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 №1

Протоколи й алгоритми електронного голосування

Тема: Простий протокол Е-голосування

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

Перевірив:
Нестерук А. О.

Київ 2024

ЗМІСТ

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ

Дослідити протокол простого Е-голосування

2 ЗАВДАННЯ

Змоделювати простий протокол Е-голосування будь-якою мовою програмування та провести його дослідження. Для кодування повідомлень використовувати метод гамування, 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. Код роботи можна розглянути у додатку А. Поверхнево опишемо архітектуру рішення:

- Модуль виборця (voter): Цей модуль представляє виборця в системі. Кожен виборець має свою пару ключів RSA. Модуль містить функціональність для створення зашифрованого голосу, який включає зашифрований ключ гамування, цифровий підпис та зашифрований голос.
- Модуль Центральної виборчої комісії (CEC): Цей модуль представляє центральний орган, який керує процесом голосування. CEC має список кандидатів, стан виборців та свою власну пару ключів RSA. Він відповідає за обробку голосів, перевірку їх дійсності та підрахунок результатів.

4 ДЕМОНСТРАЦІЯ РОБОТИ ПРОТОКОЛУ

Для початку опишемо загальну схему роботи протоколу:

1. Підготовка
   - Центральна виборча комісія (ЦВК) ініціалізується зі списком кандидатів.
   - ЦВК створює пару ключів RSA (публічний і приватний).
   - Реєструються виборці, кожному присвоюється стан "може голосувати".
   - Кожен виборець генерує власну пару ключів RSA.
2. Процес голосування
   - Виборець обирає кандидата.
   - Виборець генерує випадковий ключ гамування.
   - Голос (ім'я кандидата) шифрується ключем гамування.
   - Виборець створює хеш зашифрованого голосу.
   - Хеш підписується приватним ключем виборця.
   - Ключ гамування шифрується публічним ключем ЦВК.
3. Подання голосу

   Виборець надсилає до ЦВК: зашифрований голос, підписаний хеш зашифрованого голосу, зашифрований ключ гамування
4. Обробка голосу ЦВК
   - ЦВК перевіряє, чи виборець має право голосувати.
   - Перевіряється цифровий підпис, використовуючи публічний ключ виборця.
   - ЦВК розшифровує ключ гамування своїм приватним ключем.
   - Голос розшифровується за допомогою ключа гамування.
   - Перевіряється, чи є кандидат у списку.
   - Якщо все коректно, голос зараховується, а стан виборця змінюється на "проголосував".
5. Підрахунок результатів

   Після завершення голосування ЦВК підраховує кількість голосів за кожного кандидата.

6. Обробка помилок

На кожному етапі обробки голосу ЦВК перевіряє на можливі помилки:

- VoterNotRegistered: Якщо виборець не зареєстрований в системі.
- VoterCanNotVote: Якщо виборець не має права голосувати.
- VoterAlreadyVoted: Якщо виборець вже проголосував.
- GammingKeyHashedNotMatch: Якщо хеш зашифрованого голосу не відповідає підпису.
- FailedParseGammingKey: Якщо не вдалося розшифрувати ключ гамування.
- FailedParseCandidate: Якщо не вдалося розшифрувати ім'я кандидата.
- CandidateNotRegistered: Якщо розшифрований кандидат не зареєстрований в системі.

У разі виникнення будь-якої з цих помилок, голос не зараховується, а система зберігає інформацію про помилку для подальшого аналізу.

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

На рисунку 4.2 проведемо дослідження того, що за умови відсутності права голосу в виборця ми отримуємо помилку.

На рисунку 4.3 проведемо дослідження того, що за умови повторного голосування ми отримуємо помилку.

Таким чином ми провели мінімальне тестування справності роботи протоколу.

