EPiC Series in Computing

Volume 75, 2021, Pages 1–10

CAINE 2020. The 33rd International Conference on Computer Applications in Industry and Engineering

Personalized Secure Communication

Emdad Ahmed[1]*, Scott Payne[2]†, and Craig Matherly[3]‡

[1] Department of Computer Science and Software Engineering
Miami University, Oxford, OH USA
ahmede@miamiOH.edu
[2] paynest@miamiOH.edu
[3] mathercg@miamiOH.edu

Abstract

We propose multiple factors of authentication, more specifically a two factor. We have developed an algorithm ESC which utilizes a combination of public key and symmetric key encryption that provides a solid secured communication framework. As a proof of concept, we report here Java based implementation of the approach.

Keyphrases: RSA, public key, symmetric key, plaintext, ciphertext, digital signature.

1 Introduction

Computer and Network security was not at all well known, even about 15 years ago. Today, it is something everyone is aware of the need. Cybersecurity is one of the fastest growing fields, U.S. has an urgent need to fill 2.5 million Cybersecurity jobs Test Out. Security includes topic of threats, countermeasures, risks, stories, events and paranoia. With some mathematics, algorithms, designs and software issues mixed in yet, not enough people understand the issues and implications.

1.1 Goals of Security

These are the few goals of security:

- Integrity: Guarantee that the data is what we expect
- Authentication: Guarantee that only authorized persons can access to the resources
- Confidentiality: The information must just be accessible to the authorized people
- Reliability: Computers should work without having unexpected problems

*Corresponding author
†Did numerous tests on Java coding and provided a lot of suggestions
‡Masterminded the project

Table 1: Four Layered Model of Security

Types of attacks	**Layer**	**attack prevention methods**
Logic bugs, design flaws, code injection	Application	Sandboxing, software restrictions
Insecure defaults, platform vulnerabilities	Operating System	Patches, reconfiguration, hardening
Sniffing, spoofing, masquerading	Network	Encryption, authentication, filtering
Console access, hardware-based attacks	Physical	Guards, vaults, device data encryption

1.2 Types of Security

Due to its importance security is highly diversified:

- Computer security: Deals with programs and users on a computer. Viruses, worms etc. are part of computer security
- Network security: Deals with various OSI/ISO 7 layers (now-a-days TCP/IP 5 layers) in a network including protocols
- Cryptography: A common theme involved in security. Deals with encrypting/decrypting information to avoid malicious users from accessing the data. Cryptography is very important in networking which is the main focus of this paper. Note that cryptography can be inserted at almost any layer in the OSI model.
- Physical security: Securing data centers and physical facilities

1.3 Vulnerabilities

Common attacks include:

- Software/OS exploits: Buffer overflows, Viruses, Worms, Trojans, Rootkits
- Network Attacks: Packet sniffing, spoofing, man-in-the-middle attack, DNS hacking
- Email and HTTP/web attacks: Phishing, SQL Injection and Cross Site Scripting (XSS). Cognitive Cyber weapons and Social Engineering. Not all hackers are evil wrongdoers trying to steal your info, Ethical Hackers, Consultants, Penetration testers, Researchers.

The above discussions can be summarized in the following table 1 which shows that security is implemented in many layers. At least scientifically, we know how to do cryptography. Unsecured message travels the network as *plaintext* P. To be secured, the message need to be encrypted as *ciphertext* C. A **cipher** is a character-by-character or bit-for-bit transformation. The whole essence of cryptography is that of mapping from plaintext to ciphertext and vice versa utilizing some underlying algorithms that use some key K. Mathematically, we define encryption function $\psi : P \mapsto C$ and decryption function $\mu : C \mapsto P$. Note that encryption followed by decryption will give the same result as decryption followed by encryption, i.e. $C = E_K(P)$, similarly, $P = D_K(C)$. Then it follows that: $D_K(E_K(P)) = P$. All of the above suggest that *security* is inversely proportional to *convenience* i.e., $Security \propto \frac{1}{Convenience}$. The more secure a system is, the more constraint the users will be.

