École Polytechnique Fédérale de Lausanne

D-Voting:
e-Voting on Dela

by Auguste Baum (Master Semester Project)
Emilien Duc (Bachelor Semester Project)

Supervised by:

Prof. Dr. Bryan Ford (DEDIS)
Advisor

Noémien Kocher (DEDIS)
Supervisor

EPFL IC IINFCOM DEDIS
BC 210 (Bâtiment BC)
Station 14
CH-1015 Lausanne

January 7, 2022

Abstract

In a world with high-stakes elections and constant trust issues, blockchain technology has a high potential of easing voters' minds thanks to its auditability and lack of central third party. With this in mind, DEDIS has been a pioneer in the e-voting field, and has initiated the `D-Voting` project, an e-voting system based on the high-level blockchain components in the `Dela` project [4]. In this work, we bring substantial improvements to the previous iteration of the `D-Voting` system: more flexibility for the polling questions, increased security in the cryptographic protocols, more flexible test suite and increased robustness to failures, among others. Though we made no significant improvement to the performance of the system, we consider the project as fit to enter the last stage of its pre-launch development before it can be used in the EPFL faculty elections.

Contents

Abstract **2**

1 Introduction **5**

2 Background **7**
- 2.1 Security requirements 7
- 2.2 E-voting smart contract 7
- 2.3 Pedersen DKG 8
- 2.4 Neff shuffle 8

3 Design & Implementation **10**
- 3.1 Election process 10
  - 3.1.1 Design 10
  - 3.1.2 Implementation 11
- 3.2 Pedersen DKG protocol 11
  - 3.2.1 Design 11
  - 3.2.2 Implementation 12
- 3.3 Persistence of system state 13
  - 3.3.1 Design 13
  - 3.3.2 Implementation 13
- 3.4 Security of the Neff Shuffle algorithm 13
  - 3.4.1 Design 13
  - 3.4.2 Implementation 14
- 3.5 Diversification of election formats 15
  - 3.5.1 Design 15
  - 3.5.2 Implementation 16
- 3.6 Increasing the maximum ballot size 18
  - 3.6.1 Design 18
  - 3.6.2 Implementation 20

4 Evaluation **22**
- 4.1 Correctness tests 22

4.2 Performance 22

5 Future Work **25**

5.1 Security issues 25

5.1.1 Proof of decryption 25

5.1.2 Proof of encryption 25

5.1.3 Ballot re-encryption can de-anonymize users 26

5.2 Authentication of users 26

6 Conclusion **27**

Bibliography **29**

A Installation and testing **31**

Chapter 1

Introduction

Electronic voting is a current topic, as it stands at the meeting point between human sciences and technological sciences. Even though a large part of developed countries are governed democratically, the election process remains a contentious affair with voter suppression [8], allegations of fraud [6] and as a result, distrust from the public as a common occurrence. In this context, e-voting constitutes a (seemingly) simple solution to guarantee that the votes are reliably accounted for.

Even supposing that they are worthwhile, e-voting systems are high-stake endeavours, insofar as they introduce dependencies to the democratic process. E-voting must be considered a *complementary* method for expressing one's views, with a "low-tech" solution always present, lest some participants be unable to exercise their right to vote. Furthermore, e-voting systems must fulfill robustness and transparency guarantees that are at least as restrictive as those on conventional voting systems.

A number of countries have experimented with e-voting, with limited success so far [1]. In Switzerland, the addition of e-voting to the already existing methods is an ongoing project which is under constant scrutiny, including by the DEDIS lab. At EPFL, e-voting has been considered since at least 2018 as a means to elect faculty members, and the current voting system has been developed by DEDIS [3]. The voting system is decentralized by means of a blockchain, which reinforces trust and transparency of the voting process. Accordingly, the DEDIS lab recently developed a new blockchain components architecture called `Dela` [4]. The architecture offers stronger reliability guarantees and facilitates development thanks to its modularity. The `D-Voting` project is an e-voting system based on `Dela`; it aims to replace the current e-voting system used at EPFL, and in some respects to showcase the applications that can be facilitated by `Dela`.

We have worked to bring the `D-Voting` system closer to production level, by extending its features and completing the work that was started by our predecessors. Security wise, we

decoupled the cryptography protocols from the election procedure, so that two different elections are now more independent, and we also strengthened the ballot shuffling protocol. In terms of usability, we overcame a ballot size limitation which allowed us to implement different types of elections, such as ranked-choice voting, multiple choice and open text answers. This is a major improvement that will facilitate experimentation with other voting systems. We also worked on the test suite to make adding new tests easier.

