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 №3

## Протоколи й алгоритми електронного голосування

**Тема:** Протокол Е-голосування з двома виборчими комісіями

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 з двома виборчими комісіями

# 2 ЗАВДАННЯ

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

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 alg_utils забезпечує основні математичні операції, які часто використовуються у криптографії. Він включає функції для обчислення позитивного модуля (positive_mod), обчислення оберненого значення за модулем (modinv), модульного експоненціювання (modpow), а також генерації простих чисел у заданому діапазоні (gen_prime). Ці функції є важливими компонентами криптографічних алгоритмів, які потребують точних і безпечних математичних розрахунків.

Модуль dsa реалізує алгоритм цифрового підпису DSA (Digital Signature Algorithm). У ньому є структури PrivateKey та PublicKey для зберігання ключів, а також Signature для підпису. Функція create_keys() генерує пару ключів, а методи sign і verify відповідно створюють та перевіряють підписи. Цей модуль дозволяє хешувати дані, генерувати підписи і підтверджувати їх автентичність, що забезпечує цілісність і захист даних.

Модуль elgamal реалізує алгоритм шифрування Ел-Гамаля. Він включає структури PublicKey та PrivateKey для зберігання ключів і надає методи для шифрування (cipher) та дешифрування (decipher) даних. Модуль дозволяє зберігати зашифровані дані у структурі CipheredData, а також обробляти помилки, пов’язані з дешифруванням, за допомогою перерахування DecipherError. Це надійний метод асиметричного шифрування, який підходить для безпечної передачі інформації.

# 4 ДЕМОНСТРАЦІЯ РОБОТИ ПРОТОКОЛУ

Для початку опишемо загальну схему роботи протоколу.

Ініціалізація середовища для голосування: створюється об’єкт reg_bureau (Реєстраційне бюро) для обліку зареєстрованих виборців і сес (Центральна виборча комісія) з набором кандидатів. Потім генерується список з 100 об’єктів voter, що представляють виборців. Кожен виборець має унікальний CitizenId та VoteId.

Процес голосування: для кожного виборця викликається метод vote, де він отримує реєстраційний номер з бюро реєстрації та шифрує свій голос, використовуючи відкритий ключ сес. Голос містить ID кандидата, а також унікальний ідентифікатор голосування (VoteId) та підпис DSA, що підтверджує його автентичність. Результат шифрування та підпис зберігаються як серіалізовані дані у структурі Vote.

Обробка голосів: комісія сес отримує кожен зашифрований голос і викликає метод process_vote для його обробки. На цьому етапі сес розшифровує голос, перевіряє підпис, і якщо голос легітимний, збільшує кількість голосів за відповідного кандидата. Бюро реєстрації оновлює стан виборця, відзначаючи, що він уже проголосував.

Виведення результатів: після обробки всіх голосів комісія виводить результати виборів, показуючи кількість голосів, отриманих кожним кандидатом.

У разі виникнення будь-якої з цих помилок, відповідна операція переривається, і система зберігає інформацію про помилку для подальшого аналізу.

Для дослідження розглянемо тести. На рисунку 4.1 ми перевіряємо загальну роботу алгоритму, коли всі виборці можуть голосувати. Як бачимо, жодного повідомлення про помилку немає.

