A Decentralized and Distributed E-voting Scheme Based on Verifiable Cryptographic Shuffles

Andrea Caforio
andrea.caforio@epfl.ch
EPFL

Linus Gasser
linus.gasser@epfl.ch
EPFL

Philipp Jovanovic
philipp.jovanovic@epfl.ch
EPFL

Abstract— **In this project we propose a new electronic voting scheme that aims to leverage a decentralized and distributed architecture based on the blockchain technology in combination with verifiable cryptographic shuffles of ElGamal ciphertext pairs to achieve an efficient and somewhat scalable implementation. The system is constructed as a state machine which that the overall life cycle of an election, driven by several underlying distributed protocols that are executed at each phase of the election. Furthermore, the architecture is supposed to be agnostic in terms of the choice of the front end. It provides a lean API that can be accessed through protocol buffer messages over web sockets.**

I. INTRODUCTION

In 2008, Ben Adida published the white paper to his web-based voting scheme named Helios [1]. The rational was to use a Sako-Kilian mixnet, essentially a single shuffle, to provide an interface where any willing observer is able to audit an ongoing or completed election [2]. This means 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 has been terminated. 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.

We have to stress at this point that the Helios protocol does not opt to secure a voter's privacy but only his actual vote, thus everyone who has casted 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 [3]. This includes political elections where anonymity is crucial to avoid any potential threats against people who have casted a ballot. On the other hand it is perfectly suited for elections in smaller environments like universities or local clubs. Our project builds on the ideas used in the Helios project. In other words, the high-level goals of providing a completely verifiable election architecture where the protocol can be audited by any observer at any stage remains unchanged. In the same manner, our e-voting scheme does not hide the identity of the voter either but only protects his actual ballot. However, some of the core components of the architecture, especially the underlying protocols, have been replaced or adapted.

II. GOALS

The largest alteration to the base line Helios project is the replacement of the Sako-Kilian mixnet by Andrew Neff's verifiable secret shuffle [4]. This yields a significant performance improvement as described in more detail in the following chapter. Another large change is the actual storage architecture. Helios relies on a conventional relational PostgreSQL database model that is hosted on a single server that is also meant to perform the actual shuffle [5]. This reliance on a sole node is acknowledged in the Helios white paper which suggests a distributed architecture to mitigate this single point of failure. In our project we take it a step further by putting all the election related data on the blockchain, specifically a skipchain which is under development at the DEDIS laboratory at EPFL [6]. This switch comes with several improvements, not only is the data now administered by a collective of servers, those nodes can also be used to perform distributed protocols that enhance the reliability and aptitude of the system. For example, the public/private key pair is now generated by a distributed key generation algorithm that produces a global public key but shares the private key among the nodes such that no single node is able to decrypt the ballots or even reconstruct the master secret key on its own. This differs from the global key pair in the Helios project. Furthermore, since the election data is hosted on a blockchain the integrity of the data is provided by default. We also have to point out that the Helios project comprises both the back-end and the front-end whereas this project solely focuses on the server architecture only the providing an API and the specification a potential front end implementation has to follow in order to interface seamlessly with the back end.

We now look at the theoretical background of cryptographic shuffles, comparing both the Sako-Kilian method to Andrew Neff's shuffle protocol.

III. Theory

Both shuffling schemes work on ElGamal ciphertext pairs [7]. 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:

$$E(\mathbb{F}_p) = \{\mathcal{O}\} \cup \{(x, y) \in \mathbb{Z}_p; -x^2 + y^2 = 1 - \frac{121665}{121666} x^2 y^2\}$$

$$p = 2^{255} - 19$$

$$y_g = 4/5$$

$$n = 2^{252} + 27742317777372353535851937790883648493$$

Since the equation is quadratic in x, the x-coordinate of the base point g is set to be positive. The curve itself is isomorphic to the popular Curve25519 among others things used in Signal [9], [10]. 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. See below the correctness proof for a single re-encryption.

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 $(\mathcal{A}, \mathcal{B}, g, y)$ where

$$\mathcal{A} = \begin{pmatrix} u_i \\ v_i \end{pmatrix}$$

is a list of ElGamal ciphertext pairs and given a permutation π and a list of random values $r_i \in \mathbb{Z}_n$

$$\mathcal{B} = \begin{pmatrix} u_{\pi(i)} + gr_{\pi(i)} \\ v_{\pi(i)} + yr_{\pi(i)} \end{pmatrix}$$

is the re-encryption and permutation of $\mathcal{A}$. g and y are, as mentioned above, the base point of the curve and the public key. Now, a prover $\mathcal{P}$ needs to show to a verifier $\mathcal{V}$ that $\mathcal{B}$ has been generated from $\mathcal{A}$ 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.