2 Data Encryption and Decryption

Historically, four groups of people have used and contributed to the art of cryptography: the military, the diplomatic corps, diarists, and lovers (the famous Alice-Bob ..). Cryptography

has a long history dating back at least as far as Julius Caesar. Symmetric key cryptography algorithms attributed to Julius Caesar, known as Caesar cipher. **Monoalphabetic cipher** substitute one plaintext character with another letter of the *alphabet*. Later, to improve the scheme, **polyalphabetic encryption** was invented that uses multiple Monoalphabetic ciphers.

The idea that the *cryptanalyst* knows the algorithms and that the secrecy lies exclusively in the keys is called **Kerckhoff's principle**, stated as follows: *All algorithms must be public; only the keys are secret.*

The seminal RSA work (*although too slow for actually encrypting large volumes of data but is widely used for key distribution*) has lead us to a trapdoor mechanism based on composite *residuosity* classes, i.e., factoring a hard-to-factor num $n = pq$ where p and q are two large prime numbers. In fact RSA typically uses 256 bit, 512 bit key, but those have recently been cracked. The new standard AES uses 1024 bit key. Below we discuss two such trials: Homomorphic encryption and order preserving encryption.

2.1 Homomorphic Encryption

The *homomorphic encryption*, a long time dream of security experts, reflects the concept of homomorphism, a structure-preserving map $f(.)$ between two algebraic structures of the same type. When $f(.)$ is a one-to-one mapping, call $f^{-1} : A^{'} \mapsto A$ the inverse of $f(.)$. Then $a = f^{-1}(a')$, $b = f^{-1}(b')$, $c = f^{-1}(c')$. In this case we can carry out the composition operation $\diamond$ in the target domain and apply the inverse mapping to get the same result produced by the $\square$ composition operation in the original domain, $f^{-1}(a') \diamond f^{-1}(b') = f(a \square b)$ Unfortunately, the homomorphic encryption is not a practical solution at this time. Existing algorithms for homomorphic encryption increase the processing time with encrypted data by many orders of magnitude compared with processing of plaintext data. In this line, this is our humble tries to go above and beyond with traditional encryption paradigm.

2.2 Order Preserving Encryption

Let a order-preserving function $f : \{1 \dots M\} \mapsto \{1 \dots N\}$ with $N \gg M$ be uniquely represented by a combination of M out of N ordered items. One can show that a order preserving $f(x)$ for a given point $x \in \{1 \dots M\}$ has a Negative Hypergeometric Distribution (NHG) over a random choice of f. To encrypt plaintext x the OPE encryption algorithm performs a binary search down to x. Given the search key K the algorithm first assigns $\mathit{Encrypt}(K, M/2)$, then $\mathit{Encrypt}(K, M/4)$ if the index $m < M/2$ and $\mathit{Encrypt}(K, 3M/4)$ otherwise, and so on, until $\mathit{Encrypt}(K, x)$ is assigned. Searchable Symmetric Encryption (SSE) is used when an encrypted database Ξ is outsourced to a cloud or to a different organization. Again, this is in line with our approach to combine public key and symmetric key encryption.

3 Related Work

In [12], we reviewed the existing security mechanism that are in place in the virtual cloud platform. Any Cloud Service platform [8] requires some sort of security mechanism for login, for example Duo security where a valid user has to go through two factors authentication. Typically the one time security code is sent to any smart phone [14].

Traditional SMTP ($FROM$ address is not required, only TO is mandatory, that is a major source of email phishing) based system employ Base64 encoding which increases the required bandwidth by 33%. We envision that our work will fit well at least for person-to-person secured

email communication (example application using Pretty Good Privacy (PGP) for encrypting email.