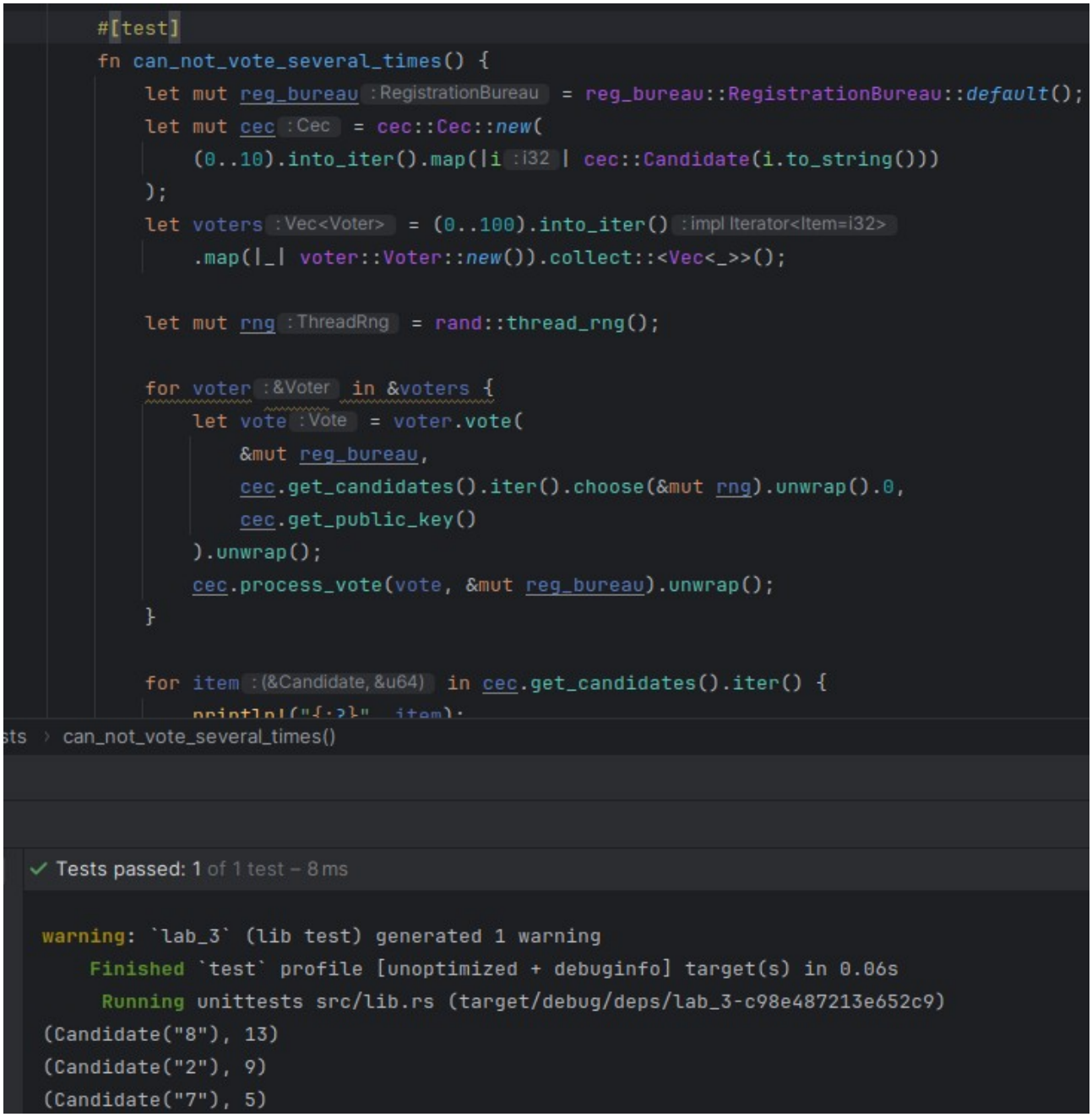


Рисунок 4.1 — Загальний тест

На рисунку 4.2 перевіряємо, що виборець не може проголосувати декілька разів.