In the following section, we shall review the technical background behind the `D-Voting` system.

Chapter 2

Background

2.1 Security requirements

As a voting system, `D-Voting` must protect the privacy of its users and prevent tampering of the election. The main security requirements are listed here:

Ballot secrecy No one can determine how a user voted.

Auditability A quorum of nodes must witness the whole process. Anyone can verify the results.

Data integrity Votes are securely stored and cannot be tampered with or deleted.

Availability A user cannot be prevented to cast their vote by just one node.

In order to fulfill the security requirements of `D-Voting`, we make use of some cryptographic primitives.

2.2 E-voting smart contract

The election process is recorded and executed via a smart contract running on the `Dela` blockchain previously developed by DEDIS [4]. The smart contract receives information from the nodes participating in the chain, and actions can be triggered by adding specific transactions to the chain.

2.3 Pedersen DKG

Distributed Key Generation (DKG) allows the splitting of a cryptographic key pair between several actors: the public key is untouched, but the secret key is divided into key *shares*, each of them belonging to a node. When the nodes wish to decrypt a message, they contribute their share to form the full secret key in such a way that the share cannot be recovered by other actors. It is easy to see how this primitive allows us to avoid giving encryption and decryption powers to a central third party, thereby reinforcing ballot secrecy and data integrity.

Importantly, the nodes should be able to produce a cryptographic proof of decryption, that is, a guarantee that the decryption process was performed correctly. This would contribute to the auditability of the system and it would help prevent tampering with the encrypted votes, as we would then expect decryption to fail. More information about DKG can be found in [10].

2.4 Neff shuffle

The Neff shuffle algorithm on ElGamal pairs is used to change the appearance of the ballots to maintain ballot secrecy. It relies on the probabilistic nature of the ElGamal cryptosystem by re-encrypting the ciphertexts without compromising the correctness of the decryption [9]. We use the implementation from the `kyber` library [5] in accordance with our predecessors on this project. They give more detailed reasoning behind this choice in their report [7].

In his paper from 2004, Neff makes the distinction between two types of shuffle, the k-shuffle, a simple shuffle of ElGamal pairs, and the ElGamal sequence shuffle, a shuffle of sequences of ElGamal pairs. The shuffle of sequences uses the same mechanisms as the simple k-shuffle, but applies to k sequences of N_Q ElGamal pairs, where k is the number of ballots and N_Q the number of ElGamal pairs per ballot. It shuffles the sequences without reordering or altering their inner content.

The `kyber` library implements both shuffles, but is limited in the sense that it can only encrypt ciphertexts that are at most 29 bytes long. In order to have bigger ciphertexts, it is necessary to split them into sequences and use the sequence shuffle.

As our predecessors already did, it is possible to generate a proof of a k-shuffle so that a verifier can verify its validity. However, in order to generate a sequence shuffle proof, the Verifier has to provide a randomly generated vector to the Prover. In `kyber`, the Prover gives to the Verifier a function which takes the random vector as parameter and returns the proof. This implementation avoids the added communication between the Prover and the Verifier that would take place if the

Verifier needed to provide the random vector to the Prover.

Chapter 3

Design & Implementation

3.1 Election process

3.1.1 Design

Each election is controlled by an *admin*. All the other participants are the *voters*.

The election process is a fixed sequence of states that are triggered by certain actions, which we list now:

1. `Open` the election;
2. `Cast` a number of (encrypted) ballots;
3. `Close` the election;
4. `Shuffle` the (encrypted) ballots;
5. `Decrypt` the ballots and publish the result.

All actions must be triggered by the admin, except for ballot-casting. The admin may also `Cancel` an election at any point after opening it.

3.1.2 Implementation

In general each of the actions presented in subsection 3.1.1 is implemented as a method on a smart contract, which is running on the `Dela` blockchain.

However, in order to uphold the safety requirements, in some cases other conditions must be fulfilled to continue to the next step: for example, after creation of a new election, a DKG RPC must be started in order to trigger the `Open`, but the RPC cannot be started before the election was `Created`. These dependencies are summarized in Figure 3.1.

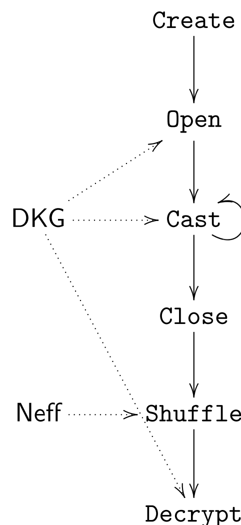


Figure 3.1: The actions that can be run on an election and their dependencies. The actions must be performed in order.

