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 №6

Протоколи й алгоритми електронного голосування

Тема: Протокол Е-голосування без підтвердження

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

Перевірив:
Нестерук А. О.

Київ 2024

ЗМІСТ

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ

Дослідити протокол Е-голосування без підтвердження.

2 ЗАВДАННЯ

Змоделювати протокол Е-голосування без підтвердження будь-якою мовою програмування та провести його дослідження. Для кодування повідомлень використовувати метод Ель-Гамаля, для кодування Е-бюлетеня використовувати метод BBS.

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 sim_env моделює середовище голосування, яке складається з кількох компонентів, включаючи реєстраційне бюро (reg_bureau), виборчу комісію (ec) та виборців (voters). Підмодуль reg_bureau відповідає за генерацію унікальних ID для виборців і зберігає публічні токени для кожного виборця. Підмодуль ec реалізує функції виборчої комісії, яка приймає зашифровані голоси та веде підрахунок голосів за кандидатів. Тут використовується ElGamal для шифрування голосів, а BBS для анонімного передачі даних між виборцями та виборчою комісією. Підмодуль voters реалізує процес голосування: виборці обирають кандидатів, шифрують свої голоси за допомогою отриманих токенів і надсилають їх виборчій комісії для підрахунку.

Тестові сценарії у модулі sim_env перевіряють процес голосування від реєстрації виборців до підрахунку голосів.

4 ДЕМОНСТРАЦІЯ РОБОТИ ПРОТОКОЛУ

Тест у модулі sim_env демонструє повну симуляцію процесу голосування з використанням протоколу е-голосування, де задіяні реєстраційне бюро (reg_bureau), виборча комісія (ec) і виборці (voters). Основне завдання тесту — перевірити, чи коректно працює система від початку до кінця, забезпечуючи безпеку, анонімність і правильний підрахунок голосів.

Спочатку тест викликає функцію ec::Ec::accept_ids_generate_tokens, яка приймає унікальні ідентифікатори (ID) від реєстраційного бюро. Реєстраційне бюро генерує ці ID за допомогою функції send_ids, яка забезпечує унікальність ідентифікаторів для кожного виборця. Для кожного виборця створюються публічні токени з використанням BBS (Blum-Blum-Shub). Токени мають публічні ключі, які будуть використовуватися виборцями для шифрування голосів, тоді як приватні ключі залишаються в комісії для дешифрування голосів після голосування.

Далі в тесті викликається функція voters::vote, яка симулює процес голосування. Кожен виборець отримує публічний токен від реєстраційного бюро і використовує його для шифрування свого голосу. Виборці обирають випадкового кандидата з набору доступних кандидатів, шифрують свій вибір за допомогою публічного ключа BBS, і шифрують дані ще раз, використовуючи публічний ключ ElGamal, який є загальним для всієї виборчої комісії. Це забезпечує додатковий рівень безпеки. Після шифрування голос надсилається до виборчої комісії.

Виборча комісія приймає ці зашифровані голоси через функцію accept_vote, розшифровує їх, використовуючи свої приватні ключі BBS, і підраховує кількість голосів за кожного кандидата. Комісія спочатку дешифровує зовнішній шар шифрування за допомогою приватного ключа ElGamal, потім використовує приватний ключ BBS для дешифрування внутрішніх зашифрованих даних, що дозволяє визначити голос виборця.

Після завершення голосування результати підрахунку голосів виводяться на екран. Тест показує кількість голосів за кожного кандидата, що дозволяє

перевірити правильність підрахунку і дешифрування голосів. Результати є важливим індикатором того, що процес шифрування, дешифрування і підрахунку голосів був виконаний коректно.

Таким чином, тест на рисунку 4.1 симулює повний цикл голосування: від реєстрації виборців у реєстраційному бюро, через анонімне шифрування і передачу голосів до виборчої комісії, і до завершального етапу — підрахунку голосів. Це демонструє роботу всіх компонентів протоколу е-голосування і забезпечує, що система дотримується вимог безпеки, анонімності і точності підрахунку голосів.

