Abertay University

Diffie-Hellman Key Exchange

Diego Andrade – 2200905@abertay.ac.uk

Introduction to Security – CMP110

BSc Ethical Hacking Year 1

2022/23

Note that Information contained in this document is for educational purposes.

.

Abstract

Symmetric ciphers depend on a key exchange to be able to encrypt and decrypt communications, these exchanges need to be secure from “man-in-the-middle” attacks. The Diffie-Hellman key exchange allows for two clients to securely exchange keys; the main limitation of this technique, however, is the speed required to generate all the necessary variables, as they are large integers. The speed of the exchange is mainly limited by the prime factorization algorithm, as trial division is the simplest, yet slowest algorithm. The security of the exchange is theoretically sound, nonetheless, in order to keep up with modern cryptographical standards, the key size needed is significantly larger than can feasibly be generated in this demonstration.

.

Contents

.

.

1 INTRODUCTION

1.1 BACKGROUND

Cryptography is essential to the modern world; most digital communications use cryptography to protect both user's privacy and their safety. Simply put, encryption is a collection of methods to translate what is called "plaintext" into "ciphertext". This "ciphertext" optimally can only be translated back into "plaintext" by the desired recipient. Kessler stated in his paper "An Overview of Cryptography", "Cryptography, then, not only protects data from theft or alteration, but can also be used for user authentication" (Kessler 2015).

An attacker can implement an attack called a "man-in-the-middle" attack, where they redirect any traffic or communications to themselves before the server, allowing them to read or modify any traffic. One way to combat an attack like this is "end-to-end" encryption. End-to-end encryption means that communications don't need to be decrypted at the server, and they stay encrypted until they reach their destination. This can be achieved through two methods: asymmetric key encryption, and symmetric key encryption. Asymmetric key encryption has a public key that can encrypt messages but not decrypt them, and a private key that is used to decrypt messages. Symmetric key encryption uses the same key to encrypt and decrypt messages. An advantage of symmetric key encryption is that the computational power needed to encrypt and decrypt a symmetric key cipher is significantly lower than an asymmetric key cipher. A disadvantage of a symmetric key cipher is that sharing the keys between clients safely is a cryptographical challenge.

1.2 AIM

This report aims to demonstrate and analyze the Diffie-Hellman key exchange as a method of securely sharing a key for a symmetric key cipher. This overall aim can be achieved through a couple sub-aims:

- Demonstrate the Diffie-Hellman key exchange in a controlled system.
- Discuss the Diffie-Hellman key exchange as an effective method by analyzing,
    - The speed of the exchange
    - The security of the exchange

2 PROCEDURE

2.1 OVERVIEW OF PROCEDURE

In this report the demonstration is written in Rust, this is because it offers easier modularity (with crates) and allows for greater memory safety when dealing with large integers. See Appendix A for *Cargo.toml* file with all necessary dependencies and all their versions.

The key aspects of the Diffie-Hellman key exchange can be listed as follows: generating large prime numbers, finding a primitive root, server-side protocols, and client-side protocols. To evaluate the security of the exchange, Wireshark can then be used to passively observe communications.

To generate large random prime numbers, a random number needs to be generated and then tested with the Miller-Rabin primality test until it is declared *probably prime* by the test. The Miller-Rabin primality test is one of the fastest and most accurate tests of a prime number. A paper called "On the Number of Witnesses in the Miller-Rabin Primality Test" it is stated that, "The probability of error after k successful iterations becomes less than $\frac{1}{4^k}$. The only type of error in the Rabin' procedure is defining a composite integer as prime" (Ishmukhametov, Mubarakov and Rubtsova, 2020). When tested with 40 witnesses, a number that tested prime has an error probability of 1 in 1.20×10^{24}. See Table 1 for error probability to witness number.

Witnesses (k)	*Error probability*
1	0.25
10	9.536743×10^{-7}
20	9.094947×10^{-13}
30	8.673617×10^{-19}
40	8.271806×10^{-25}

Table 1: Error Probability of Miller-Rabin test with k witnesses, 4^{-k}.

To find a primitive root of the resultant prime number, n, the prime factors of n need to be computed, and each number from 2 to n needs to be checked by modular exponentiation to verify if all the possible remainders coprime to n are present.

