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 №2

Протоколи й алгоритми електронного голосування

Тема: Протокол Е-голосування зі сліпими підписами

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 зі сліпими підписами

2 ЗАВДАННЯ

Змоделювати протокол Е-голосування зі сліпими підписами будь-якою мовою програмування та провести його дослідження. Для кодування повідомлень використовувати шифрування RSA, для реалізації ЕЦП використовувати алгоритм RSA.

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.

Модуль виборця відповідає за операції, пов'язані з діями виборця в системі. Основною структурою є `Voter`, яка представляє виборця. Цей модуль містить методи для створення зашифрованих пакетів з бюлетенями (`produce_packets`) та для обробки підписаного пакету і здійснення голосування (`accept_signed_packet_and_vote`).

Модуль ЦВК керує процесом голосування з боку Центральної Виборчої Комісії. Головною структурою є `Cec`, яка містить інформацію про кандидатів та стан виборців. Ключові методи включають `consume_packets` для обробки пакетів від виборців та `process_vote` для обробки та підрахунку голосів. Цей модуль також відповідає за перевірку легітимності голосів та запобігання подвійному голосуванню.

4 ДЕМОНСТРАЦІЯ РОБОТИ ПРОТОКОЛУ

Для початку опишемо загальну схему роботи протоколу:

1. Підготовка

• Центральна виборча комісія (ЦВК) ініціалізується зі списком кандидатів та створює пару ключів RSA.

• Реєструються виборці, кожному присвоюється стан "NotRegistered" в структурі VoterState.

• Кожен виборець створюється як екземпляр структури Voter з унікальним VoterId.

2. Процес голосування

• Виборець викликає метод produce_packets, який створює набір зашифрованих бюлетенів.

• Кожен бюлетень містить список всіх кандидатів та ідентифікатор виборця.

• Виборець застосовує операцію засліплення (blinding) до бюлетенів.

• Засліплені бюлетені шифруються публічним ключем ЦВК.

3. Подання голосу

• Виборець надсилає зашифровані та засліплені бюлетені до ЦВК через метод consume_packets.

4. Обробка голосу ЦВК

• ЦВК розшифровує пакети своїм приватним ключем.

• Перевіряється легітимність бюлетенів та ідентифікатор виборця.

• ЦВК підписує один з бюлетенів своїм приватним ключем.

• Стан виборця змінюється на "Registered".

• ЦВК повертає підписаний бюлетень виборцю.

5. Голосування

• Виборець отримує підписаний бюлетень через метод accept_signed_packet_and_vote.

• Виборець знімає засліплення з бюлетеня.

• Виборець обирає кандидата з бюлетеня.

- Вибір шифрується публічним ключем ЦВК та відправляється назад.

6. Підрахунок голосів

- ЦВК отримує зашифрований голос через метод process_vote.
- ЦВК розшифровує голос, перевіряє легітимність виборця та кандидата.
- Якщо все коректно, голос зараховується, а стан виборця змінюється на "Voted".
- Після завершення голосування, ЦВК підраховує кількість голосів за кожного кандидата.

7. Обробка помилок На кожному етапі обробки голосу ЦВК перевіряє на можливі помилки:

- VoterIdAbsent: Якщо ідентифікатор виборця відсутній в системі.
- VoterInvalidState: Якщо виборець має неправильний стан для голосування.
- InvalidCandidate: Якщо обраний кандидат не зареєстрований в системі.
- FailedToDecipherPacket: Якщо не вдалося розшифрувати пакет даних.
- FailedToDeserializePackets: Якщо не вдалося десеріалізувати дані пакетів.
- PacketDataIsEmpty: Якщо дані пакету порожні.
- Інші помилки, пов'язані з криптографічними операціями та обробкою даних.

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.