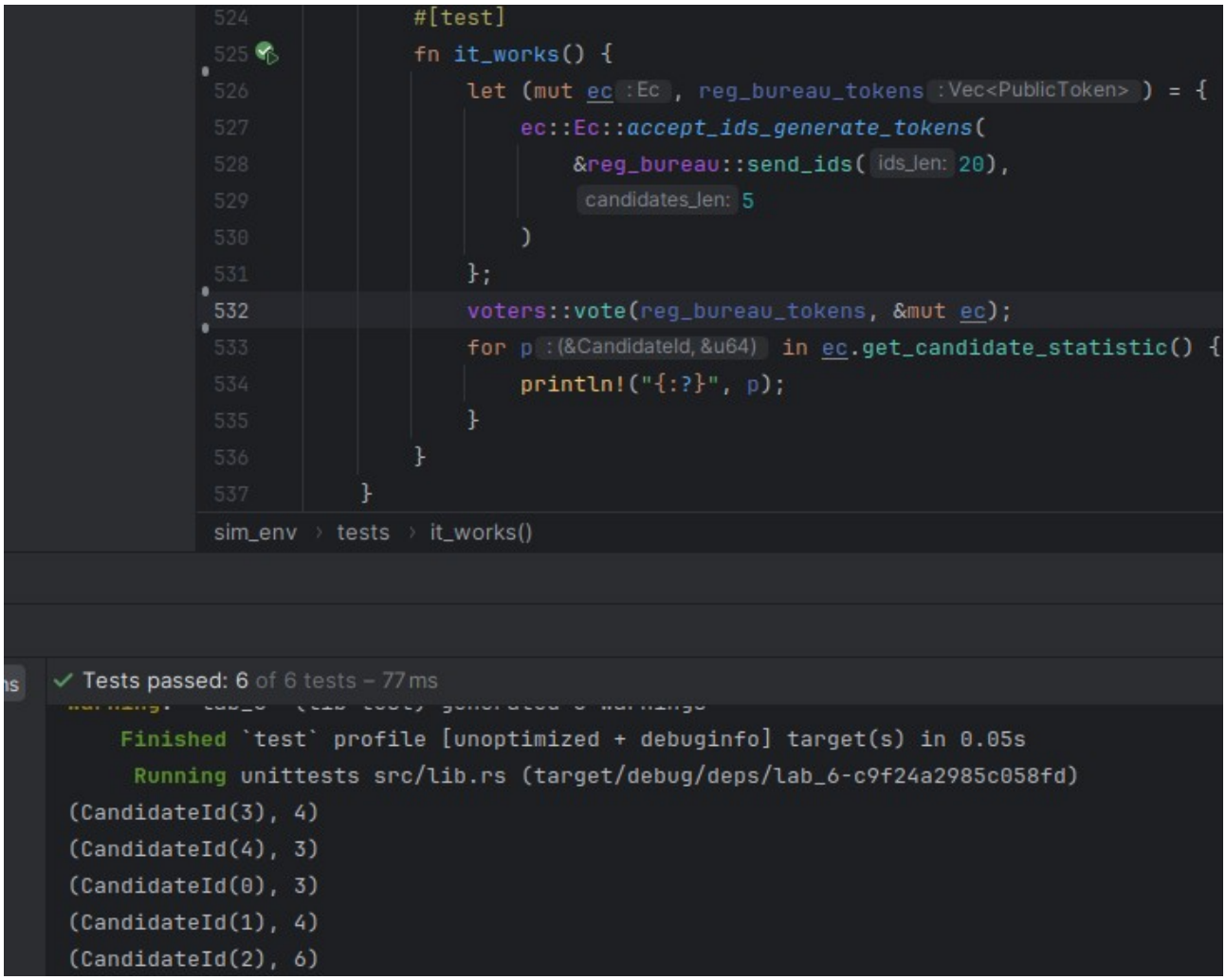

Рисунок 4.1 — Тестування процесу голосування

5 ДОСЛІДЖЕННЯ ПРОТОКОЛУ

Чи можуть голосувати ті, хто не має на це права? Протокол е-голосування гарантує, що голосувати можуть тільки ті виборці, які були зареєстровані через реєстраційне бюро. Це досягається шляхом генерації унікальних ID для кожного виборця, а також видачі їм публічних токенів. Токени пов’язані з конкретним виборцем, і тільки ті, хто має дійсний токен, можуть подати свій голос. Виборча комісія приймає лише ті голоси, які належать авторизованим виборцям, що виключає можливість голосування сторонніми особами.

