Міністерство освіти і науки України

Національний технічний університет України

“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Лабораторна робота №4

Протоколи й алгоритми електронного голосування

Тема: Протокол Е-голосування з перемішуванням

Виконали
студенти групи ІП-11:
Панченко С. В.,
Лисенко А. Ю.

Перевірив:
Нестерук А. О.

Київ 2024

ЗМІСТ

1 Мета лабораторної роботи.....6

2 Завдання.....7

3 Виконання.....8

4 Демонстрація роботи протоколу.....9

5 Дослідження протоколу.....11

Висновок.....13

ДОДАТОК А ТЕКСТИ ПРОГРАМНОГО КОДУ.....14

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ

Дослідити протокол Е-голосування з перемішуванням.

2 ЗАВДАННЯ

Змоделювати протокол Е-голосування з перемішуванням будь-якою мовою програмування та провести його дослідження. Для кодування повідомлень використовувати метод RSA, для реалізації ЕЦП використовувати алгоритм Ель-Гамаля.

Умови: В процесі голосування повинні приймати участь не менше 2 кандидатів та не менше 4 виборців. Повинні бути реалізовані сценарії поведінки на випадок порушення протоколу (виборець не проголосував, проголосував неправильно, виборець не має права голосувати, виборець хоче проголосувати повторно, виборець хоче проголосувати замість іншого виборця та інші).

На основі змодельованого протоколу провести його дослідження (Аналіз повинен бути розгорнутим та враховувати всі можливі сценарії подій під час роботи протоколу голосування):

1. Перевірити чи можуть голосувати ті, хто не має на це права.
2. Перевірити чи може виборець голосувати кілька разів.
3. Чи може хтось (інший виборець, ЦВК, стороння людина) дізнатися за кого проголосували інші виборці?
4. Перевірити чи може інший виборець чи стороння людина проголосувати замість іншого зареєстрованого виборця.
5. Чи може хтось (інший виборець, ЦВК, стороння людина) таємно змінити голос в бюлетені?
6. Чи може виборець перевірити, що його голос врахований при підведенні кінцевих результатів?

3 ВИКОНАННЯ

Для виконання роботи використовували мову програмування Rust.

У проєкті реалізовані кілька модулів, кожен із яких виконує окремі функції для підтримки криптографічних операцій та симуляції голосування.

Модуль voter реалізує сам протокол голосування. В ньому використовуються функції для генерування шуму, наприклад, create_noise, що додають випадкові дані для перемішування зашифрованих повідомлень, роблячи їх важко відстежуваними. Процес голосування відбувається в кілька етапів. Функція cipher реалізує шифрування голосів на першому та другому етапах, використовуючи алгоритм RSA, тоді як функції decipher_second_stage і decipher_sign_first_stage відповідають за їх дешифрування. Після дешифрування голоси розшифровуються, перевіряються, і в кінці підраховуються. Для цього також перевіряється, чи всі голоси є справжніми. У модулі реалізовані тестові сценарії, які симулюють процес голосування, обчислюють кількість голосів і перевіряють їх справжність після дешифрування.

4 ДЕМОНСТРАЦІЯ РОБОТИ ПРОТОКОЛУ

Тест, що перевіряє роботу протоколу, виконує повну симуляцію процесу е-голосування з перемішуванням і складається з кількох ключових етапів, кожен з яких перевіряє правильність роботи різних частин протоколу. Спершу, на етапі підготовки, створюється список виборців та кандидатів. У цьому прикладі генерується 15 виборців, кожен із яких має свої RSA та ElGamal ключі. Також створюється список із 5 кандидатів, кожен із яких отримує унікальне ім'я. Це забезпечує базу для подальшого шифрування та голосування.

Наступним етапом є шифрування голосів. Тут кожен виборець шифрує свій голос за одного з кандидатів, використовуючи функцію, яка відповідає за шифрування даних через RSA. Для ускладнення дешифрування додається випадковий шум. Після завершення цього процесу для кожного виборця створюються об'єкти, що містять інформацію про шум першого етапу, таблицю шуму другого етапу та зашифровані дані. Ці об'єкти будуть використовуватися на наступних етапах дешифрування.

На третьому етапі відбувається дешифрування другого етапу. Використовуючи функцію дешифрування, кожен виборець за допомогою свого приватного ключа розшифровує відповідні дані, видаляючи доданий раніше шум. Цей етап є важливим для забезпечення анонімності виборців, оскільки шифровані дані перемішуються та стають важко відстежуваними.

Після завершення дешифрування другого етапу викликається функція для дешифрування та підписання першого етапу. Кожен виборець знову перевіряє свій голос, додає свій підпис і передає його далі. Це гарантує справжність кожного голосу та відсутність змін у процесі голосування.

