École Polytechnique Fédérale de Lausanne

E-voting on DELA

Anas Ibrahim, Vincent Parodi

CYBERSECURITY MASTER SEMESTER PROJECT

Supervisors : Noémien Kocher, Gaurav Narula
Responsible : Prof. Bryan Ford

EPFL IC IINFCOM DEDIS
BC 210 (Bâtiment BC)
Station 14
CH-1015 Lausanne
June 10, 2021

***Abstract*—E-Voting is a very interesting, actual, fancy, and useful use case to demonstrate the utility of a distributed ledger technology. The aim of this project is to implement a new E-Voting system based in Dela [1], the latest blockchain-based distributed ledger from the DEDIS lab. We implement the back-end part of a complete e-voting system, from the creation of an election to the publishing of the results. We provide a rest API such that any front end can make use of the system.**

I. Introduction

E-Voting is a nice showcase to demonstrate the utility of a system based on a distributed ledger. Its utility is well understood by the general public and it addresses a very actual concern.

The DEDIS lab has already implemented an E-Voting solution that has been successfully used in the past for EPFL elections [2]. This system is using the previous DEDIS ledger, cothority/onet [3], which is now being replaced by a more recent system: Dela [1]. The cothority (collective authority) project provides a framework for development, analysis, and deployment of decentralized, distributed (cryptographic) protocols. A given set of servers running these protocols is referred to as a collective authority or cothority.

In 2008, Ben Adida published the white paper to his web-based voting scheme named Helios [4]. The idea was to use a Sako-Kilian mixnet, essentially a single shuffle, so that a voter can verify the encryption of his ballot, audit the shuffle to check if his ballot has been included and finally verify the integrity of the decrypted ballot once the election is over. The whole Helios protocol works as follows:

1) Alice encrypts and casts as many ballots as she wishes. Ensuring that each ballot has been encrypted correctly.
2) Alice's encrypted ballot is published on the election's public bulletin board.
3) A shuffle is performed when the election closes, providing a verifiable proof of correctness according to Sako-Kilian.
4) Any observer is now able to verify the shuffle, specifically Alice can check if her ballot has been included.
5) When no complaints have been made by any auditor the shuffled ballots are decrypted with a decryption proof for each one of them.
6) The tally is performed on the decrypted ballots.

The Helios protocol does not secure a voter's privacy but only his actual vote, thus everyone who has cast a ballot is part of the public bulletin board. As a consequence the Helios software is not meant to be used in elections where voter coercion is an issue [5]. This includes political elections where anonymity is crucial to avoid any potential threats against people who have cast a ballot. On the other hand it is perfectly suited for elections in smaller environments like universities or local clubs. The current implementation builds on the ideas used in the Helios project. The goals are to provide a completely verifiable election architecture where the protocol can be audited by any observer at any stage of the process. This e-voting scheme does not hide the identity of the voters either but insures that voters and ballots are unlikable. Some of the components of the Helios project, especially the underlying protocols, have been replaced or adapted.

The Sako-Kilian mixnet is replaced by Andrew Neff's verifiable secret shuffle [6]. This yields a significant performance improvement as demonstrated in [7]. Another change is the actual storage architecture. Helios relies on a conventional relational PostgreSQL [8] database model that is hosted on a single server that is also meant to perform the actual shuffle. In the current implementation all the election related data is put on the blockchain instead, using cothority [3]. This mitigates the single point of failure since the data is now administered by a collective of servers. Also, the servers can run distributed protocols that enhance the reliability and aptitude of the system. For example, in Helios ballots are encrypted using a global key pair whereas the current implementation uses a distributed key generation protocol to create a collective public key. Each node only get one share of the collective secret key (which of course is not enough to retrieve the secret key) which means no single node can decrypt the ballots. Furthermore, the election data is hosted on a blockchain so the integrity of the data is provided if at least 2/3 of the nodes are honest, which is a lot safer than having a single point of failure as in Helios.

The goal of our project is to implement a proof-of-concept of a similar system using Dela [1] instead of cothority. We still aim to provide a completely verifiable election architecture where the protocol can be audited by any observer at any stage. This e-voting scheme will not hide the identity of the votes either, it will only protect their actual ballots.

II. Goals of the project

A. Motivation and contribution

Motivation : E-voting already has some use cases, for instance EPFL elections have been conducted using the current implementation. [2] Dela is the new distributed ledger infrastructure of the DEDIS lab, it serves the same purpose as cothority but it is more modular, more maintainable, testable and stable. Our project should offer the same functionalities as the current implementation, but it benefits of the advantages of Dela over cothority. In addition, it is the first practical work based on Dela.

Contribution : The general goal is to have a first proof-of-concept of a simple E-Voting system that runs on Dela, as explained before we are looking for the same functionalities as the current implementation. In practice, we will have to modify the DKG service of Dela, implement a shuffling service, an e-voting smart contract, and an http server that allows a front-end to use the system.

B. Requirements

The system should be usable by two actors:

- Admin: responsible for managing an election.
- Voter: end-user that is able to cast a vote and get the result of an election.

We want the following to hold :

- Votes are anonymous.