A. *Sako-Kilian*

The protocol suggested by Kazuoe Sako and Joe Kilian is an interactive zero-knowledge proof but can easily be converted into a non-interactive version using the Fiat-Shamir technique [11]. The protocol unfolds as follows:

1) The prover $\mathcal{P}$ selects uniformly at random a list of random values $t_i \in \mathbb{Z}_n$ and a random permutation λ. He calculates
$$\mathcal{C} = \begin{pmatrix} u_{\lambda(i)} + gt_{\lambda(i)} \\ v_{\pi(i)} + yt_{\lambda(i)} \end{pmatrix}$$
and sends $\mathcal{C}$ to the verifier $\mathcal{V}$.
2) $\mathcal{V}$ prompts $\mathcal{P}$ with probability $1/2$ to show λ and t_i. Then $\mathcal{V}$ verifies that $\mathcal{C}$ is consistent with $\mathcal{A}$ by recomputing the shuffle with λ and t_i.
3) $\mathcal{V}$ prompts $\mathcal{P}$ with probability $1/2$ to show $\lambda' = \lambda \circ \pi^{-1}$ and $t'_i = t_i - r_i$. $\mathcal{V}$ then checks if $\mathcal{C}$ is consistent with $\mathcal{B}$ by recomputing the shuffle with t'_i and λ'.

Clearly, an honest prover $\mathcal{P}$ is able to respond to both queries from $\mathcal{V}$, whereas a malicious shuffler may only reply to at most one query correctly. Naturally, we can increase the strength of the proof by iteratively repeating the protocol. The correctness can then be assessed with probability

$$1 - (1/2)^n$$

where n is the number iterations. This need to recompute the protocol several times introduces a significant overhead in terms of computational complexity. Andrew Neff's verifiable shuffle avoids this performance weakness by proposing a single-pass proof protocol.

B. *Neff Shuffle*

Neff's protocol is more involved than the one from Sako and Kilian, therefore we only give a sketch at this point and leave it to the reader to dive into the details. In the most basic terms the protocol is a multi-dimensional extension of the Chaum-Pedersen protocol used to prove the equality of two discrete logarithms [12]. Applied to an ElGamal ciphertext $(u, v) = (gr, m + yr)$ where a prover $\mathcal{P}$ wants to show his knowledge of the secret key x to a verifier $\mathcal{V}$, the protocol unfolds like this:

1) $\mathcal{P}$ randomly chooses $w \in \mathbb{Z}_n$, then transits $A = gw$ and $B = uw$ to $\mathcal{V}$.
2) $\mathcal{V}$ responds with a challenge $c \in \mathbb{Z}_n$.

3) $\mathcal{P}$ sends back $t = w + xc$.
4) $\mathcal{V}$ verifies that $gt = A + yc$ and $ut = B + (v - m)c$.

More specifically, in Neff's protocol we can employ the Chaum-Pedersen protocol to show that given a shuffle of ElGamal ciphertext pairs

$$(\overline{\mathcal{U}}, \overline{\mathcal{V}}) = (\mathcal{U}_{\pi(i)} + g r_{\pi(i)}, \mathcal{V}_{\pi(i)} + y t_{\pi(i)})$$

it is possible that $\mathcal{P}$ proves the existence of some permutation π such that $r_i = t_i$ to $\mathcal{V}$. More generally, we can show that for a random vector $\vec{s}$ we have that

$$\vec{s} \cdot \vec{r} = \vec{s} \cdot \vec{t}.$$

C. Benchmarks

Even though, Neff's shuffle protocol introduces more overhead than the rather simple protocol of Sako and Kilian it still performs better than the iterative approach. This benchmark test was run on a standalone dual-core 2.4 GHz machine. The algorithm shuffles and performes a non-interactive proof on increasing number of ElGamal pairs.

