message can be recovered back by this scheme. Therefore, we conclude that the RSA asymmetric non-public key encryption algorithm works conveniently in the Shamir three-pass protocol.

Since the robustness of this scheme is beyond the scope of this work, more studies are needed to prove that this scheme is secure enough against all kinds of cryptanalyst attacks. If it is proven to be secure, then it may be expected that some other public key encryption algorithms can also be treated as asymmetric non-public key encryption algorithms to be used in the three-pass protocol.

ACKNOWLEDGEMENTS

We gratefully acknowledge that this research is funded by Lembaga Penelitian Universitas Sumatera Utara. The support is under the research grant TALENTA USU of Year 2017 Contract Number : 5338/UN5.1.R/PPM/2017.

REFERENCES

[1] Diffie, Whitfield, and Martin Hellman. "New directions in cryptography." *IEEE transactions on Information Theory* 22.6 (1976): 644-654.

[2] Rivest, Ronald L., Adi Shamir, and Leonard Adleman. "A method for obtaining digital signatures and public-key cryptosystems." *Communications of the ACM* 21.2 (1978): 120-126.

[3] Pomerance, Carl. "The quadratic sieve factoring algorithm." *Workshop on the Theory and Application of Cryptographic Techniques*. Springer, Berlin, Heidelberg (1984).

[4] Boneh, Dan, and Ramarathnam Venkatesan. "Breaking RSA may not be equivalent to factoring." *Advances in Cryptology—EUROCRYPT'98* (1998): 59-71.

[5] Katz, Jonathan, A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography*. CRC press. (1996).

[6] Yang, Li, Ling-An Wu, and Songhao Liu. "Quantum three-pass cryptography protocol." *Quantum Optics in Computing and Communications.* Vol. 4917. International Society for Optics and Photonics (2002).

[7] Smart, Nigel P. "Cryptography made simple". Springer International Publishing, (2016).

[8] Rachmawati, Dian, and Mohammad Andri Budiman. "An implementation of the H-Rabin algorithm in the Shamir three-pass protocol." *2nd International Conference on Automation, Cognitive Science, Optics, Micro Electro-Mechanical System, and Information Technology (ICACOMIT)* IEEE (2017): 28-33.

[9] Schneier, Bruce. *Applied cryptography: protocols, algorithms, and source code in C*. Indianapolis, IN. Wiley (2015).

[10] Carlsen, Ulf. "Cryptographic protocol flaws: know your enemy." *Computer Security Foundations Workshop VII*. Proceedings. IEEE (1994).

[11] Ambika, R., S. Ramachandran, and K. R. Kashwan. "Securing Distributed FPGA System using Commutative RSA Core." *Global Journal of Research In Engineering* (2013).

[12] Hsu, Jen-Chieh, Raylin Tso, Yu-Chi Chen, and Mu-En Wu. "Oblivious Transfer Protocols Based on Commutative Encryption." *International Conference on New Technologies, Mobility and Security (NTMS), 2018 9th IFIP*. IEEE (2018).

[13] Budiman, Mohammad Andri, Dian Rachmawati, and M. R. Parlindungan. "An implementation of super-encryption using RC4A and MDTM cipher algorithms for securing PDF Files on android." *Journal of Physics: Conference Series.* Vol. 978. No. 1. IOP Publishing (2018).

[14] Rachmawati, Dian, Mohammad Andri Budiman, and Indra Aulia. "Super-Encryption Implementation Using Monoalphabetic Algorithm and XOR Algorithm for Data Security." *Journal of Physics: Conference Series.* Vol. 979. No. 1. IOP Publishing (2018).

[15] Budiman, Mohammad Andri, Amalia, and N. I. Chayanie. "An Implementation of RC4+ Algorithm and Zig-zag Algorithm in a Super Encryption Scheme for Text Security." *Journal of Physics: Conference Series.* Vol. 978. No. 1. IOP Publishing (2018).

[16] Agrawal, Manindra, Neeraj Kayal, and Nitin Saxena. "PRIMES is in P." *Annals of mathematics* (2004): 781-793.