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 №5

## Протоколи й алгоритми електронного голосування

**Тема:** Протокол Е-голосування з розділенням комісії на незалежні частини

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 розділенням на незалежні комісії.

# 2 ЗАВДАННЯ

Змоделювати протокол Е-голосування з розділенням комісії на незалежні частини будь-якою мовою програмування та провести його дослідження. Для кодування повідомлень використовувати метод RSA, 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 sim_env відповідає за симуляцію процесу голосування. Він включає кілька підмодулів, таких як voter і cec. Підмодуль voter містить структуру виборця і функції для голосування. Кожен виборець шифрує свій голос, обирає випадкові множники для кандидата і підписує свої голоси цифровим підписом DSA. Підмодуль cec моделює центральну виборчу комісію (ЦВК), яка зберігає дані про виборців та кандидатів, а також проводить процес голосування і підрахунок голосів. Кандидати реєструються через структуру Candidate, а голоси шифруються та перевіряються через різні комісії. Всі дані шифруються і розшифровуються за допомогою RSA, що забезпечує безпеку процесу голосування.

# 4 ДЕМОНСТРАЦІЯ РОБОТИ ПРОТОКОЛУ

Тест у модулі sim_env моделює повний процес голосування і підрахунку голосів у системі е-голосування з розділенням виборчої комісії на дві частини, перевіряючи коректність роботи всієї симуляції.

Спочатку створюється об'єкт cec::Initialization, який представляє початковий етап голосування, де реєструються виборці та кандидати. Тест генерує 15 виборців та 5 кандидатів, причому кожен виборець отримує унікальний ідентифікатор. Кандидати також отримують унікальні ідентифікатори і зберігаються у структурі Candidate. Після цього процес голосування активується через виклик методу activate_vote, який ініціалізує дві виборчі комісії (ec) і повертає їх разом з активною сесією голосування (ActiveVote).

Кожен виборець виконує голосування через функцію vote. Під час голосування виборець обирає випадкового кандидата, розбиває його ідентифікаційний номер на два множники (для кожної частини голосу) і шифрує їх за допомогою RSA. Потім обидві частини голосу передаються двом різним виборчим комісіям, і кожна частина підписується виборцем через DSA. Виборчі комісії перевіряють підписи і зберігають зашифровані частини голосів.

Після того як всі виборці проголосували, тест завершує голосування, викликаючи метод end_vote. Цей метод обробляє результати з двох виборчих комісій. Спочатку зашифровані частини голосів, що зберігалися у виборчих комісіях, публікуються, а потім ЦВК збирає ці дані, дешифрує їх і об'єднує для визначення голосу кожного виборця. Цей етап також перевіряє правильність даних, зіставляючи ідентифікатори кандидатів.

Результати голосування підраховуються і виводяться на екран. Тест на рисунку 4.1 забезпечує перевірку, чи були всі голоси правильно зашифровані, розшифровані, підраховані і чи відповідають кандидати результатам голосування.