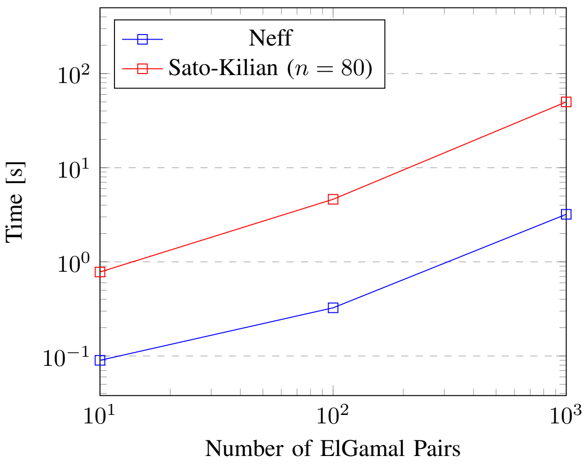

Fig. 1. Comparison of Shuffle Verification Protocols

Having dealt with the theoretical parts of the project, we can now turn our attention to the actual architectural design and its implementation.

IV. Architecture

The project's structure is divided into several more or less self-sustaining parts that are glued together by a collective authority (cothority) [13]. In the following subsections we discuss the individual parts in more detail. We start with the cothority as it is the project's backbone.

A. Cothority

The collective authority is a research project at the DEDIS laboratory at EPFL. It is meant to be a framework that eases the deployment of decentralized and distributed services and protocols. A cothority comprises a set of servers, called conodes, which handle the workload in a distributed manner. Furthermore, the cothority provides a platform for handling arbitrary data blobs with the means of a blockchain, specifically an alternative implementation called skipchain. As already mentioned, all the accumulated data regarding elections is stored within skipchains.

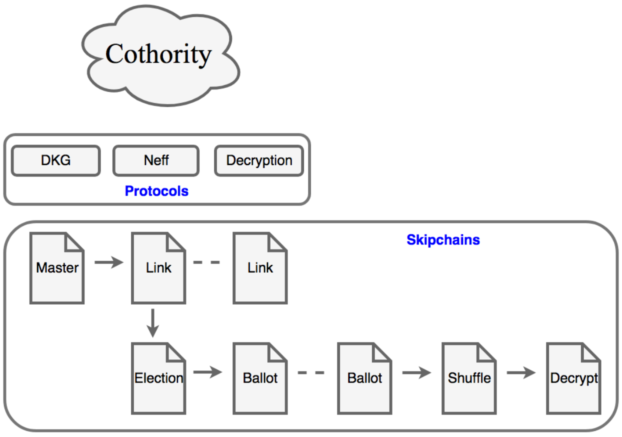

Fig. 2. Architecture Composition

B. Master Skipchain

The master skipchain handles configuration data that is global to a set of elections. In the genesis block an administrator with physical access to one of the servers can store a list of servers which are meant to handle the protocols and all the skipchains including the master skipchain and a public key of a potential front-end application (e.g. a web interface). Further, a list of election administrators can be added as well. Election administrators have the privilege to create new elections.

Creating a new election goes hand in hand with the generation of its companion election skipchain. Skipchains are identified by a 32-byte identifier. Thus to avoid having to remember elections, their identifier is appended in a separated block to the master skipchains. Those blocks referencing election skipchains are called links. In order to simplify the interaction with the master skipchain its genesis block also contains its own identifier.

Master (Genesis)
Server list (roster)
Front-end public key
Election admins
Skipchain ID

Link
Election skipchain ID

Fig. 3. Content of Master Skipchain Blocks

C. Election Skipchain

After being created by an administrator an election skipchain contains all the data related to it. This includes the actual election data (i.e. name, creator, voters etc.) and the ballots casted by participating voters. As with the genesis block of the master skipchain the election genesis block also contains the server list and the skipchain identifier. More importantly, since every election is assigned a public/private key pair upon creation, the genesis block also stores the public key whereas the secret key is shared among the servers. See the DKG section for more on the distributed key generation protocol. After the creator, and only the creator, closes an election the shuffled ballots and then in the end the decrypted ballots are appended. The shuffled and decrypted ballots differ from the ordinary encrypted ballots in the point that they are accumulated in a single block whereas every casted ballot is stored in a separated block. This is due to the fact that once a skipchain block has been added it cannot be altered anymore. An election skipchain has thus always the same structure where the encrypted ballots follow the genesis block and the accumulated shuffled and decrypted ballots make up the tail of the skipchain. Every other format must be considered invalid, rendering the entire election corrupt.