Завершальним етапом є підрахунок голосів. Після того, як всі голоси дешифровані та перевірені, вони підраховуються за допомогою хеш-таблиць. Для кожного кандидата ведеться підрахунок голосів, і результат відображається або перевіряється. Якщо голос було віддано за певного кандидата, його лічильник збільшується.

Результати тесту показують на рисунку 4.1, що протокол е-голосування

від початку до кінця працює коректно. Підрахунок голосів, перевірка підписів та дешифрування відбуваються без помилок, що підтверджує правильність реалізації протоколу.

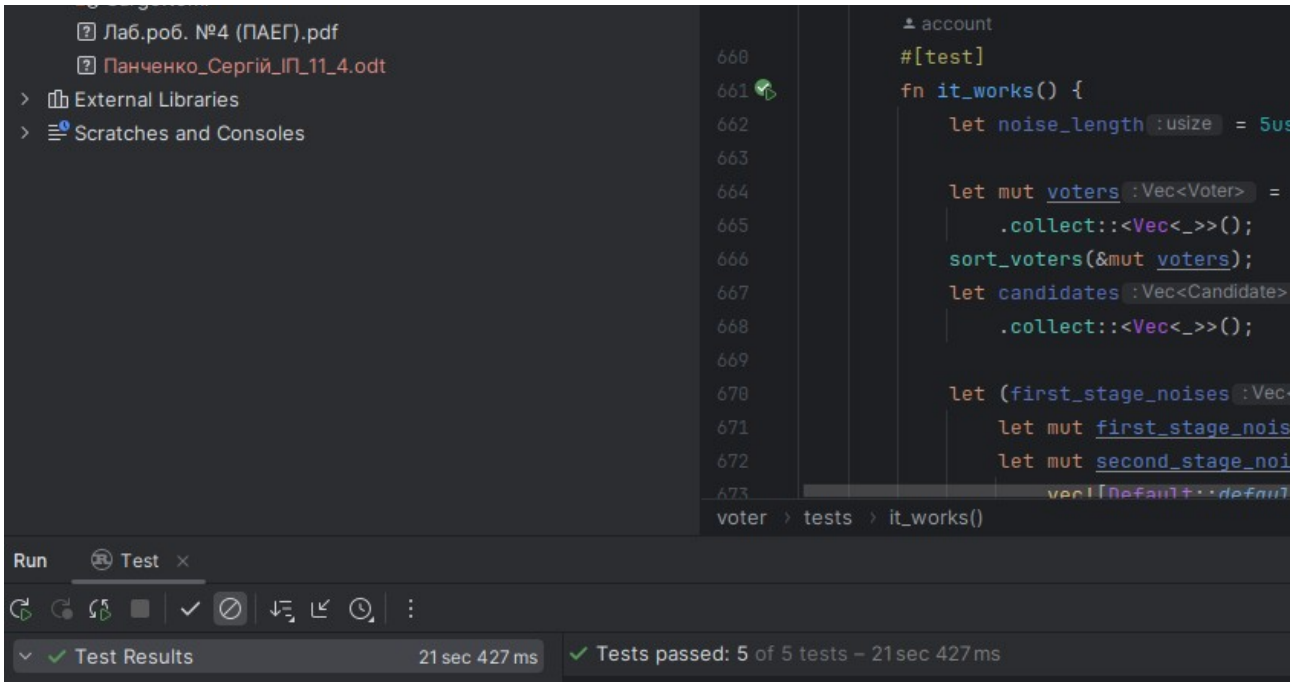

Рисунок 4.1 — Тестування процесу голосування

5 ДОСЛІДЖЕННЯ ПРОТОКОЛУ

Чи можуть голосувати ті, хто не має на це права? Алгоритм е-голосування забезпечує, що лише ті виборці, які мають відповідні ключі, можуть брати участь у голосуванні. Кожен виборець використовує унікальний ключ для шифрування свого бюлетеня. Якщо хтось намагається проголосувати без наявності ключа або спробує підробити ключ, це буде виявлено під час шифрування або перевірки бюлетеня на етапі його надсилання і розшифрування іншими учасниками. Оскільки кожен бюлетень проходить багаторазове шифрування та перевірку підписів, будь-яка невідповідність швидко буде виявлена. Таким чином, сторонні особи, що не мають відповідного ключа, не зможуть брати участь у голосуванні.

Чи може виборець голосувати кілька разів? Голосування кілька разів для одного виборця неможливе через те, що кожен бюлетень пов'язаний з унікальним ключем виборця і шифрується окремо. Будь-яка спроба подати кілька бюлетенів буде виявлена під час шифрування і розшифрування, оскільки система вимагає, щоб кількість бюлетенів відповідала кількості виборців. Якщо один із виборців спробує проголосувати більше одного разу, це буде виявлено під час перевірки кількості зашифрованих бюлетенів.