Чи може виборець голосувати кілька разів? Протокол забороняє подвійне голосування, оскільки кожен виборець отримує унікальний токен, який може бути використаний лише один раз для подання голосу. Після того як голос виборця зашифрований і відправлений до виборчої комісії, цей токен більше не може бути використаний. Якщо виборець спробує повторно проголосувати, комісія відхилить голос, оскільки токен уже був використаний, що гарантує, що кожен виборець може проголосувати тільки один раз.

Чи може хтось (інший виборець, ВК, стороння людина) дізнатися, за кого проголосували інші виборці? Протокол забезпечує високий рівень анонімності голосів через використання двоетапного шифрування. Кожен голос шифрується спочатку публічним ключем BBS, а потім публічним ключем ElGamal, який є загальним для всієї виборчої комісії. Навіть виборча комісія не може дізнатися, за кого конкретно проголосував виборець, поки не буде проведено остаточне розшифрування всіх голосів після завершення голосування. Це унеможливлює визначення особи, за кого проголосував конкретний виборець, іншими виборцями, членами комісії або сторонніми людьми.

Чи може інший виборець чи стороння людина проголосувати замість іншого зареєстрованого виборця? Кожен виборець використовує унікальний публічний токен і приватний ключ BBS, що захищає від можливості проголосувати за когось іншого. Якщо хтось спробує використати чужий токен або ID, система виявить це, оскільки токен не буде відповідати відповідному

приватному ключу. Виборча комісія перевіряє автентичність кожного голосу за допомогою приватних ключів, що робить неможливим голосування замість іншого виборця або підробку його голосу.

Чи може хтось (інший виборець, ВК, стороння людина) таємно змінити голос у бюлетені? Протокол захищає від зміни голосів за рахунок використання криптографічних алгоритмів. Після того як голос виборця шифрується за допомогою токену BBS і публічного ключа ElGamal, ніхто не може змінити його без доступу до приватних ключів виборчої комісії. Будь-які спроби змінити шифр або дані голосу призведуть до того, що голос стане недійсним, оскільки система виявить некоректність шифрування. Це забезпечує цілісність голосів і запобігає їх таємній зміні сторонніми особами.

Чи може виборець перевірити, що його голос врахований при підведенні кінцевих результатів? Протокол свідомо не дозволяє виборцю перевірити, що його голос був врахований у підсумках голосування. Це зроблено для того, щоб уникнути можливості підкупу або примусу виборців. Якщо б виборці могли підтвердити свій голос, це могло б створити можливість шантажу або вимагання від них підтвердження голосу на користь певного кандидата. Анонімність голосів зберігається на всіх етапах, і виборці не можуть перевірити, чи врахований їхній голос, що захищає їх від зовнішнього тиску та зловживань.

ВИСНОВОК

У результаті виконання лабораторної роботи було досліджено та реалізовано протокол е-голосування, що забезпечує анонімність виборців та безпеку голосування. Протокол ефективно захищає виборчий процес від несанкціонованого доступу, підробки голосів і спроб дізнатися вибір конкретних виборців. Завдяки використанню криптографічних алгоритмів ElGamal та BBS гарантується, що лише зареєстровані виборці можуть голосувати, кожен виборець може проголосувати лише один раз, а голоси залишаються таємними до кінця виборчого процесу.

Протокол також забезпечує захист від підкупу або примусу виборців, оскільки виборець не може перевірити або довести, що його голос був врахований, що запобігає можливості шантажу. Крім того, двоетапне шифрування голосів забезпечує надійний захист від зміни голосів сторонніми особами.

Протокол демонструє високий рівень безпеки і конфіденційності, що робить його ефективним інструментом для проведення електронних голосувань у сучасному цифровому суспільстві. Водночас, подальші дослідження можуть бути спрямовані на підвищення прозорості та масштабованості системи для використання в реальних умовах.

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 alg_utils {
    use num_prime::nt_funcs::is_prime64;
    use rand::distributions::uniform::SampleRange;
    use rand::Rng;
    use thiserror::Error;

    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;
            }
        }
    }

    pub 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)
    }
}

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 rand::Rng;
    use crate::alg_utils;
    use crate::alg_utils::{modinv, positive_mod, ModInvError};

    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
    }

    #[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,
        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 = alg_utils::gen_prime(MIN_P..MAX_P);
            let g = alg_utils::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, 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 bbs {
    use getset::Getters;
    use num_integer::{lcm, gcd};
    use rand::Rng;
    use crate::alg_utils::{gen_prime};
    use crate::elgamal::{modpow};