- A quorum of nodes must witness the whole process.
- Votes are securely stored and cannot be tempered/deleted.
- Anyone can verify the election's result.
- A single node cannot prevent a user to cast his/her vote.

C. Scenarios

A scenario describes the high-level use of our system, from the actor's point of view. It implies that all requirements are followed. The initial goal of our system is to handle the following scenario that describes the minimal proof-of-concept :

Happy scenario :

1) (admin) set up a new election
2) (voter) cast a vote
3) (admin) closes election (shuffle ...)
4) (voter) read the results

However, we went a bit further in our implementation. Indeed, our system is able to handle multiple elections, multiples users, multiple votes per users (only the last cast vote is saved) and users are able to monitor the election. Also, we considered security threats caused by malicious nodes and we tried to fix some of these issues (more about this in the security analysis in section VII).

D. Threat Model

We consider active adversaries, which include the nodes participating in the distributed protocols as well as the users of the system.

Dela offers the primitives to run distributed protocols and smart contracts, handled by a set of Byzantine fault-tolerant nodes. It assumes less than 1/3 of the nodes are malicious, thus we assume as much, since our implementation is based on Dela.

III. Election process

Before going deeper into the architecture and the implementation details, we give an overview of the election process. First, the administrator runs the Dela nodes, performs the setup of the blockchain, initiates the distributed key generation protocol (DKG) plus the shuffling protocol and starts the e-voting http server. Then, he registers the election's information on a smart contract. Before storing this information in the blockchain, every node retrieves the collective public key from the DKG. This public key is stored along with the election's information and can be retrieved by any voters. To keep their votes anonymous, voters would precisely use this key to encrypt their ballots locally before casting it. Encrypted ballots are then stored in the blockchain, along with the corresponding election and can not be tampered with nor deleted thanks to Dela properties. Once the administrator closes the election, nodes shuffle the ballots before proceeding with the decryption (again, this prevents adversaries from linking a ballot to a voter). And finally, the result of the election is published and anyone can access it. During the whole process, the administrator and the voters interact with our system through our e-voting http server.

IV. Architecture

A. DELA

Dela stands for DEDIS Ledger Architecture. It provides a modular framework that describes a minimal and extended set of abstractions for a distributed ledger architecture. It also provides several modules implementations that can be combined to run a distributed ledger.

The system is built around three **abstractions** that offer an API to clients so that one can propagate transactions, create and validate that it will be accepted, and finally wait for a confirmation that it is included, or rejected, by the public ledger.

A transaction is what triggers a change in the ledger set of values, where it is left to the ledger implementation to define how values are stored and identified. The transaction is created with a targeted execution environment alongside the input parameters.

1) The first abstraction which allows one to propagate transactions is defined as a pool that can work independently from the other abstractions. In fact, you could run a distributed system that only shares transactions.
2) The second is a service that accepts a transaction and can return if it will be included in the next block, or if it will be rejected. Furthermore, it defines the validity of the transaction so that it cannot be reused in a replay attack for instance.
3) Finally, the third abstraction will collect the transactions from the pool and create a chain of blocks, block after block according to a consensus algorithm. One can register to the service to receive notifications when a new block is created and learn about the transactions included in the block, and which of them have been accepted.

The corresponding **modules implementations** :

1) *Transaction Pool*. Each participant of a distributed ledger needs to collect a bunch of transactions when it is trying to create a block. The pool is there to provide that service, so that clients can send their transactions to one of the pool of the distributed system while the ordering services (see point 3) wait for enough transactions to be discovered to start a new block. The purpose is to offer a single entry point for the client and then the system will take care of spreading the transactions so that any reachable node participating in the distributed ledger will at some point learn about it.
2) *Validation service*. The validation service is there to protect the system against malicious behaviour. We already mentioned a replay attack which allows an attacker to listen for transactions and reuse them to double spent, or similar operation. Bitcoin is protected because of the UTXOs, while Dela is using a nonce that is monotonically increasing for each address/identity. Another potential issue is the input parameters provided by the transaction which could allow a malicious client to execute a smart contract in a incorrect way. The

validation service makes sure the result of a transaction execution only updated what is allowed. Finally, because it is the validation that defines what the transaction looks like, it also provides a manager that will help to create and sign transactions. For instance, in the case of Dela, a client needs to use the correct nonce for the transaction to be accepted.

3) *Ordering service*. A distributed ledger backed with a blockchain evolves block after block. Each block contains a list of transactions that will be execute sequentially or in parallel depending on the implementation. Each execution will produce zero, one or several changes in the ledger values. The ordering service is responsible for creating the blocks and, in another extent, the chain. The abstraction does not provide any property on how the consensus is decided, but it defines the ways to read or get the values of the ledger.

B. Election

In practice we store the election information in the structure 1, and these election structures are stored in the blockchain and managed by the e-voting smart contract. The Ballot structure is just a string, which means the front end can submit any type of ballot as long as it can parse it back correctly. The back-end would not perform any check on the ballot. Indeed, since our system cannot access the content of the ballot at an early stage (the content is encrypted until the decryption phase, at the end of an election), it cannot check whether the cast ballot follows the format of the corresponding election. Therefore, it is up to the front-end to provide a meaningful format to allow other voters to know which questions are being asked and how they can answer to these questions.

