CS333 Final Report

A Method for Obtaining Digital Signatures and Public-Key Cryptosystems

Arman Hasanzade
dept. Computer Science
Özyeğin University
Istanbul, Turkey
arman.hasanzade@ozu.edu.tr

Ahmet Burak Yıldırım
dept. Computer Science
Özyeğin University
Istanbul, Turkey
burak.yildirim@ozu.edu.tr

Güneş Büyükgönenç
dept. Computer Science
Özyeğin University
Istanbul, Turkey
gunes.buyukgonenc@ozu.edu.tr

Cem Denizsel
dept. Computer Science
Özyeğin University
Istanbul, Turkey
cem.denizsel@ozu.edu.tr

***Abstract*—This paper is about a mathematical way to encrypt and decrypt any message, without any data loss. In addition to this capability it also ensures that the message is sent by the intended sender using signatures and confirms that it has not been hijacked or changed. It is also shown how to develop a basic client/server application by using the methods explained in the paper. By applying this method, it is proven that the encrypted message can be sent from one end to another, and only the receiver can decrypt and read the message. It is also shown that if the message is caught by an unintended person, they will not be able to decrypt and read that message because of the security procedures which depends on the difficulty of factoring prime numbers.**

***Index Terms*—encryption, decryption, ciphertext, plaintext, scrambled-text, difficulty of computation, basic tcp server/client**

I. Introduction

The era of paper mail has been changed with electronic mails and properties of the current paper mail system needed to be preserved. Those properties are the privacy and the signature of the messages. In this paper, a method of implementing a public-key cryptosystem (asymmetric key encryption) is proposed to ensure those capabilities in an electronic mail system. The security of the cryptosystems depend on the difficulty of factoring large prime numbers. It allows establishing secure communications without the need of a courier to carry keys for encrypting and decrypting the message and it allows signing digitized documents. This method provides an implementation of a "public-key cryptosystem" which was invented by Diffie and Hellman.

To clarify some terms used in the paper such as "Public" or "Private" keys, it must be explained what the message being sent is. The message is generally a "String", which is also called a plain text. Anyone who has access to this plain text, can read it without any problem. After applying the encryption algorithm, the plain text becomes an encrypted message or a cipher text. A ciphertext is not readable and the length of the ciphertext is also different from the plain text. The encryption and decryption process is done by using the combination of public and private keys. The details of how these keys are created and used will be discussed in the following sections.

The process of sending a message from a sender "Bob" to a receiver "Alice" is as follows. There are 2 essential parts of this process for Alice. Understanding if the message is meant to be sent to them, and understanding if the message is actually sent by Bob. First, Bob needs to encrypt the message with his own private key, which is called "signing". Doing this will help Alice to make sure that the message is sent by Bob.

Bob also needs to encrypt the message using Alice's public key. Since the only person who can decrypt a message that is encrypted by Alice's public key is themselves, this will ensure that potential attackers will not be able to decrypt any message and will help Alice to understand that the message is intended for them. So if Alice can first decrypt the message using their private key and then decrypt it with Bob's public key, it means that they received the correct message and it is certainly sent by Bob. As other similar algorithms, Diffie and Hellman's algorithm [1] [2], Symmetric or single key encryption [3] can be mentioned. The method described in the paper is an improved version of Diffie Hellman algorithm. This algorithm also has a lot of applications such as its usage in networking protocols like HTTPS [4] [5] and SSL [6], chatting applications (which we are implementing a smaller scale of it), and finally digital signature.

Another interesting use of asymmetric key encryption is Discrete Cosine Transform (DCT) on JPEG images [12]. This is a fairly complicated way to hide crucial information in JPEG image files. This is done by first acquiring the cosine waves from the image and then their coefficients are calculated. Using these values the message is created and then it is encrypted using asymmetric key encryption. After these steps the message is put into the JPEG image. This is done by editing the least significant bits of each pixel. The receiver of the message must first decrypt the message using the methods described in this paper, and then extract the DCT. This procedure very complicated and cannot be explained in few sentences and is above the scope of this project.

In Section II, we discuss how the Public-Key cryptosystems works in general, privacy of the given methods and the signature idea that identifies the sender of the message. In Section III, we discuss how the necessary keys are found and applied on the message sent in order to make the connection secure. In Section IV, we explained the algorithms in detail which are mentioned in Section III. In Section V, we explained how the implementations of the algorithms are done in order to send a message from client to server. In Section VI, the test results of the implementation and its efficiency is discussed. In Section VII, the paper is summarized.