There are many financial institutions and secured sites require that user must be authenticated using multiple factors. Our proposed work is in line with those kind of applications requirement. Our work in systems programming and systems administration leads fairly naturally to security. UNIX operating system provides *crypt* command to encode data, the same command can be used to decode as well. However, user may NOT like to use it and even this command is not available on all systems. In particular, for reasons of national security, this command is not supposed to be available on systems that are shipped outside of the United States. Due to the ease of breaking it, it is considered to be obsolete.

We can think of ontology [1, 3, 4, 2] when pair of communicating parties use public and private key. More relevant recent works reported in: [11], [5], [6], [15]. Lee et al [10] proposed a holistic approach to security mechanism. [18] reports implementation of a secured IoT communication, whereas our approach encompasses broader audience.

4 Materials and Method

We implemented the Rivest, Shamir and Adleman (RSA) algorithm utilizing extended Euclid's algorithm [7]. A running example of RSA can be found in [13] as well as in [17]. For readers' conveniences we mention here one such example as well: $p = 3$ and $q = 11$, giving $n = 33$ and $z = 20$, a suitable value for d is $d = 7$ since 7 and 20 have no common factors, e can be found by solving the equation $7e = 1(mod 20)$ which yields $e = 3$.

The basis of our approach is the *scientific method*: we develop hypothesis about performance, create mathematical models, and run experiments to test them, repeating the process as necessary. [16]. There are infinitely many primes number, example: first few primes to be $2, 3, 5$, then multiplying all together and just add 1, we will get 31 which is another prime (aka Euclid Prime), we just need to pick some of them suitably as the public and private key. When two parties communicate securely, they need to exchange keys (example Diffee-Hellman key exchange). There are certification authority (CA) like Kerberos and Key Distribution Center (KDC). There are company which can certify digitally signed Web sites (example verisign).

We employ public key as well as symmetric key encryption. First, one party will send a short message to the other party to be the encrypted key. The other party will decrpyt to know the actual key value. As for symmetric encryption, one party can use *Ceasar cipher* simply shifting (i.e., adding) the key value to the data. The other party, upon receiving the message, simply can decrypt by subtracting the key. For symmetric key encryption, it is important that inverse function do exist (for our case add and sub is the example of inverse function). This is the high level understanding of the whole scenario.

As a running example, lets consider a plaintext message simply to be **abc** and suppose the Ceasar cipher key to be 7.Then sender will send **hij** as ciphertext (shifting and adding each character by 7) whereas the receiver will use the same key value 7 and this time she will subtract the key value from the ciphertext to get back the plaintext. Here is our contribution: in our approach, even before sending and receiving ciphertext, both the communicating parties need to handshake and agree on public and private key, again utilizing the famous RSA algorithm. Suppose the value 3 to be public key and 7 be the corresponding private key. So the decision to encrypt and decrypt will depend on public key and private key. We also note that we have to use modulus operation as the character set wraps around. For the above example, both the party will know that public key is 3 and the private key is 7.

Sender will encode its message so that only a receiver with the *key computation capability*

will be able to decode the message. Note that cryptography is a field of study unto itself, with large and small complexities and subtleties as discussed forward in run time analysis subsection.

The details of the Java code is appended at the end of this paper. Due to space brevity, we attach the client side code. Actually this type of application is NOT client-server model, rather it is P2P where no one is client and server, every one may be deemed client and server. We assume that the two parties already have some informal prior knowledge how they should proceed.

5 Results and Discussions

We utilized the modular arithmetic, specifically modular multiplicative inverse. Inverse do exist, sometimes we observed that decryption key turned out to be negative, then we simply added the ϕ value to it. We also used $MODULAR\_EXPONENTIATION$ taking advantage of "repeated squaring" and shifting, of course using the loop invariants by properly "initializing, maintenance and termination". RSA enables "digital signature" to the end of an electronic message. It is the perfect tool for electronically signed business contracts, electronic check, electronic purchase orders, and other electronic communications that parties wish to authenticate.