We detail the election structure in listing 1.

- *Title* : Title of the election.
- *ElectionID* : ID of the election, used by users to refer to a particular election.
- *AdminID* : ID of the admin of the election.
- *Status* : Status of the election : *Open* when the election is up and accepts new ballots, *Closed* when the election no longer accepts new ballots, *ShuffledBallots* when the ballots have been shuffled, *ResultAvailable* when the election is over (the shuffled ballot have been decrypted) and the results are available, *Canceled* when the election is canceled.
- *Pubkey* : Collective public key generated via DKG and used to encrypt votes.
- *EncryptedBallots* : Encrypted vote of each user.
- *ShuffledBallots* : Shuffled ballots for each round of the shuffling protocol.
- *Proofs* : Corresponding proof of the shuffle of each round.
- *DecryptedBallots* : Content of the ballots in plaintext.
- *ShuffleThreshold* : Threshold of nodes that needs to provide a correct shuffling (see section V-B).
- *Members* : The address and the public key of each node participating in the shuffling protocol.

```
type ID string

type status uint16

const (
        Open            = 1
        Closed          = 2
        ShuffledBallots = 3
        ResultAvailable = 5
        Canceled        = 6
)

type Election struct {
        Title            string
        ElectionID       ID
        AdminId          string
        Status           status
        Pubkey           []byte
        EncryptedBallots map[string][]byte
        ShuffledBallots  map[int][][]byte
        Proofs           map[int][]byte
        DecryptedBallots []Ballot
        ShuffleThreshold int
        Members          []CollectiveAuthorityMember
        Format           string
}

type Ballot struct {
        Vote string
}
```

Listing 1: Election structure

- *Format* : The front end is free to put any information in this field. It allows to encode and decode a ballot as mentioned previously

C. Smart contract

There is no notion of “instance of a smart contract” in Dela, therefore we have to consider a smart contract as a single entity that performs its read and write operations on the storage with a predefined set of keys.

The simplest and naive solution for our system is to have a single smart contract that stores everything : the elections, the ballots for each election, and the results. This is like having a giant XML/JSON file that contains everything, with the smart contract being the interface that handles that.

Transactions can perform the following actions, handled by our e-voting smart contract :

- *createElection* : Creates an election and stores it in the blockchain.
- *castVote* : Stores the encrypted ballot of the user who casts the vote in the given election (specified by its id), in the blockchain. If a user casts more than one vote then vote $n+1$ overwrites vote n to avoid user voting more than once. The election needs to be open.
- *closeElection* : Closes the given election, in practice only updates the status of the election in the blockchain to closed. The election needs to be open.
- *shuffleBallots* : For a given election, verifies a shuffle of the ballots and stores the shuffle along with its proof on the blockchain if the shuffle is correct. Used in the

Shuffle protocol, see section V-B. The election needs to be closed.

- *decryptBallots* : Updates the election in the blockchain with the decrypted ballots. The ballots must have been shuffled.
- *cancelElection* : Cancels an election, effectively just sets the election status to canceled, which forbids any action on this election.

Only the administrator of the election can trigger *closeElection*, *shuffleBallots*, *decryptBallots* and *cancelElection* on this particular election. The arguments of the transactions are listed in listing 2.

```
type CreateElectionTransaction struct {
        ElectionID       string
        Title            string
        AdminId          string
        ShuffleThreshold int
        Members          []CollectiveAuthorityMember
        Format           string
}

type CastVoteTransaction struct {
        ElectionID string
        UserId     string
        Ballot     []byte
}

type CloseElectionTransaction struct {
        ElectionID string
        UserId     string
}

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

type DecryptBallotsTransaction struct {
        ElectionID       string
        UserId           string
        DecryptedBallots []Ballot
}

type CancelElectionTransaction struct {
        ElectionID string
        UserId     string
}
```

Listing 2: Transactions arguments

V. Protocols

The project is driven by several protocols. The distributed key generation (DKG) protocol has to be executed before the creation of an election to generate a collective public key which is used to encrypt the ballots, insuring that no single node can decrypt them. The shuffling protocol is executed once all votes have been submitted, so that the ballots become unlikable to the voters. And finally, the decryption protocol is executed once the votes have been shuffled, to retrieve the content of the ballots which is then shared to users.

A. Distributed key generation

1) Background: We use the implementation in kyber [9] of the protocol from Pedersen [10], where to mitigate the possibility that one server hold all power over public/private key pair a node of the roster only holds a share of the secret without the ability to reconstruct the global secret on its own. Each participating server S_i of a set of servers S starts off by randomly selecting a polynomial of degree t, $fi(z) = a_{i,0} + a_{i.1}z + ... + a_{i,t}z^t$ with coefficients in $\mathbb{Z}_n$. Here, $z_i = a_{i,0} = f_i(0)$ is each party's secret. The protocol works even if some nodes are malicious. It is possible to specify a threshold for the number of nodes that have to be honest in the implementation in kyber. The shared secret is then reconstructed with $x = \sum_i z_i$. From this, the public key calculation follows with $y = gx$. The raw implementation of the above protocol is part of the DEDIS code base and is thus easily included in our project. Before creating an election, an admin has to run the DKG protocol, the collective public key generated will then be used as the public key of the new election. Naturally, the public key produced by the protocol can then be used by a front-end application to encrypt ballots before casting them.