Чи може хтось (інший виборець, ВК, стороння людина) дізнатися, за кого проголосували інші виборці? Алгоритм е-голосування з перемішуванням гарантує анонімність голосів завдяки шифруванню та додаванню випадкових рядків, які знищують можливість стежити, за кого саме проголосував виборець. Після кожного етапу шифрування бюлетені перемішуються, що ускладнює можливість встановлення зв'язку між виборцем і його голосом. Навіть інші виборці або виборча комісія не можуть визначити, за кого було подано голос, що забезпечує високий рівень конфіденційності.

Чи може інший виборець чи стороння людина проголосувати замість іншого зареєстрованого виборця? Алгоритм передбачає використання унікальних ключів для кожного виборця, що унеможливлює можливість голосування іншої особи замість зареєстрованого виборця. Якщо хтось спробує

зімітувати ключ іншого виборця або подати підроблений бюлетень, система виявить це завдяки перевірці підпису. Бюлетені мають бути підписані цифровим підписом, що прив'язаний до приватного ключа, і якщо підпис не збігається з очікуваним, бюлетень буде відхилено.

Чи може хтось (інший виборець, ВК, стороння людина) таємно змінити голос у бюлетені? Шифрування бюлетенів на багатьох рівнях і додавання цифрових підписів кожним виборцем робить неможливим зміну голосу після його подання. Кожен бюлетень підписується виборцем, і будь-яка спроба зміни його змісту призведе до невідповідності підпису, що буде одразу виявлено іншими учасниками. Тому змінити голос після його шифрування неможливо, а будь-яка спроба змінити дані призведе до виявлення шахрайства.

Чи може виборець перевірити, що його голос врахований при підведенні кінцевих результатів? Алгоритм дозволяє кожному виборцю переконатися, що його голос був врахований. Оскільки бюлетені шифруються з використанням унікальних ключів і підписуються на кожному етапі, виборці можуть перевірити свої підписи на бюлетенях під час підрахунку голосів. Це гарантує, що жоден голос не буде загублено або змінено, а виборець може бути впевнений, що його голос врахований при підведенні підсумків голосування.

ВИСНОВОК

У результаті виконання лабораторної роботи було досліджено протокол е-голосування з перемішуванням, що забезпечує анонімність і надійність голосування без необхідності в довіреній третій стороні, як-от виборча комісія. Протокол базується на криптографічних алгоритмах RSA та Ель-Гамаля, які використовуються для шифрування та підпису бюлетенів. Під час реалізації було розглянуто кілька ключових аспектів, таких як анонімність виборців, захист від повторного голосування та можливість виявлення шахрайства.

Протокол гарантує, що кожен виборець може проголосувати лише один раз і що ніхто не може дізнатися, за кого був поданий голос. Система також унеможливлює підробку бюлетенів або зміни голосів без їх виявлення, що робить її надійним засобом для електронного голосування. Крім того, кожен виборець може переконатися, що його голос був правильно врахований у фінальному підрахунку.

Незважаючи на високу безпеку протоколу, його недоліком є складність реалізації в умовах великої кількості виборців, що може ускладнити масштабування для реальних виборів. Водночас протокол є важливим інструментом для забезпечення безпечного та прозорого е-голосування в умовах обмеженої кількості учасників.

ДОДАТОК А ТЕКСТИ ПРОГРАМНОГО КОДУ

Тексти програмного коду

(Найменування програми (документа))

Жорсткий диск

(Вид носія даних)

(Обсяг програми (документа), арк.)