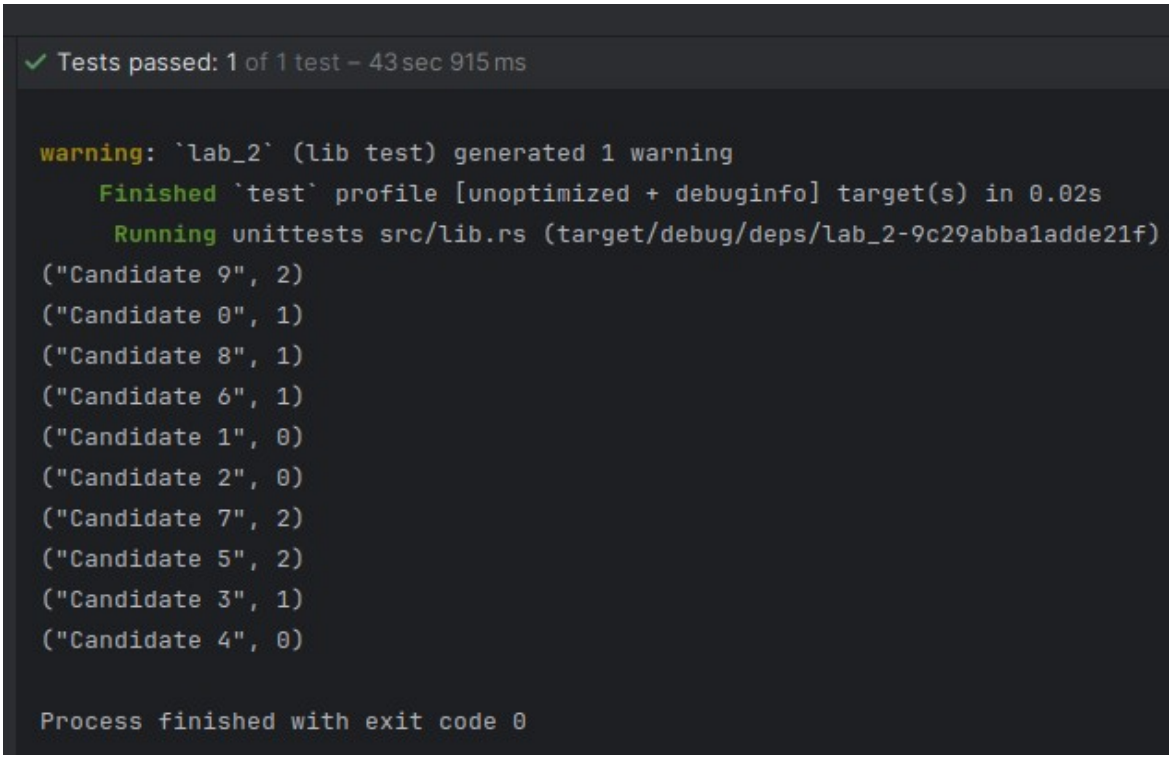

Рисунок 4.1 — Загальний тест

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

Рисунок 4.2 — Перевірка неможливості проголосувати декілька разів

На рисунку 4.3 перевіримо, що людина, яка не має права голосувати і

справді не в змозі це зробити.

Рисунок 4.3 — Перевірка повідомлення про помилку за відсутності права голосувати

5 ДОСЛІДЖЕННЯ ПРОТОКОЛУ

Чи можуть голосувати ті, хто не має на це права? Система має ефективні механізми для запобігання голосуванню неавторизованих осіб. У методі consume_packets класу Cec здійснюється перевірка стану виборця перед обробкою його пакету. Важливо зазначити, що стан виборця змінюється з NotRegistered на Registered тільки після успішної перевірки пакету. Крім того, у методі process_vote проводиться додаткова перевірка, чи має виборець стан Registered перед зарахуванням голосу. Ці механізми ефективно запобігають голосуванню тих, хто не має на це права.

Чи може виборець голосувати кілька разів? Система має надійний захист від багаторазового голосування. Після успішного голосування стан виборця автоматично змінюється з Registered на Voted. При кожному наступному зверненні до методу process_vote проводиться перевірка, чи має виборець стан Registered. Якщо виявляється, що стан вже змінено на Voted, голос не буде зараховано. Таким чином, система ефективно запобігає можливості багаторазового голосування одним виборцем.

Чи може хтось (інший виборець, ВК, стороння людина) дізнатися за кого проголосували інші виборці? Для забезпечення конфіденційності голосування система використовує шифрування RSA та техніку засліплення (blinding). Ці механізми захищають дані при передачі та приховують зв'язок між виборцем та його вибором. ЦВК отримує лише зашифрований голос, який розшифровується тільки в момент підрахунку. Однак, варто зазначити, що ЦВК має доступ до розшифрованих голосів під час підрахунку. Теоретично, це створює можливість для зловмисника з доступом до системи ЦВК пов'язати голоси з конкретними виборцями. Таким чином, хоча система забезпечує високий рівень анонімності, повна гарантія відсутня через централізований характер ЦВК.