2) Implementation: This protocol is already implemented in Dela.

B. Shuffle

1) Background: We use Neff shuffle on ElGamal ciphertext pairs. As a reminder, the ElGamal encryption algorithm can either be performed on a multiplicative prime order group or an elliptic curve. For the project we chose to go with the Twisted Edwards Curve over a prime order field [8]. For the remainder of this text all operations are meant to be performed over this curve with additive notation :

$E(\mathbb{F}_p) = \{\mathcal{O}\} \cup \{(x, y) \in \mathbb{Z}_p; -x^2 + y^2 = 1 - \frac{121665}{121666}x^2y^2\}$
$p = 2255 - 19$
$y_g = 4/5$
$n = 2^{252} + 27742317777372353535851937790883648493$

Since the equation is quadratic in x, the x-coordinate of the base point g is set to be positive.

The ElGamal encryption algorithm proceeds as follows:

1) Bob picks a random $x \in \mathbb{Z}_n$ and computes $y = gx$. Let x be Bob's private key and y is his public key.
2) Alice encrypts a message $m \in \langle g \rangle$ by selecting a random value $r \in \mathbb{Z}_n$ then computing $u = gr$ and $v = m + yr$.
3) (u, v) is the ciphertext that is transmitted to Bob.
4) To decrypt the message, Bob recomputes the secret $s = ux = grx = yr$ and extracts the message with $m = v - s = m + yr - yr$.

We can use the fact that the ElGamal crypto system is probabilistic and re-encrypt the ciphertexts multiple times without compromising the correctness of the decryption.
For a single re-encryption, we get :

1) Given the ciphertext (u, v) choose a random value $r' \in \mathbb{Z}_n$ and calculate $(u', v') = (gr' + u, yr' + v)$.

2) The decryption algorithm remains unchanged. $s = u'x = (gr' + u)x = gr'x + ux = yr' + yr$ followed by $v' - s = yr' + m + yr - yr' - yr = m$.

The re-encryption property of the ElGamal cipher is a cornerstone in both shuffling algorithms. Proving the correctness of a shuffle means that given a tuple (A, B, g, y) where $A = \begin{pmatrix} u_i \\ v_i \end{pmatrix}$ is a list of ElGamal ciphertext pairs and given a permutation π and a list of random values $r_i \in \mathbb{Z}_n$, $B = \begin{pmatrix} u_{\pi(i)} + gr_{\pi(i)} \\ v_{\pi(i)} + yr_{\pi(i)} \end{pmatrix}$ is the re-encryption and permutation of A. g and y are, as mentioned above, the base point of the curve and the public key. Now, a prover P needs to show to a verifier V that B has been generated from A without revealing neither the permutation π nor the random values r_i or the secret key x. In terms of an election this means that a voter can verify whether his ballot has been correctly included in the shuffle and has not been disregarded by the system.

It is possible to use Neff shuffle [6] to perform such a shuffle. This is what is implemented in the kyber library [9], and that we use in our implementation : Neff shuffle gives a proof of the shuffle that is a seven-move, public coin proof of knowledge which is computational zero knowledge. Indeed, distinguishing between real and simulated proofs is as hard as the decision Diffie-Hellman problem. The proof is of course hard to forge, more details can be found in the paper [6] and in kyber's implementation [9].

2) Implementation: When an admin closes an election, the shuffle protocol has to be executed to insure that ballots and voters are unlinkable. More precisely, the protocol unfolds as follows: Let N be the set of nodes that participate in the protocol.

1) $\forall n \in N, n$ retrieves the encrypted ballots if it is the first round, or the shuffled ballots from round $r - 1$ if it is round r.
2) $\forall n \in N, n$ performs a Neff shuffle on the retrieved ballots.
3) $\forall n \in N, n$ try to store the resulting shuffled ballots and the proof of the shuffle in the blockchain.
4) only the first valid shuffle is stored in the blockchain. To validate a shuffle, every node would perform the verification of the shuffle using its corresponding proof. Let us say it was the shuffle from node n_s, then n_s exits the protocol, and $N \leftarrow N \setminus \{n_s\}$
5) $r \leftarrow r + 1$
6) if $r = threshold$ then done otherwise go to 1)

Of all the nodes that shuffle the ballots at some point during the protocol, we need at least one to be honest otherwise they could all collude which would reveal the permutations and ultimately the mapping between users and votes. Therefore we can set the threshold to 1/3 of the total number of nodes, since we assume in the threat model that less than 1/3 nodes are malicious. Also only a shuffle with a valid proof of correctness can be accepted at step 4), thus it is not possible to temper the ballots during the shuffling protocol. After the completion of the protocol the shuffle with the proof of correctness can be retrieved by any observer.

C. Decryption

The decryption protocol is always executed at the end of the election to get the votes in plaintext so that we are able to give the result of the election.