3.2 Pedersen DKG protocol

3.2.1 Design

Previously the Pedersen DKG implementation was taken from the `dela` project, and modified to integrate with the election process. However, an important modification was yet to be made: the DKG protocol is run *just once* per chain, which means that the same credentials are used in any election run on that chain. Concretely, consider a situation where all doorlocks on a street have the same keyhole: it might be difficult to forge a key but once it is done, the damage can be considerable.

Thus, one of our first tasks was to further tailor the DKG implementation to our use case,

including requiring a new DKG actor for each new election.

3.2.2 Implementation

The previous implementation of the DKG protocol could not support multiple elections on the same chain in a safe manner. A pseudo-code view of the former API is given in Listing 3.1.

```
type DKG interface {
    NewActor() (Actor, error)
    GetLastActor() (Actor, error)
    // ...
}

type Actor interface {
    Setup() (PublicKey, error)
    GetPublicKey() (PublicKey, error)
    Encrypt(pt Plaintext) (Ciphertext, error)
    Decrypt(ct Ciphertext, electionID string) (Plaintext, error)
    // ...
}
```

Listing 3.1: An approximation of the previous DKG implementation

`DKG` can be seen as a starter and access point to the `Actor`; it is the `Actor` which runs the DKG primitives. Notice that by and large, the API does not include any reference to the election, whereas we would like each DKG action to be linked to an existing election.

For example, `DKG` only supports accessing the *last* `Actor`, whereas we would like to be able to access all `Actors` that deal with running elections. Hence, we replace `GetLastActor()` with `GetActor(electionID)`. Similarly, `DKG` should be able to create an actor for any running election, so we replace `NewActor()` with `NewActor(electionID)`.

Finally `Decrypt` should only be run on ballots of the election that the `Actor` is linked with, so we replace `Decrypt(ct, electionID)` with `Decrypt(ct)`.

3.3 Persistence of system state

3.3.1 Design

Since each node in the system stores ballots and contributes to the shuffling and decryption process, it is necessary to guarantee that a node can recover the necessary information in case of a crash. Indeed, if too many nodes lose their key share, it would not be possible to decrypt the encrypted ballots, and the election would have to be cancelled.

Thus, we added a facility to save the distributed key and the corresponding private share, as well as the key pair used to communicate with the other nodes via the RPC.

3.3.2 Implementation

Whenever an `Actor` is created or set up (i.e. a key pair is generated), the relevant data should be saved to persistent storage.

The saving to disk is performed using the `kv.DB` primitive from `Dela`, which assures atomic read and write operations.

3.4 Security of the Neff Shuffle algorithm

3.4.1 Design

One of our first task on the project was to address a security issue of the Neff shuffle algorithm. In order to make sure that at least one shuffle submitted on the chain is valid, we need to ensure that at least $\frac{1}{3} + 1$ shuffles have been made by different nodes, since up to $\frac{1}{3}$ of the nodes can be malicious according to the specifications of Dela [4]. To that aim, the previous algorithm specified was that all nodes should attempt to shuffle the ballots and keep trying until either their shuffle is accepted or the required threshold is reached.
While the algorithm is correct, the smart contract allowed a node to submit more than one shuffle, so that a malicious node or quorum of nodes could submit enough shuffles to reach the threshold on their own. This was an issue because the node performing the shuffle is able to revert it, thus if all shuffles are made by malicious nodes we would completely lose voter anonymity.

To overcome this, we keep track of who has already submitted a shuffle and require that

nodes sign their shuffle.

3.4.2 Implementation

In order to identify which node makes a shuffle, we completed the `ShuffleBallotsTransaction` to contain the public key of the node making the shuffle, and a signature of the transaction from the node, as can be seen in Listing 3.2.

```
type ShuffleBallotsTransaction struct {
    ElectionID      string
    Round           int
    ShuffledBallots EncryptedBallots
    Proof           []byte

    // Signature is the signature of the result of
    // HashShuffle() with the private key
    // corresponding to PublicKey
    Signature       []byte

    // PublicKey is the public key of the signer
    PublicKey       []byte
}
```

Listing 3.2: The new `ShuffleBallotTransaction` with signature and public key of the shuffler node

With this additional information, we were able to adapt the smart contract so that it verifies that no other shuffle has been submitted by the signer. It can then verify that the given public key is indeed associated to a node in the roster, reproduce the hash that was signed and verify that it corresponds to the content of the signature.