Чи може інший виборець чи стороння людина проголосувати замість іншого зареєстрованого виборця? У даній системі кожен виборець має унікальний VoterId, який відомий лише самому виборцю та ЦВК. Важливо відзначити, що система не використовує підписування повідомлень приватним

ключем виборця для аутентифікації. Безпека в цьому аспекті повністю залежить від надійності зберігання та передачі VoterId. У випадку компрометації VoterId, теоретично можливе голосування від імені іншого виборця. Тому рівень безпеки системи в цьому аспекті напряму залежить від надійності захисту VoterId.

Чи може хтось (інший виборець, ВК, стороння людина) таємно змінити голос в бюлетені? Всі голоси в системі шифруються публічним ключем ЦВК, що забезпечує їх захист від зміни сторонніми особами. Проте, важливо зазначити, що система повністю довіряє ЦВК. Це означає, що теоретично ЦВК має можливість змінити голос без виявлення, оскільки має доступ до розшифрованих даних. Інші виборці або сторонні особи не мають можливості змінити голос через використання шифрування.

Чи може виборець перевірити, що його голос вірно врахований при підведенні кінцевих підсумків? У поточній реалізації системи відсутній прямий механізм для виборця перевірити правильність врахування свого голосу після його подачі. Система не надає квитанцію або інший доказ голосування, який можна було б використати для подальшої перевірки. Таким чином, виборці змушені повністю довіряти системі та ЦВК щодо правильного підрахунку голосів, не маючи можливості індивідуальної перевірки свого голосу після його подачі.

ВИСНОВОК

У ході виконання лабораторної роботи було досліджено та реалізовано систему електронного голосування, що використовує криптографічні методи для забезпечення безпеки та анонімності процесу голосування.

Основні результати дослідження:

1. Реалізована система успішно запобігає голосуванню неавторизованих осіб та багаторазовому голосуванню завдяки ефективному механізму контролю стану виборців.
2. Використання шифрування RSA та техніки сліпих підписів забезпечує високий рівень конфіденційності, ускладнюючи можливість зв'язати конкретний голос з виборцем.
3. Система базується на довірі до Центральної Виборчої Комісії (ЦВК), яка має доступ до розшифрованих голосів під час підрахунку, що створює потенційний ризик порушення анонімності.
4. Безпека голосування значною мірою залежить від збереження конфіденційності ідентифікатора виборця (VoterId), оскільки система не використовує додаткові механізми аутентифікації.
5. Поточна реалізація не передбачає можливості для виборця перевірити правильність врахування свого голосу після його подачі, що може знизити довіру до системи.

Аналіз показав, що реалізований протокол має ряд сильних сторін, зокрема у запобіганні несанкціонованому голосуванню та забезпеченні базового рівня анонімності. Проте, виявлено і деякі обмеження, пов'язані з централізованою природою системи та відсутністю механізмів індивідуальної верифікації голосів.

Загалом, проведене дослідження демонструє потенціал та складності реалізації систем електронного голосування, підкреслюючи важливість балансу між безпекою, анонімністю та прозорістю в таких системах. Реалізований протокол Е-голосування зі сліпими підписами надає базовий рівень захисту та анонімності, але також виявляє деякі обмеження, характерні для