For our case, we use RSA during the first phase handshake of the communicating parties. We use the key as "nonce", i.e., once in a life time for the entire duration of the subsequent communication session. RSA relies on two factors: (a) Ease of finding large prime, and (b) Difficulty of factoring the product of two large primes.

5.1 Run Time Analysis

Let β be the number of bits required to represent the keys. We know $\lg e = \mathcal{O}(1)$ and $\lg d \leq \beta$, and $\lg n \leq \beta$. Then public key requires $\mathcal{O}(1)$ modular multiplication and $\mathcal{O}(\beta^2)$ bit operation, secret keys require $\mathcal{O}(\beta)$ modular multiplication using $\mathcal{O}(\beta^3)$ bit operations.

Again, we note here is passing that we are using RSA in "hybrid" or "key-management" mode. In this mode, after successfully hand shaking mechanism, we employ symmetric key encryption, i.e., encryption and decryption use the same key, we can just simply $+(add)$ during encryption and $-(sub)$ during decryption

5.2 Overall Algorithm

Now that we have discussed in details how our personalized secure communication will take place, we present below the overall algorithm ESC. By *Sender* we abstract everything related to the sender side of the communication, i.e., IP address, port, domain name, user name etc. Similarly, by *Receiver* we abstract everything related to the receiver side of the communication, i.e., IP address, port, domain name, user name etc. Also we use two sub-algorithm A_1 to be public key algorithm and A_2 to be any symmetric key algorithm. Then Personalized Secure Communication ESC. is a 6-tuple(S, R, A1, A2, Key, M).

1. Sender sends the key using A_1

2. Receiver computes the secret key using A_1

3. Sender sends the encrypted message using A_2

4. Receiver decrypts the ciphertext using secret key of A_2

6 Conclusion and Future Works

In this paper, we have proposed a personalized secure communication system that combines RSA and symmetric encryption into an unified view and a simple prototype Java implemented system that uses users chosen public key, private key and symmetric encryption key. We hope to expand our system in the future to include more factors into considerations and a more secured communication mechanism. More research should be done to discover optimal key size by their own individual preference. Further work can be done such as cross validating our recommendations with a system like DES, AES etc. From our research, it is expected that our system will be highly scalable allowing for a vast selection of domains and datasets to be applied in the future. As for the client server simulating, we used TCP. Future works might use UDP to see how much performance improvement in terms of response time gain, of course at the cost of probability of dropping call.

We have been able to discover a multitude of new research directions as a result of our current work. We have shown a prototype implementation using Java.net framework, whereas C++ is believed to be twice as faster than Java when employing Boost C++ socket library. We wish to run more tests using both Java and C++ [9].

7 Acknowledgment

The authors thankfully acknowledge the generous support of Miami University.

References

[1] Emdad Ahmed. Use of ontologies in software engineering. In *17th International Conference on Software Engineering and Data Engineering (SEDE)*, pages 145–150. ISCA, Los Angeles, USA, June 30 - July 2 2008.

[2] Emdad Ahmed. Achieving classification and clustering in one shot - lesson learned from labeling anonymous datasets. In *4th International Conference on Semantic Computing (ICSC)*, pages 228–231. IEEE, Carnegie Mellon University, USA, September 22-24 2010.

[3] Emdad Ahmed. Discriminative power of l v pattern for labeling big anonymous datasets (best paper award). In *29th Pennsylvania Association of Computer and Information Science Educators Conference*, pages 19–28. PACISE, California University of Pennsylvania, USA, April 4-5 2014.

[4] Emdad Ahmed. Theoretical proof of discriminative power of l v pattern for labeling big anonymous dataset. In *CATA*, pages 131–136. ISCA, March 9-11 2015.

[5] Yosef Ashibani, Dylan Kauling, and Hassan Mahasneh. A context aware authentication framework for smart homes. In *30th IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 312–318. University of Windsor, CANADA, April 30 - May 3 2017.