II. Background

A. Public-Key Cryptosystems

In the Public-Key Cryptosystems, there are two major procedures named encryption (E) and decryption (D). Encryption procedures of each user are stored in a public directory. On the other hand, the decryption procedures are kept secretly by their owners. These procedures have the following four properties:

1) Deciphering an enciphered message results in getting the original form of the message. Formally:
$$D(E(M)) = M$$
2) Both E and D can be computed easily.
3) Since E is known by everyone, users should guarantee that calculating D is almost impossible which means that only the target person can decrypt the message and the owner of the message can encrypt the message in an efficient way.
4) Enciphering an deciphered message results in getting the original form of the message. Formally:
$$E(D(M)) = M$$

Public keys are used to encipher a message M to obtain the enciphered form of the message which is called the ciphertext C. This encryption procedure can be applied by everyone. However, applying this procedure becomes impossible if the keys of these procedures are secure. Knowing the algorithm of the E yields to try all possible messages M till an $E(M) = C$ is found in order to decrypt C. If the 3rd property is satisfied, computing $D(C)$ in this approach becomes impractical.

If E satisfies property 1 and 3, then it is a "trap-door one-way function". If it also satisfies property 4, then it is called a "trap-door one-way permutation" which is a concept introduced by *Diffie and Hellman* but they did not present any examples. The "one-way" word means that the function can be easily computed in one direction but it is very hard to do the computation in the opposite way. If the property 3 is also satisfied, then every message becomes a ciphertext for a specific message and every ciphertext is a specific permissible message which creates a one-to-one relationship. Property 4 is only used for "signatures".

In the scenario of this paper, A as *Alice* and B as *Bob* are the users of the public-key cryptosystem. Their encryption E and decryption D procedures are named as follows; E_A, D_A, E_B, D_B. Encryption procedure of any person X E_x is done by X's public-key. and the decryption procedure D_x is done by the private-key of person X.

B. Privacy

Encryption is used to convert a communication to a private session. The owner of the message encrypts it before sending it. Only the receiver knows how to decrypt the encrypted message to obtain the original form of the message. If there is an attacker wanting to obtain the message sent to the receiver, they see meaningless messages and they can't decrypt it.

In the current years, many sensitive and private data are kept in cloud systems and being transferred between phones which makes encryption important for keeping these data secret. In the classical encryption systems, distributing the key of a private session was requiring another private session. In this case, a private courier is used to transfer the key from the sender to receiver. This technique was not applicable for rapid mail systems. A public-key cryptosystem does not require any private courier, since all the public keys are visible for everyone.

If *Bob* wants to send a message M to *Alice*, he gets *Alice*'s Public-key to compute $E_A(M) = C$. Then he sends the ciphertext to *Alice*. After receiving the ciphertext, *Alice* decrypts it by computing $D_A(E_A(M)) = D_A(C) = M$. In this case, only *Alice* knows D_A procedure due to property 3, so that only she can decipher the ciphertext. She can use $E_B(M)$ to respond to the message which is public information and only *Bob* can decipher it. By this procedure, *Alice* and *Bob* can communicate with each other in private without requiring another private session.

C. Signatures

Signature is a unique stamp of the sender *Bob* that informs the receiver *Alice* about who wrote the message received. Signatures are message-dependent and signer-dependent.

When *Bob* sends a message to *Alice*, he firstly computes the signature *S*:
$$S = D_B(M) \quad \text{With his private key}$$
In order to make the message only readable if
$$E_B(S) = E_B(D_B(M)) = M$$
procedure is used which is available for everyone. In this example, finding M by using S is easy, since everybody can access E_B procedure. However, nobody can send any message that can be transformed to its original form by using E_B due to the 4 fact that they need to know D_B to replace S with another one but procedure 3 makes it impossible. So that, the S is message-dependent.

Furthermore, S is signer-dependent because it can only transform to the original message by using E_B which is a procedure identical to *Bob*. This proves that the message was sent by *Bob* because nobody can create any S transformable by E_B without knowing D_B which is the procedure that only *Bob* knows.

Bob can also add any additional text to his *S* which informs the receiver that the sender is *Bob* so that the receiver uses E_B to get the message M.

Another concern is that an attacker can also read the S by using E_B which is a public information. To prevent this, *Bob* computes $C = E_A(S)$ procedure which is again a public information. If the attacker obtains C, he can't get the signature S because E_A can only be reversed by using D_A which is only known by *Alice*.