централізованих систем електронного 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.

```
extern crate derive_more;

use num_bigint::BigUint;
use num_traits::ToPrimitive;

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

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

    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 struct ApplyBlindingOps {
        mask_multiplicative: BigUint,
        product_number: ProductNumber
    }

    impl ApplyBlindingOps {
        pub fn apply(&self, data: &[u8]) -> Vec<BigUint> {
            data.iter().map(|byte| {
                let m: BigUint = (*byte).into();
                (m * &self.mask_multiplicative) % &self.product_number.0
            })
                .collect::<Vec<_>>()
        }
    }

    #[derive(Serialize, Deserialize)]
    pub struct UnapplyBlindingOps {
        mask_multiplicative_inverse: BigUint,
        product_number: ProductNumber
    }

    impl UnapplyBlindingOps {
        pub fn apply(&self, mut data: Vec<BigUint>) -> Vec<BigUint> {
            for unit in &mut data {
                *unit *= &self.mask_multiplicative_inverse;
                *unit %= &self.product_number.0;
            }
            data
        }
    }
```

```
    pub fn create_blinding_ops(public_key_ref: &PublicKeyRef) ->
(ApplyBlindingOps, UnapplyBlindingOps) {
        let (r, r_inv) = loop {
            let r =
rand::thread_rng().gen_biguint_below(public_key_ref.product_number);
            if let Some(r_inv) = r.modinv(public_key_ref.product_number) {
                break (r, r_inv);
            }
        };

        let (e, n) = public_key_ref.get_parts();
        let mask_multiplicative = modpow(&r, e, n).unwrap();

        (
            ApplyBlindingOps { mask_multiplicative, product_number: n.clone() },
            UnapplyBlindingOps { mask_multiplicative_inverse: r_inv,
product_number: n.clone() }
        )
    }

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

        #[test]
        fn test_blinding() -> Result<(), CipherDataError> {
            let receiver = KeyPair::new();

            let message = "message";

            let (apply_ops, unapply_ops) = {
                create_blinding_ops(&receiver.get_public_key_ref())
            };

            let original_data = {
                let unblinded_signed_data = {
                    let signed_blinded_data = {
                        let unciphered_blinded_data = {
                            let ciphered_blinded_data = {
                                let blinded_data =
apply_ops.apply(message.as_bytes());
                                cipher_data_biguint(
```

```
                                    &receiver.get_public_key_ref(), blinded_data
                                )?
                            };
                            cipher_data_biguint(
                                &receiver.get_private_key_ref(),
ciphered_blinded_data
                            )?
                        };
                        cipher_data_biguint(
                            &receiver.get_private_key_ref(),
unciphered_blinded_data
                        )?
                    };
                    unapply_ops.apply(signed_blinded_data)
                };

                cipher_data_biguint(
                    &receiver.get_public_key_ref(), unblinded_signed_data
                )?
            };

            let original_bytes = original_data.into_iter()
                .map(|e| {
                    e.to_u8().unwrap()
                }).collect::<Vec<_>>();

            let original_message = String::from_utf8(original_bytes).unwrap();

            assert_eq!(original_message, message);
            println!("{}", original_message);
            Ok(())
        }

        #[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 voter {
    use std::num::NonZeroUsize;
    use derive_more::From;
    use getset::Getters;
    use num_bigint::BigUint;
    use rand::prelude::SliceRandom;
    use serde::{Deserialize, Serialize};
    use thiserror::Error;
    use crate::{convert_to_bytes, rsa};

    #[derive(Serialize, Deserialize)]
    pub(crate) struct PacketsData {
        pub blinded_candidate_id_vec_vec: Vec<BlindedCandidateIdVec>,
        pub unapply_blind_ops: rsa::UnapplyBlindingOps,
    }

    #[derive(
        Serialize, Deserialize, Clone,
        Default, PartialEq, Eq, Hash, Debug
    )]
    pub struct VoterId(pub BigUint);

    #[derive(Default, Getters)]
    #[getset(get = "pub with_prefix")]
    pub struct Voter {
        // key_pair: rsa::KeyPair,
        id: VoterId
    }

    #[derive(Serialize, Clone)]
    struct CandidateIdRef<'a> {
        candidate: &'a str,
        id: &'a VoterId
    }

    #[derive(Serialize, Deserialize, Getters, Default, Clone)]
```

```
#[getset(get = "pub with_prefix")]
pub struct CandidateId {
    candidate: String,
    id: VoterId
}

#[derive(Deserialize, Default, Clone)]