3.5 Diversification of election formats

3.5.1 Design

3.5.1.1 Election configuration

A major feature we worked on this semester is a more sophisticated format for elections, more generic and with some freedom for the administrator to create any kind of poll. We identified 3 types of question that can be asked in a poll:

Select a select question requires the user to choose one or more options among multiple choices from 1 to N, where N is the total number of choices. Examples might be "Select candidate A or B", or "Select two candidates from A, B, and C".

Rank a rank question requires the user to rank 1 or more choices among multiple choices from 1 to N, where N is the total number of choices. Examples might be "Rank each from the list based on your preference, starting with 1 your favorite, to N".

Text a text question requires the user to enter free text in 1 to N fields, where N is the total number of choices. Examples might be "Enter the name of 1 or 2 of your favorite candidate(s)".

3.5.1.2 Encoding of ballots

Given elections can now have multiple questions, we need a common encoding of the answers to make sure all ballots can be encoded in a unique way, up to the ordering of questions.

In order to maintain complete voter anonymity and nontraceability of ballots throughout the election process, it is important that all encrypted ballots have the same size. To this aim, the election data structure should contain the information `BallotSize`, which is the size all ballots should have before they are encrypted. This size represents the maximum size a ballot can have for this poll. Smaller ballots should then be padded before encryption.

3.5.2 Implementation

3.5.2.1 Election configuration

In order to gather all the elements related to the questions and the layout of a poll, a `Configuration` is now in the data structure of an election. A `Configuration` contains the main title of the election, and a scaffold of subjects. A `Subject` is essentially a container for questions and possibly other nested subjects. It allows the administrator to split the election in different sections and subsections, give each of them a title and choose the order in which they will appear in the poll. For that purpose and also to identify the answers, each `Configuration`, `Subject` and `Question` is associated to a unique identifier. This can be seen in Listing 3.3 and Listing 3.4.

```
type Configuration struct {
    MainTitle string
    Scaffold  []Subject
}
```

Listing 3.3: The `Configuration` of an Election

```
type Subject struct {
    ID ID
    Title string

    // Order defines the order of the different questions
    // which all have a unique identifier.
    // This is purely for display purposes.
    Order []ID

    Subjects []Subject
    Selects  []Select
    Ranks    []Rank
    Texts    []Text
}
```

Listing 3.4: A `Subject` is used as a container for questions

Each of the three new types of questions contain an `ID`, a `Title`, a maximum number of answers `MaxN` and a minimum `MinN`, which apply on the `Choices`, the name or text associated to each possible answer. We decided to restrict the `ID` to be base64-encoded in order to avoid

any undesirable special characters (such as `\n`) that could corrupt a ballot during decryption. In addition to these attributes, an open text question also has the attributes `MaxLength` in order to bound the length of the answers, and `Regex` to interpret the answers as regular expressions. We show the structure for a `Text` question in Listing 3.5.

```
type Text struct {
    ID ID

    Title     string
    MaxN      uint
    MinN      uint
    MaxLength uint
    Regex     string
    Choices   []string
}
```

Listing 3.5: Data structure representing an open text question in a poll

The configuration of an election is submitted at the creation of the election in the `CreateElectionTransaction`. The smart contract then validates the configuration, for example by checking the encoding of the `ID`s and the coherence of the question parameters (such as `MaxN` > `MinN`).

3.5.2.2 Encoding of ballots

We decided that a ballot would be encoded with one answer per line, with each line in the format shown in Listing 3.6.

```
<question type>:<id>:<answers>

<question type> = ‘select’|‘text’|‘rank’
<id> = bytestring up to 3 bytes long, base64-encoded
<answers> = <answer>[‘,’<answer>]*
<answer> = <select_answer>|<text_answer>|<rank_answer>
<select_answer> = ‘0’|‘1’
<rank_answer> = empty if not selected, else an int in [0, MaxN[
<text_answer> = base64-encoded string
```

Listing 3.6: Format of a line in a ballot

To denote the end of the ballot and the start of the padding, we use an empty line (`\n\n`). The padding will be ignored at decryption but should be done in an encoding that does not contain the character '`\n`'.

Using this specific encoding and the `Configuration` of an election we can then easily predict the maximum size a ballot can have when the election is created. This information is then contained in the data structure representing an Election, available to the front end which will add the padding upon encryption.