The server-side protocols consist of negotiating constants, attempting the key exchange, and resetting the constants. The constants that need to be negotiated are the modulus, and the base (the primitive root of the modulus). The exchange protocol does not do any calculations, as it does not have all the available information to calculate the shared key, instead it passes the public combination of the secrets and the constants to the clients. The reset protocol resets the server and prepares it for a new exchange. The client-side protocols are the same as the server-side protocols; however, the negotiation and exchange protocols compute the constants and keys.

2.2 GENERATING LARGE PRIME NUMBERS

This demonstration will use the “num-bigint” crate to create an arbitrarily large unsigned integer type. In the “get_odd_bytes” function, random bytes of a desired amount will be generated using the “rand” crate, then, using a bitwise OR operation, the first and last bits of the array will be set. This will make sure the number has the desired number of bits and that it is odd, since the only even prime is 2. See Appendix B lines 5 to 17 for the full code.

The Miller-Rabin primality test is a key part of the key exchange, as the secret keys and modulus are large prime numbers. First, choose a random witness, a, between $2 < a < n - 2$. Then find r and d:

Where r is the largest positive integer such that $2^r \; rem \; (n-1) = 0$

Where $d = \frac{n-1}{2^r}$ and d is odd.

With those values:

Calculate x $= a^d mod\; n$, if $x = 1$ or $x = (n \; - \; 1)$

continue to the next witness; otherwise,

Calculate $x = x^2 \; mod \; n$.

If $x = (n \; - \; 1)$, n is composite, else: repeat the previous step $r \; - \; 1$ times. If all the witnesses are computed without returning composite, then n is probably prime. The ”miller_rabin” function first checks all the common cases: 0, 2, 3, and even numbers and then computes the previously described algorithm.

The ”get_prime” function calls “get_odd_bytes” with a given number of bytes and tests the returned integer with the “miller_rabin” function, and generates another number until it is prime. It is not computationally expensive to generate another random number, thus, it is not more efficient to add 1 to the previous number and test again.

2.3 COMPUTING PRIMITIVE ROOT OF GIVEN PRIME

To find the primitive root of a given prime, the factors of the prime are essential. In this report, the method used to find factors is trial division. Trial division is a simple method to find factors; to use it, compute with all numbers in range: $[2, \sqrt{n}]$:

For every number, i, if $n\ rem\ i = 0$, then i is a factor.

To continue, divide n by i and repeat until $n = 1$ or $n\ rem\ i \neq 0$.

In the “get_factors” function, this algorithm is applied, while the factors are pushed to a vector and the vector is returned at the end.

The ”get_primitive_root” function returns an Option structure. In Rust, an Option structure allows the programmer to return either “Some(value)” or “None”. This allows for simpler error checking without having to use a try-catch block. Since finding a primitive root is not always guaranteed (in this report, assume it is), we will return None if no primitive root is found. As explained in the previous page, the function will run from 2 to n and check, by modular exponentiation, that all possible remainders are present. If they are, it will return an Option structure containing the primitive root, which is then unwrapped in the main function. See Appendix B lines 138 to 160 for the full code.

2.4 SERVER-SIDE PROTOCOLS

The first protocol the server handles is the negotiation protocol; when the server receives a request to the “/negotiate” route, it will store the attached modulo and base unless they have already been set. In the case that they have already been set, the server responds with a status code of 409 (conflict) and sends a JSON object with the modulo and base.

The second protocol the server handles is the exchange protocol; when the server receives a request to the “/key” route, it will store the attached key in a temporary exchange variable and respond with a status code of 209 (accepted). If it receives a request when the key is already set, and the key isn’t the same as the one already stored, it will respond with the temporary key variable in a JSON object and sets the temporary key variable to the request’s attached key.

The last protocol is the reset protocol. In order to do another exchange, it is necessary to reset all the server variables. When the server receives a request to the “/reset” route, it will set all the variables to empty strings. See Appendix C for the full code.

2.5 CLIENT-SIDE PROTOCOLS

The client protocols and data are all stored on a Client struct, in order to execute HTTP requests, the “reqwest” crate is used. Throughout the code, hash maps are used to store the data before the request is sent, this is because the “reqwest” crate automatically converts hash maps into JSON objects with the request builder’s “json” function.