In general, *Bob* computes $C = E_A(D_B(M))$ and sends C to *Alice*. Then *Alice* computes $M = E_B(D_A(C))$. As a result, the message M is only computable by *Alice* due to property 3 and *Alice* gets the message without a need for another private session to send the keys.

III. The Encryption and Decryption Methods

In the previous parts, it is explained how decryption and encryption procedures are being used to transfer a message from sender to receiver in private. Now, we are going to explain what these procedures are in detail.

Encryption procedure of a message M requires a public encryption key (e, n) to apply the computation (e and n are positive integers). In order to be able to make this computation, M should be an integer between 0 and $n-1$. Thus it is necessary to convert the message into many parts satisfying this requirement. In the implementation, messages are converted to integer form of its *ASCII* characters less than n which is a product of two large prime numbers. The encryption and decryption algorithms are explained below:

$$\text{Encryption: } C \equiv E(M) \equiv M^e \pmod{n},$$

$$\text{Decryption: } D(C) \equiv C^d \pmod{n}$$

Note that in the equations above C is the ciphertext and M is the message.

These calculations are not applicable for any random keys n, e and d. Each user calculates their 3 keys. Then they expose key n and e as public keys which are the identical keys for the user who created them. Key d is kept secretly by each user which is the private key of the related user. Finding these keys is the main concern of this paper. Firstly, it is needed to calculate n to find e and d. Let p and q two very large prime numbers chosen randomly (finding these large prime numbers will be explained in the following parts). After finding p and q, n is calculated by taking their product:

$$n = p \cdot q$$

After n is found, calculating d is done by finding a large integer which is relatively prime to $(p-1)\cdot(q-1)$. Checking if d is relatively prime to the result of this equation is done by confirming the equation given below:

$$gcd(d, (p-1)\cdot(q-1)) = 1$$

This equation is satisfied when the random large number d is a prime number greater than max(p,q). In the implementation, calculations are done in this way.

At the end, e is calculated by finding a very large random number which satisfies the following equation:

$$e \cdot d \equiv 1(mod(p-1)\cdot(q-1))$$

In the implementation part, it is explained how to check if e satisfies this equation. After finding n, d and e, they become an identical key of a specific user. On the other hand, q and p numbers are not going to be used after these calculations.

IV. Analysis of the Given Formulas

In this section, it is going to be shown the correctness of the given encryption and decryption procedures. Assuming M (converted to integer) is a relatively prime number to n, As *Euler and Fermat [7]* states:

$$M\Phi(n) \equiv 1(modn)$$

In this formula, $\Phi(x)$ is the *Euler Totient* which gives the number of the positive integers relatively prime to n. If there is a prime number p instead of x, then:

$$\Phi(p) = p - 1$$

Let $a = b \cdot c$ (b and c are prime factors of a), one of the properties of the *totient* function [7] is:

$$\Phi(a) = \Phi(b) \cdot \Phi(c)$$

These 2 properties can be applied to our formula of $n = p \cdot q$:

$$\begin{aligned}\Phi(n) &= \Phi(p)\cdot\Phi(q)\\ &= (p-1)\cdot(q-1)\\ &= n-(p+q)+1\end{aligned}$$

It is stated that $d \cdot e$ is relatively prime to $(p-1)\cdot(q-1)$ which is equals to $\Phi(n)$. Formally:

$$e \cdot d \equiv 1(mod\ \Phi(n))$$

$$e \cdot d = k \cdot \Phi(n) + 1 \quad \text{(k can be any integer number)}$$

As it is known that:

$$D(E(M)) \equiv M^{e\cdot d}\,(mod\,n)$$

$$E(D(M)) \equiv M^{e\cdot d}\,(mod\,n)$$

Then the following equation can be obtained:

$$E(D(M)) \equiv M^{k\cdot\Phi(n)+1}\,(mod\,n)$$

When the first and second equations are considered:

$$M^{\Phi(p)} \equiv 1\,(mod\,p) \quad \text{p is a prime number}$$

$$M^{p-1} \equiv 1(mod\,p)$$

Due to the fact that $(p-1)$ divides $\Phi(n)$:

$$M^{k\cdot\Phi(n)+1} \equiv M(mod\,p)$$

Number q was a prime number, so that the equation can be transformed to the following equation:

$$M^{k\cdot\Phi(n)+1} \equiv M(mod\,q)$$

When the last two equations are combined, it can be written that:

$$M^{e\cdot d} \equiv M^{k\cdot\Phi(n)+1} \equiv M(mod\,n)$$

This equation proves that applying E and D procedures together on a message M where $0 \le M < n$, it results in M again.