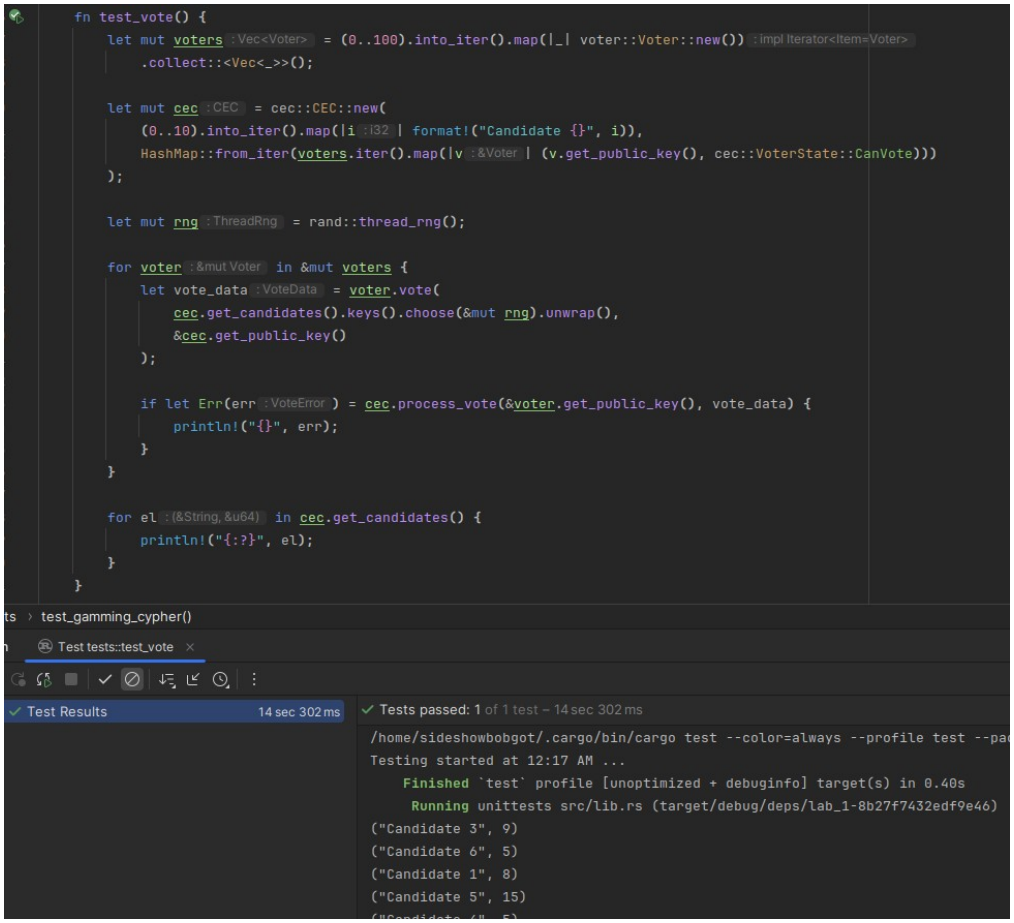

Рисунок 4.1 — Тестування з виборцями, що можуть голосувати

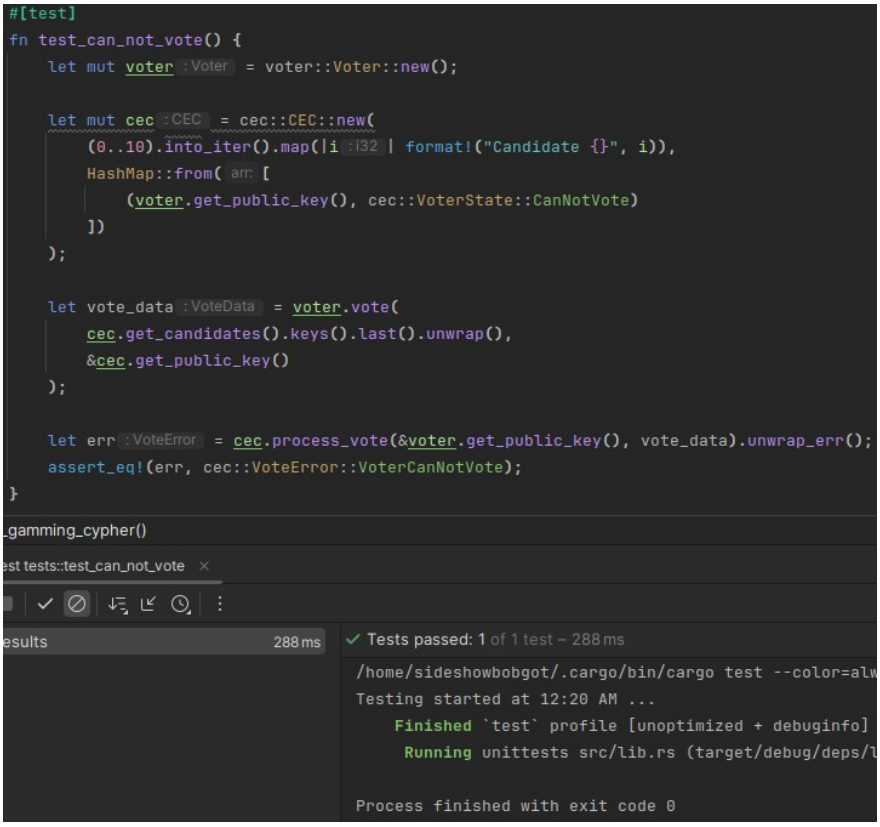

Рисунок 4.2 — Тестування можливості голосувати за умови відсутності права голосу

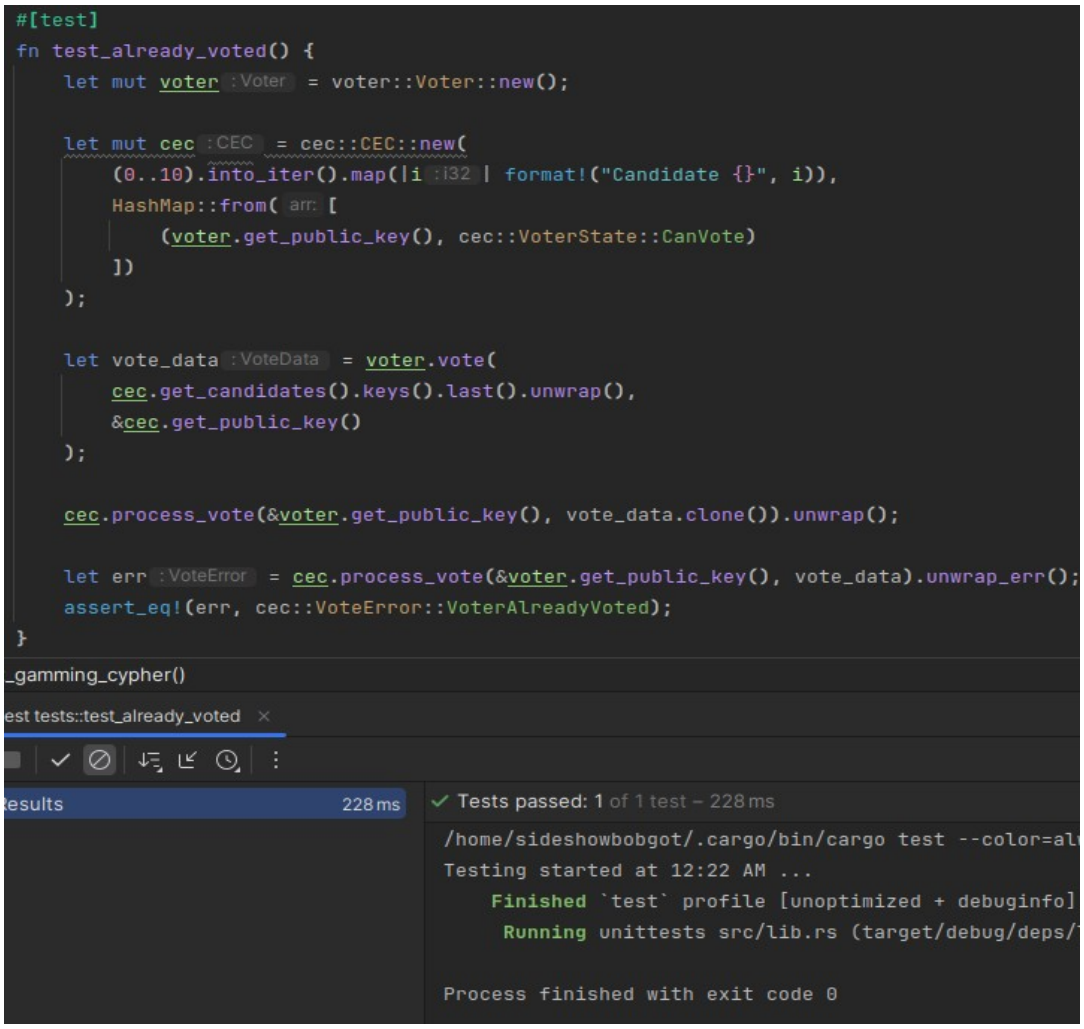

Рисунок 4.3 — Тестування можливості голосувати за умови, що виборець вже проголосував

5 ДОСЛІДЖЕННЯ ПРОТОКОЛУ

Чи можуть голосувати ті, хто не має на це права: протокол перевіряє стан виборця перед обробкою голосу. Якщо стан виборця "CanNotVote", голос не буде зарахований і буде повернута помилка VoterCanNotVote. Отже, ті, хто не має права голосувати, не зможуть це зробити.