pub struct CandidateIdVec(pub Vec<CandidateId>);

#[derive(Serialize, Default, Clone)]
pub struct CandidateIdRefVec<'a>(Vec<CandidateIdRef<'a>>);

#[derive(Serialize, Deserialize, Clone, Default)]
pub struct SerializedCandidateIdVec(pub Vec<u8>);

#[derive(Serialize, Deserialize, Default, Clone)]
pub struct BlindedCandidateIdVec(pub Vec<BigUint>);

#[derive(Error, Debug)]
pub enum ProducePacketsError {
    #[error("Failed to serialize candidate_id: {0}")]
    SerializedCandidateId(bincode::Error),
    #[error("Failed to serialize packets_data: {0}")]
    SerializePackets(bincode::Error),
    #[error("Failed to cipher packets_data: {0}")]
    CipherPacketsData(#[from] rsa::CipherDataError),
}

#[derive(Error, Debug, From)]
#[error("Failed to convert unciphered data to bytes")]
pub struct UncipheredConvertToBytesError(BigUint);

#[derive(Error, Debug)]
pub enum VoteError {
    #[error("Failed to uncipher vote: {0}")]
    FailedUncipherVote(#[from] rsa::CipherDataError),
    #[error(transparent)]
    FailedUncipheredConvertToBytes(#[from] UncipheredConvertToBytesError),
    #[error("Failed to deserialize candidate id vec: {0}")]
    FailedDeserializeCandidateIdVec(bincode::Error),
    #[error("Candidate id vec is empty")]
    CandidateIdVecIsEmpty,
    #[error("Failed serialize candidate id")]
    FailedSerializeCandidateId(bincode::Error),
```

```
        #[error("Failed serialize candidate id")]
        FailedCipherCandidateId(rsa::CipherDataError)
    }

    impl Voter {
        pub fn new() -> Self {
            Self {
                // key_pair: rsa::KeyPair::new(),
                id: VoterId(rsa::generate_num())
            }
        }

        pub fn produce_packets<'a>(
            &'a self,
            candidate_id_vec_number: NonZeroUsize,
            candidates: impl Iterator<Item=&'a String> + Clone,
            cec_public_key: &'a rsa::PublicKeyRef
        ) -> Result<Vec<BigUint>, ProducePacketsError> {
            let ser_candidate_id_vec = {
                let candidate_id_pairs = CandidateIdRefVec(
                    candidates.clone().map(
                        |c| CandidateIdRef {candidate: c.as_ref(), id:
&self.id }
                    ).collect::<Vec<_>>()
                );
                bincode::serialize(&candidate_id_pairs)
                    .map_err(ProducePacketsError::SerializedCandidateId)
                    .map(|serialized_candidate_ids| vec![
                        SerializedCandidateIdVec(serialized_candidate_ids);
                        candidate_id_vec_number.get()
                    ])?
            };
            let (apply_blind_ops, unapply_blind_ops) = {
                rsa::create_blinding_ops(cec_public_key)
            };
            let blinded_candidate_id_vec_vec = {
                ser_candidate_id_vec.into_iter().map(|candidate_id| {

BlindedCandidateIdVec(apply_blind_ops.apply(&candidate_id.0))
                }).collect::<Vec<_>>()
            };
            let packets_data = PacketsData{ blinded_candidate_id_vec_vec,
unapply_blind_ops };
            let serialized_packet = bincode::serialize(&packets_data)
```

```
                .map_err(ProducePacketsError::SerializePackets)?;
            rsa::cipher_data_u8(cec_public_key,
&serialized_packet).map_err(Into::into)
        }

        pub fn accept_signed_packet_and_vote(
            &self,
            signed_blinded_vote: Vec<BigUint>,
            unapply_blinding_ops: rsa::UnapplyBlindingOps,
            cec_public_key: &rsa::PublicKeyRef
        ) -> Result<Vec<BigUint>, VoteError> {
            let signed_vote = unapply_blinding_ops.apply(signed_blinded_vote);
            rsa::cipher_data_biguint(cec_public_key, signed_vote)
                .map_err(VoteError::FailedUncipherVote)
                .and_then(|data|
                    convert_to_bytes::<UncipheredConvertToBytesError>(&data)
                        .map_err(VoteError::from)
                )
                .and_then(|byte_data|
                    bincode::deserialize::<CandidateIdVec>(&byte_data)
                        .map_err(VoteError::FailedDeserializeCandidateIdVec)
                )
                .and_then(|candidate_id_vec| {
                    candidate_id_vec.0.choose(&mut rand::thread_rng())
                        .ok_or(VoteError::CandidateIdVecIsEmpty)
                        .and_then(|candidate_id| {
                            bincode::serialize(&candidate_id)
                                .map_err(VoteError::FailedSerializeCandidateId)
                        })
                })
                .and_then(|serialized_candidate_id| {
                    rsa::cipher_data_u8(cec_public_key,
&serialized_candidate_id)
                        .map_err(VoteError::FailedCipherCandidateId)
                })
        }
    }
}

mod cec {
    use std::collections::HashMap;
    use derive_more::From;
    use num_bigint::BigUint;
    use num_integer::Integer;
```

```
use thiserror::Error;