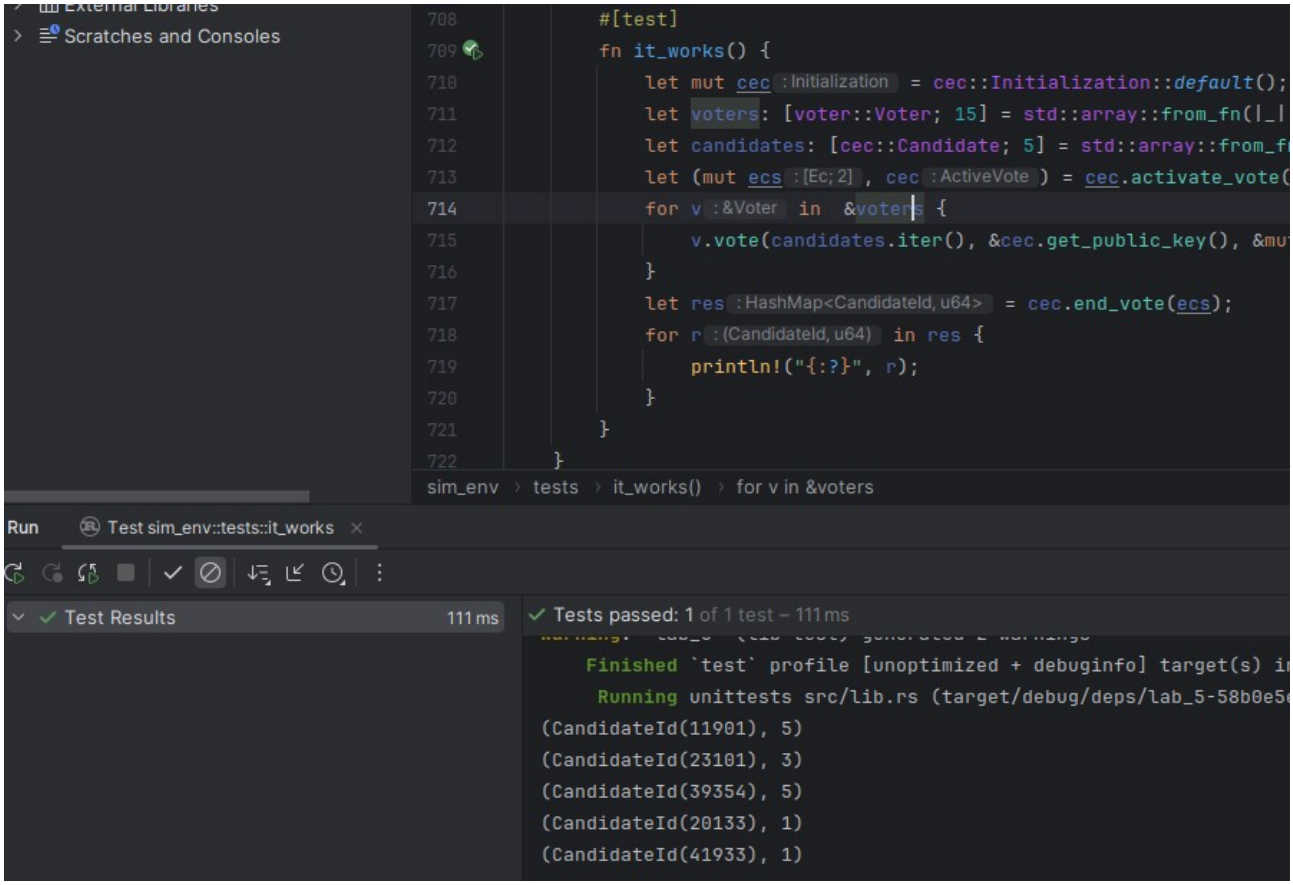


Рисунок 4.1 — Тестування процесу голосування

# 5 ДОСЛІДЖЕННЯ ПРОТОКОЛУ

Чи можуть голосувати ті, хто не має на це права? Протокол е-голосування гарантує, що лише зареєстровані виборці можуть брати участь у голосуванні. Кожен виборець отримує унікальний ідентифікатор (ID), який перевіряється виборчими комісіями під час подачі голосу. Кожна комісія зберігає ідентифікатори виборців і фіксує, чи отримала бюлетені від кожного зареєстрованого учасника. Якщо хтось спробує проголосувати без наявності відповідного ID, комісії не приймуть такі голоси. Це забезпечує захист від участі неавторизованих осіб у голосуванні.

Чи може виборець голосувати кілька разів? Протокол також захищає від повторного голосування одним виборцем. Під час голосування кожен виборець відправляє зашифровані частини свого бюлетеня до двох виборчих комісій. Кожна комісія відмічає, що отримала бюлетень від конкретного виборця за його ID, і не дозволяє подавати бюлетень вдруге. Якщо виборець спробує проголосувати повторно, це буде виявлено під час перевірки його ID, і нові голоси будуть відхилені. Таким чином, система запобігає спробам подвійного голосування.

Чи може хтось (інший виборець, ВК, стороння людина) дізнатися, за кого проголосували інші виборці? Протокол забезпечує високий рівень анонімності голосування. Під час голосування бюлетені шифруються окремими частинами за допомогою RSA, і кожна частина надсилається до різних виборчих комісій. Оскільки кожна комісія отримує лише частину зашифрованого голосу, вони не можуть самостійно дізнатися, за кого був поданий голос. Тільки після об'єднання обох частин і розшифрування центральною виборчою комісією можна визначити результат. Це робить практично неможливим стеження, за кого конкретно проголосував виборець.

Чи може інший виборець чи стороння людина проголосувати замість іншого зареєстрованого виборця? Кожен виборець має унікальний ID і використовує цифровий підпис DSA для підтвердження свого голосу. Під час голосування кожен бюлетень підписується особистим ключем виборця, що

гарантує автентичність голосу. Виборчі комісії перевіряють цифрові підписи, і якщо підпис не збігається з ID виборця, голос не приймається. Це захищає систему від можливих спроб проголосувати за когось іншого або підробити голос виборця.