1) Background: The decryption protocol unfolds as follows :

1) The server that is first contacted by the election creator prompts the other nodes that participate in the protocol to decrypt the shuffled ballots with their own secrets and to send the result back to the this node.
2) A single node cannot fully decrypt the encrypted ballots, the initiator node runs its own decryption pass with its secret then reconstructs the ballot plaintexts using all the partially decrypted ballots from the other participating nodes.
3) The result is published in the blockchain.

We have to note that at no point during the decryption protocol is it necessary that one server accumulates the private shares of the private key, which would allow this server to reconstruct the private key and decrypt the ballots at will. In other words, it remains true that no server is able to decrypt the ballots by itself. Right now the initiator node asks and waits for the partial decryption of all the other nodes participating in the protocol. This is a problem because a malicious node could chose to not send his partial decryption and this would block the system. However, it remains true that no single node can decrypt the ballots. Also, this is easily fixable as we just need to make sure the initiator node proceeds once it has collected a threshold t of partial decryption, which would mean that the system cannot be blocked if at least t nodes are honest. We could set $t = (2/3) * (number of nodes)$ since we assume that at least 2/3 of the nodes are honest.

The Helios project also provides a Chaum-Pedersen non-interactive proof of decryption. At the current stage of the project this feature has not been implemented.

2) Implementation: The protocol was for the most part already implemented in Dela. However, the current implementation does not perform any kind of verification on the ciphertext to decrypt. Indeed, we must only decrypt shuffled ballots. We would not want a malicious node to initiate the decryption of a non shuffled ballot. Therefore, we adapted the current implementation : every node participating in the decryption first check that the ciphertext is indeed a shuffled ballots of the corresponding election before sending its public share.

VI. Implementation

A. Golang

The whole Dela project as well as the advanced cryptography library kyber are almost entirely written in Go [11], therefore we naturally chose to implement our e-voting project in Go as well.

B. Project structure

We briefly describe the project structure. Note that it might change in the future.

- memcoin : A command line interface for the administrator to setup the system.
- evoting : Contains the implementation of the evoting smart contract, the http server and every structures we use in our system.
- dkg : Contains the implementation of the DKG protocol and the decryption protocol.
- shuffle : Contains the implementation of the shuffling protocol.

C. Front end

We did not implement the front-end part of the system ourselves. The only requirement for a seamless interaction of the front-end with the back-end is the availability of a Twisted Edwards Curve implementation necessary to perform the encryption of ballots and the actual verification of the shuffles. Users could access the front-end application through a browser that is for example hosted by a node.js server and which authenticates the user toward a database [12]. In our case this could be the EPFL's internal authentication service [13]. As soon as the user is authenticated the front-end relays the communication towards our back-end. During this project we worked in collaboration with another MASTER student who was in charge of implementing the front-end. Once the front-end managed to encrypt the ballots, it was able to correctly interact with our back-end using the API, see section VI-D.

D. HTTP server

We implemented an HTTP server that allows a front-end to interact with our system. For each smart contract action, there exists an endpoint that pushes a transaction triggering the corresponding action. Also, we added a few GET methods :

- *getElectionInfoEndpoint* : Retreives the election's information corresponding to the provided election id.
- *getAllElectionsInfoEndpoint* : Retreives the information of all the elections stored in the blockchain.
- *getAllElectionsIdsEndpoint* : Retreives the id of all the elections stored in the blockchain.
- *getElectionResultEndpoint* : Retreives the result of a specific election.

Also, you can find the API messages in the appendix. Note that it might change in the future.

E. Tests

We implemented a few unit tests for the shuffling protocol, decryption protocol and the evoting smart contract. The coverage is shown in table I.

As we can see there is still room for improvement. Also, we miss integration tests for the http server, though we tested scenarios manually (with the cli) and with the help of Sarah who was in charge of implementing the front-end.

package	coverage [%]
smart contract	86.59
dkg	89.72
shuffle	92.87

TABLE I: Test coverage for our packages

As a comparison, the coverage of the current implementation is shown in table II. Note that the evoting smart contract of our implementation does not match exactly the evoting package of the current implementation. The point is that it seems to be easier to test a system which uses Dela, compared to a system using cothority.

package	coverage [%]
evoting	37
dkg	65.38
shuffle	65.54

TABLE II: Test coverage of the current implementation

F. Benchmark

The costs were computed on a Windows 10 64-bit architecture laptop with a 4-cores Intel(R) Core(TM) i7-1065G7@ 1.30GHz processor. We evaluated the performance of our system by benchmarking the shuffling part and the decryption part. We designed a scenario in which we vary the number of ballots and the number of nodes to show the effects of these to parameters on the computation time of the shuffling and the decryption. This scenario consists of :

1) Setting up a DKG protocol and a Shuffle protocol with a certain number of nodes.
2) Creating an election.
3) Casting a certain number of votes.
4) Requesting the closing of the election.
5) Requesting the shuffling of the ballots by calling the corresponding endpoint.
6) Requesting the decryption of the shuffled ballots by calling the corresponding endpoint.

We perform an end to end evaluation, we compute the time from the call to the endpoints to receiving the response from the server. This benchmark essentially allows us to know whether our system scales well, and if it is not the case, we can figure out what is going wrong and try to come up with a solution.