Студентів групи ІП-11 4 курсу

Панченка С. В

Лисенка А. Ю.

```
mod rsa {
    use std::ops::Rem;
    use derive_more::Deref;
    use getset::Getters;
    use lazy_static::lazy_static;
    use num_bigint::{BigUint};
    use num_traits::{One, ToPrimitive, Zero};
    use rand::Rng;
    use serde::{Deserialize, Serialize};
    use thiserror::Error;

    #[derive(Error, Debug)]
    #[error("Failed convert to byte: {0}")]
    pub struct ConvertToBytesError(BigUint);

    pub fn convert_to_bytes(i: &[BigUint]) -> Result<Vec<u8>,
ConvertToBytesError> {
        i.iter()
            .try_fold(Vec::new(), |mut acc, unit| {
                unit.to_u8().ok_or_else(|| {
                    ConvertToBytesError(unit.clone())
                }).map(|byte| {
                    acc.push(byte);
                    acc
                })
            })
    }

    const MIN_GENERATED_NUMBER: u32 = u16::MAX as u32;
    const MAX_GENERATED_NUMBER: u32 = u32::MAX;

    pub fn generate_num() -> BigUint {

BigUint::from(rand::thread_rng().gen_range(MIN_GENERATED_NUMBER..MAX_GENERATED_N
UMBER))
    }

    fn generate_num_by_condition(predicate: fn(&BigUint) -> bool) -> BigUint {
        loop {
            let num = generate_num();
            if predicate(&num) {
                return num;
            }
        }
```

```
}

lazy_static! {
    static ref TWO: BigUint = BigUint::from(2u32);
}

fn is_prime(n: &BigUint) -> bool {
    if *n < *TWO {
        return false;
    }
    let mut i = TWO.clone();
    let n_sqrt = n.sqrt();
    while i < n_sqrt {
        if n.clone().rem(&i).is_zero() {
            return false;
        }
        i += BigUint::one();
    }
    true
}

fn generate_prime() -> BigUint {
    generate_num_by_condition(is_prime)
}

lazy_static! {
    pub static ref PUBLIC_NUMBER: BigUint = BigUint::from(65537u32);
}

#[derive(Debug, Clone, Deref, Serialize, Deserialize)]
pub struct ProductNumber(BigUint);

#[derive(Debug, Getters)]
pub struct PrivateKeyRef<'a> {
    private_number: &'a BigUint,
    #[get = "pub with_prefix"]
    product_number: &'a ProductNumber
}

#[derive(Debug, Getters)]
pub struct PublicKeyRef<'a> {
    #[get = "pub with_prefix"]
    product_number: &'a ProductNumber
}
```

```
    #[derive(Debug, Clone, Getters)]
    pub struct KeyPair {
        #[get = "pub with_prefix"]
        product_number: ProductNumber,
        private_number: BigUint,
    }

    impl Default for KeyPair {
        fn default() -> Self {
            DEFAULT_KEY_PAIR.clone()
        }
    }

    lazy_static! {
        pub static ref DEFAULT_KEY_PAIR: KeyPair = KeyPair::new();
    }

    #[derive(Error, Debug)]
    #[error("Base {base} is bigger or equal to {modulus}")]
    pub struct ModPowError {
        base: BigUint,
        modulus: BigUint,
    }

    fn modpow(base: &BigUint, exp: &BigUint, modulus: &BigUint) ->
Result<BigUint, ModPowError> {
        if base >= modulus {
            Err(ModPowError{base: base.clone(), modulus: modulus.clone()})
        } else {
            Ok(base.modpow(exp, modulus))
        }
    }

    impl KeyPair {
        pub fn new() -> Self {
            loop {
                let p = generate_prime();
                let q = generate_prime();
                let n = ProductNumber(p.clone() * q.clone());
                let phi = (p.clone() - BigUint::one()) * (q.clone() -
BigUint::one());
                if *PUBLIC_NUMBER < phi {
                    if let Some(d) = PUBLIC_NUMBER.modinv(&phi) {
```

```
                        return Self { product_number: n, private_number: d }
                    }
                }
            }
        }

        pub fn get_private_key_ref(&self) -> PrivateKeyRef {
            PrivateKeyRef { product_number: &self.product_number,
private_number: &self.private_number }
        }

        pub fn get_public_key_ref(&self) -> PublicKeyRef {
            PublicKeyRef { product_number: &self.product_number }
        }
    }

    pub trait KeyRef {
        fn get_parts(&self) -> (&BigUint, &ProductNumber);
    }

    impl<'a> KeyRef for PrivateKeyRef<'a> {
        fn get_parts(&self) -> (&BigUint, &ProductNumber) {
            (self.private_number, self.product_number)
        }
    }

    impl<'a> KeyRef for PublicKeyRef<'a> {
        fn get_parts(&self) -> (&BigUint, &ProductNumber) {
            (&PUBLIC_NUMBER, self.product_number)
        }
    }

    #[derive(Error, Debug)]
    #[error(transparent)]
    pub struct CipherDataError(#[from] ModPowError);

    pub fn cipher_data_biguint(key: &impl KeyRef, mut data: Vec<BigUint>) ->
Result<Vec<BigUint>, CipherDataError> {
        let (part, product_number) = key.get_parts();
        for c in &mut data {