[6] Long Chen and Shervin Erfani. A note on security management of the internet of things. In *30th IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 933–939. University of Windsor, CANADA, April 30 - May 3 2017.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald Rivest, and C. Stein. *Introduction to Algorithms, 3rd edition*. MIT Press, USA, 2009.

[8] Narayan Debnath, Mario Peralta, Carlos Salgado, Lorena Baigorria, Germán Montejano, Daniel Riesco, and Emdad Ahmed. “a strategy to evaluate and measure the degree of openness of cloud

services environment. In *32ND International Conference on Computer Applications industry and Engineering (CAINE2019)*, pages 1–10. ISCA, September 30 - October 2 2019.

[9] Paul Deitel and Harvey Deitel. *C++ How to Program Tenth Edition*. Pearson, 2017.

[10] Maximilian Etschmaier and Gordon Lee. A holistic approach to the design of a secure system. In *CATA*, pages 107–114. ISCA, April 4-6 2016.

[11] Ibraheem Al Hejri. A comparative analysis of aes common modes of operation. In *30th IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 303–309. University of Windsor, CANADA, April 30 - May 3 2017.

[12] Jeremy St Clair Jeremy Black, Bryan Freed and Emdad Ahmed. A survey of security in the virtual computing cloud. In *29th Pennsylvania Association of Computer and Information Science Educators Conference*, pages 62–68. PACISE, California University of Pennsylvania, USA, April 4-5 2014.

[13] James F. Kurose and Keith W. Ross. *Computer Networking A Top Down Approacj Seventh Edition*. Pearson, 2017.

[14] Phillip Osial, Kalle Kauranen, and Emdad Ahmed. Smartphone recommendation system using web data integration techniques. In *30th IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 1182–1186. University of Windsor, CANADA, April 30 - May 3 2017.

[15] Shawon Rahman and Yvonne May. Wireless security vulnerabilities and countermeasures for an airport. In *CATA*, pages 431–436. ISCA, March 9-11 2015.

[16] Robert Sedgewick and Kevin Wayne. *Algorithms Fourth Edition*. Addison-Wesley, 2011.

[17] Andrew S. Tanenbaum and David J. Wetherall. *Computer Networks Fifth Edition*. Prentice Hall, 2011.