V. Implementation and Algorithms

In this part of the paper, it is explained how the efficient algorithms of the calculations of encryption and decryption methods are developed and their implementations are shown one by one.

Calculation of $M^e(mod\, n)$ is done with the binary form of the numbers which is called "exponentiation by repeated squaring and multiplication". The method is explained below:

Step 1. Let $e_k\, e_{k-1} \dots e_1\, e_0$ be the binary representation of e.

Step 2. Set the variable C to 1.

Step 3. Repeat steps 3a and 3b for $i = k, k-1, \dots, 0$:

Step 3a. Set C to the remainder of C^2 when divided by n.

Step 3b. If $e_i = 1$, then set C to the remainder of $C \cdot M$ when divided by n.

Step 4. Halt. Now C is the encrypted form of M.

The calculation also can be done by using key d in a similar way. Implementation of the calculations with key d and e are given in *Appendix A*.

Finding the large prime numbers is one of the most challenging parts of the overall calculations and the numbers found by this calculation which are used in calculation of the keys of encryption and decryption plays a crucial role in the security part. In order to increase the security, the keys must be as large as possible so that the factorization of the $n = p \cdot q$ becomes not computationally feasible for anyone. It is recommended choosing the p and q numbers as 100 digit primes. It yields n to become a 200 digit number that cannot be factorized with a feasible computation. One way of making these computations is generating random odd numbers until finding the prime one. For generating 100-digit prime numbers, this approach takes about $\dfrac{\ln 10^{100}}{2} = 115$ try by the prime number theorem [7]. It is not an efficient method for large numbers. Because of that reason, in our implementation we used the probabilistic approach developed by *Solovay and Strassen* [10]. The idea of this approach is selecting a number t randomly to test if it is prime or not. In order to test its primality, a number k between 1 and $t-1$. After selecting k, apply the equation below:

$$gcd(k,t) = 1 \; and \; J(k,t) = \frac{k(t-1)}{2} \; (mod\, t)$$

where $J(k,t)$ stands for *Jacobi* calculation [7]. If t is prime, then the equation results as true. If t is not prime, the equation results as false with the probability of $\frac{1}{2}$. In order to increase correctness of the primality, we select 100 random k numbers and put them into the equation over and over again for 100 times. If all the results are equal to true, then it means that t is not prime with the probability of $(\frac{1}{2})^{100}$. Even if the calculation finds number t as a not prime number, then the system is going to be unable to do all the encryption and decryption calculations. So, it can easily notice that the number is not correct and it does the related calculations again. The *Jacobi* calculations are done with the algorithm below after finding the numbers k and t with *gcd = 1*:

$$J(k,t) = \text{if } k = 1, then\ 1$$

$$else \text{ if } k\ (mod\, 2) = 0,\ then\ J(\frac{k}{2},\ t) \cdot (-1)^{\frac{(t^2-1)}{8}}$$

$$else\ J(t\ (mod\, k), k) \cdot (-1)^{\frac{(k-1)\cdot(t-1)}{4}}$$

This algorithm does not test the primality by factoring the number. Because of that it is efficient compared to those types of algorithms. Implementation of finding N-Digit prime numbers (for finding p and q) with respect to explained algorithms are shown in *Appendix B*. After finding p and q, n is calculated by multiplying these two numbers.

Another concern is finding number d which is relatively prime to $\Phi(n)$. Any prime number greater than the $max(p,\, q)$ satisfies this condition. So, d is found by generating the prime number with the calculations explained in previous parts which has the length greater than the length of $max(p,\, q)$.

After finding the key d, the calculations of finding key e is done by *Euclid's gcd algorithm*. To find $gcd(\Phi(n),\, d)$. Algorithm uses series calculated by the following steps:

$$x_0 = \Phi(n)$$

$$x_1 = d$$

$$x_{i+1} = x_{i-1}(mod\, x_i)\ until\ an\ x_k = 0,$$

$$then\ gcd(x_0,\, x_1) = x_{k-1}$$

$gcd(x_0,\, x_1)$ is equal to key e. Do the calculations till e is less than $\log_2(n)$. If it is not, choose another d value and do the calculations again. Implementation of finding d and e is shown in *Appendix C*.

VI. Discussion

We have implemented all the algorithms by developing a basic client and server approach which are explained as they referenced to their related appendix. There is one server and one client, and the connection between them is established by using TCP sockets. Without using any of the encryption algorithms discussed previously, the users of the application can send messages back and forth. These messages are what we have defined as plain text.