use crate::{convert_to_bytes, rsa, voter};

pub struct Cec {
    candidates: HashMap<String, u64>,
    voters_state: HashMap<voter::VoterId, VoterState>,
    key_pair: rsa::KeyPair,
}

#[derive(Error, Debug, From)]
#[error("Failed to convert unciphered to bytes: {0}")]
pub struct ConvertUncipheredToBytesError(BigUint);

#[derive(Error, Debug)]
pub enum VoteError {
    #[error("Failed to decipher vote: {0}")]
    FailedDecipherVote(rsa::CipherDataError),
    #[error(transparent)]
    ConvertUncipheredToBytes(#[from] ConvertUncipheredToBytesError),
    #[error("Failed to deserialize candidate id vec: {0}")]
    FailedToDeserializeCandidateId(#[from] bincode::Error),
    #[error("Voter id is absent: {0:?}")]
    VoterIdAbsent(voter::VoterId),
    #[error("Voter has invalid state: {0:?}")]
    VoterInvalidState(VoterState),
    #[error("Invalid candidate: {0}")]
    InvalidCandidate(String),
}

#[derive(Debug, PartialEq, Eq, Clone)]
pub enum CanVoteState {
    NotRegistered,
    Registered,
    Voted
}

#[derive(Debug, Clone, PartialEq, Eq)]
pub enum VoterState {
    CanVote(CanVoteState),
    CanNotVote,
}

#[derive(Error, Debug, From)]
#[error("Failed to convert deciphered packet to byte data: {0}")]
```

```
pub struct ConvertToByteDataError(BigUint);

#[derive(Error, Debug, From)]
#[error("Failed to convert unsigned candidate_id_vec to byte data: {0}")]
pub struct ConvertUnsignedCandidateIdVecToByteDataError(BigUint);

#[derive(Error, Debug)]
pub enum ConsumePacketsError {
    #[error("Failed to decipher packet: {0}")]
    FailedToDecipherPacket(rsa::CipherDataError),
    #[error(transparent)]
    FailedConvertToByteData(#[from] ConvertToByteDataError),
    #[error("Failed to deserialize packets_data: {0}")]
    FailedToDeserializePackets(bincode::Error),
    #[error("Failed to sign candidate_id_vec: {0}")]
    FailedSignCandidateIdVec(rsa::CipherDataError),
    #[error("Packet data is empty")]
    PacketDataIsEmpty,
    #[error("Failed to unsign candidate_id: {0}")]
    FailedUnsignCandidateIdVec(rsa::CipherDataError),
    #[error(transparent)]
    FailedConvertUnsignedCandidateIdVecToByteData(
        #[from] ConvertUnsignedCandidateIdVecToByteDataError
    ),
    #[error("Failed to deserialize candidate_id: {0}")]
    FailedDeserializedCandidateId(bincode::Error),
    #[error(transparent)]
    CheckCandidateIdError(#[from] CheckCandidateIdError),
}

#[derive(Error, Debug)]
pub enum CheckCandidateIdError {
    #[error("Invalid candidate: {0}")]
    InvalidCandidateError(String),
    #[error("Invalid voter id: {0:?}")]
    InvalidVoterId(voter::VoterId),
    #[error("Invalid voter state: {0:?}")]
    InvalidVoterState(VoterState),
    #[error("Ids are not the same: {0:?} != {1:?}")]
    IdsAreNotTheSame(voter::VoterId, voter::VoterId),
    #[error("Real: {0}, packet: {1}")]
    CandidatesListLenNotMatch(usize, usize)
}
```

```
    fn check_packet<'a>(
        candidate_id_vec_vec: &'a [voter::CandidateIdVec],
        candidates: &'a HashMap<String, u64>,
        voters_state: &'a HashMap<voter::VoterId, VoterState>,
    ) -> Result<voter::VoterId, CheckCandidateIdError> {
        for candidate_id_vec in candidate_id_vec_vec.iter() {
            if candidate_id_vec.0.len() != candidates.len() {
                return Err(
                    CheckCandidateIdError::CandidatesListLenNotMatch(
                        candidates.len(), candidate_id_vec.0.len()
                    )
                )
            }
            for (candidate_id, true_candidate) in
candidate_id_vec.0.iter().zip(candidates.keys()) {
                if candidate_id.get_candidate() != true_candidate {
                    return
Err(CheckCandidateIdError::InvalidCandidateError(candidate_id.get_candidate().cl
one()));
                }
                if let Some(voter_state) =
voters_state.get(candidate_id.get_id()) {
                    match voter_state {
                        VoterState::CanVote(can_vote_state) => {
                            match can_vote_state {
                                CanVoteState::NotRegistered => (),
                                _ => return
Err(CheckCandidateIdError::InvalidVoterState(voter_state.clone())),
                            }
                        },
                        VoterState::CanNotVote => {
                            return
Err(CheckCandidateIdError::InvalidVoterState(voter_state.clone()))
                        }
                    }
                } else {
                    return
Err(CheckCandidateIdError::InvalidVoterId(candidate_id.get_id().clone()))
                }
            }
        }

        let first = &candidate_id_vec_vec[0].0.first().unwrap();
        for candidate_id_vec in candidate_id_vec_vec.iter() {
```

```
            for candidate_id in candidate_id_vec.0.iter() {
                if first.get_id() != candidate_id.get_id() {
                    return Err(CheckCandidateIdError::IdsAreNotTheSame(