[18] Priyanka Thota and Ju-Yeon Jo. Implementation of secure iot communication. In *CATA*, pages 327–334. ISCA, April 4-6 2016.

```
/*
* Author Dr. Emdad Ahmed, Craig Matherly and Scott Payne
*/
public class encryptRSA
{
  int  e= 13, M, n = 437;
  String line = "";
   public encryptRSA()
   {
      // int M is the message char integer
     this.M = 0;
   } // end constructor
   int Modular_Exponentiation(int message)
   {
     // compute M^e mod n using repeated squaring and shifting
     int i, k, c = 0, d = 1;
     int[] bb = new int[32];

     for (i = 0; i < 32; i++)
     bb[i] = 0; // initialize binary representation

     bb = decimal_to_binary(e); // convert the decimal exponent into binary

     k = bb.length; // assuming 32 bit integer, length is 32
```

```
     for (i = k - 1; i >= 0; i--)
     {
       c = c * 2; // actually c is not required, just for book keeping
       d = (d * d) % this.n; // squaring

       if (bb[i] == 1)
       {
         c++;
         d = (d * message) % this.n; // shifting if bit value is 1
       }
     }
     return d;
    } // method Modular_Exponentiation

      int[] decimal_to_binary(int num)
      {
        int[] local_array = new int[32]; // convert a decimal number to binary
        int i;
        for (i = 0; i < 32; i++)
           local_array[i] = 0; // initialize binary array
        i = 0;
        while (num >= 1)
        {
          local_array[i] = num % 2; // store the mod value
          num = num / 2; // divide the number by 2
          i++; // next upper significant bit position
         }
         return local_array;
       } // method decimal_to_binary
} // class crypt

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
public class decryptRSA
{
   static int[] array = new int[3]; // represent d,x,y for Extended_Euclid
   static int[] temparray = new int[3];
   int n, phi, p = 23, q = 19, e = 13;

    public decryptRSA()
    {
        this.n = this.p * this.q; // modulus
        this.phi = (this.p - 1) * (this.q - 1); // phi relatively prime to e
         for (int i = 0; i < 3; i++)
         { // initialize d,x,y value
            array[i] = 0;
            temparray[i] = 0;
         }
    } // end constructor
   public int Modular_Exponentiation(int message)
   {
     Extended_Euclid(phi, e); // will return Multiplicative inverse of e modulo phi
```

```
  if (temparray[2] < 0)
  temparray[2] += phi; // adjust if returned quotient is negative
  // compute a^b mod n using repeated squaring and shifting
  int i, k, c = 0, d = 1;
  int[] bb = new int[32];

  for (i = 0; i < 32; i++)
    bb[i] = 0; // initialize binary representation

  bb = decimal_to_binary(temparray[2]); // convert the decimal exponent into binary
  k = bb.length;

  for (i = k - 1; i >= 0; i--)
  {
     c = c * 2; // actually c is not required, just for book keeping
     d = (d * d) % n; // squaring
     if (bb[i] == 1)
     {
       c++;
       d = (d * message) % n; // shifting if bit value is 1
      }
   }
   return d;
} // method Modular_Exponentiation
private int[] decimal_to_binary(int num)
{
  int[] local_array = new int[32]; // convert a decimal number to binary
  int i;

  for (i = 0; i < 32; i++)
  local_array[i] = 0; // initialize binary array

  i = 0;
  while (num >= 1)
  {
    local_array[i] = num % 2; // store the mod value
    num = num / 2; // divide the number by 2
    i++; // next upper significant bit position
   }
   return local_array;
 } // method decimal_to_binary
private int[] Extended_Euclid(int a, int b)
{
  if (b == 0)
  {   // compute GCD, d of a and b, as well as
     temparray[0] = a; // some x and y such that d = ax + by
     temparray[1] = 1;
     temparray[2] = 0;
     return temparray;
  }
   Extended_Euclid(b, a % b);
   array[0] = temparray[0];
```

```
      array[1] = temparray[2];
      array[2] = temparray[1] - (a / b) * temparray[2];
      temparray[0] = array[0];
      temparray[1] = array[1];
      temparray[2] = array[2];

      return array;
   } // method Extended_Euclid
/************************************************************/
} // class decrypt

import java.io.*;
import java.net.*;
class TCPClientRSAEncrypt
{
   public static void main(String argv[]) throws Exception
   {
     String sentence;
     String modifiedSentence;
     BufferedReader inFromUser = new BufferedReader(new InputStreamReader(System.in));
     Socket clientSocket = new Socket("10.33.10.119", 6789);
     DataOutputStream outToServer = new DataOutputStream(clientSocket.getOutputStream());
     BufferedReader inFromServer = new BufferedReader(new InputStreamReader
     (clientSocket.getInputStream()));
    System.out.println("Enter sentence to transmit using RSA ");
    sentence = inFromUser.readLine();

    //sends the lower case message to the encrypt method.
    sentence = encryptRSA(sentence);
    System.out.println("After encryption and before sending to server: " + sentence);

    outToServer.writeBytes(sentence + '\n');
    modifiedSentence = inFromServer.readLine();
    System.out.println("FROM SERVER: " + modifiedSentence);
    clientSocket.close();
  }
  public static String encryptRSA(String input)
  {
     encryptRSA encryptTest = new encryptRSA();
     String temp = "";
     for (int i = 0; i < input.length(); i++)
     {
        temp += encryptTest.Modular_Exponentiation(input.charAt(i))+" ";
     }
     return temp;
  }
}
```