Чи може виборець голосувати кілька разів: після успішного голосування стан виборця змінюється на "Voted". При спробі проголосувати знову, система поверне помилку VoterAlreadyVoted. Таким чином, протокол запобігає повторному голосуванню.

Чи може хтось дізнатися, за кого проголосували інші виборці: голос шифрується ключем гамування, який в свою чергу шифрується публічним ключем ЦВК. Теоретично, тільки ЦВК може розшифрувати голос. Однак, якщо приватний ключ ЦВК буде скомпрометовано, конфіденційність голосів може бути порушена.

Чи може інший виборець чи стороння людина проголосувати замість іншого зареєстрованого виборця: кожен голос підписується приватним ключем виборця. Без доступу до цього ключа, неможливо створити дійсний підпис. Отже, якщо приватний ключ виборця не скомпрометовано, ніхто інший не зможе проголосувати від його імені.

Чи може хтось таємно змінити голос в бюлетені: голос шифрується і підписується. Будь-яка зміна зашифрованого голосу призведе до невідповідності підпису, і такий голос буде відхилено. Однак, якщо приватний ключ ЦВК буде скомпрометовано, теоретично можливо розшифрувати, змінити і знову зашифрувати голос.

Чи може виборець перевірити, що його голос врахований при підведенні кінцевих результатів: у поточній реалізації протоколу немає механізму для виборця перевірити, чи його голос був врахований правильно. Це є суттєвим недоліком системи з точки зору прозорості та аудиту.

ВИСНОВОК

У ході лабораторної роботи було досліджено простий протокол електронного голосування, реалізований на мові програмування Rust. Протокол включав використання методу гамування для шифрування голосів та алгоритму RSA для реалізації електронного цифрового підпису.

Основні компоненти системи включали модулі для виборців, Центральної виборчої комісії (ЦВК), та криптографічних операцій. Було реалізовано сценарії для обробки різних випадків порушення протоколу, таких як спроби повторного голосування, голосування неавторизованими особами, та некоректні дані голосування.

Дослідження протоколу показало, що він ефективно забезпечує базовий рівень безпеки та конфіденційності. Зокрема:

1. Система успішно запобігає голосуванню осіб, які не мають на це права. Механізм зміни стану виборця після голосування ефективно запобігає повторному голосуванню.
2. Шифрування голосів забезпечує їх конфіденційність, хоча це значною мірою залежить від безпеки приватного ключа ЦВК.
3. Використання цифрових підписів ефективно запобігає голосуванню від імені іншого виборця.
4. Цілісність голосів захищена від несанкціонованих змін завдяки використанню цифрових підписів.

Однак, були виявлені й певні обмеження протоколу:

1. Відсутність механізму для виборців перевірити, чи їхній голос був правильно врахований, що знижує прозорість системи.
2. Потенційна вразливість у випадку компрометації приватного ключа ЦВК, що може призвести до порушення конфіденційності голосів.
3. Відсутність механізмів для зовнішнього аудиту процесу голосування та підрахунку голосів.

Загалом, досліджений протокол демонструє базову функціональність та безпеку електронного голосування, але для реального застосування потребував

би значних удосконалень. Зокрема, необхідно було б впровадити механізми для перевірки голосів виборцями, покращити захист ключів ЦВК, додати можливості зовнішнього аудиту та підвищити загальну прозорість системи.

Ця лабораторна робота дала цінний досвід у розумінні принципів роботи систем електронного голосування, їх потенційних вразливостей та викликів, пов'язаних із забезпеченням безпеки та прозорості таких систем.

ДОДАТОК А ТЕКСТИ ПРОГРАМНОГО КОДУ

Тексти програмного коду

(Найменування програми (документа))

Жорсткий диск

(Вид носія даних)

(Обсяг програми (документа), арк.)