    #[derive(Getters)]
    #[get = "pub with_prefix"]
    pub struct KeyPair {
        public_key: PublicKey,
        private_key: PrivateKey,
    }

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

```
    q: u16,
}

#[derive(Copy, Clone)]
pub struct PublicKey(u32);

impl KeyPair {

    pub fn new() -> Self {
        fn gen_prime_34() -> u16 {
            loop {
                let p = gen_prime(u8::MAX as u16..u16::MAX);
                if p % 4 == 3 {
                    break p;
                }
            }
        }

        let p = gen_prime_34();
        let q = gen_prime_34();
        Self {
            public_key: PublicKey((p as u32) * (q as u32)),
            private_key: PrivateKey{ p, q },
        }
    }
}

macro_rules! get_last_bit {
    ($v:expr) => {
        $v & 1 == 1
    };
}

fn u8_to_bits(mut v: u8) -> [bool; 8] {
    std::array::from_fn(|_| {
        let b = get_last_bit!(v);
        v >>= 1;
        b
    })
}

fn bits_to_u8(bits: [bool; 8]) -> u8 {
    let mut a: u8 = 0;
    for (i, b) in bits.iter().enumerate() {
```

```
        let ba = (*b as u8) << (i as u8);
        a |= ba;
    }
    a
}

impl PublicKey {
    pub fn cipher(&self, data: &mut [u8]) -> u32 {
        let x = loop {
            let x = rand::thread_rng().gen_range(0..self.0);
            if gcd(x as u64, self.0 as u64) == 1 {
                break x;
            }
        };
        let next_x = |x: u64| {
            ((x * x) % (self.0 as u64)) as u32
        };
        let x_0 = next_x(x as u64);
        let mut x = x_0;
        data.iter_mut().for_each(|v| {
            let bits = u8_to_bits(*v);
            let xor_bits: [bool; 8] = std::array::from_fn(|i| {
                let b = bits[i] ^ get_last_bit!(x);
                x = next_x(x as u64);
                b
            });
            *v = bits_to_u8(xor_bits);
        });
        x_0
    }
}

impl PrivateKey {
    pub fn decipher(&self, data: &mut [u8], x_0: u32) {
        let lcm_lam = lcm((self.p - 1) as u32, (self.q - 1) as u32);
        let n = self.p as u32 * self.q as u32;
        let next_x = |i: usize| {
            let ppow = modpow(2, i as u64, lcm_lam as u64);
            modpow(x_0 as u64, ppow, n as u64) as u32
        };
        let mut x = next_x(0);
        let mut x_index = 1;
        data.iter_mut().for_each(|v| {
            let bits = u8_to_bits(*v);
```

```
                let xor_bits: [bool; 8] = std::array::from_fn(|i| {
                    let b = bits[i] ^ get_last_bit!(x);
                    x = next_x(x_index);
                    x_index += 1;
                    b
                });
                *v = bits_to_u8(xor_bits);
            });
        }
    }

    #[cfg(test)]
    mod tests {
        use crate::bbs::{bits_to_u8, u8_to_bits};

        #[test]
        fn bits_u8() {
            for v in 0..u8::MAX {
                assert_eq!(bits_to_u8(u8_to_bits(v)), v);
            }
        }

        #[test]
        fn it_works() {
            let message = "message";
            let mut c_message = bincode::serialize(&message).unwrap();
            let key_pair = super::KeyPair::new();
            let x_0 = key_pair.public_key.cipher(&mut c_message);
            key_pair.private_key.decipher(&mut c_message, x_0);
            let original_message: String =
bincode::deserialize(&c_message).unwrap();
            assert_eq!(original_message, message);
        }
    }
}

mod sim_env {

    mod reg_bureau {
        use rand::Rng;
        use serde::{Deserialize, Serialize};
        use crate::sim_env::ec::PublicToken;

        type InnerId = u16;
```

```
        #[derive(Serialize, Deserialize, Debug, Copy, Clone, PartialEq, Eq,
Hash)]
        pub struct Id(InnerId);


        pub struct RegBureau {
            public_tokens: Vec<PublicToken>
        }

        pub fn send_ids(ids_len: u16) -> Vec<Id> {
            let mut ids = Vec::with_capacity(ids_len as usize);
            for i in 0..ids_len {
                loop {
                    let id = Id(rand::thread_rng().gen_range(0..InnerId::MAX));
                    if !ids.contains(&id) {
                        ids.push(id);
                        break;
                    }
                }
            }
            ids
        }
    }

    mod ec {