Of course, we only discover the content of the ballots upon decryption, which raises an important question: how do we interpret a ballot that has not been formatted correctly? During an on-paper election, if a vote is written on something other than the official ballot, then the vote is not counted. In our case, it makes sense not to count this ballot as well. However, there must be a clear equality between the format created by the front end and the one accepted by the back end.

3.6 Increasing the maximum ballot size

3.6.1 Design

In order to support multiple-question elections, we also needed to increase the maximum size of a ballot, which was previously capped to 29 bytes due to a limitation of the `kyber` library. Indeed, `kyber` can only encrypt a ciphertext into ElGamal pairs if its total size is less than 29 bytes [5]. Hence, we decided to represent a ballot as a slice of multiple chunks of 29 bytes. That way, a ballot is no longer bounded in size and we do not have to limit the configuration of polls.

3.6.1.1 Adaptation of the protocols

Such a change in the format of ballots requires multiple changes in the encryption, the shuffle and the decryption processes. The encryption is handled by the front end. The decryption process changed from the decryption of one *ballot* at a time to the decryption of one *chunk of ballot* at a time, which makes the decryption process N times slower with N the number of chunks per ballot. For the shuffle, we needed more profound changes given the nature of the ballots changed. Explicitly, we now needed to shuffle *sequences* of ElGamal pairs instead of simple ElGamal pairs.

We use Neff's shuffle of sequence to shuffle the ballots, as presented in section 2.4. Each of the k ballots is a sequence of N_Q ElGamal pairs. The implementation of the shuffle of sequence in

`kyber` is close to Neff's paper, which implies a very specific layout in memory of the sequences to be shuffled.

3.6.1.2 Proving the shuffle of sequence

Let us denote the prover by P (i.e. the node making the shuffle and generating the proof). Generating a proof for a shuffle of sequence can no longer be done by P alone. Recall that in the original paper, the node can only generate a proof using a random vector generated by the verifier V and in `kyber` P gives instead to V a function to get this proof using a random vector.

Two problems arise from this situation. First of all, having P give a function to generate the proof is not feasible, because the shuffle is submitted in a transaction on the chain and the content of this transaction needs to be marshalled into bytes. If we marshal a function into bytes, we lose the primary purpose of this function which was to not communicate its inner content.
Therefore, we needed to find a way to generate this random vector beforehand. Random generation on a blockchain is a well-known problem because the nodes need to agree on a state, which is orthogonal with the concept of randomness.

A first solution we considered was to trust one node to generate the random vector in the previous transaction. Every transaction that precedes a shuffle (either a `CloseElection` or `ShuffleBallots` transaction) would also contain the random vector to be used in the following shuffle. This is a valid solution, but the problem is that to ensure a shuffle has been made by a valid node, with a valid random vector, we need to know that two transactions in a row have been made by valid different nodes. This is a problem because it would double our current threshold; we would need a total of $\frac{2}{3} + 2$ of the nodes to make a shuffle and it would take twice as long to shuffle the ballots.

To avoid the extra complexity added by the need to trust two nodes, we decided to go with semi-randomness: the prover would use the hash of the shuffled ballots and the ID of the election as a seed for a random generator. That way, the verifiers can recover the same generator and generate the same random vector as the prover. This generator would be different for each shuffle since it depends on the shuffled ballots, and is not purely dependent on the prover either since it is generated from the election ID as well. We decided to adopt this solution since it allows us to keep the original flow of the shuffle at very small cost while still following a strong security argument.

3.6.2 Implementation

Changing the format of ballots implied a lot of refactoring on the types related to ballots. We took the opportunity to also give the ballot types more intuitive and representative names. For example, an encrypted ballot used to be represented by the type `CipherText`, which way simply an ElGamal pair. Now, we renamed this type to `EncryptedBallot`, and it represents an array `[]Ciphertext` since it now contains multiple ElGamal pairs. Here are the main changes:

- `CipherText` = ElGamal Pairs $\xrightarrow{\text{became}}$ `EncryptedBallot` = `[]CipherText`
- `CipherTexts` = `[]Ciphertext` $\xrightarrow{\text{became}}$ `EncryptedBallots` = `[]EncryptedBallot`
- `EncryptedBallots` = Map from users to ballots $\xrightarrow{\text{renamed to}}$ `PublicBulletinBoard`

Intuitively, the ballots are stored in the form of an array of ballots. In this form, it is easier to add ballots one by one during the voting phase, as well as to delete or replace ballots in the public bulletin board. However, as illustrated in Figure 3.2, we need the ElGamal pairs to be in a transposed layout in order to perform the shuffle of sequence. We decided to do this transposition only once we unmarshal the ElGamal pairs in order to use them, so that we avoid any unnecessary computation. This happens both during shuffling and decryption. During shuffling, we get the ElGamal pairs in the desired form and we account for the fact that the pairs are transposed in the decryption process. This trade-off allows us to be as efficient as possible in all use cases, without duplicating the storage of the encrypted ballots.