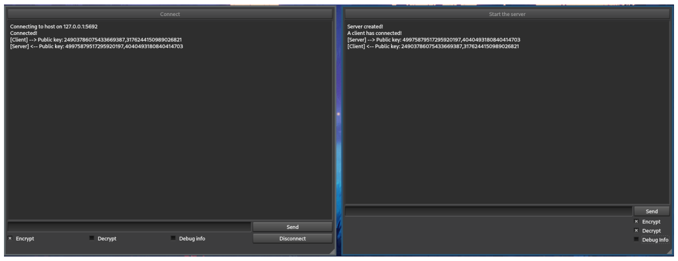
We have developed the application using C++ and Qt [8] library. In addition to those, to calculate the public and private keys, we have used an external library called *InfInt* [9]. *InfInt* is a C++ library which provides its users with almost infinite integers. Since the algorithm requires us to generate, multiple, and in general work with very large prime numbers. Doing these tasks would be impossible by using the primitive and default data structures, so that we have decided to use *InfInt* library.

By using this library, we have designed another library called *Crypto utils*. This library is the heart and soul of the implementation of the algorithm described in the paper.

The client and server that we created have options of sending a message in various ways. First when they are connected to each other, they send their n value and public keys to each other in order to be able to do the calculations explained before. After completing the connection, they have options of sending the message without encrypting or with it. Also another option is receiving a message with decrypting it or not. All the permutations of these options are tested by sending various messages and all the options worked as expected in 500 tests. The tests are done with 20 digit prime numbers because of not having enough hardware to compute larger prime numbers. Also the *Jacobi* test is decreased from 100 to 1 due to the same limitations. The algorithms explained before worked without any error even though they had a possibility of error with a larger size because of *Jacobi* test number.

An example of our project can be seen in the Figure 1. Before going on and explaining the details, the source code of our project can be found in our GitHub repository [11].

Figure 1 shows us the state and the exchange of the values d and n which are explained before. The Server sends its pair to the client when it receives a new connection and the Client sends its pairs to the Server when it receives the pairs from the Server.



```
Connecting to host on 127.0.0.1:5692
Connected!
[Client] --> Public key: 24903786075433669387,3176244150989026821
[Server] <-- Public key: 49975879517295920197,4040493180840414703
```

```
Server created!
A client has connected!
[Server] --> Public key: 49975879517295920197,4040493180840414703
[Client] <-- Public key: 24903786075433669387,3176244150989026821
```

Fig. 1. Key exchange between client and the server.

Other functionalities are as follows; The user can uncheck the default options, which are *encrypt* and *decrypt* to send plain text. For instance one side can uncheck the *decrypt* checkbox and see how the encrpted message looks like. Finally the user can check the *Debug info* checkbox to see the process of signing and encrypting with the other connection's E value. More images from our application showing different states can be found in the Appendix section.

When it comes to security, it cannot be proven if the encryption method is secure or not however, the factorization problem with the large numbers is a well-known problem that the mathematicians have been trying to solve since 400 years ago which started with Fermat and Legendre. It can be said that the methods we use are "partially" proven by the previous efforts to find an efficient way of factorization. On the other hand, IBM tried to attack the encrypted messages that are encrypted with the explained algorithms in order to obtain the message sent. But they failed and the algorithm has successfully resisted. It can be said that the methods are secure because they resisted such organized attacks.

VII. Conclusion and Future Work

In this paper, we explained how encryption keys and signature algorithms are applied and tested by the client-server applications implemented in C++ language. In the algorithms that we used, there are public private keys and the signature idea that does not require ant couriers compared to the other approaches. Public/Private keys construct secure connections so that no one can change the message sent or obtain it (trap-door one-way permutation) and the signature idea makes the message reliable because it is used to identify exactly who sent the message. The algorithms that are explained in this paper are commonly used in different areas to provide security. Because of the development of the technology (like quantum computers), a way of breaking the algorithms used in encryption can be found in the future and present systems can become insecure until new algorithms are developed for security.

When it comes to our implementation and our C++ application, future improvements such as a *Peer-to-peer* connection instead of local-host connection can be made. Furthermore our crypto tools library can also be improved to perform faster and work with prime numbers with more digits.