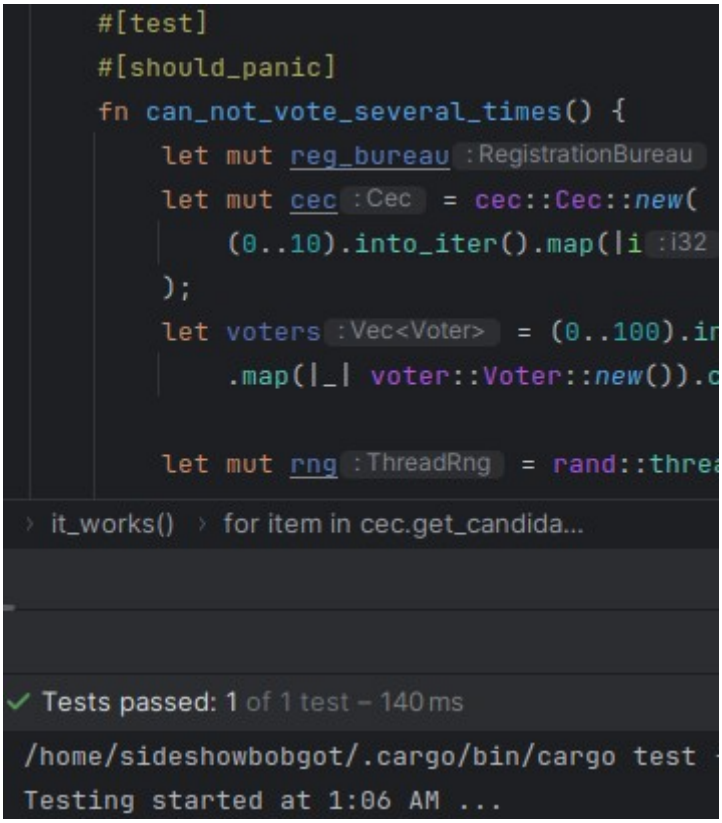


Рисунок 4.2 — Перевірка неможливості проголосувати декілька разів

# 5 ДОСЛІДЖЕННЯ ПРОТОКОЛУ

Чи можуть голосувати ті, хто не має на це права? У даному протоколі тільки бюро реєстрації (БР) видає реєстраційні номери виборцям, що мають право голосу. БР зберігає відповідність номерів виборцям, забезпечуючи, що лише зареєстровані особи можуть проголосувати. Проте, оскільки БР має доступ до реєстраційних записів, воно теоретично може створити неіснуючих виборців або дозволити голосування особам, які не мають на це права.

Чи може виборець голосувати кілька разів? Протокол передбачає перевірку кожного реєстраційного номера на унікальність та одноразове використання. Кожен виборець отримує один реєстраційний номер, який перевіряється комісією (ВК) перед зарахуванням голосу та видаляється зі списку після обробки. Це унеможливлює повторне голосування виборцем.

Чи може хтось (інший виборець, ВК, стороння людина) дізнатися, за кого проголосували інші виборці? Завдяки шифруванню та цифровим підписам повідомлень сторонні особи, включаючи інших виборців, не можуть дізнатися, за кого проголосував виборець. Тільки ВК має змогу розшифрувати бюлетень, але навіть за наявності повідомлення сторонні особи не зможуть отримати доступ до його вмісту без необхідних ключів.

Чи може інший виборець чи стороння людина проголосувати замість іншого зареєстрованого виборця? Це можливо лише у випадку, якщо стороння особа вгадає реєстраційний номер виборця, що є малоймовірним через значну кількість можливих комбінацій. Протокол передбачає випадкову генерацію реєстраційних номерів, кількість яких перевищує кількість виборців, зменшуючи ймовірність вгадування.

Чи може хтось (інший виборець, ВК, стороння людина) таємно змінити голос в бюлетені? ВК не може змінити вміст бюлетеня без втрати його автентичності, оскільки бюлетені підписані виборцями за допомогою DSA. Зміна вмісту бюлетеня призведе до порушення підпису, що одразу буде помітно. Однак, якщо ВК знаходиться в змові з БР, вони можуть змінювати

результати, створюючи фальшиві голоси або коригуючи бюлетені.

Чи може виборець перевірити, що його голос врахований при підведенні кінцевих результатів? Після завершення виборів ВК публікує список, де наведені ідентифікаційні номери виборців і відповідні їм бюлетені. Кожен виборець може знайти свій номер у цьому списку та переконатися, що його голос зарахований правильно, що забезпечує прозорість процесу підрахунку.

# ВИСНОВОК

У ході виконання лабораторної роботи було реалізовано та досліджено протокол Е-голосування, що базується на криптографічних методах захисту даних та розподілі ролей між бюро реєстрації (БР) і виборчою комісією (ВК). Цей підхід спрямований на забезпечення безпеки голосування, анонімності виборців та прозорості підрахунку голосів.

Основні результати роботи:

Реалізована система дозволяє контролювати доступ до голосування та запобігає багаторазовому голосуванню завдяки унікальним реєстраційним номерам від БР. Це забезпечує високий рівень захисту від спроб дублювання голосів та несанкціонованого доступу.

Використання шифрування Ел-Гамаля та цифрових підписів DSA гарантує, що зміст бюлетеня зберігається у таємниці, а інформація про виборців не розголошується стороннім особам.