Чи може хтось (інший виборець, ВК, стороння людина) таємно змінити голос у бюлетені? Оскільки голоси шифруються за допомогою криптографічних алгоритмів і підписуються цифровими підписами, зміна голосу після його подання практично неможлива. Будь-яка спроба змінити зашифровані частини голосу або підробити підпис буде відразу виявлена під час розшифрування або перевірки підписів. Це гарантує, що ніхто, включно з іншими виборцями, виборчими комісіями чи сторонніми особами, не може таємно змінити голос виборця після його подання.

Чи може виборець перевірити, що його голос врахований при підведенні кінцевих результатів? Протокол дозволяє кожному виборцю перевірити, що його голос був врахований у підсумках голосування. Після завершення голосування виборчі комісії публікують списки ID виборців і зашифровані частини бюлетенів. Центральна виборча комісія публікує повні бюлетені після їх розшифрування, що дозволяє кожному виборцю переконатися, що його голос не був змінений і правильно врахований під час підрахунку. Це забезпечує прозорість процесу і дає виборцю можливість перевірити свій внесок у результат виборів.

# ВИСНОВОК

У результаті виконання лабораторної роботи було досліджено та реалізовано протокол е-голосування з розділенням виборчої комісії на незалежні частини. Основною перевагою цього протоколу є високий рівень безпеки та анонімності, який досягається за рахунок розподілу бюлетеня на дві частини, що шифруються окремо і передаються до різних виборчих комісій. Завдяки цьому жодна з комісій не може самостійно дізнатися, за кого проголосував виборець, що захищає від спроб підробки результатів голосування або втручання в процес.

Протокол гарантує, що лише зареєстровані виборці можуть голосувати, виключає можливість подвійного голосування та забезпечує автентичність кожного голосу через використання цифрового підпису DSA. Крім того, кожен виборець може перевірити, що його голос був врахований при підведенні підсумків виборів, що підвищує прозорість процесу голосування.

Однак, попри високу безпеку, залишається ймовірність шахрайства, якщо всі виборчі комісії вступлять у змову. Водночас, у випадку чесної роботи комісій, протокол забезпечує високий рівень надійності та захисту голосів. У цілому, цей протокол може стати основою для безпечних і прозорих систем електронного голосування, що особливо важливо в умовах сучасного цифрового суспільства.

# 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.*

```
extern crate derive_more;

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 crate::alg_utils::{gen_prime, modinv, modpow};
    use getset::Getters;
    use num_prime::nt_funcs::is_prime64;
    use rand::Rng;
    use serde::{Deserialize, Serialize};
    use std::hash::{DefaultHasher, Hasher};
    use std::ops::Rem;

    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, Debug)]
    pub struct Signature {
        r: u16,
        s: u16,
    }

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

```
impl Default for KeyPair {
    fn default() -> Self {
        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);

        KeyPair{
            public_key: PublicKey { q, p, g, y },
            private_key: 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 key_pair = super::KeyPair::default();
            let signature = key_pair.private_key.sign(message.as_bytes());
            let res = key_pair.public_key.verify(&signature,
message.as_bytes());
            assert!(res);
        }
    }
}

mod rsa {
    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 std::ops::Rem;
    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)
        })
    }

    pub fn cipher_data_u8_arr<const N: usize>(
        key: &impl KeyRef, data: &[u8; N]
    ) -> Result<[BigUint; N], CipherDataError> {
        let (part, product_number) = key.get_parts();
        let mut res: [BigUint; N] = std::array::from_fn(|_| Default::default());
        for (r, d) in res.iter_mut().zip(data.iter()) {
            *r = modpow(&BigUint::from(*d), part, product_number)?;
        }
        Ok(res)
    }

    pub fn cipher_data_biguint_arr<const N: usize>(
        key: &impl KeyRef, mut data: [BigUint; N]
    ) -> Result<[BigUint; N], CipherDataError> {
        let (part, product_number) = key.get_parts();
        for d in data.iter_mut() {
            let dn = modpow(&d, part, product_number)?;
            *d = dn;
        }
        Ok(data)
    }

    #[cfg(test)]
```

```
    mod tests {
        use super::{cipher_data_biguint, cipher_data_u8, KeyPair};
        use num_traits::ToPrimitive;