The “negotiate_constants” function uses the “get_prime” and “get_primitive_root” to generate the base and modulus constants, then they are inserted into a hash map and sent in an HTTP POST request. If the response has a 409 (conflict) response then it destructs the response into a hash map and sets the clients constants to the ones saved on the server.

The “exchange” function generates its own public key by calculating:

$$g^a \ mod \ p,$$

Where g is the global base, a is the client’s private key (a random prime number generated when the client is initialized using the “init” function), and p is the global modulus.

It then sends the public key and the name of the client (arbitrary name i.e., Alice or Bob) in an HTTP POST request. If the response has a status code of 202 (accepted), then it will wait 1 second and try again. When the response is 200 (successful), it computes the shared secret key by:

$$B^a \ mod \ p,$$

Where B is the other client’s public key.

The “reset’ function sets all the constants and the shared secret key to 0, and generates a new random prime for the private key. It then sends a HTTP GET request to the “/reset” route to reset the server.

2.6 MAIN FUNCTION

The main function is very vague, anything can be done here as long as the client is initialized, and the “negotiate_constants” and “exchange” functions are run. This demonstration allows the user to enter a name, key size (in bytes), and decide if they want to run the demo (negotiate and then exchange), reset, or quit. See Appendix D lines 6 to 39 for full code.

2.7 WIRESHARK

The demonstration runs on localhost port 3000, this allows us to use the “Adapter for loopback traffic capture” Wireshark interface. Start the capture in Wireshark, run the demo, stop the capture, and filter by HTTP packets. The requests and responses should be listed with their respective response codes. Within the packets, the JSON objects should be visible without any encryption, and all the contained values should be legible.

3 Results

This report of the Diffie-Hellman key exchange is aimed to assess its effectiveness and performance in being a fast and secure key exchange protocol. The following results were obtained on an MSI laptop with an i9-11900H running at 2.5GHz:

3.1 Generating Large Prime Numbers

3.1.1 Speed

To test the speed of generating large prime numbers, an example was set up to generate a random prime number with a variable size, a variable number of times. The average time elapsed was then taken and output. See Table 2 for time taken to generate random prime numbers with sizes 4-8 over 1000 iterations.

Size (bytes)	*Average time elapsed (s)*
4	8.73×10^{-3}
5	9.26×10^{-3}
6	1.00×10^{-2}
7	1.01×10^{-2}
8	1.13×10^{-2}

Table 2: Average time elapsed for prime numbers of varying sizes over 1000 iterations.

3.1.2 Security

The cryptographic security of a random number generator is based on two criteria: the “next-bit” test, and “forward secrecy”. If a random number generator passes the “next-bit” test, it means given the first k bits generated by the generator, the next bit $(k+1)$ cannot be predicted with a success rate of greater than 50%. Forward secrecy means that if part or all of the generator’s state is compromised, it should not be possible to calculate the previous random numbers that have been generated, or future numbers that can be generated. The random number generator used in this demonstration is ChaCha12. ChaCha12 passes the “next-bit” test, but does not have forward secrecy. See Table 3 for RNG security table.

Name	*Next-bit test*	*Forward secrecy*
ChaCha8	✓	✗
ChaCha12	✓	✗
ChaCha20	✓	✗

Table 3: CSPRNG requirements met by ChaCha RNGs.

3.2 COMPUTING PRIMITIVE ROOT OF GIVEN PRIME

3.2.1 Speed

The speed of this algorithm has proven to be highly dependent on the speed of the "get_factors" function; the speed of the rest of the function is negligible in comparison to the speed of the trial division algorithm. See Figure 4 for time complexity of trial division. To test the speed, a similar test example was set up to run the "get_primitive_root" function with a set byte size over multiple iterations. See Table 5 for time taken to calculate primitive root with sizes 4-6 bytes over 4 iterations.

Size (bytes)	*Average time elapsed (s)*
4	4.84×10^{-2}
5	1.35
6	18.2

Table 5: Average time elapsed for primitive roots of varying sizes for 4 iterations.

3.2.2 Security

The security of this algorithm is always the same, as the primitive root is only used as a generator for the modular exponentiation, allowing for equal distribution in the function, and all values between 0 and n-1 to be possible. If the primitive root is, in fact, a primitive root, then the security cannot be increased.