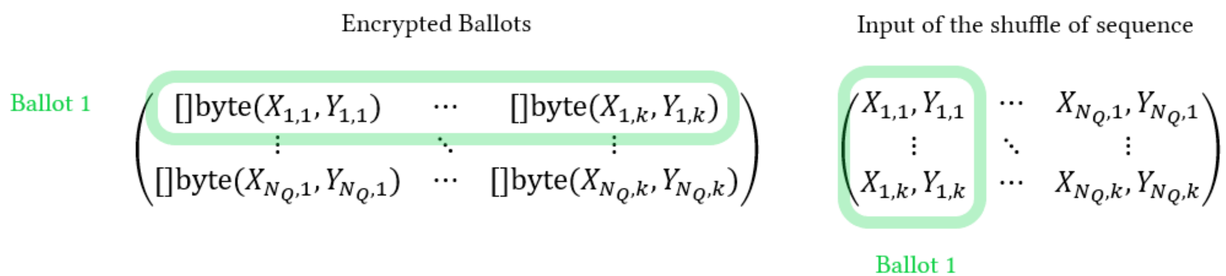


Figure 3.2: Layout of the ElGamal Pairs in memory. $X_{i,j}$, $Y_{i,j}$ is the jth ElGamal pair of the ith ballot.

3.6.2.1 Sanitization of ballots

A security issue that existed in the previous implementation was that no check was done on ballots on casting. If someone submitted an empty ballot, it would crash the entire shuffle protocol and we would not be able to finish the election. In order to fix this problem, the smart contract

now verifies that a ballot has the right number of ElGamal pairs (i.e. that it has been padded) and that we can unmarshal those pairs.

Chapter 4

Evaluation

4.1 Correctness tests

A comprehensive test suite is an essential component for a transparent and auditable system, and of course to facilitate the development process. Our predecessors achieved fairly high coverage with unit tests, though the large refactoring that was done before we started working on `D-Voting` broke a number of them. They also wrote a "test scenario" that can be launched from the CLI; in this scenario, 3 nodes create an election and 3 dummy votes are cast. Many of these tests were adapted as the corresponding API changes were made, and we took the opportunity to create more reusable mocks. The coverage of the current unit tests is listed in Table 4.1.

Module	Coverage
Contract	75.8%
DKG	86.3%
Shuffle	88.6%

Table 4.1: Coverage of unit tests for the different modules

4.2 Performance

The main performance critical operations, as already pointed out by our predecessors, are shuffle and decryption. Based on the solution introduced in section 3.6, we reevaluated the time taken to process ballots in different conditions.

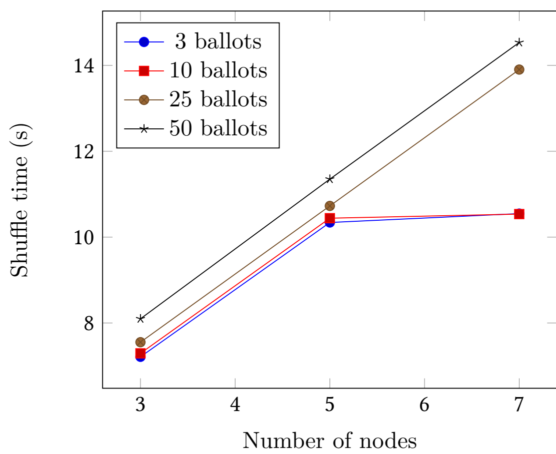
Measuring the shuffling and decryption times, we ran elections with different combinations

of parameters:

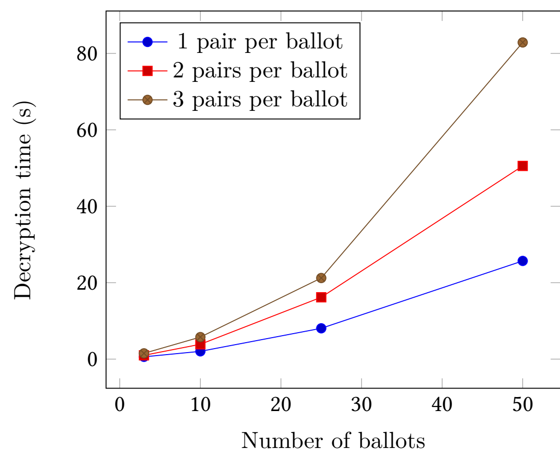
- the number of nodes cooperating (3, 5 or 7);
- the number of ballots cast in the election (3, 10, 25 or 50), and
- the number of ElGamal pairs in each ballot (1, 2 or 3).