            *c = modpow(c, part, product_number)?;
        }
        Ok(data)
    }
```

```
    pub fn cipher_data_u8(key: &impl KeyRef, data: &[u8]) ->
Result<Vec<BigUint>, CipherDataError> {
        let (part, product_number) = key.get_parts();
        data.iter().try_fold(Vec::new(), |mut acc, byte| {
            acc.push(modpow(&BigUint::from(*byte), part, product_number)?);
            Ok(acc)
        })
    }

    #[cfg(test)]
    mod tests {
        use num_traits::ToPrimitive;
        use super::{cipher_data_biguint, cipher_data_u8, KeyPair};

        #[test]
        fn test_ciphering() {
            let r = KeyPair::new();
            let message = "message";
            let ciphered_data = cipher_data_u8(
                &r.get_public_key_ref(), message.as_bytes()
            ).unwrap();
            let deciphered_data = cipher_data_biguint(
                &r.get_private_key_ref(), ciphered_data
            ).unwrap().into_iter().map(|e| e.to_u8().unwrap()).collect();
            let deciphered_message =
String::from_utf8(deciphered_data).unwrap();
            assert_eq!(deciphered_message, message);
            println!("{}", deciphered_message);
        }
    }
}

mod elgamal {
    use std::hash::{DefaultHasher, Hasher};
    use std::ops::Rem;
    use num_bigint::BigUint;
    use num_traits::{Pow, ToPrimitive};
    use serde::{Deserialize, Serialize};
    use thiserror::Error;
    use num_prime::nt_funcs::is_prime64;
    use rand::distributions::uniform::SampleRange;
    use rand::Rng;
```

```
fn positive_mod(a: i64, n: i64) -> i64 {
    ((a % n) + n) % n
}

#[derive(Error, Debug)]
#[error("Can not get inverse: {z}, {a}")]
pub struct ModInvError { z: u16, a: u16 }

pub fn modinv(z: u16, a: u16) -> Result<u16, ModInvError> {

    if z == 0 {
        return Err(ModInvError{z, a});
    }

    if a == 0 {
        return Err(ModInvError{z, a});
    }

    if z >= a {
        return Err(ModInvError{z, a});
    }

    let mut i = a;
    let mut j = z;
    let mut y_2: i64 = 0;
    let mut y_1: i64 = 1;

    while j > 0 {
        let quotient = i / j;
        let remainder = i % j;
        let y = y_2 - (y_1 * quotient as i64);
        i = j;
        j = remainder;
        y_2 = y_1;
        y_1 = y;
    }
    if i != 1 {
        return Err(ModInvError{z, a});
    }
    Ok(positive_mod(y_2, a as i64) as u16)
}

pub fn modpow(base: u64, exp: u64, modulus: u64) -> u64 {
    let mut result = 1;
```

```
    let mut base = base % modulus;
    let mut exp = exp;

    while exp > 0 {
        if exp % 2 == 1 {
            result = (result * base) % modulus;
        }
        exp >>= 1;
        base = (base * base) % modulus;
    }

    result
}

pub fn gen_prime<R: SampleRange<u16> + Clone>(r: R) -> u16 {
    loop {
        let n = rand::thread_rng().gen_range(r.clone());
        if is_prime64(n as u64) {
            break n;
        }
    }
}

#[derive(Serialize, Deserialize)]
pub struct CipheredData {
    pub a: u16,
    pub bs: Vec<u16>
}

const MIN_P: u16 = u8::MAX as u16 + 1;
const MAX_P: u16 = u16::MAX;

#[derive(Serialize, Deserialize, Clone)]
pub struct PublicKey {
    g: u16,
    y: u16,
    p: u16,
}

fn generate_k(p: u16) -> u16 {
    rand::thread_rng().gen_range(1..(p - 1))
}

impl PublicKey {
```

```
        pub fn cipher(&self, data: &[u8]) -> CipheredData {
            let (a, u) = {
                let k = generate_k(self.p);
                (
                    modpow(self.g as u64, k as u64, self.p as u64) as u16,
                    modpow(self.y as u64, k as u64, self.p as u64) as u16
                )
            };
            CipheredData {
                a,
                bs: data.iter()
                    .map(|m| {
                        ((u as u32 * (*m as u32)) % self.p as u32) as u16
                    })
                    .collect()
            }
        }

        pub fn verify(&self, data: &[u8], signature: &Signature) -> bool {
            let y = BigUint::from(self.y);
            let r = BigUint::from(signature.r);
            let left = (y.pow(&r) *
r.pow(signature.s)).rem(BigUint::from(self.p)).to_u16().unwrap();

            let m = calculate_hash(data);
            let right = modpow(self.g as u64, m as u64, self.p as u64) as u16;

            let is_valid = left == right;
            is_valid
        }
    }

    #[derive(Error, Debug)]
    pub enum DecipherError {
        #[error("Can not create inverse")]
        ModInv(ModInvError),
        #[error("Can not convert to byte")]
        CanNotConvertToByte(u16),
    }

    #[derive(Clone)]
    pub struct PrivateKey {
        p: u16,
        g: u16,
```

```
        y: u16,
        x: u16,
    }

    fn calculate_hash(data: &[u8]) -> u16 {