3.3 SERVER-SIDE AND CLIENT-SIDE PROTOCOLS

3.3.1 Speed

The time spent on server-side protocols was found to be mostly in the "desync" when exchanging keys. This is caused by the frequency of the client sending requests to check if the exchange has been completed by the other client. The client requests once a second, meaning the maximum desync time is 1s.

3.3.2 Security

HTTP is very insecure for general communications between clients and servers because it is not encrypted, thus everything is sent in plaintext. Using Wireshark to capture all packets between the clients and the server, all the JSON objects containing all the variables are visible in plaintext; these include: the base, modulus, and both generated public keys. See Appendix E for full packet capture.

4 Discussion

4.1 General Discussion

4.1.1 Speed

The time taken to generate a large prime number is minimal when using the Miller-Rabin primality test, this report found that using random numbers every iteration is not noticeably slower than adding 1 to the initially generated number; however, in terms of calculations needed, adding 1 to the random number is more efficient.

The speed of the key exchange was most hindered by the generation of the primitive root, more specifically, the calculation of prime factors. The trial division algorithm is the slowest algorithm used to calculate factors; other algorithms include Pollard's Rho, quadratic sieve, and general number field sieve. These algorithms consist of significantly more code and more complicated mathematics that are out of this report's scope. A major problem with the trial division method is its speed, as it needs to do $\sqrt{n}$ remainder operations, which causes a drastic increase in time taken with larger numbers. This could be improved with multithreading, however, with this demonstration, it proved to slow down the overall speed, this might be because the factors vector needs to be locked by each thread when they add a new factor, thus causing the other threads to wait before pushing their factor to the end of the vector. Another technique that could be used to improve prime factorization speed is GPU acceleration; however, there are currently no Rust crates that are stable enough to use with ease, NVIDIA CUDA programming has the most support with C++, thus for GPU acceleration, C++ would be the better alternative. With this demonstration, the maximum size recommended is 6 bytes, as it takes less than 1 minute to generate all the factors necessary, as seen in the previous section.

4.1.2 Security

The Wireshark capture in Appendix E is similar to what a third-party performing a "man-in-the-middle" attack would find when attacking a program using a Diffie-Hellman key exchange. In the capture, variables A, B, g, and p are publicly available. This does not mean that a, b, or the shared secret are able to be calculated. To find the a or b, an attacker would have to solve a discrete logarithm problem, which takes significant processing power and time with large enough numbers. An algorithm such as:

$$A = g^a mod\ p$$

is considered a one-way algorithm, meaning that finding a from the result is significantly more difficult than finding the result, even if all variables not including a are known by the third-party. This is because there are multiple possible values of a for the same value of A, with a large enough p (modulus), there are too many possible values to compute in a feasible amount of time. Even if all the values were to be computed, it is hard to test if any of the values are the correct value, as b would have to be found and decryption using a test secret would have to be attempted. This only holds true if g (base) is a primitive root of p, thus the speed of generating a primitive root is important, as using a large value of p is key. Most modern keys use 1024-bit and 2048-bit key sizes, meaning this demonstration is not up to standard, however it does a good job of demonstrating the key exchange and its security.

4.2 FUTURE WORK

In a more secure and efficient demonstration, the next step in improving the program would be optimizing the “get_factors” function using the general number field sieve method. This would require a deeper understanding of number fields and polynomial mathematics. Generating a 1024-bit or larger prime number would also require optimization of the “get_prime” function, possibly using multiple threads to test multiple prime numbers simultaneously. With smaller prime numbers, this may hinder the speed for the same reason multithreading did not work with the “get_factors” function, with that in mind, numbers of that size do cause the Miller-Rabin test to slow down, thus, there would be fewer accesses to the same variable, causing less threads to wait on each other and slow down the overall speed.

4.3 CONCLUSIONS

This report has shown that a Diffie-Hellman key exchange is theoretically secure; nevertheless, it has also proven that it is difficult to generate numbers large enough to have a key exchange hold up to modern standards. This is due to the speed of dealing with large integers, although this is fixable, as stated in the Future Work section, this demonstration does not meet that standard.

REFERENCES

Kessler, G.C. (2015). *An Overview of Cryptography*. [online] Auerbach, p.2. Available at: https://www.academia.edu/download/38411944/An_Overview_of_Cryptography.pdF [Accessed 20 May 2023].