Студентів групи ІП-11 4 курсу

Панченка С. В

Лисенка А. Ю.

```
mod rsa {
    use std::ops::Rem;
    use lazy_static::lazy_static;
    use num_bigint::{BigUint};
    use num_traits::{One, Zero};
    use rand::Rng;
    pub use keys::{KeyPair, PublicKey};

    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)
    }

    mod keys {
        use lazy_static::lazy_static;
        use num_bigint::BigUint;
        use num_traits::{One, Zero};
        use crate::rsa::{generate_prime, TWO};

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

        fn quad_fold_hash(n: &BigUint, data: &[u8]) -> BigUint {
            data.iter().fold(BigUint::zero(), |acc, x| {
                (acc + BigUint::from(*x)).modpow(&TWO, n)
            })
        }

        fn apply_key(data: &BigUint, exp: &BigUint, modulus: &BigUint) ->
Option<BigUint> {
            if data > modulus {
                return None;
            }
            Some(data.modpow(exp, modulus))
        }

        pub struct PrivateKey {
            d: BigUint,
            n: BigUint
        }

        impl PrivateKey {
            pub fn apply(&self, data: &BigUint) -> Option<BigUint> {
                apply_key(data, &self.d, &self.n)
            }
        }

        #[derive(Debug, Hash, PartialEq, Eq)]
        pub struct PublicKey {
            n: BigUint
        }
```

```
impl PublicKey {
    pub fn apply(&self, data: &BigUint) -> Option<BigUint> {
        apply_key(data, &E_PRIME, &self.n)
    }

    pub fn quad_fold_hash(&self, data: &[u8]) -> BigUint {
        quad_fold_hash(&self.n, data)
    }
}

#[derive(Debug, Clone)]
pub struct KeyPair {
    n: BigUint,
    d: BigUint,
}

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

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

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

    pub fn quad_fold_hash(&self, data: &[u8]) -> BigUint {
```

```
                quad_fold_hash(&self.n, data)
            }

            pub fn get_private_key(&self) -> PrivateKey {
                PrivateKey { n: self.n.clone(), d: self.d.clone() }
            }

            pub fn get_public_key(&self) -> PublicKey {
                PublicKey { n: self.n.clone() }
            }
        }
        #[cfg(test)]
        mod tests {
            use super::{KeyPair};

            #[test]
            fn test_sign_verify() {
                let r = KeyPair::new();
                let message_hash = r.quad_fold_hash("message".as_bytes());

                let encrypted_hash =
r.get_private_key().apply(&message_hash).unwrap();
                assert_eq!(r.get_public_key().apply(&encrypted_hash).unwrap(),
message_hash);

                let encrypted_hash =
r.get_public_key().apply(&message_hash).unwrap();
                assert_eq!(r.get_private_key().apply(&encrypted_hash).unwrap(),
message_hash);
            }
        }
    }
}

fn gamming_cipher(message: &[u8], key: &[u8]) -> Vec<u8> {
    message.iter().zip(key.iter().cycle()).map(|(m, k)| m ^ k).collect()
}

mod voter {
    use getset::Getters;
    use num_bigint::BigUint;
    use crate::{gamming_cipher, rsa};

    #[derive(Getters, Clone)]
```

```
    #[getset(get = "pub with_prefix")]
    pub struct VoteData {
        encrypted_gamming_key: BigUint,
        rsa_signature: BigUint,
        gammed_vote: Vec<u8>
    }

    #[derive(Default)]
    pub struct Voter { key_pair: rsa::KeyPair }
    impl Voter {
        pub fn new() -> Self {
            Self { key_pair: rsa::KeyPair::new() }
        }

        pub fn vote(&mut self, candidate: &str, cec_public_key: &rsa::PublicKey)
-> VoteData {
            let gamming_key = rsa::generate_num();

            let gammed_vote = gamming_cipher(candidate.as_bytes(),
&gamming_key.to_bytes_le());
            let gammed_vote_hash = self.key_pair.quad_fold_hash(&gammed_vote);

            let rsa_signature =
self.key_pair.get_private_key().apply(&gammed_vote_hash).unwrap();
            let encrypted_gamming_key =
cec_public_key.apply(&gamming_key).unwrap();

            VoteData { encrypted_gamming_key, rsa_signature, gammed_vote }
        }

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

mod cec {
    use std::collections::HashMap;
    use thiserror::Error;
    use crate::rsa;
    use crate::{gamming_cipher, voter};
    use crate::rsa::PublicKey;

    pub struct CEC {
```

```
        candidates: HashMap<String, u64>,
        voters_state: HashMap<rsa::PublicKey, VoterState>,
        key_pair: rsa::KeyPair,
    }

    #[derive(Error, Debug, PartialEq, Eq)]
    pub enum VoteError {
        #[error("Gamming keys do not match")]
        GammingKeyHashedNotMatch,
        #[error("Failed parse gamming key")]
        FailedParseGammingKey,
        #[error("Failed parse candidate")]
        FailedParseCandidate,
        #[error("Invalid candidate")]
        CandidateNotRegistered,
        #[error("Voter has already voted")]
        VoterAlreadyVoted,
        #[error("Voter can not vote")]
        VoterCanNotVote,
        #[error("Voter is not registered")]
        VoterNotRegistered
    }

    #[derive(Debug)]
    pub enum VoterState {
        CanVote,
        CanNotVote,
        Voted,
    }

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

        pub fn process_vote(&mut self, voter_key: &rsa::PublicKey, vote_data:
voter::VoteData)
            -> Result<(), VoteError> {
```

```
            if let Some(state) = self.voters_state.get_mut(voter_key) {
                match state {
                    VoterState::Voted => Err(VoteError::VoterAlreadyVoted),
                    VoterState::CanNotVote => Err(VoteError::VoterCanNotVote),
                    VoterState::CanVote => {
                        let hash =
voter_key.quad_fold_hash(vote_data.get_gammed_vote());
                        let decrypted_hash =
voter_key.apply(vote_data.get_rsa_signature()).unwrap();

                        if hash == decrypted_hash {
                            if let Some(gamming_key) =
self.key_pair.get_private_key()
                                .apply(vote_data.get_encrypted_gamming_key()) {
                                if let Ok(candidate) = String::from_utf8(
                                    gamming_cipher(vote_data.get_gammed_vote(),
&gamming_key.to_bytes_le())
                                ) {
                                    if let Some(score) =
self.candidates.get_mut(candidate.as_str()) {
                                        *state = VoterState::Voted;
                                        *score += 1;
                                        Ok(())
                                    } else {
                                        Err(VoteError::CandidateNotRegistered)
                                    }
                                } else {
                                    Err(VoteError::FailedParseCandidate)
                                }
                            } else {
                                Err(VoteError::FailedParseGammingKey)
                            }
                        } else {
                            Err(VoteError::GammingKeyHashedNotMatch)
                        }
                    }
                }
            } else {
                Err(VoteError::VoterNotRegistered)
            }
        }

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