        use std::collections::HashMap;
        use num_integer::Integer;
        use serde::{Deserialize, Serialize};
        use crate::{bbs, elgamal};
        use crate::elgamal::CipheredData;
        use crate::sim_env::reg_bureau::Id;
        use crate::sim_env::voters::VoteData;

        pub struct PublicToken {
            pub id: Id,
            pub public_key: bbs::PublicKey
        }

        #[derive(Debug, Serialize, Deserialize, PartialEq, Eq, Hash, Clone,
Copy)]
        pub struct CandidateId(u16);

        pub struct Ec {
```

```
        elgamal: elgamal::KeyPair,
        candidates: HashMap<CandidateId, u64>,
        private_tokens: HashMap<Id, bbs::PrivateKey>,

    }

    impl Ec {

        pub fn accept_ids_generate_tokens(
            ids: &[Id],
            candidates_len: u16
        ) -> (Self, Vec<PublicToken>) {
            let mut public_tokens = Vec::with_capacity(ids.len());
            let mut private_tokens = HashMap::with_capacity(ids.len());
            for id in ids {
                let key_pair = bbs::KeyPair::new();
                public_tokens.push(PublicToken {
                    id: *id,
                    public_key: *key_pair.get_public_key()
                });
                private_tokens.insert(*id, *key_pair.get_private_key());
            }
            (
                Self {
                    elgamal: elgamal::KeyPair::default(),
                    candidates: HashMap::from_iter(
                        (0..candidates_len).map(|i| (CandidateId(i), 0u64))
                    ),
                    private_tokens
                },
                public_tokens
            )
        }

        pub fn get_public_key(&self) -> &elgamal::PublicKey {
            &self.elgamal.public_key
        }

        pub fn get_candidates(&self) -> impl Iterator<Item=&CandidateId> {
            self.candidates.keys()
        }

        pub fn get_candidate_statistic(&self) -> impl
Iterator<Item=(&CandidateId, &u64)> {
```

```
                self.candidates.iter()
            }

            pub fn accept_vote(&mut self, ciphered_data: CipheredData) {
                let cand_id: CandidateId = {
                    let data =
self.elgamal.private_key.decipher(&ciphered_data).unwrap();
                    let mut vote_data: VoteData =
bincode::deserialize(&data).unwrap();
                    let private_key =
self.private_tokens.get(&vote_data.id).unwrap();
                    private_key.decipher(&mut vote_data.data, vote_data.x_0);
                    bincode::deserialize(&vote_data.data).unwrap()
                };
                self.candidates.get_mut(&cand_id).unwrap().inc();
            }
        }
    }

    mod voters {
        use rand::prelude::{IteratorRandom, SliceRandom};
        use serde::{Deserialize, Serialize};
        use crate::sim_env::ec;
        use crate::sim_env::ec::PublicToken;
        use crate::sim_env::reg_bureau::Id;

        #[derive(Debug, Serialize, Deserialize)]
        pub struct VoteData {
            pub id: Id,
            pub x_0: u32,
            pub data: Vec<u8>
        }

        pub fn vote(reg_bureau_tokens: Vec<PublicToken>, ec: &mut ec::Ec) {
            reg_bureau_tokens.into_iter().for_each(|token| {
                let candidate = ec.get_candidates()
                    .choose(&mut rand::thread_rng()).unwrap();
                let mut data = bincode::serialize(candidate).unwrap();

                let x_0 = token.public_key.cipher(&mut data);
                let ser_vote_data = bincode::serialize(
                    &VoteData { id: token.id, x_0, data }
                ).unwrap();
                let cipher_data = ec.get_public_key().cipher(&ser_vote_data);
```

```
                ec.accept_vote(cipher_data);
            });
        }
    }


    #[cfg(test)]
    mod tests {
        use crate::sim_env::{ec, reg_bureau, voters};

        #[test]
        fn it_works() {
            let (mut ec, reg_bureau_tokens) = {
                ec::Ec::accept_ids_generate_tokens(
                    &reg_bureau::send_ids(20),
                    5
                )
            };
            voters::vote(reg_bureau_tokens, &mut ec);
            for p in ec.get_candidate_statistic() {
                println!("{:?}", p);
            }
        }
    }
}
```