Ishmukhametov, S.T., Mubarakov, B.G. and Rubtsova, R.G. (2020). On the Number of Witnesses in the Miller–Rabin Primality Test. *Symmetry*, 12(6), p.890. doi:https://doi.org/10.3390/sym12060890.

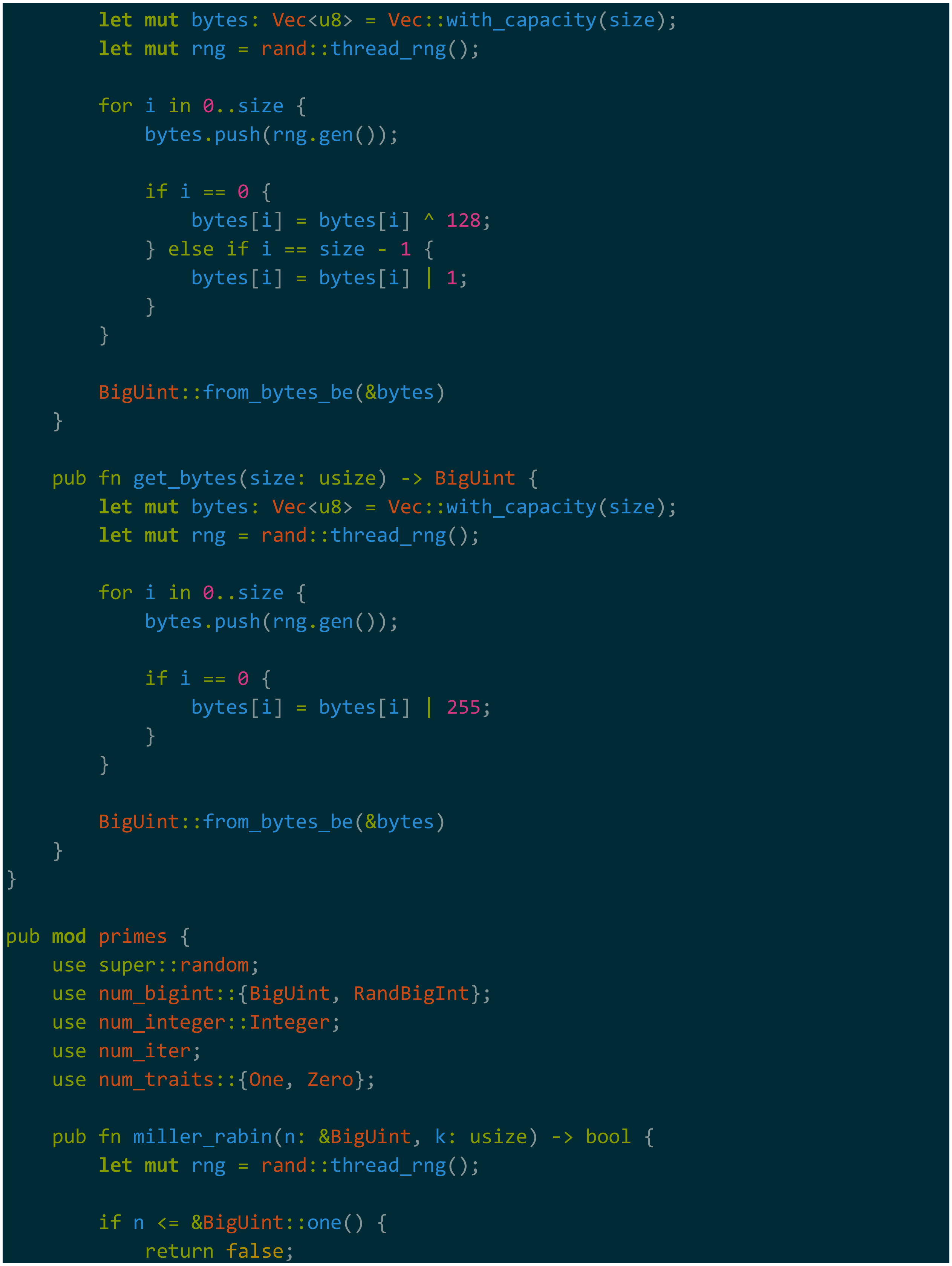
APPENDICES

APPENDIX A – CARGO.TOML

```toml
[package]
name = "diffie-hellman-rust"
version = "0.1.0"
edition = "2021"

[dependencies]
rand = "0.8.5"
num-bigint = { version = "0.4.3", features = ["rand"] }
num-integer = "0.1.45"
num-traits = "0.2.15"
num-iter = "0.1.43"
num-prime = "0.4.3"
reqwest = { version = "0.11.17", features = ["json"] }
tokio = { version = "1.28.0", features = ["full"] }
text_io = "0.1.12"
```