        let mut hasher = DefaultHasher::new();
        hasher.write(data);
        hasher.finish() as u16
    }

    pub struct Signature{ r: u16, s: u16 }

    impl PrivateKey {
        pub fn decipher(&self, c_data: &CipheredData) -> Result<Vec<u8>,
DecipherError> {
            let s_inv = {
                let s = modpow(c_data.a as u64, self.x as u64, self.p as u64) as
u16;
                modinv(s, self.p).map_err(DecipherError::ModInv)?
            };
            c_data.bs.iter().try_fold(Vec::new(), |mut acc, c| {
                let prob_byte = ((*c as u32 * s_inv as u32) % self.p as u32) as
u16;
                let byte =
prob_byte.to_u8().ok_or(DecipherError::CanNotConvertToByte(prob_byte))?;
                acc.push(byte);
                Ok(acc)
            })
        }

        pub fn sign(&self, data: &[u8]) -> Signature {
            let m = calculate_hash(data);
            let (k, k_inv) = loop {
                let k = generate_k(self.p);
                if let Ok(k_inv) = modinv(k, self.p - 1) {
                    break (k, k_inv);
                }
            };
            let r = modpow(self.g as u64, k as u64, self.p as u64) as i64;
            let s = positive_mod ((m as i64 - (self.x as i64) * r) * (k_inv as
i64), self.p as i64 - 1);
            Signature { r: r as u16, s: s as u16 }
        }
    }
```

```
    #[derive(Clone)]
    pub struct KeyPair {
        pub private_key: PrivateKey,
        pub public_key: PublicKey,
    }

    impl Default for KeyPair {
        fn default() -> Self {
            let p = gen_prime(MIN_P..MAX_P);
            let g = gen_prime(1..p);
            let x = rand::thread_rng().gen_range(1..(p - 2));
            let y = modpow(g as u64, x as u64, p as u64) as u16;
            Self { public_key: PublicKey { g, y, p }, private_key: PrivateKey
{ p, g, y, x } }
        }
    }

    #[cfg(test)]
    mod tests {

        #[test]
        fn it_works() {
            let message = "message";
            let key_pair = super::KeyPair::default();
            let c_data = key_pair.public_key.cipher(message.as_bytes());
            let data = key_pair.private_key.decipher(&c_data).unwrap();
            let deciphered_message = String::from_utf8(data).unwrap();
            assert_eq!(message, deciphered_message);
        }

        #[test]
        fn test_modinv() {
            let vals = [(23, 6577), (10, 7919), (17, 3181)];
            for (z, a) in vals {
                let res = super::modinv(z, a).unwrap();
                let b = z * res;
                let c = b % a;
                assert_eq!(c, 1);
            }
        }

        #[test]
        fn sign_verify() {
```

```
            let message = "message";
            let key_pair = super::KeyPair::default();
            let signature = key_pair.private_key.sign(message.as_bytes());
            let is_valid = key_pair.public_key.verify(message.as_ref(),
&signature);
            assert!(is_valid);
        }
    }
}

mod voter {
    use std::mem;
    use num_bigint::BigUint;
    use num_traits::{FromPrimitive, ToPrimitive};
    use rand::prelude::IteratorRandom;
    use rand::Rng;
    use serde::{Deserialize, Serialize};
    use thiserror::Error;
    use crate::{elgamal, rsa};

    #[derive(Default, Clone)]
    struct FirstStageNoise(Vec<BigUint>);

    #[derive(Default, Clone)]
    struct SecondStageNoise(Vec<BigUint>);

    #[derive(Default, Clone)]
    struct SecondStageNoiseRow(Vec<SecondStageNoise>);

    #[derive(Default, Clone)]
    struct SecondStageNoiseTable(Vec<SecondStageNoiseRow>);

    #[derive(Default, Clone)]
    struct Voter {
        rsa: rsa::KeyPair,
        elgamal: elgamal::KeyPair,
    }

    fn create_noise(noise_length: usize) -> Vec<BigUint> {
        (0..noise_length).into_iter().map(|_| {

BigUint::from_u8(rand::thread_rng().gen_range(0u8..u8::MAX)).unwrap()
        }).collect::<Vec<_>>()
    }
```

```
    #[derive(Default)]
    struct SecondStageCipherData(Vec<BigUint>);

    #[derive(Serialize, Deserialize, Debug, Clone, Eq, PartialEq, Hash)]
    pub struct Candidate(pub String);

    pub fn sort_voters(voters: &mut Vec<Voter>) {
        voters.sort_by(|a, b| {
            let a_p = a.rsa.get_product_number();
            let b_p = b.rsa.get_product_number();
            a_p.cmp(&b_p)
        });
    }

    #[derive(Error, Debug)]
    pub enum CipherError {
        #[error("Can not chose candidate")]
        CanNotChooseCandidate,
        #[error(transparent)]
        SerializeCandidate(bincode::Error),
        #[error(transparent)]
        FirstStageCipherData(rsa::CipherDataError),
        #[error(transparent)]
        SecondStageCipherData(rsa::CipherDataError),
    }

    #[derive(Serialize, Deserialize)]