Decryption performance evaluation : The results are shown in figure 1. The number of nodes seems to have a small influence on the computation time. Indeed, with 5 nodes, the performance is only decreased by 10% compared to the 4 nodes case. However, the computation time seems to grow exponentially with the number of ballots, and we can provide an explanation for this. Let's recap how the decryption works : to make sure nodes does not decrypt a non-shuffled ballots, every node checks that the ciphertext is indeed a shuffled ballot of a certain election, and if it is not the case, it simply does not take part in the decryption protocol. Also, the current implementation of the DKG protocol only allows to decrypt

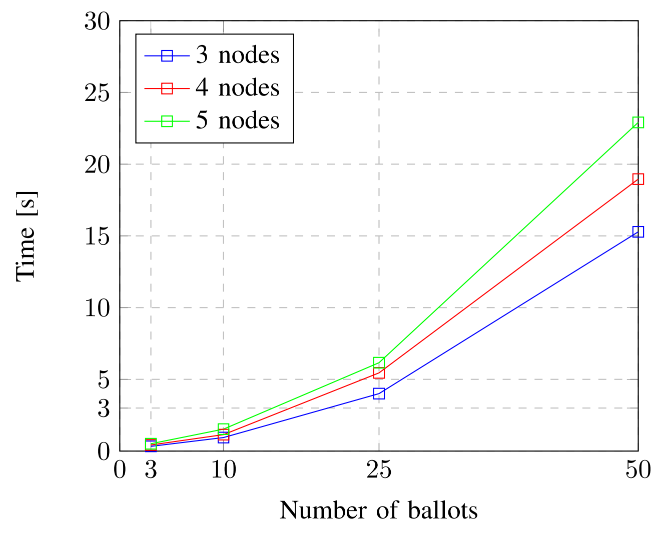

Fig. 1: Decryption protocol benchmark

one ciphertext at a time. Thus, the more ballots we have in an election, the more checks the nodes must perform. Also, the list of the shuffled ballots is also longer (making the checks slower). Not to mention the fact that the nodes would retreive the election object from the blockchain for each ciphertext decryption. To fix this issue, we can improve the DKG implementation by allowing rosters of nodes to decrypt multiple ciphertexts at once. This way, nodes would perform only one read per election, and more importantly, nodes would perform only one DKG decryption per election (and not per ballot).

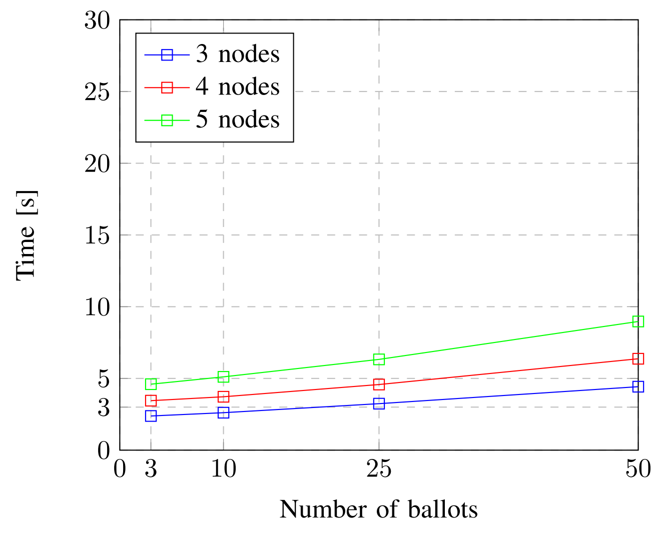

Fig. 2: Shuffling protocol benchmark

Shuffling performance evaluation : The results are shown in figure 2. Unlike the decryption phase, the shuffling protocol seems to scale well. Both the number of nodes and the number of ballots influences the computation time but the shape of the line does not grow exponentially with any of these two parameters.

VII. Security analysis

We consider active adversaries, which include the nodes participating in the protocols as well as the users. We rely on Dela so we can handle at most 1/3 of the nodes to be malicious. Our shuffling protocol can handle at most $n-1$ malicious nodes if n is the number of participating nodes. (if there is one honest node then we have one correct shuffle which is technically enough, given that the node that performed it is honest). The DKG protocol needs t honest nodes. Since we need at least 2/3 of the nodes to be honest because of Dela, we can set $t = (2/3) * n$. Currently the decryption protocol needs all the nodes to be honest, but as explained in section V-C, we can easily modify it so that it needs only t honest nodes. Since we need at least 2/3 of the nodes to be honest because of Dela, we can set $t = (2/3) * n$. Since this problem can only cause the system to block, we assume in the remaining of this section that we can make it work with only $t = (2/3) * n$ honest nodes. Therefore the entire system can handle at most 1/3 of the nodes to be malicious, otherwise attacks targeted directly at Dela are possible.

The communication between users and the http server is not secure, thus it is vulnerable to man-in-the-middle attacks. An adversary could sniff the traffic or even alter the content of the packets. A solution would be to use HTTPS instead of HTTP.

What can happen if the node hosting http server is malicious ?