APPENDIX B – LIBRARY CODE

```rust
pub mod random {
    use num_bigint::BigUint;
    use rand::Rng;

    pub fn get_odd_bytes(size: usize) -> BigUint {
        let mut bytes: Vec<u8> = Vec::with_capacity(size);
        let mut rng = rand::thread_rng();

        for _ in 0..size {
            bytes.push(rng.gen());
        }

        bytes[0] = bytes[0] | 128;
        bytes[size - 1] = bytes[size - 1] | 1;

        BigUint::from_bytes_be(&bytes)
    }

    pub fn get_even_bytes(size: usize) -> BigUint {
```



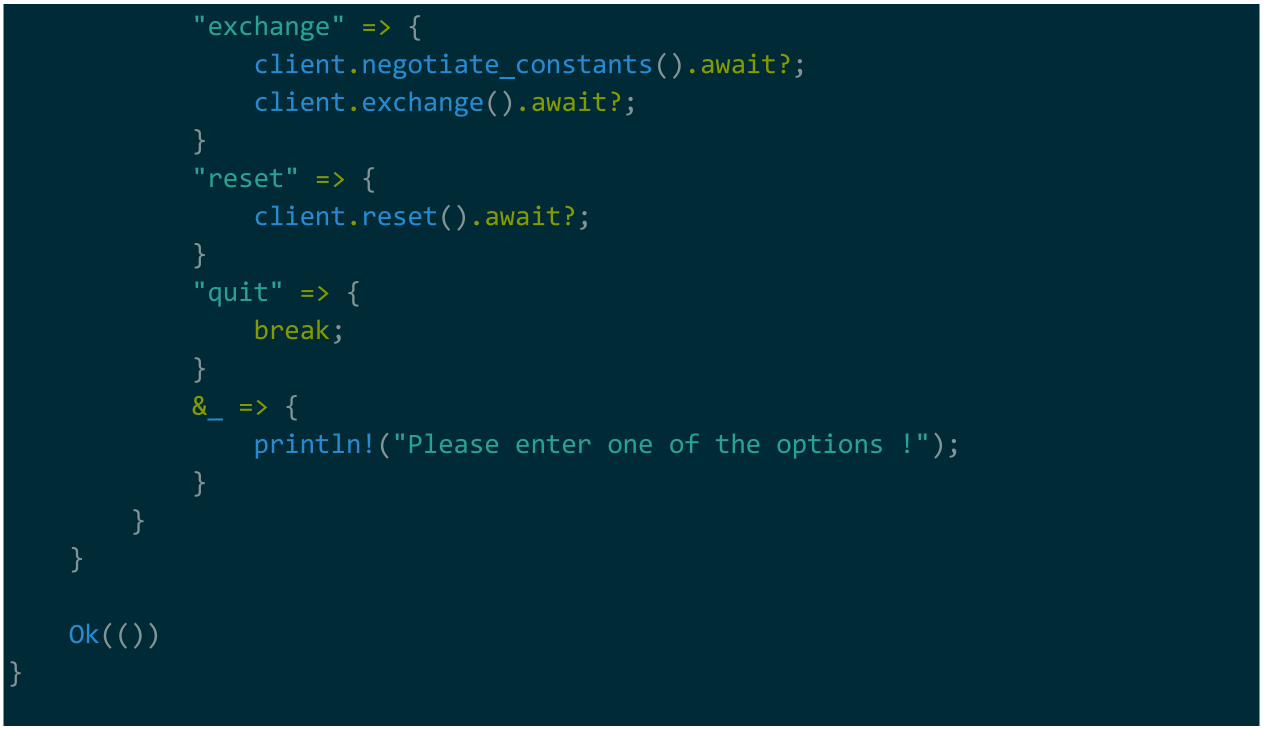
```
        let mut bytes: Vec<u8> = Vec::with_capacity(size);
        let mut rng = rand::thread_rng();

        for i in 0..size {
            bytes.push(rng.gen());

            if i == 0 {
                bytes[i] = bytes[i] ^ 128;
            } else if i == size - 1 {
                bytes[i] = bytes[i] | 1;
            }
        }

        BigUint::from_bytes_be(&bytes)
    }

    pub fn get_bytes(size: usize) -> BigUint {
        let mut bytes: Vec<u8> = Vec::with_capacity(size);
        let mut rng = rand::thread_rng();

        for i in 0..size {
            bytes.push(rng.gen());

            if i == 0 {
                bytes[i] = bytes[i] | 255;
            }
        }

        BigUint::from_bytes_be(&bytes)
    }
}

pub mod primes {
    use super::random;
    use num_bigint::{BigUint, RandBigInt};
    use num_integer::Integer;
    use num_iter;
    use num_traits::{One, Zero};

    pub fn miller_rabin(n: &BigUint, k: usize) -> bool {
        let mut rng = rand::thread_rng();

        if n <= &BigUint::one() {
            return false;
        }

        if n == &BigUint::from(2_u8) || n == &BigUint::from(3_u8) {
            return true;
        }

        if n.is_even() {
            return false;
        }

        let mut r: u64 = 0;
        let mut d: BigUint = n - 1_u8;

        while d.is_even() {
            r += 1;
            d >>= 1;
        }

        for _ in 0..k {
            let a = rng.gen_biguint_range(&BigUint::from(2_u8), &(n - 2_u8));
            let mut x = a.modpow(&d, n);

            if x == BigUint::one() || x == (n - BigUint::one()) {
                continue;
            }

            for _ in 1..r {
                x = x.modpow(&BigUint::from(2_u8), n);

                if x == (n - BigUint::one()) {
                    break;
                }
            }

            if x != (n - BigUint::one()) {
                return false;
            }
        }

        true
    }

    pub fn get_prime(size: usize) -> BigUint {
        let mut prime = random::get_odd_bytes(size);
        while !miller_rabin(&prime, 40) {
            prime = random::get_odd_bytes(size);
        }

        prime
    }

    pub fn get_factors(num: &BigUint) -> Vec<BigUint> {
        let mut factors: Vec<BigUint> = Vec::new();
        let mut n = num.clone();

        for i in num_iter::range_step_inclusive(BigUint::from(2u8), n.sqrt(),
BigUint::one()) {
            while &n % &i == BigUint::zero() {
                factors.push(i.clone());
                n /= &i;
            }

            if n == BigUint::one() {
                break;
            }
        }

        if n > BigUint::one() {
            factors.push(n);
        }

        factors
    }

    pub fn get_primitive_root(p: &BigUint) -> Option<BigUint> {
        let phi = p.clone() - BigUint::one();

        let factors = get_factors(&phi);

        for g in num_iter::range_step_inclusive(BigUint::from(2u8), p.clone(),
BigUint::one()) {
            let mut is_primitive_root = true;

            for factor in &factors {
                let exp = &phi / factor;
                if g.modpow(&exp, p) == BigUint::one() {
                    is_primitive_root = false;
                    break;
                }
            }