References

[1] Diffie, W., and Hellman, M. "New directions in cryptography". IEEE Trans. Inform.Theory IT-22, (Nov. 1976), 644-654.
[2] Diffie, W., and Hellman, M. "Exhaustive cryptanalysis of the NBS data encryption standard". Computer 10 (June 1977), 74-84.
[3] S. Chandra, S. Bhattacharyya, S. Paira and S. S. Alam, "A study and analysis on symmetric cryptography," 2014 International Conference on Science Engineering and Management Research (ICSEMR), Chennai, 2014, pp. 1-8, doi: 10.1109/ICSEMR.2014.7043664.
[4] Tools.ietf.org. 2000. RFC 2818 - HTTP Over TLS. [online] Available at: ¡https://tools.ietf.org/html/rfc2818¿ [Accessed 13 April 2020].
[5] Tools.ietf.org. 1999. RFC 2660 - "The Secure Hypertext Transfer Protocol". [online] Available at: ¡https://tools.ietf.org/html/rfc2660¿ [Accessed 13 April 2020].
[6] Tools.ietf.org. 2011. RFC 6101 - "The Secure Sockets Layer (SSL) Protocol" Version 3.0 [online] Available at: ¡https://tools.ietf.org/html/rfc6101¿ [Accessed 13 April 2020].
[7] Niven, I., and Zuckerman, H.S. "An Introduction to the Theory of Numbers". Wiley, New York, 1972.
[8] Nord, Havard, and Erik Chambe-Eng. "Qt, a cross-platform application development framework for desktop, embedded and mobile" Qt Io , https://www.qt.io/.
[9] Tutar, Sercan. "GitHub InfInt Library". GitHub InfInt Library ,https://github.com/sercantutar/infint.
[10] Solovay, R., and Strassen, V. "A Fast Monte-Carlo test for primality". SIAM J. Comptng. (March 1977), 84-85.
[11] Hasanzade, Arman. Buyukgönenç, Güneş. Burak Yıldırım, Ahmet. Denizsel, Cem." TCP Server/Client with Public/Private key encryption", https://github.com/ArmanHZ/CS333_project
[12] Shoeran, Ms Anjali. Sikha, Taruna. "Image Encryption and Decryption Using Discrete Cosine Transform (DCT)" Department of Electronics & Communication Engineering, Spiet Rohtak. [online] Available at: https://pdfs.semanticscholar.org [Accessed 23 May 2020]

Appendix

A. Encryption and Decryption Algorithms

```
auto encryptMessage(const string& message, const InfInt& key, const InfInt& n) -> string {
    auto binary = bitsetInfInt(key);
    string result;
    InfInt c;
    for (const auto& letter : message) {
        c = 1;
        for (const auto& bit : binary) {
            c = inf_pow(c, 2, n);
            if (bit == '1')
                c = c * letter % n;
        }
        for (size_t i{c.numberOfDigits()}; i < n.numberOfDigits(); ++i)
            result += '0';
            result += c.toString();
    }
    return result;
}

auto decryptMessage(const string& message, const InfInt& key, const InfInt& n) -> string {
    auto binary = bitsetInfInt(key);
    auto message_format = std::vector<InfInt>{};
    string result;
    for (size_t i{}; i < message.length(); i += n.numberOfDigits())
        message_format.emplace_back(message.substr(i, n.numberOfDigits()));
    InfInt c;
    for (const auto& letter : message_format) {
        c = 1;
        for (const auto& bit : binary) {
            c = inf_pow(c, 2, n);
            if (bit == '1')
                c = c * letter % n;
        }
        result += static_cast<char>(c.toInt());
    }
    return result;
}
```