        #[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 sim_env {

    mod voter {
        use crate::sim_env::cec::ec::AcceptVote;
        use crate::sim_env::cec::{Candidate, VoterId};
        use crate::{dsa, rsa};
        use num_bigint::BigUint;
        use rand::prelude::IteratorRandom;
        use serde::{Deserialize, Serialize};

        pub struct Voter {
            dsa: dsa::KeyPair,
            id: VoterId
        }

        fn get_multiplicative_pairs(value: u16) -> Vec<[u16; 2]> {
            let mut pairs = Vec::new();
            for i in 1..=value {
                if value % i == 0 {
                    pairs.push([i, value / i]);
                }
            }
```

```
        pairs
    }

    #[derive(Serialize, Deserialize, Debug)]
    pub struct MulPart(pub [BigUint; 2]);

    #[derive(Serialize, Deserialize, Debug)]
    pub struct VoteData {
        pub mul_part: MulPart,
        pub voter_id: VoterId,
    }

    impl Voter {
        pub fn new(id: VoterId) -> Self {
            Self {
                id,
                dsa: Default::default(),
            }
        }

        pub fn vote<'a, T: AcceptVote>(
            &self,
            candidates: impl Iterator<Item=&'a Candidate>,
            cec_public_key: &rsa::PublicKeyRef,
            ecs: &mut [T; 2]
        ) -> () {
            let candidate = candidates.choose(&mut
rand::thread_rng()).unwrap();
            let mul_pairs = get_multiplicative_pairs(candidate.get_id().0)
                .into_iter().choose(&mut rand::thread_rng()).unwrap();
            let vote_data_arr: [VoteData; 2] = std::array::from_fn(|i| {
                VoteData {
                    mul_part: MulPart(
                            rsa::cipher_data_u8_arr(
                            cec_public_key, &mul_pairs[i].to_be_bytes()
                        ).unwrap()
                    ),
                    voter_id: self.id
                }
            });
            let signatures: [dsa::Signature; 2] = std::array::from_fn(|i| {
                self.dsa.get_private_key()
                    .sign(&bincode::serialize(&vote_data_arr[i]).unwrap())
            });
```

```
                for ((vote_data, signature), ec) in vote_data_arr.into_iter()
                    .zip(signatures.iter()).zip(ecs.iter_mut()) {
                    ec.accept_vote(vote_data, signature,
self.dsa.get_public_key());
                }
            }
        }
    }

    mod cec {
        use crate::rsa;
        use crate::sim_env::cec::ec::{Ec, PublishVotes, VoteState};
        use derive_more::From;
        use getset::Getters;
        use num_integer::Integer;
        use num_traits::ToPrimitive;
        use rand::Rng;
        use serde::{Deserialize, Serialize};
        use std::collections::HashMap;
        use std::mem;

        #[derive(Getters)]
        #[getset(get = "pub with_prefix")]
        pub struct Candidate {
            name: String,
            id: CandidateId
        }

        macro_rules! declare_id {
            ($name:ident) => {
                #[derive(
                    Serialize, Deserialize, Debug,
                    Eq, PartialEq, Clone, Copy, From, Hash,
                    Ord, PartialOrd
                )]
                pub struct $name (pub u16);
            };
        }

        declare_id!(VoterId);
        declare_id!(CandidateId);

        #[derive(Default)]
        pub struct Initialization {
```

```
            voter_ids: Vec<VoterId>,
            candidate_ids: Vec<CandidateId>
        }

        #[derive(Default)]
        pub struct ActiveVote {
            rsa: rsa::KeyPair,
            candidate_ids: Vec<CandidateId>
        }

        fn register_id<Id: From<u16> + Copy + PartialEq<Id>>(ids: &mut Vec<Id>)
-> Id {
            loop {
                let id = Id::from(rand::thread_rng().gen_range(u8::MAX as
u16..u16::MAX));
                if ids.iter().find(|el| (*el).eq(&id)).is_none() {
                    ids.push(id);
                    break id;
                }
            }
        }

        impl Initialization {
            pub fn register_voter(&mut self) -> VoterId {
                register_id::<VoterId>(&mut self.voter_ids)
            }

            pub fn register_candidate(&mut self, name: String) -> Candidate {
                let id = register_id::<CandidateId>(&mut self.candidate_ids);
                Candidate { name, id }
            }

            pub fn activate_vote(self) -> ([Ec; 2], ActiveVote) {
                (
                    std::array::from_fn(|_| Ec::new(self.voter_ids.iter())),
                    ActiveVote {
                        candidate_ids: self.candidate_ids,
                        rsa: Default::default()
                    },
                )
            }
        }