            if is_primitive_root {
                return Some(g);
            }
        }

        None
    }
}

pub mod crypto {
    use num_bigint::BigUint;
    use num_traits::Zero;
    use std::{
        collections::HashMap,
        str::{self, FromStr},
        thread, time,
    };

    use super::primes;

    pub struct Client {
        modulus: BigUint,
        base: BigUint,
        private_key: BigUint,
        shared_key: BigUint,
        client: reqwest::Client,
        server_name: String,
        name: String,
        bytes: usize,
    }

    impl Client {
        pub fn init(name: &str, server_name: &str, bytes: usize) -> Client {
            Client {
                name: String::from(name),
                client: reqwest::Client::new(),
                server_name: String::from(server_name),
                private_key: primes::get_prime(bytes),
                shared_key: BigUint::zero(),
                modulus: BigUint::zero(),
                base: BigUint::zero(),
                bytes,
            }
        }

        pub async fn negotiate_constants(&mut self) -> Result<(), reqwest::Error>
{
            self.modulus = primes::get_prime(self.bytes);
            self.base = primes::get_primitive_root(&self.modulus).unwrap();

            let mut map = HashMap::new();

            map.insert("modulus", self.modulus.to_string());
            map.insert("base", self.base.to_string());
            map.insert("name", self.name.to_string());

            let res = self
                .client
                .post(self.server_name.to_owned() + "/negotiate")
                .json(&map)
                .send()
                .await?;

            if res.status() == reqwest::StatusCode::CONFLICT {
                let body: HashMap<String, String> = res.json().await?;

                self.modulus =
BigUint::from_str(body.get("modulus").unwrap()).unwrap();
                self.base =
BigUint::from_str(body.get("base").unwrap()).unwrap();
            }

            println!("{}, {}", self.modulus, self.base);

            Ok(())
        }

        pub async fn exchange(&mut self) -> Result<(), reqwest::Error> {
            let public_key = self.base.modpow(&self.private_key, &self.modulus);

            let mut map = HashMap::new();
            map.insert("key", public_key.to_string());
            map.insert("name", self.name.to_string());
            let mut res = self
                .client
                .post(self.server_name.to_owned() + "/key")
                .json(&map)
                .send()
                .await?;

            while res.status() == reqwest::StatusCode::ACCEPTED {
                thread::sleep(time::Duration::from_secs(1));

                res = self
                    .client
                    .post(self.server_name.to_owned() + "/key")
                    .json(&map)
                    .send()
                    .await?;
            }

            let body: HashMap<String, String> = res.json().await?;

            let other_public_key =
BigUint::from_str(body.get("key").unwrap()).unwrap();

            println!("mine: {}", public_key);
            println!("other: {}", other_public_key);

            self.shared_key = other_public_key.modpow(&self.private_key,
&self.modulus);

            println!("{}", self.shared_key);

            Ok(())
        }

        pub async fn reset(&mut self) -> Result<(), reqwest::Error> {
            self.base = BigUint::zero();
            self.modulus = BigUint::zero();
            self.private_key = primes::get_prime(self.bytes);
            self.shared_key = BigUint::zero();

            self.client
                .get(self.server_name.to_owned() + "/reset")
                .send()
                .await?;
            Ok(())
        }
    }
}
```