Протокол базується на довірі до БР, яка має доступ до реєстраційних записів, та ВК, що виконує розшифрування бюлетенів під час підрахунку голосів. Це створює певні ризики, оскільки можливий зговір між БР і ВК для внесення фальсифікацій у результати.

Протокол дозволяє кожному виборцю після голосування перевірити, що його голос врахований правильно, завдяки публікації списку з ідентифікаційними номерами та бюлетенями, що підвищує прозорість системи.

Захист системи від підробки голосів сторонніми особами залежить від випадковості реєстраційних номерів. Хоча існує можливість вгадати номер, протокол мінімізує цей ризик завдяки широкому діапазону потенційних номерів.

Аналіз показав, що реалізований протокол забезпечує високий рівень безпеки та прозорості, однак має певні ризики, пов'язані з централізованою природою системи та необхідністю довіри до БР і ВК. Незважаючи на ці обмеження, дослідження демонструє потенціал протоколу для захисту електронного голосування і підкреслює важливість продуманої архітектури для

забезпечення балансу між анонімністю, прозорістю та безпекою в системах Е-голосування.

# 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;

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

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

        if z == 0 {
            return None;
        }

        if a == 0 || !is_prime64(a as u64) {
            return None;
        }

        if z >= a {
            return None;
        }

        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 None;
        }
        Some(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;
            }
        }
    }

    #[cfg(test)]
    mod tests {
        use super::modinv;

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


mod dsa {
    use std::hash::{DefaultHasher, Hasher};
```

```
use std::ops::Rem;
use rand::Rng;
use crate::alg_utils::{gen_prime, modinv, modpow};
use num_prime::nt_funcs::is_prime64;
use serde::{Deserialize, Serialize};

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

pub struct PrivateKey {
    q: u16,
    p: u64,
    g: u64,
    x: u16
}

#[derive(Serialize, Deserialize)]
pub struct PublicKey {
    q: u16,
    p: u64,
    g: u64,
    y: u64,
}

#[derive(Serialize, Deserialize)]
pub struct Signature {
    r: u16,
    s: u16,
}

pub fn create_keys() -> (PublicKey, PrivateKey) {
    let (q, p) = 'qp_loop: loop {
        const N: usize = 16;
        const MIN: u16 = (1 << (N - 1)) as u16;
        const MAX: u16 = ((1 << N) - 1) as u16;
        let q = gen_prime(MIN..MAX);
        let mut p_1 = (q as u64) * 2;
        loop {
            if u64::MAX - p_1 < q as u64 {
                continue 'qp_loop;
            }
```

```
                let p = p_1 + 1;
                if is_prime64(p) && p_1 % q as u64 == 0 {
                    break 'qp_loop (q, p);
                }
                p_1 += q as u64;
            }
        };

        let g = loop {
            let h = rand::thread_rng().gen_range(1..p-1);
            let g = modpow(h, (p - 1) / q as u64, p);
            if g > 1 {
                break g;
            }
        };

        let x = rand::thread_rng().gen_range(0..q);
        let y = modpow(g, x as u64, p);

        (PublicKey { q, p, g, y }, PrivateKey { q, p, g, x })
    }

    impl PrivateKey {
        pub fn sign(&self, data: &[u8]) -> Signature {
            let data_hash = calculate_hash(data);
            loop {
                let k = rand::thread_rng().gen_range(0..self.q);
                let r = modpow(self.g, k as u64, self.p).rem(self.q as u64) as
u16;
                if r != 0 {
                    if let Some(k_inv) = modinv(k, self.q) {
                        let part = ((data_hash as u64 + self.x as u64 * r as
u64) % self.q as u64) as u16;
                        let s = ((k_inv as u64 * part as u64) % self.q as u64)
as u16;
                        if s != 0 {
                            break Signature { r, s };
                        }
                    }
                }
            }
        }
    }
```