        impl ActiveVote {
```

```
            pub fn get_public_key(&self) -> rsa::PublicKeyRef {
                self.rsa.get_public_key_ref()
            }

            pub fn end_vote<T: PublishVotes + Default>(self, mut ecs: [T; 2]) ->
HashMap<CandidateId, u64> {
                let mut candidates_votes = HashMap::from_iter(
                    self.candidate_ids.iter().map(|id| (*id, 0u64))
                );
                mem::take(&mut ecs[0]).publish_votes()
                    .into_iter().zip(mem::take(&mut
ecs[1]).publish_votes().into_iter())
                    .for_each(|(v1, v2)| {
                        assert_eq!(v1.0, v2.0);
                        match (v1.1, v2.1) {
                            (VoteState::Voted(vd_0), VoteState::Voted(vd_1)) =>
{
                                let mut vds = [vd_0, vd_1];
                                let mul_parts: [u16; 2] = std::array::from_fn(|
i| {
                                    let data = rsa::cipher_data_biguint_arr(
                                        &self.rsa.get_private_key_ref(),
mem::take(&mut vds[i].0)
                                    ).expect("Failed ciphering data");
                                    let byte_data: [u8; 2] =
std::array::from_fn(|i| {
                                        data[i].to_u8().unwrap()
                                    });
                                    u16::from_be_bytes(byte_data)
                                });

                                let candidate_id = CandidateId(mul_parts[0] *
mul_parts[1]);
                                if let Some(vote_counter) =
candidates_votes.get_mut(&candidate_id) {
                                    vote_counter.inc();
                                } else {
                                    panic!("Invalid candidate id: {:?}",
candidate_id);
                                }
                            },
                            _ => panic!("unexpected vote state")
                        }
                    }
```

```
            );
            candidates_votes
        }
    }

    pub mod ec {
        use crate::dsa;
        use crate::sim_env::cec::VoterId;
        use crate::sim_env::voter::{MulPart, VoteData};
        use std::collections::HashMap;

        pub enum VoteState {
            NotVoted,
            Voted(MulPart)
        }

        #[derive(Default)]
        pub struct Ec(HashMap<VoterId, VoteState>);

        pub trait AcceptVote {
            fn accept_vote(
                &mut self,
                vote_data: VoteData,
                signature: &dsa::Signature,
                public_key: &dsa::PublicKey,
            );
        }

        impl AcceptVote for Ec {
            fn accept_vote(
                &mut self,
                vote_data: VoteData,
                signature: &dsa::Signature,
                public_key: &dsa::PublicKey,
            ) {
                assert!(
                    public_key.verify(signature,
&bincode::serialize(&vote_data).unwrap()),
                    "invalid signature"
                );
                let vote_state = self.0.get_mut(&vote_data.voter_id)
                    .expect("Voter id must be registered");
                match vote_state {
                    VoteState::Voted(_) => {
```

```
                            panic!("Voter id already voted: {:?}",
vote_data.voter_id)
                        },
                        VoteState::NotVoted => {
                            *vote_state = VoteState::Voted(vote_data.mul_part)
                        }
                    }
                }
            }

            pub trait  PublishVotes {
                fn publish_votes(self) -> Vec<(VoterId, VoteState)>;
            }

            impl PublishVotes for Ec {
                fn publish_votes(self) -> Vec<(VoterId, VoteState)> {
                    let mut v: Vec<(VoterId, VoteState)> =
self.0.into_iter().collect();
                    v.sort_by(|v1, v2| v1.0.cmp(&v2.0));
                    v
                }
            }

            impl Ec {
                pub(super) fn new<'a>(voter_ids: impl Iterator<Item=&'a
VoterId>) -> Self {
                    Ec(HashMap::from_iter(voter_ids.map(|id| (*id,
VoteState::NotVoted))))
                }
            }
        }
    }

    #[cfg(test)]
    mod tests {
        use crate::sim_env::{cec, voter};

        #[test]
        fn it_works() {
            let mut cec = cec::Initialization::default();
            let voters: [voter::Voter; 15] = std::array::from_fn(|_|
voter::Voter::new(cec.register_voter()));
            let candidates: [cec::Candidate; 5] = std::array::from_fn(|i|
cec.register_candidate(i.to_string()));
```

```
            let (mut ecs, cec) = cec.activate_vote();
            for v in  &voters {
                v.vote(candidates.iter(), &cec.get_public_key(), &mut ecs);
            }
            let res = cec.end_vote(ecs);
            for r in res {
                println!("{:?}", r);
            }
        }
    }
}
```