1) If a user communicates with a malicious node then the node could refuse his vote, and even make it so the user does not realise (if the user looks at the blockchain via this node).
2) Users fetch the collective public key using *getElection-Info* endpoint, the malicious server can reply with his own public key. The user will then use this public key to encrypt his vote, which he will send to the server. The server can then decrypt the vote, associate its value with the user, re encrypt the vote using the real public key of the election, and upload this to the blockchain.
3) It could simply block the whole system, make it so that the election never reaches an end. Basically it can block the system at any time
4) When a user wants the result of an election, it queries the server, thus if it is malicious it can send fake ballots so that the result is changed.

Would some sort of distributed http server mitigate this kind of issues ? In our case if we ran the server on all nodes it would not help in the sense that if the node which the user communicates with is malicious, it would be able to trick the user into thinking that everything is fine and the user would not check with other nodes. An idea would be for each user to communicate with every node and look for inconsistencies. Then, it keeps the response sent by at least 2/3 of the nodes (this response must exist since we consider at least 2/3 of non malicious nodes). In this setting, in 1) if a client sends his vote to a malicious node, even if the malicious nodes tells the client that his vote has been included, the client will be able to verify that it is not the case, and maybe send his vote again to one of the 2/3 nodes that are consistent. In 2) the client has to query the *getElectionInfo* endpoint of all nodes and use the public key that is in at least 2/3 of the replies. In 3), if several nodes run servers then one node cannot block the whole system. In 4) the user could query the result to all nodes and accept a result that is given by at least 2/3 of the nodes.

More generally, we thought of some attacks, and whether they would work against our system :

- It is not possible to execute different steps of the election out of order because of the status field of election.
- Attack on election public key : A node might want to corrupt the public key associated to an election so that it can decrypt the votes. However, a node cannot try to associate his own public key to an election. Indeed, when an election is created the collective public key generated by the running instance of DKG is automatically associated with this election. Therefore the public key of an election should be safe as long as the DKG protocol is not compromised.
- Attacks on the http server : see above
- Attacks on ballots : if the node hosting http server is malicious, see above. Otherwise the transaction to store the ballot on the blockchain is accepted and so if Dela is not compromised we should be good, a node cannot prevent a user from casting his vote, or modify the votes already stored. A voter cannot vote twice, only his last vote will be considered.
- Attacks exploiting the lack of authentication and access control : Users could try to close an election, run the shuffling protocol and the decryption protocol. Even now this would not work because we compare user ids with the admin id.
- A node could cast as much votes as he wants generating fake user ids. This attack will not work when a proper authentication system will be implemented.
- A node can basically interfere with the flow of an election (cancel...), just to make it so that the election we want to run is never performed. This attack will not work when a proper authentication system will be implemented.
- A node can close the election (ignore legitimate ballots), then trigger the shuffling and the decryption. The shuffling will not happen if there is only one ballot, so it cannot be used directly to infer the vote of a user, but if the adversary has an accomplice among users, then he can cast his vote (which the adversary knows) and then close the election when only one other user has caster his vote, the shuffle will happen and the adversary will be able to infer the user's vote because even if there are 2 shuffled votes he knows the user (accomplice) who cast one of these votes. This attack will not work when a proper authentication system will be implemented.
- Deleting an election : Right now a malicious node can create an election with the same id as an election that is currently on the blockchain and it will overwrite this election. This is easy to fix though.
- Attack on the shuffling protocol : anyone could try to discover the mapping between users and votes. The fact that we use Neff shuffle and that all nodes verify the proof of a given shuffle before it can be stored in the blockchain means that a user or a node cannot infer the mapping between users and votes from the encrypted votes (which are linked to user ids), the shuffled votes, the shuffling proof and the resulting decrypted votes.
- Attacks against integrity of ballots : a node cannot forge a shuffle proof (Neff shuffle) which means a node cannot temper with ballots at the shuffle phase. To recreate the mapping from user to votes, all nodes that have shuffled the ballots would have to collude, since we take the threshold for the number of nodes that have to shuffle to be at least 1/3 of the nodes, then more than 1/3 of the nodes would have to be malicious and collude with each other, they would then be able to perform this attack but also to attack Dela itself.
- Attack on the decryption protocol : A node could try to execute the decryption protocol on the ballots before they are shuffled, which would give him the votes of each user in clear text. We deal with this case in the protocol, it will trigger an error.
- We do not have an implementation for the proofs of decryption yet.

VIII. Future works

We did not have time to implement a real authentication system, and as such it could be the target of future works, specifically we thought of using EPFL's internal authentication service Tequila. Also, we only ran the nodes on the same computer. Deploying the system on different machines over a widespread network would allow to perform more tests, and We summarize our ideas for future works here :

- We did not have time to implement a real authentication system, specifically we thought of using EPFL's internal authentication service Tequila.
- We only ran the nodes on the same computer. It would be interesting to deploy the project on different machines over a widespread network. This would allow to perform more tests and benchmarks as well as to use the system for real elections, such as elections within the university.
- We could execute the DKG protocol on the blockchain
- The encryption and shuffling scheme currently implemented in kyber are limited respectively in term of the size of the plaintext to be encrypted and in term of the size of the ciphertexts to be shuffled. This could be dealt with in future works.
- We did not implement the proof of decryption of ballots. This would be good to have to complete the system.
- Improve the decrypt protocol implementation to allow the decryption of multiple ciphertext at once.