APPENDIX C – SERVER CODE

```
const express = require('express');
const app = express();
const port = 3000;

let modulus = '';
let base = '';

let temp_key = '';

app.use(express.json());

app.post('/negotiate', (req, res) => {
    console.log('request recieved !');

    if (!base && !modulus) {
        modulus = req.body.modulus;
        base = req.body.base;

        res.status(200);
        res.send();
    } else {
        res.status(409);
        res.json({ modulus: modulus, base: base });
    }
});

app.post('/key', (req, res) => {
    if (!temp_key || req.body.key == temp_key) {
        console.log('exchange attempted');
        temp_key = req.body.key;
        res.status(202);
        res.send();
    } else {
        res.status(200);
        res.json({ key: temp_key });
        temp_key = req.body.key;
    }
});

app.get('/reset', (req, res) => {
    temp_key = '';
    modulus = '';
    base = '';
    res.status(200);
    res.send();
});

app.listen(port, () => {
    console.log(`listening on port ${port}`);
});
```

APPENDIX D – CLIENT CODE

```
use diffie_hellman_rust::crypto;

use text_io::read;

#[tokio::main]
async fn main() -> Result<(), reqwest::Error> {
    print!("Enter client name: ");
    let name: String = read!();

    print!("Enter keysize in bytes: ");
    let bytes: usize = read!();

    let mut client = crypto::Client::init(&name, "http://localhost:3000", bytes);

    let mut input: String;

    loop {
        print!("Enter [exchange], [reset], or [quit]: ");
        input = read!();

        match input.as_str() {
```



```
            "exchange" => {
                client.negotiate_constants().await?;
                client.exchange().await?;
            }
            "reset" => {
                client.reset().await?;
            }
            "quit" => {
                break;
            }
            &_ => {
                println!("Please enter one of the options !");
            }
        }
    }

    Ok(())
}
```

APPENDIX E

exchange_capture.pcapng