In Figure 4.1a, we show the shuffle time depending on the number of nodes. Each curve corresponds to an election with the given number of ballots cast, and the number of ElGamal pairs in each ballot is fixed at 3. Notice how with enough ballots, the shuffle time starts following a linear trend in the number of nodes: this makes sense because the number of shuffles needed is a direct function of the number of nodes. Additionally, the small time difference between the time taken for each number of ballots highlights the fact that the time taken for a single shuffle is a small part of the protocol and scales well in the number of ballots we need to shuffle. The time does not seem to increase between 5 and 7 nodes for 3 and 10 ballots. This might be due to transaction conflicts which are more prevalent when there are many nodes and ballots.

In Figure 4.1b, we show the decryption time depending on the number of ballots. Each curve corresponds to an election with the given number of pairs per ballot, and the number of nodes is fixed at 7 because it has a negligible influence on the decryption time. The decryption time appears to grow exponentially with the number of ballots, which was already observed last semester for reasons explained in detail in their final report [7]. Since we did not change the decryption protocol, it makes sense that we still observe this complexity now. The fact that the decryption time is linear in the number of pairs per ballot is also very intuitive: instead of decryption ballots (which used to be a single ElGamal pair) one by one, we now decrypt ElGamal pairs one by one, so the protocol is similar to decrypting *#Ballots* × *#Pairs per ballot* in the previous implementation, with a small overhead added by the additional operations of splitting the ballots into pairs and merging them into a single ballot after decryption. While this complexity is far from optimal, it can be much improved in the future by changing the implementation of the protocol to decrypt not just a single ElGamal pair at a time, but rather all the ballots together.



(a) Shuffle time depending on number of nodes, for various numbers of ballots, with fixed number of pairs per ballot



(b) Decryption time depending on number of ballots, for various numbers of pairs per ballots, with fixed number of nodes

Figure 4.1: Performance results

Chapter 5

Future Work

5.1 Security issues

5.1.1 Proof of decryption

As mentioned in section 2.3, it is possible to give evidence that the decryption process was performed as intended. However, currently the relevant information is not stored by the smart contract.

A possible implementation to solve this might be the following (suggested by our supervisor Noémien): once the shuffle step is done, each node participating in the election will compute their public share, which only they can do. They will then sign it and send a transaction to `Dela` to have their share saved with the election data. The smart contract will verify the shares are well signed and store them. Once all nodes have had their share verified, the decryption process can start.

5.1.2 Proof of encryption

While we managed to filter the cast ballots so that they are proper ElGamal pairs that can be shuffled, we currently have no way of verifying if a ballot has been encrypted with the public key of the election or not. When a ballot that has been encrypted with the wrong key is submitted, the decryption protocol cannot finish. Possible solutions to this include a proof of encryption to make sure the right public key has been used to encrypt the ballot, or drop a ballot when we cannot successfully decrypt it.

While dropping ballots that we cannot decrypt is a simpler solution, the situation is similar to the one discussed in section 3.6: it is a choice between allowing the chain to drop cast ballots or expose a vulnerability. We believe that being able to drop ballots is a requirement to maintain a fail-safe property and protect an election from denial of services or undefined behaviours. As long as a ballot is submitted with the right format and encrypted with the right key, it should not be dropped. For full auditability, users should have a direct feedback when they submit their vote which would assure them that their true vote has been cast and will be counted. This property was described by Adida and Neff in 2006 as *Ballot casting assurance* [2] and will have to be thoroughly tested in the future if this choice of implementation is made: having full trust in the front-end is a vulnerability.

5.1.3 Ballot re-encryption can de-anonymize users

A critical threat to voter anonymity is the possibility of ballot re-encryption. The cryptographic function used to encrypt ballots is deterministic, so two encryption of the same ballots with the same key will be equal. Since the key is public, it is possible for an attacker to re-encrypt the ballots with random user IDs in a brute force manner, until they find a match with the submitted ballots. This is only a problem for elections with few participants, or where ballots are mostly unique (e.g. with an open text question).