```
        }

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

#[cfg(test)]
mod tests {
    use std::collections::HashMap;
    use rand::seq::IteratorRandom;
    use super::{gamming_cipher, cec, voter};

    #[test]
    fn test_gamming_cypher() {
        let message = "spongebob";
        let key = "squarepants";

        let gammed = gamming_cipher(message.as_bytes(), key.as_bytes());
        let ungammed = String::from_utf8(gamming_cipher(&gammed,
key.as_bytes())).unwrap();

        assert_eq!(ungammed, message);
    }

    #[test]
    fn test_vote() {
        let mut voters = (0..100).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_public_key(),
cec::VoterState::CanVote)))
        );

        let mut rng = rand::thread_rng();

        for voter in &mut voters {
            let vote_data = voter.vote(
                cec.get_candidates().keys().choose(&mut rng).unwrap(),
                &cec.get_public_key()
            );
```

```
            if let Err(err) = cec.process_vote(&voter.get_public_key(),
vote_data) {
                println!("{}", err);
            }
        }

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

    #[test]
    fn test_can_not_vote() {
        let mut voter = voter::Voter::new();

        let mut cec = cec::CEC::new(
            (0..10).into_iter().map(|i| format!("Candidate {}", i)),
            HashMap::from([
                (voter.get_public_key(), cec::VoterState::CanNotVote)
            ])
        );

        let vote_data = voter.vote(
            cec.get_candidates().keys().last().unwrap(),
            &cec.get_public_key()
        );

        let err = cec.process_vote(&voter.get_public_key(),
vote_data).unwrap_err();
        assert_eq!(err, cec::VoteError::VoterCanNotVote);
    }

    #[test]
    fn test_already_voted() {
        let mut voter = voter::Voter::new();

        let mut cec = cec::CEC::new(
            (0..10).into_iter().map(|i| format!("Candidate {}", i)),
            HashMap::from([
                (voter.get_public_key(), cec::VoterState::CanVote)
            ])
        );
```

```
        let vote_data = voter.vote(
            cec.get_candidates().keys().last().unwrap(),
            &cec.get_public_key()
        );

        cec.process_vote(&voter.get_public_key(), vote_data.clone()).unwrap();

        let err = cec.process_vote(&voter.get_public_key(),
vote_data).unwrap_err();
        assert_eq!(err, cec::VoteError::VoterAlreadyVoted);
    }
}
```