B. Finding Random N-Digit Prime Numbers

```cpp
auto getNDigitRandomPrimeNumber(uint32_t n) {
    InfInt result;
    while (true) {
        auto b = getRandomNumber(n);
        if (b % 2 == 0) ++b;
        if (b % 5 == 0) b += 2;
        int cycle_size = 1;
        int i{};
        for (; i < cycle_size; ++i) {
            auto a = getRandomNumber(n - 1);
            if(inf_gcd(a, b) != 1 || Jacobi(a, b) != (inf_pow(a, (b - 1) / 2, b))) break;
        }
        if (i == cycle_size) {
            result = b;
            break;
        }
    }
    return result;
}
```

```cpp
auto Jacobi(const InfInt& a, const InfInt& b) -> InfInt {
    if (a == 1) return 1;
    if (a % 2 == 0) return Jacobi(a / 2, b) * inf_pow(-1, (inf_pow_nomod(b, 2) - 1) / 8, 2);
        return Jacobi(b % a, a) * inf_pow(-1, (a - 1) * (b - 1) / 4, 2);
}
```

```cpp
auto getRandomNumber(uint32_t n) -> InfInt {
    static auto rng = std::mt19937(std::chrono::steady_clock::now().time_since_epoch().count()
    static auto generator = std::uniform_int_distribution<std::mt19937::result_type>{0, 9};
    string str_result;
    auto value = generator(rng);
    while (value == 0)
        value = generator(rng);
    str_result += std::to_string(value);
    for (uint32_t i{1}; i < n; ++i)
        str_result += std::to_string(generator(rng));
    auto result = InfInt{str_result};
    return result;
}
```

```cpp
auto inf_gcd(const InfInt& x, const InfInt& y) -> InfInt {
    if (y == 0) return x;
        return inf_gcd(y, x % y);
}
```

```cpp
auto inf_pow(const InfInt& x, const InfInt& y, const InfInt& my_mod) -> InfInt {
    InfInt temp;
    if (y == 0)
        return 1;
    temp = inf_pow(x, y / 2, my_mod);
    if ((y % 2) == 0)
        return temp * temp % my_mod;
    else
        return x * temp * temp % my_mod;
}
```

```cpp
auto inf_pow_nomod(const InfInt& x, const InfInt& y) -> InfInt {
    InfInt temp;
    if (y == 0)
        return 1;
    temp = inf_pow_nomod(x, y / 2);
    if ((y % 2) == 0)
        return temp * temp;
    else
        return x * temp * temp;
}
```

C. Finding Public and Private Keys

```cpp
auto inf_log2(InfInt& n) -> InfInt {
    InfInt logValue = -1;
    while (n > 0) {
        logValue++;
        n /= 2;
    }
    return logValue;
}
```

```cpp
auto inf_gcdExtended(const InfInt& x, const InfInt& y, InfInt& a, InfInt& b) -> InfInt {
    // Base Case
    if (x == 0) {
        a = 0;
        b = 1;
        return y;
    }
    InfInt a1, b1; // To store results of recursive call
    auto gcd = inf_gcdExtended(y % x, x, a1, b1);
    // Update x and y using results of
    // recursive call
    a = b1 - (y / x) * a1;
    b = a1;
    return gcd;
}
```

```cpp
auto getNDigitRandomPrimeNumber(uint32_t n) {
    InfInt result;
    while (true) {
        auto b = getRandomNumber(n);
        if (b % 2 == 0) ++b;
        if (b % 5 == 0) b += 2;
        int cycle_size = 1;
        int i{};
        for (; i < cycle_size; ++i) {
            auto a = getRandomNumber(n - 1);
            if(inf_gcd(a, b) != 1 || Jacobi(a, b) != (inf_pow(a, (b - 1) / 2, b))) break;
        }
        if (i == cycle_size) {
            result = b;
            break;
        }
    }
    return result;
}
```

```cpp
auto calculateD_E(const InfInt& p, const InfInt& q, const InfInt& n) {
    auto fi_n = n - (p + q) + 1;
    auto n_here = n;
    InfInt max;
    auto calculateD = [&max](const InfInt& p, const InfInt& q) -> InfInt {
        if (p > q)
            max = p;
        else
            max = q;
        auto random_num = getNDigitRandomPrimeNumber(max.numberOfDigits() + 1);
        max = random_num;
        return max;
    };
    InfInt a, b, d;
    do {
        d = calculateD(p, q);
        a = 0, b = 0;
        inf_gcdExtended(fi_n, d, a, b);
    } while (inf_log2(n_here) > b);
    return std::make_pair(d, b);
}
```

D. Project images

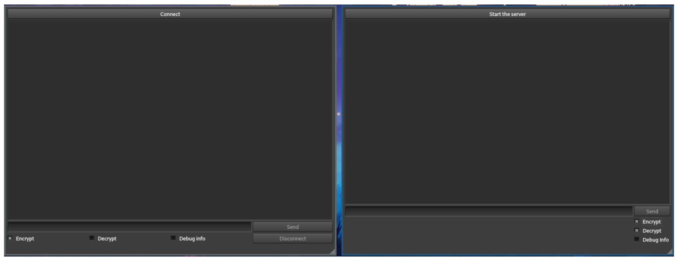
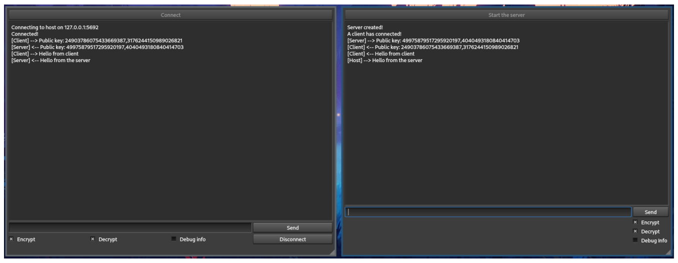