Election (Genesis)
Name
Description
Creator
Election public key
Voter list
Server list (roster)
Skipchain ID

Ballot Enc.
User ID
ElGamal pair

Ballot Dec.
User ID
Ballot data

Fig. 4. Content of Election Skipchain Blocks I.

In order to reduce the data load, a ballot only contains a user's identifier (some number) and his encrypted vote in form of an ElGamal ciphertext pair. In the same spirit the decrypted ballots only contain the user identifier and the decrypted vote in raw bytes. It is then up to the front-end application to make sense of the actual election data.

Shuffle (Box)
List Ballot Enc.

Decryption (Box)
List Ballot Dec.

Fig. 5. Content of Election Skipchain Blocks II.

V. Protocols

The project is driven by its underlying protocol that are executed upon the creation of an election, to generate the shuffle of the encrypted ballots and after termination to produce the plaintext ballots for tallying. The cothority infrastructure makes it easy deploy and test complex distributed protocols without the hassle to implement the actual networking and exchange of messages.

A. DKG

The distributed key generation protocol is a direct implementation of a proposal by Gennaro et. al., 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 [14]. The protocol itself is an improvement over the initial work by Pedersen in his proposal for a verifiable secret sharing protocol [15]. Each participating server $\mathcal{S}_i$ of a set of servers $\mathcal{S}$ starts off by randomly selecting a polynomial of degree t, $f_i(z) = a_{i0} + a_{i1}z + \ldots + a_{it}z^t$ with coefficients in $\mathbb{Z}_n$. Here, $z_i = a_{i0} = f_i(0)$ is each party's secret. The protocol then proceeds with a cascade of message exchanges until ending up with a set of qualified servers *QUAL* excluding potentially dishonest parties from further participating. The shared secret is then reconstructed with $x = \sum_{i \in QUAL} z_i$. From this, the public key calculation follows with $y = gx$.

The raw implementation of the above protocol is part of the cothority code base and is thus easily included in our project. This means that whenever an administrator opens a fresh election the protocol is executed to set up a shared secret. The nodes which are part of the qualified set of honest servers make up the server list stored in the election's genesis block. Naturally, the public key produced by the protocol can then be used by a front-end application to encrypt ballots before casting them.

B. Shuffle

Closing an election goes hand in hand with the execution of the shuffle protocol which re-encrypts and permutes the ballots several times before storing the output back on the election skipchain providing a verifiable prove of the shuffle. In more detail, the protocol unfolds as follows:

1) The server that is first contacted by the election creator collects all the casted ballots only taking the last casted ballot of each voter.
2) It then performs the first re-encryption of the ElGamal ciphertext pairs with simultaneous permutation.
3) The shuffled ballots are then passed on to a random server of the *QUAL* set alongside a prover function which then first checks the integrity of the received shuffle before repeating the re-encryption and permutation before sending it to a next node.
4) After calculating its shuffle the last remaining server of the *QUAL* set sends its result to the initial server which then appends the product to the election skipchain.

After the completion of the protocol the shuffle with the prove of correctness can be retrieved by any observer.

C. Decryption

An election is always terminated with the execution of the decryption protocol. It proceeds in a similar way as the shuffle protocol.

1) The server that is first contacted by the election creator prompts the remaining nodes of the *QUAL* set to decrypt the shuffled ballots with their own secrets and to send the result back to the root node.

2) Since a single node can not fully decrypt the encrypted ballots, the root node itself runs its own decryption pass with its secret then reconstructs the ballot plaintexts by a Lagrange interpolation over all the partially decrypted ballots as indicated in Gennaro et. al..
3) The result is finally appended in the concluding block of the election skipchain.

We have to note that at no point during the decryption protocol is it necessary that one server accumulates the shared secrets, hence the decryption does not invalidate the premise of the distributed key generation protocol. Furthermore, the Helios project provides a Chaum-Pedersen non-interactive proof of decryption. At the current stage of the project this feature has not been implemented yet, however it would be part of a future enhancement plan.

VI. Implementation

A. Golang

Due to the fact that the whole cothority project and its companion advanced crypto library kyber are almost entirely written in Go, it was natural that our e-voting scheme has to be coded in the same language. The sole exception is a standalone shell script that sets up a given number of cothority servers (conodes) [16], [17].