                        first.get_id().clone(), candidate_id.get_id().clone()
                    ))
                }
            }
        }
        Ok(first.get_id().clone())
    }

    impl Cec {
        pub fn new(
            candidates: impl Iterator<Item=String>,
            voters_state: HashMap<voter::VoterId, VoterState>
        ) -> Self {
            Self {
                candidates: HashMap::from_iter(
                    candidates.map(|c| (c, 0u64))
                ),
                voters_state,
                key_pair: rsa::KeyPair::new()
            }
        }

        pub fn consume_packets(
            &mut self,
            ciphered_data: Vec<BigUint>
        ) -> Result<(Vec<BigUint>, rsa::UnapplyBlindingOps),
ConsumePacketsError> {
            let (unapply_blind_ops, mut signed_blinded_candidate_id_vec_vec) = {
                let packet_data: voter::PacketsData = {
                    let deciphered_byte_data = {
                        let deciphered_data = rsa::cipher_data_biguint(
                            &self.key_pair.get_private_key_ref(), ciphered_data
                        ).map_err(|err| {
                            ConsumePacketsError::FailedToDecipherPacket(err)
                        })?;

convert_to_bytes::<ConvertToByteDataError>(&deciphered_data)?
                    };
                    bincode::deserialize(&deciphered_byte_data)
                        .map_err(ConsumePacketsError::FailedToDeserializePackets
)?
```

```
                };
                (
                    packet_data.unapply_blind_ops,
                    packet_data.blinded_candidate_id_vec_vec
                        .into_iter()
                        .try_fold(
                            Vec::new(),
                            |mut acc, candidate_id_vec| {
                                acc.push(
                                    rsa::cipher_data_biguint(
                                        &self.key_pair.get_private_key_ref(),
                                        candidate_id_vec.0
                                    )?
                                );
                                Ok(acc)
                            }
                        )
                        .map_err(ConsumePacketsError::FailedSignCandidateIdVec)?
                )
            };

            if signed_blinded_candidate_id_vec_vec.is_empty() {
                Err(ConsumePacketsError::PacketDataIsEmpty)
            } else {
                let last = signed_blinded_candidate_id_vec_vec.pop().unwrap();
                let unblinded_signed_candidate_id_vec_vec =
signed_blinded_candidate_id_vec_vec
                    .into_iter()
                    .map(
                        |candidate_id_vec|
unapply_blind_ops.apply(candidate_id_vec)
                    )
                    .collect::<Vec<_>>();
                let check_candidate_id_vec_vec = {
                    unblinded_signed_candidate_id_vec_vec.into_iter()
                        .try_fold(
                            Vec::new(),
                            |mut acc, signed_candidate_id_vec| {
                                rsa::cipher_data_biguint(
                                    &self.key_pair.get_public_key_ref(),
                                    signed_candidate_id_vec
                                ).map_err(ConsumePacketsError::FailedUnsignCandi
dateIdVec)
                                    .and_then(|ser_big_candidate_id_vec| {
```

```
convert_to_bytes::<ConvertUnsignedCandidateIdVecToByteDataError>(&ser_big_candid
ate_id_vec)
                                        .map_err(Into::into)
                                    })
                                    .and_then(|ser_candidate_id_vec| {

bincode::deserialize::<voter::CandidateIdVec>(&ser_candidate_id_vec)
                                        .map_err(ConsumePacketsError::Failed
DeserializedCandidateId)
                                    })
                                    .map(|candidate_id_vec| {
                                        acc.push(candidate_id_vec);
                                        acc
                                    })
                            }
                        )?
                };
                let voter_id = check_packet(&check_candidate_id_vec_vec,
&self.candidates, &self.voters_state)?;
                *self.voters_state.get_mut(&voter_id).unwrap() =
VoterState::CanVote(CanVoteState::Registered);
                Ok((last, unapply_blind_ops))
            }
        }

        pub fn process_vote(&mut self, vote: Vec<BigUint>) -> Result<(),
VoteError> {
            rsa::cipher_data_biguint(&self.key_pair.get_private_key_ref(), vote)
                .map_err(VoteError::FailedDecipherVote)
                .and_then(|ser_big_candidate_id_vec| {

convert_to_bytes::<ConvertUncipheredToBytesError>(&ser_big_candidate_id_vec)
                        .map_err(VoteError::from)