```
    impl PublicKey {
        pub fn verify(&self, signature: &Signature, data: &[u8]) -> bool {
            if ![signature.s, signature.r].iter().all(|num| {
                *num > 0 && *num < self.q
            }) {
                return false;
            }
            if let Some(w) = modinv(signature.s, self.q) {
                let data_hash = calculate_hash(data);
                let u_one = (data_hash as u64 * w as u64).rem(self.q as u64) as
u16;
                let u_two = (signature.r as u64 * w as u64).rem(self.q as u64)
as u16;
                let v_one = modpow(self.g, u_one as u64, self.p) as u128;
                let v_two = modpow(self.y, u_two as u64, self.p) as u128;
                let v = ((((v_one * v_two) % self.p as u128) as u64) % self.q as
u64) as u16;
                v == signature.r
            } else {
                false
            }
        }
    }

    #[cfg(test)]
    mod tests {

        #[test]
        fn it_works() {
            let message = "message";
            let (public_key, private_key) = super::create_keys();
            let signature = private_key.sign(message.as_bytes());
            let res = public_key.verify(&signature, message.as_bytes());
            assert!(res);
        }
    }
}

mod elgamal {
    use num_traits::ToPrimitive;
    use rand::Rng;
    use serde::{Deserialize, Serialize};
    use thiserror::Error;
    use crate::alg_utils::{gen_prime, modinv, modpow};
```

```
use crate::elgamal::DecipherError::CanNotConvertToByte;

#[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)]
pub struct PublicKey {
    g: u16,
    y: u16,
    p: u16,
}

impl PublicKey {
    pub fn cipher(&self, data: &[u8]) -> CipheredData {
        let (a, u) = {
            let k = rand::thread_rng().gen_range(1..(self.p - 1));
            (
                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()
        }
    }
}

#[derive(Error, Debug)]
pub enum DecipherError {
    #[error("Can not create inverse")]
    CanNotCreateInverse{ s: u16, p: u16 },
    #[error("Can not convert to byte")]
    CanNotConvertToByte(u16),
```

```
    }

    pub struct PrivateKey {
        p: u16,
        x: 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).ok_or(DecipherError::CanNotCreateInverse { s,
p: self.p })?
            };
            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(CanNotConvertToByte(prob_byte))?;
                acc.push(byte);
                Ok(acc)
            })
        }
    }

    pub struct KeyPair {
        pub private_key: PrivateKey,
        pub public_key: PublicKey,
    }

    pub fn create_keys() -> KeyPair {
        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;
        KeyPair { public_key: PublicKey { g, y, p }, private_key: PrivateKey
{ p, x } }
    }

    #[cfg(test)]
    mod tests {
```

```
        #[test]
        fn it_works() {
            let message = "message";
            let key_pair = super::create_keys();
            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);
        }
    }
}

mod sim_env {
    use rand::Rng;

    fn gen_large_num() -> u64 {
        const MIN: u64 = u32::MAX as u64;
        const MAX: u64 = u64::MAX;
        rand::thread_rng().gen_range(MIN..MAX)
    }

    mod voter {
        use serde::{Deserialize, Serialize};
        use thiserror::Error;
        use crate::{dsa, elgamal};
        use crate::dsa::Signature;
        use crate::sim_env::{cec, gen_large_num, reg_bureau};

        #[derive(Serialize, Deserialize, PartialEq, Eq, Debug, Clone, Copy)]
        pub struct CitizenId(u64);

        #[derive(Serialize, Deserialize)]
        pub struct VoteId(pub u64);

        pub struct Voter {
            citizen_id: CitizenId,
            vote_id: VoteId,
        }

        #[derive(Serialize)]
        struct VoteDataRef<'a> {
            vote_id: VoteId,
            reg_num: reg_bureau::RegistrationNumber,
            candidate: &'a cec::Candidate,
```

```
    public_key: dsa::PublicKey
}

#[derive(Deserialize)]
pub struct VoteData {
    pub vote_id: VoteId,
    pub reg_num: reg_bureau::RegistrationNumber,
    pub candidate: cec::Candidate,
    pub public_key: dsa::PublicKey
}

pub struct Vote {
    pub ser_ciphered_data: Vec<u8>,
    pub signature: Signature
}