IX. Conclusion

The goal of our project was to implement a first proof-of-concept of a simple E-Voting system that runs on Dela. Our system allows admins to create several elections, multiple users can then cast several ballots, and admins can then close the election, trigger the shuffling of the ballots, and finally request the decryption of the ballots and publish their values in the blockchain so that anyone can verify the result of the election. The encrypted ballots, the shuffled ballots and the proofs of the shuffles, as well as the decrypted ballots are stored in the blockchain so that anyone can monitor the whole process. Also, the votes remain anonymous. Indeed, the ballots are encrypted with a collective public key so that more than 1/3 of the nodes would have to collude to decrypt the ballots. The ballots are shuffled using Neff's shuffle in such a way that more than 1/3 of the nodes would have to collude to discover which user cast which vote. All the information related to elections are stored in a blockchain so the integrity of the data, including the ballots, is guaranteed. A single node cannot prevent a correct transaction to perform the corresponding action, particularly a single node cannot prevent a user to cast his vote. Therefore, our system handles the scenario in section II-C satisfy the requirements defined in section II-B under the threat model defined in section II-D.

References

[1] Dela, the new dedis distributed ledger. https://github.com/dedis/dela. Last consulted 10.06.2021.

[2] "Epfl uses blockchain technology to secure e-voting systems." https://actu.epfl.ch/news/epfl-uses-blockchain-technology-to-secure-e-voting. Last consulted 10.06.2021.

[3] Cothority, the current dedis distributed ledger. https://github.com/dedis/cothority. Last consulted 10.06.2021.

[4] B. Adida, helios, Web-based open-audit voting.

[5] Coercion, https://en.wikipedia.org/wiki/Electoral_fraud#Intimidation. Last consulted 10.06.2021.

[6] C. A. Neff, "A verifiable secret shuffle and its application to e-voting." http://web.cs.elte.hu/~rfid/p116-neff.pdf. Last consulted 10.06.2021.

[7] P. J. Andrea Caforio, Linus Gasser, "A decentralized and distributed e-voting scheme based on verifiablecryptographic shuffles." 2017, https://www.epfl.ch/labs/dedis/wp-content/uploads/2020/01/report-2017-2-andrea_caforio-evoting.pdf. Last consulted 10.06.2021.

[8] PostgreSQL, https://www.postgresql.org/. Last consulted 10.06.2021.

[9] Kyber, dedis cryptographic library. https://github.com/dedis/kyber. Last consulted 10.06.2021.

[10] T. P. Pedersen, "A threshold cryptosystem without a trusted party." https://dl.acm.org/citation.cfm?id=1754929. Last consulted 10.06.2021.

[11] The go programming language. https://golang.org/. Last consulted 10.06.2021.

[12] Node.js. https://nodejs.org/en/. Last consulted 10.06.2021.

[13] Tequila, federated identity management system. https://tequila.epfl.ch/. Last consulted 10.06.2021.

Appendix

API

```
type LoginResponse struct {
        UserID string
        Token  string
}

type CollectiveAuthorityMember struct {
        Address   string
        PublicKey string
}

type CreateElectionRequest struct {
        Title            string
        AdminId          string
        Token            string
        Members          []CollectiveAuthorityMember
        ShuffleThreshold int
        Format           string
}

type CreateElectionResponse struct {
        ElectionID string
}

type CastVoteRequest struct {
        ElectionID string
        UserId     string
        Ballot     []byte
        Token      string
}

type CastVoteResponse struct {
}

// Wraps the ciphertext pairs
type Ciphertext struct {
        K []byte
        C []byte
}

type CloseElectionRequest struct {
        ElectionID string
        UserId     string
        Token      string
}

type CloseElectionResponse struct {
}

type ShuffleBallotsRequest struct {
        ElectionID string
        UserId     string
        Token      string
}

type ShuffleBallotsResponse struct {
        Message string
}

type DecryptBallotsRequest struct {
        ElectionID string
        UserId     string
        Token      string
}

type DecryptBallotsResponse struct {
}

type GetElectionResultRequest struct {
        ElectionID string
        Token string
}

type GetElectionResultResponse struct {
        Result []Ballot
}

type GetElectionInfoRequest struct {
        ElectionID string
        Token string
}

type GetElectionInfoResponse struct {
```

```go
        ElectionID string
        Title      string
        Status     uint16
        Pubkey     string
        Result     []Ballot
        Format     string
}

type GetAllElectionsInfoRequest struct {
        Token string
}

type GetAllElectionsInfoResponse struct {
        AllElectionsInfo []GetElectionInfoResponse
}

type GetAllElectionsIdsRequest struct {
        Token string
}

type GetAllElectionsIdsResponse struct {
        ElectionsIds []string
}

type CancelElectionRequest struct {
        ElectionID string
        UserId     string
        Token      string
}

type CancelElectionResponse struct {
}
```