                })
                .and_then(|ser_candidate_id_vec| {

bincode::deserialize::<voter::CandidateId>(&ser_candidate_id_vec)
                        .map_err(VoteError::FailedToDeserializeCandidateId)
                })
                .and_then(|candidate_id| {
                    self.voters_state.get_mut(candidate_id.get_id())
                        .ok_or_else(||
VoteError::VoterIdAbsent(candidate_id.get_id().clone()))
```

```
                        .and_then(|voter_state| {
                            if *voter_state !=
VoterState::CanVote(CanVoteState::Registered) {

Err(VoteError::VoterInvalidState(voter_state.clone()))
                            } else {

self.candidates.get_mut(candidate_id.get_candidate())
                                    .ok_or_else(||
VoteError::InvalidCandidate(candidate_id.get_candidate().clone()))
                                    .map(|votes_number| {
                                        votes_number.inc();
                                        *voter_state =
VoterState::CanVote(CanVoteState::Voted);
                                    })
                            }
                        })
                })
        }

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

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

#[cfg(test)]
mod tests {
    use std::collections::HashMap;
    use std::num::NonZeroUsize;
    use crate::{cec, voter};

    #[test]
    fn test_vote() {
        let mut voters = (0..10).into_iter().map(|_| voter::Voter::new())
            .collect::<Vec<_>>();

        let mut cec = cec::Cec::new(
            (0..10).into_iter().map(|i| format!("Candidate {}", i)),
            HashMap::from_iter(voters.iter().map(|v| (
```

```
                v.get_id().clone(),
cec::VoterState::CanVote(cec::CanVoteState::NotRegistered)))
            )
        );

        for voter in &mut voters {
            let packet = voter.produce_packets(
                NonZeroUsize::new(10).unwrap(), cec.get_candidates().keys(),
&cec.get_public_key()
            ).unwrap();
            let signed_candidate_id_vec = cec.consume_packets(packet).unwrap();
            let ciphered_vote = voter.accept_signed_packet_and_vote(
                signed_candidate_id_vec.0, signed_candidate_id_vec.1,
&cec.get_public_key()
            ).unwrap();
            cec.process_vote(ciphered_vote).unwrap()
        }

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

    #[test]
    fn test_double_vote() {
        let mut voters = (0..10).into_iter().map(|_| voter::Voter::new())
            .collect::<Vec<_>>();

        let mut cec = cec::Cec::new(
            (0..10).into_iter().map(|i| format!("Candidate {}", i)),
            HashMap::from_iter(voters.iter().map(|v| (
                v.get_id().clone(),
cec::VoterState::CanVote(cec::CanVoteState::NotRegistered)))
            )
        );

        for voter in &mut voters {
            let packet = voter.produce_packets(
                NonZeroUsize::new(10).unwrap(), cec.get_candidates().keys(),
&cec.get_public_key()
            ).unwrap();
            let signed_candidate_id_vec = cec.consume_packets(packet).unwrap();

            let packet = voter.produce_packets(
```

```
                NonZeroUsize::new(10).unwrap(), cec.get_candidates().keys(),
&cec.get_public_key()
            ).unwrap();

            let signed_candidate_id_vec = cec.consume_packets(packet);

            match signed_candidate_id_vec {
                Err(
                    cec::ConsumePacketsError::CheckCandidateIdError(
                        cec::CheckCandidateIdError::InvalidVoterState(

cec::VoterState::CanVote(cec::CanVoteState::Registered)
                        )
                    )
                ) => {},
                _ => assert!(false)
            }
        }
    }

    #[test]
    fn test_illegal_vote() {
        let mut voters = (0..10).into_iter().map(|_| voter::Voter::new())
            .collect::<Vec<_>>();

        let mut cec = cec::Cec::new(
            (0..10).into_iter().map(|i| format!("Candidate {}", i)),
            HashMap::from_iter(voters.iter().map(|v| (
                v.get_id().clone(), cec::VoterState::CanNotVote))
            )
        );

        for voter in &mut voters {
            let packet = voter.produce_packets(
                NonZeroUsize::new(10).unwrap(), cec.get_candidates().keys(),
&cec.get_public_key()
            ).unwrap();
            let signed_candidate_id_vec = cec.consume_packets(packet);

            match signed_candidate_id_vec {
                Err(
                    cec::ConsumePacketsError::CheckCandidateIdError(
                        cec::CheckCandidateIdError::InvalidVoterState(
                            cec::VoterState::CanNotVote
```

```
                        )
                    )
                ) => {},
                _ => assert!(false)
            }
        }
    }
}
```