#[derive(Error, Debug)]
pub enum VoteError {
    #[error(transparent)]
    GiveRegNumber(reg_bureau::GiveRegNumberError),
    #[error(transparent)]
    VoteSerialization(bincode::Error),
    #[error(transparent)]
    CipherSerialization(bincode::Error),
}

impl Voter {
    pub fn new() -> Self {
        Self {
            citizen_id: CitizenId(gen_large_num()),
            vote_id: VoteId(gen_large_num()),
        }
    }

    pub fn vote(
        &self,
        reg_bureau: &mut impl reg_bureau::GiveRegNumber,
        candidate: &cec::Candidate,
        cec_public_key: &elgamal::PublicKey,
    ) -> Result<Vote, VoteError> {
        let sign_keys = dsa::create_keys();

        let (signature, ser_ciphered_data) = {
            let (signature, ciphered_data) = {
```

```
                        let ser_vote_data = {
                            let reg_num =
reg_bureau.give_reg_num(self.citizen_id)
                                .map_err(VoteError::GiveRegNumber)?;
                            let vote_data_ref = VoteDataRef {
                                vote_id: VoteId(gen_large_num()),
                                reg_num, candidate, public_key: sign_keys.0
                            };
                            bincode::serialize(&vote_data_ref)
                                .map_err(VoteError::VoteSerialization)?
                        };
                        let signature = sign_keys.1.sign(&ser_vote_data);
                        (signature, cec_public_key.cipher(&ser_vote_data))
                    };
                    (
                        signature,
                        bincode::serialize(&ciphered_data)
                            .map_err(VoteError::CipherSerialization)?
                    )
                };
                Ok(Vote { ser_ciphered_data, signature })
            }
        }
    }

    mod cec {
        use std::collections::HashMap;
        use serde::{Deserialize, Serialize};
        use thiserror::Error;
        use crate::elgamal;
        use crate::sim_env::{reg_bureau, voter};
        use crate::sim_env::voter::Vote;

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

        pub struct Cec {
            key_pair: elgamal::KeyPair,
            voter_ids: Vec<voter::VoteId>,
            candidates: HashMap<Candidate, u64>,
        }

        impl Cec {
            pub fn new(candidates: impl Iterator<Item=Candidate>) -> Self {
```

```
            Self {
                key_pair: elgamal::create_keys(),
                voter_ids: Default::default(),
                candidates: HashMap::from_iter(candidates.map(|c| (c, 0))),
            }
        }

        pub fn get_candidates(&self) -> &HashMap<Candidate, u64> {
            &self.candidates
        }

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

    #[derive(Error, Debug)]
    pub enum ProcessVoteError {
        #[error(transparent)]
        DeserializeCipherData(bincode::Error),
        #[error(transparent)]
        Decipher(#[from] elgamal::DecipherError),
        #[error(transparent)]
        DeserializeVoteData(bincode::Error),
        #[error("Invalid candidate: {0:?}")]
        InvalidCandidate(Candidate),
        #[error("Failed verification")]
        Verification,
        #[error(transparent)]
        UpdateRegistration(#[from] reg_bureau::UpdateRegistrationError)
    }

    impl Cec {
        pub fn process_vote(
            &mut self,
            vote: Vote,
            reg_bureau: &mut impl reg_bureau::UpdateRegistration
        ) -> Result<(), ProcessVoteError> {

            let data = {
                let ciphered_data =
bincode::deserialize::<elgamal::CipheredData>(&vote.ser_ciphered_data)
                    .map_err(ProcessVoteError::DeserializeCipherData)?;
                self.key_pair.private_key.decipher(&ciphered_data)
```

```
                        .map_err(ProcessVoteError::Decipher)?
                };
                let vote_data = bincode::deserialize::<voter::VoteData>(&data)
                        .map_err(ProcessVoteError::DeserializeVoteData)?;
                if vote_data.public_key.verify(&vote.signature, &data) {
                    if self.candidates.contains_key(&vote_data.candidate) {
                        reg_bureau.update_registration(vote_data.reg_num)
                            .map_err(ProcessVoteError::UpdateRegistration)
                            .map(|_| {

*self.candidates.get_mut(&vote_data.candidate).unwrap() += 1;
                                self.voter_ids.push(vote_data.vote_id);
                            })
                    } else {

Err(ProcessVoteError::InvalidCandidate(vote_data.candidate))
                    }
                } else {
                    Err(ProcessVoteError::Verification)
                }
            }
        }
    }

    mod reg_bureau {
        use serde::{Deserialize, Serialize};
        use thiserror::Error;
        use crate::sim_env::{gen_large_num, voter};