A possible mitigation is to use a random seed when encrypting the ballot, for example in the form of padding, that would not be shared during decryption.

5.2 Authentication of users

Over the course of the semester our colleague Ambroise Borbely, who worked on the front end, incorporated the authentication of users using the Tequila service from EPFL. This change is not yet supported by the API, and it needs to be adapted so that we ensure the requests have been signed by the authenticator.

Chapter 6

Conclusion

`D-Voting` is an e-voting system based on the `Dela` blockchain. As such, it allows for avoiding a central third party, and it guarantees that the election process can be transparently audited, without compromising the security of ballots.

In this work, we refined and improved the system created by our predecessors. The main changes we made were:

- Adapting the DKG implementation to support several elections on the same chain in a safe way;
- Adding facilities to save DKG parameters to persistent storage to minimize consequences of crashing;
- Correcting a security flaw in the Neff shuffle implementation by requiring that nodes sign their shuffle;
- Adding support for different types of ballots, in order to facilitate the implementation of different types of elections, and
- Overcoming a limitation on the maximum ballot size.

However, our changes caused some performance hits, particularly in the decryption process, with a 50-ballot election with large ballots taking as long as 80 seconds to decrypt. We see correcting these performance issues as a high priority for our successors. In particular, the decryption process can be made more efficient by decrypting more ballots at once.

On the security side, the proofs of encryption and decryption still need to be implemented, and there are some edge cases that still need to be covered in the ballot validation process.

On the usability side, though substantial advances were made in order to increase the flexibility of the polling facilities, the junction with the front end still needs to be made.

With these refinements in place, we believe the system will be usable in a real election.

Bibliography

[1] ACE Electoral Knowledge Network. *Countries with E-Voting Projects.* URL: https://aceproject.org/ace-en/focus/e-voting/countries (visited on 12/26/2021).

[2] Ben Adida and C. Andrew Neff. “Ballot Casting Assurance”. In: *Proceedings of the USENIX/Accurate Electronic Voting Technology Workshop 2006 on Electronic Voting Technology Workshop*. EVT’06. Vancouver, B.C., Canada: USENIX Association, 2006, p. 7.

[3] Sarah Aubort. “EPFL uses blockchain technology to secure e-voting systems”. In: (June 2018). URL: https://actu.epfl.ch/news/epfl-uses-blockchain-technology-to-secure-e-voting/.

[4] DEDIS. *Dela: Dedis Ledger Architecture.* URL: https://dedis.github.io/dela/.

[5] DEDIS. *Kyber: Advanced crypto library for the Go language.* URL: https://github.com/dedis/kyber/.

[6] Andrew C. Eggers, Haritz Garro, and Justin Grimmer. “No Evidence for Systematic Voter Fraud: A Guide to Statistical Claims about the 2020 Election”. In: *Proceedings of the National Academy of Sciences* 118.45 (Nov. 9, 2021), e2103619118. ISSN: 0027-8424, 1091-6490. DOI: 10.1073/pnas.2103619118. URL: http://www.pnas.org/lookup/doi/10.1073/pnas.2103619118 (visited on 12/24/2021).

[7] Anas Ibrahim and Vincent Parodi. “E-Voting on DELA”. In: (June 2021). URL: https://www.epfl.ch/labs/dedis/wp-content/uploads/2021/07/report-2021-1-Vincent-Anas_EvotingDela.pdf (visited on 01/04/2022).

[8] Tonda McCharles. “Robocalls: Widespread but ‘Thinly Scattered’ Vote Suppression Didn’t Affect Election, Judge Rules”. In: *Toronto Star* (May 23, 2013). URL: https://www.thestar.com/news/canada/2013/05/23/robocalls_widespread_but_thinly_scattered_vote_suppression_didnt_affect_election_judge_rules.html (visited on 12/24/2021).

[9] C. Andrew Neff. “Verifiable mixing (shuffling) of ElGamal pairs”. In: (Jan. 2004). URL: http://courses.csail.mit.edu/6.897/spring04/Neff-2004-04-21-ElGamalShuffles.pdf.

[10] Torben Pryds Pedersen. “A Threshold Cryptosystem without a Trusted Party”. In: *Advances in Cryptology — EUROCRYPT ’91*. Ed. by Donald W. Davies. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1991, pp. 522–526. ISBN: 978-3-540-46416-7. DOI: 10.1007/3-540-46416-6_47.

Appendix A

Installation and testing

The `D-Voting` project is hosted on GitHub at `https://github.com/dedis/d-voting`. The repository contains instructions on how to set up the project and run an election.