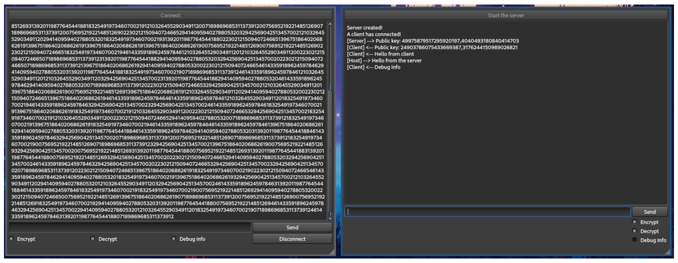
Fig. 2. Initial state of Server and Client when user first launches them.



```
Connecting to host on 127.0.0.1:5692
Connected!
[Client] --> Public key: 24903786075433669387,3176244150989026821
[Server] <-- Public key: 49975879517295920197,4040493180840414703
[Client] --> Hello from client
[Server] <-- Hello from the server
```

```
Server created!
A client has connected!
[Server] --> Public key: 49975879517295920197,4040493180840414703
[Client] <-- Public key: 24903786075433669387,3176244150989026821
[Client] <-- Hello from client
[Host] --> Hello from the server
```

Fig. 3. Client sending the Server encrypted message.



```
851269313920119877645441881832549197346070021912103264552903491120071898696853113739120075695219221485126907
18986968531137391200756952192214851269022302121509407246652941409594027880532032942569042513457002121032645
5290349112029414095940278805320183254919734607002193139201198776454418802230212150940724665139675186402068
62619139675186402068626191396751864020686261913967518640206862619007569521922148512690075695219221485126902
230212150940724665183254919734607002194614335918962459784612103264552903491120121032645529034911200223021215
0940724665071898696853113739123139201198776454418829414095940278805320329425690425134570020223021215094072
4665071898696853113739121396751864020686261929414095940278805320022302121509407246654614335918962459784629
41409594027880532031392011987764544188183254919734607002190718986968531137391246143359189624597846121032645
52903491120121032645529034911203294256904251345700231392011987764544188294140959402788053204614335918962 45
9784629414095940278805320071898696853113739120223021215094072466532942569042513457002121032645529034911201
39675186402068626190075695219221485126913967518640206862619121032645529034911202941409594027880532002230212
1509407246651396751864020686261946143359189624597846461433591896245978461210326455290349112018325491973460
700219461433591896245978463294256904251345700232942569042513457002461433591896245978461832549197346070021
91396751864020686261918325491973460700219121032645529034911200223021215094072466532942569042513457002183254
91973460700219121032645529034911200223021215094072466529414095940278805320071898696853113739121832549197346
07002191396751864020686261918325491973460700219461433591896245978464614335918962459784613967518640206862 61
92941409594027880532031392011987764544188461433591896245978462941409594027880532031392011987764544188461 43
35918962459784632942569042513457002071898696853113739120075695219221485126907189869685311373912183254919734
60700219007569521922148512690718986968531137391232942569042513457002139675186402068626190075695219221485126
93294256904251345700200756952192214851269313920119877645441880075695219221485126931392011987764544188313920 1
19877645441880075695219221485126932942569042513457002022302121509407246652941409594027880532032942569042 51
345700246143359189624597846329425690425134570020223021215094072466532942569042513457002329425690425134570
02071898696853113739120223021215094072466513967518640206862619183254919734607002190223021215094072466546143
35918962459784629414095940278805320183254919734607002191396751864020686261932942569042513457002121032645 52
90349112029414095940278805320121032645529034911203294256904251345700246143359189624597846313920119877645 44
18846143359189624597846183254919734607002191832549197346070021900756952192214851269294140959402788053200 22
302121509407246650075695219221485126913967518640206862619071898696853113739120075695219221485126900756952192
21485126918325491973460700219294140959402788053203139201198776454418800756952192214851269461433591896245978
46329425690425134570022941409594027880532012103264552903491120183254919734607002190718986968531137391246 14
33591896245978463139201198776454418807189869685311373912
```

```
Server created!
A client has connected!
[Server] --> Public key: 49975879517295920197,4040493180840414703
[Client] <-- Public key: 24903786075433669387,3176244150989026821
[Client] <-- Hello from client
[Host] --> Hello from the server
[Client] <-- Debug info
```

Fig. 4. Client sending an encrypted message with debug info on.