        #[derive(Serialize, Deserialize, PartialEq, Eq, Clone, Copy, Debug)]
        pub struct RegistrationNumber(pub u64);

        struct Row {
            citizen_id: voter::CitizenId,
            reg_num: RegistrationNumber,
            vote_state: VoteState
        }

        #[derive(Default)]
        pub struct RegistrationBureau(Vec<Row>);

        enum VoteState {
            Voted,
            NotVoted
```

```
        }

        #[derive(Error, Debug)]
        #[error("Registration number is already voted: {0:?}")]
        pub struct UpdateRegistrationError(RegistrationNumber);

        pub trait UpdateRegistration {
            fn update_registration(&mut self, reg_num: RegistrationNumber)
                -> Result<(), UpdateRegistrationError>;
        }

        impl UpdateRegistration for RegistrationBureau {
            fn update_registration(&mut self, reg_num: RegistrationNumber)
                -> Result<(), UpdateRegistrationError> {
                self.0.iter_mut().find(|r| r.reg_num == reg_num)
                    .ok_or(UpdateRegistrationError(reg_num))
                    .map(|r| r.vote_state = VoteState::Voted)
            }
        }

        #[derive(Error, Debug)]
        #[error("Registration number exists: {0:?}")]
        pub struct GiveRegNumberError(voter::CitizenId);

        pub trait GiveRegNumber {
            fn give_reg_num(&mut self, citizen_id: voter::CitizenId) ->
Result<RegistrationNumber, GiveRegNumberError>;
        }

        impl GiveRegNumber for RegistrationBureau {
            fn give_reg_num(&mut self, citizen_id: voter::CitizenId) ->
Result<RegistrationNumber, GiveRegNumberError> {

                if self.0.iter().any(|r| {
                    r.citizen_id == citizen_id
                }) {
                    Err(GiveRegNumberError(citizen_id))
                } else {
                    loop {
                        let reg_num = RegistrationNumber(gen_large_num());
                        if self.0.iter().all(|r| r.reg_num != reg_num) {
                            self.0.push(Row { citizen_id, reg_num, vote_state:
VoteState::NotVoted });
                            break Ok(reg_num);
```

```
                    }
                }
            }
        }
    }
}

#[cfg(test)]
mod tests {
    use rand::prelude::IteratorRandom;
    use super::{reg_bureau, cec, voter};

    #[test]
    fn it_works() {
        let mut reg_bureau = reg_bureau::RegistrationBureau::default();
        let mut cec = cec::Cec::new(
            (0..10).into_iter().map(|i| cec::Candidate(i.to_string()))
        );
        let voters = (0..100).into_iter()
            .map(|_| voter::Voter::new()).collect::<Vec<_>>();

        let mut rng = rand::thread_rng();

        for voter in &voters {
            let vote = voter.vote(
                &mut reg_bureau,
                cec.get_candidates().iter().choose(&mut rng).unwrap().0,
                cec.get_public_key()
            ).unwrap();
            cec.process_vote(vote, &mut reg_bureau).unwrap();
        }

        for item in cec.get_candidates().iter() {
            println!("{:?}", item);
        }
    }

    #[test]
    #[should_panic]
    fn can_not_vote_several_times() {
        let mut reg_bureau = reg_bureau::RegistrationBureau::default();
        let mut cec = cec::Cec::new(
            (0..10).into_iter().map(|i| cec::Candidate(i.to_string()))
        );
```

```
            let voters = (0..100).into_iter()
                .map(|_| voter::Voter::new()).collect::<Vec<_>>();

            let mut rng = rand::thread_rng();

            for voter in &voters {
                let vote = voter.vote(
                    &mut reg_bureau,
                    cec.get_candidates().iter().choose(&mut rng).unwrap().0,
                    cec.get_public_key()
                ).unwrap();

                let vote = voter.vote(
                    &mut reg_bureau,
                    cec.get_candidates().iter().choose(&mut rng).unwrap().0,
                    cec.get_public_key()
                ).unwrap();

                cec.process_vote(vote, &mut reg_bureau).unwrap();
            }

            for item in cec.get_candidates().iter() {
                println!("{:?}", item);
            }
        }
    }
}
```