The project is split into more or less independent packages each serving one major point of the project.

```
project
 ├ api
 │ └ ...
 │
 ├ chains
 │ └ ...
 │
 ├ cli
 │ └ ...
 │
 ├ crypto
 │ └ ...
 │
 ├ decrypt
 │ └ ...
 │
 ├ dkg
 │ └ ...
 │
 ├ service
 │ └ ...
 │
 └ shuffle
   └ ...
```

Fig. 6. Project Directory Structure

- **api**: Contains the API specification, meaning the concret protocol buffer message in both a .proto and .go file.
- **chains**: Here all the skipchain related mechanisms are handled. This includes everything storage and retrieval related. This package also defines the actual election, master and ballot structures.
- **cli**: A command line interface for administrators to set up a master skipchain. This naturally requires physical access to one of the servers.
- **crypto**: The crypto package contains the definition of the elliptic curve as well as the ElGamal encryption decryption suite.
- **decrypt**: Location of the decryption protocol code.
- **dkg**: Location of the distributed key generation protocol code. This is mostly a rewrite of the code located in the kyber library.
- **service**: Core of the application. Here, all the incoming messages are received and processed. This means all actions within the protocol originate in the service protocol. Furthermore, this packages handles the user login procedures.
- **shuffle**: Location of the shuffle protocol code.

B. Front-end

As already previously mentioned, this project leaves the front-end implementation to other sides. The only requirement for a seamless interaction 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. Furthermore Google's protocol buffers are required for sending and receiving messages [18]. Depicted below is a possible full architecture. Here, users access the front-end application through a browser that is for example hosted by a node.js server and which authenticates the user towards a database [19]. In the our case this would be the EPFL's internal authentication service [20]. As soon as the user is authenticated the front-end relays the communication towards our back-end.

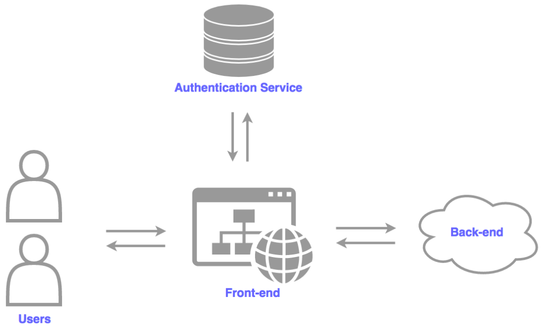

Fig. 7. Integration Sketch

C. API

Below is a shortened listing of the most important API messages and responses to create and handle elections. Note that both the actual election and ballot message objects have been omitted for brevity. Also both the master and genesis fields refer to the skipchain identifier converted to a base64 string.

```
message Login {
    required string master = 1;
    required uint32 user = 2;
    required bytes signature = 3;
}

message LoginReply {
    required string token = 1;
    required bool admin = 2;
    repeated Election elections = 3;
}

message Open{
    required string token = 1;
    required string master = 2;
    required Election election = 3;
}

message OpenReply {
    required string genesis = 1;
    required bytes key = 2;
}

message Cast {
    required string token = 1;
    required string genesis = 2;
    required Ballot ballot = 3;
}

message CastReply {
    required uint32 index = 1;
}

message Aggregate {
    required string token = 1;
    required string genesis = 2;
    required uint32 type = 3;
}

message AggregateReply {
    required Box box = 1;
}

message Shuffle {
    required string token = 1;
    required string genesis = 2;
}

message ShuffleReply {
    required Box shuffled = 1;
}

message Decrypt {
    required string token = 1;
    required string genesis = 2;
}

message DecryptReply {
    required Box decrypted = 1;
}
```

Listing 1. Protobuf API Specification

D. Benchmarks

We want to end this chapter with some indicative benchmarks regarding the run-times of the shuffle and decryption protocols since the this project aims to be scalable in the sense that it should be able to handle several hundred to thousands of ballots without unbearable waiting times. The benchmark have been conducted in a local network with an increasing number of participating servers and ballots.