    struct FirstStageCipherData(Vec<BigUint>);

    fn cipher(
        candidates: &[Candidate],
        voters: &[Voter],
        noise_length: usize,
        mut second_stage_noise_table: SecondStageNoiseTable,
    ) -> Result<(FirstStageNoise, SecondStageNoiseTable, SecondStageCipherData),
CipherError> {
        let pub_keys = voters.iter().map(|v| v.rsa.get_public_key_ref());
        let (first_stage_noise, mut ser_first_stage_cipher_data) = {
            let (first_stage_noise, first_stage_cipher_data) = {
                let (cipher_data, first_stage_noise) = {
                    let candidate = candidates.iter().choose(&mut
rand::thread_rng())
                        .ok_or(CipherError::CanNotChooseCandidate)?;
```

```
                    let first_stage_noise =
FirstStageNoise(create_noise(noise_length));
                    (
                        bincode::serialize(&candidate)
                            .map_err(CipherError::SerializeCandidate)?
                            .into_iter()
                            .map(|v| BigUint::from_u8(v).unwrap())
                            .chain(first_stage_noise.0.iter().map(Clone::clone))
                            .collect::<Vec<_>>(),
                        first_stage_noise
                    )
                };
                (
                    first_stage_noise,
                    FirstStageCipherData(
                        pub_keys.clone()
                            .try_fold(cipher_data, |acc, pub_key_ref| {
                                rsa::cipher_data_biguint(&pub_key_ref, acc)
                                    .map_err(CipherError::FirstStageCipherData)
                            })?
                    )
                )
            };
            (
                first_stage_noise,
                bincode::serialize(&first_stage_cipher_data)
                    .map_err(CipherError::SerializeCandidate)?
                    .into_iter()
                    .map(|v| BigUint::from_u8(v).unwrap())
                    .collect::<Vec<BigUint>>()
            )
        };

        for (pub_key, second_stage_noise_row) in
pub_keys.zip(second_stage_noise_table.0.iter_mut()) {
            let noise = create_noise(noise_length);
            ser_first_stage_cipher_data.extend(noise.iter().map(Clone::clone));
            second_stage_noise_row.0.push(SecondStageNoise(noise));
            ser_first_stage_cipher_data = rsa::cipher_data_biguint(&pub_key,
mem::take(&mut ser_first_stage_cipher_data))
                .map_err(CipherError::SecondStageCipherData)?;
        }

        Ok((first_stage_noise, second_stage_noise_table,
```

```
SecondStageCipherData(ser_first_stage_cipher_data)))
    }
    #[derive(Error, Debug)]
    pub enum DecipherSecondStageError {
        #[error("Second stage noise row is empty")]
        SecondStageNoiseRowEmpty,
        #[error(transparent)]
        CipherData(rsa::CipherDataError),
        #[error(transparent)]
        RemoveNoise(RemoveNoiseError),
        #[error("Can not convert to byte: {0}")]
        CantConvertToByte(BigUint),
        #[error(transparent)]
        DeserializeFirstStageCipherData(bincode::Error),
    }

    #[derive(Error, Debug)]
    pub enum RemoveNoiseError {
        #[error("Decrypted data {data_len} is shorter than noise {noise_len}")]
        DataShorterThanNoise { data_len: usize, noise_len: usize },
        #[error("Noises does not match: {0}, {1}")]
        NoisesMismatch(BigUint, BigUint),
    }

    fn remove_noise(data: Vec<BigUint>, noise: &[BigUint]) ->
Result<Vec<BigUint>, RemoveNoiseError> {
        if data.len() < noise.len() {
            Err(RemoveNoiseError::DataShorterThanNoise {
                data_len: data.len(), noise_len: noise.len()
            })
        } else {
            for (a, b) in noise.iter().rev().zip(data.iter().rev()) {
                if a != b {
                    return Err(RemoveNoiseError::NoisesMismatch(a.clone(),
b.clone()));
                }
            }
            Ok(data.into_iter().rev().skip(noise.len()).rev().collect())
        }
    }

    fn decipher_second_stage(
        voters: &[Voter],
        second_stage_cipher_data_vec: Vec<SecondStageCipherData>,
```

```
        mut second_stage_noise_table: SecondStageNoiseTable,
    ) -> Result<Vec<FirstStageCipherData>, DecipherSecondStageError> {
        let private_keys = voters.iter().map(|v| v.rsa.get_private_key_ref());

        second_stage_cipher_data_vec.into_iter().rev()
            .try_fold(Vec::new(), |mut acc, second_stage_cipher_data| {
                let data = private_keys.clone().zip(
                    second_stage_noise_table.0.iter_mut()
                ).rev().try_fold(second_stage_cipher_data.0, |data,
(private_key, row)| {

row.0.pop().ok_or(DecipherSecondStageError::SecondStageNoiseRowEmpty)
                        .and_then(|noise| {
                            let decrypted_data =
rsa::cipher_data_biguint(&private_key, data)
                                .map_err(DecipherSecondStageError::CipherData)?;
                            remove_noise(decrypted_data, &noise.0)
                                .map_err(DecipherSecondStageError::RemoveNoise)
                        })
                })?;

                let byte_data = data.into_iter().try_fold(Vec::new(), |mut acc,
v| {
                    let byte =
v.to_u8().ok_or(DecipherSecondStageError::CantConvertToByte(v.clone()))?;
                    acc.push(byte);
                    Ok(acc)
                })?;
                acc.push(
                    bincode::deserialize::<FirstStageCipherData>(&byte_data)
                        .map_err(DecipherSecondStageError::DeserializeFirstStage
CipherData)?
                );
                Ok(acc)
            })
    }

    fn decipher_sign_first_stage(
        voters: &[Voter],
        mut first_stage_cipher_data_vec: Vec<FirstStageCipherData>,
        first_stage_noise_vec: Vec<FirstStageNoise>
    ) -> Vec<Candidate> {
        for v in voters.iter().rev() {
            for cipher_data in &mut first_stage_cipher_data_vec {
```

```
                cipher_data.0 = rsa::cipher_data_biguint
                    (&v.rsa.get_private_key_ref(), mem::take(&mut cipher_data.0)
                ).expect("Can not cipher data");
                let byte_data = bincode::serialize(&cipher_data.0).unwrap();
                let signature = v.elgamal.private_key.sign(&byte_data);
                assert!(
                    v.elgamal.public_key.verify(&byte_data, &signature),
                    "Invalid signature"
                );
            }
        }

first_stage_cipher_data_vec.into_iter().rev().zip(first_stage_noise_vec.into_ite
r())
            .fold(Vec::new(), |mut acc, (first_stage_cipher_data, noise)| {
                let data = remove_noise(first_stage_cipher_data.0, &noise.0)
                    .expect("Can not remove noise")
                    .into_iter()
                    .map(|v| v.to_u8().expect("Can not convert to byte"))
                    .collect::<Vec<_>>();
                let candidate = bincode::deserialize::<Candidate>(&data)
                    .expect("Can not deserialize to candidate");
                acc.push(candidate);
                acc
            })
    }

    #[cfg(test)]
    mod tests {
        use std::collections::HashMap;
        use std::mem;
        use crate::voter::{cipher, decipher_second_stage,
decipher_sign_first_stage, sort_voters, Candidate, SecondStageNoiseTable,
Voter};

        #[test]
        fn it_works() {
            let noise_length = 5usize;

            let mut voters = (0..15).into_iter().map(|_| Voter::default())
                .collect::<Vec<_>>();
            sort_voters(&mut voters);
            let candidates = (0..5).into_iter().map(|i|
Candidate(i.to_string()))
```

```
                .collect::<Vec<_>>();

            let (first_stage_noises, mut second_stage_noise_table,
second_stage_cipher_data_vec) = {
                let mut first_stage_noises = Vec::new();
                let mut second_stage_noises_table = SecondStageNoiseTable(
                    vec![Default::default(); voters.len()]
                );
                let mut second_stage_cipher_data_vec = Vec::new();

                for _ in &voters {
                    let res = cipher(
                        &candidates, &voters,
                        noise_length, mem::take(&mut second_stage_noises_table)
                    ).unwrap();
                    first_stage_noises.push(res.0);
                    second_stage_noises_table = res.1;
                    second_stage_cipher_data_vec.push(res.2);
                }
                (first_stage_noises, second_stage_noises_table,
second_stage_cipher_data_vec)
            };

            let first_stage_cipher_vec = decipher_second_stage(
                &voters, second_stage_cipher_data_vec, second_stage_noise_table
            ).unwrap();

            let votes = decipher_sign_first_stage(&voters,
first_stage_cipher_vec, first_stage_noises);

            let mut candidates_votes: HashMap<Candidate, u64> =
HashMap::from_iter(
                candidates.into_iter().map(|candidate| (candidate, 0u64))
            );

            for v in votes {
                *candidates_votes.get_mut(&v).unwrap() += 1;
            }

            println!("{:?}", candidates_votes);
        }
    }

}
```