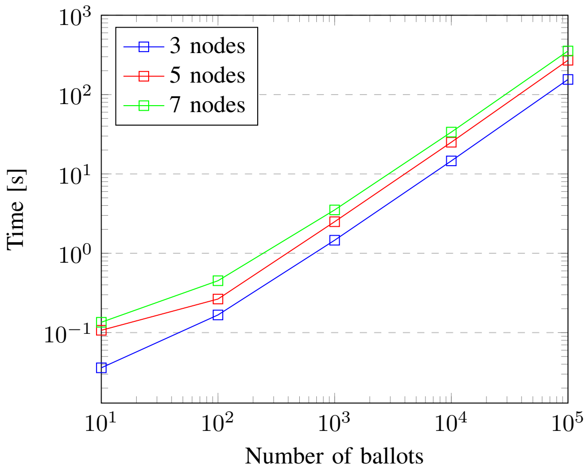

Fig. 8. Shuffle Protocol Benchmarks

We note that increasing the number of participating servers only marginally slows down the protocol thus indicating the most of the execution time is spent with permuting and re-encrypting the ballots rather than propagating messages throughout the network. The shuffling algorithm has complexity of $\mathcal{O}(n)$ where n is the number of ballots, although with rather heavy computations inside the iteration. The decryption protocol benchmarks are similar to those from the shuffling protocol, although asymptotically faster. However, the composition of the algorithm is different. Extracting the plaintexts from the encrypted ballots is a $\mathcal{O}(v * n)$ operation where the v is the number of servers and n the number of ballots.

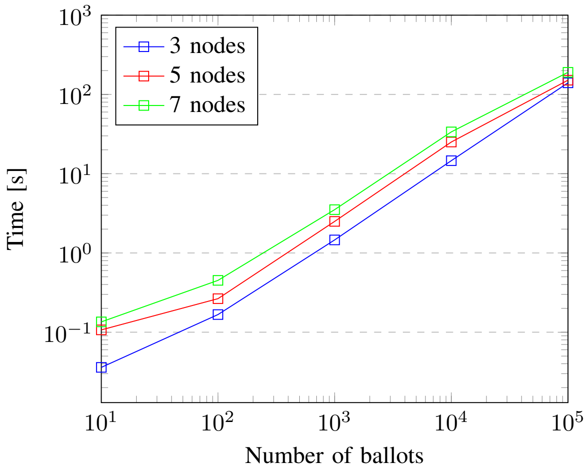

Fig. 9. Decryption Protocol Benchmarks

VII. OUTLOOK

The main goal of this project is a real-world integration for some future elections, specifically elections within the university since they fit the project the best. Any deployment requires a robust front-end that interfaces well with the back-end. Here, it is still to see if the Andrew Neff's verification protocol for proving the integrity of secret shuffles can be implemented in a front-end context. For the moment the non-interactive proof protocol only exists as part of the DEDIS advanced crypto library. On the other hand, integrating the project on physically separated machines over a widespread network will pose new challenges on itself that could not have been assessed yet as part of this project. However, in a spin-off bachelor-level semester project a proof-of-work Javascript web front-end is being created whose goals is to establish a concrete platform on which election for handling elections [21].

VIII. Conclusion

This project was part of the CS-498 Project in Computer Science II course offered by EPFL. It was conducted during the entire second half of the year 2017 by Andrea Caforio under the supervision of Linus Gasser and Phillip Jovanovic and was offered by the Distributed and Decentralized Systems Laboratory at EPFL lead by Prof. Bryan Ford. The project's source is hosted on Github at https://github.com/dedis/student_17_evoting.

IX. Installation

The project requires a installation of the latest of Go compiler with a set and working Go path. The code can then be downloaded as follows:

```
git clone github.com/dedis/student_17_evoting
```

To set up a local cothority cluster with five servers, run the below snippet:

```
cd student_17_evoting
./setup.sh run 5 3
```

The cothority is now up and running and can be accepted through the addresses in the created public.toml file. Please refer to the documentation in Github repository for more detailed installation instructions.

References

[1] Ben Adida. Helios: Web-based open-audit voting.

[2] Kazue Sako and Joe Kilian. Receipt-free mix-type voting scheme. In *Advances in Cryptology—EUROCRYPT'95*, pages 393–403. Springer, 1995.

[3] Electoral fraud. https://en.wikipedia.org/wiki/Electoral_fraud#Intimidation, 2017.

[4] C Andrew Neff. A verifiable secret shuffle and its application to e-voting. In *Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 116–125. ACM, 2001.

[5] Postgresql. https://www.postgresql.org, 2017.

[6] Kirill Nikitin, Lefteris Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Justin Cappos, and Bryan Ford. Chainiac: Proactive software-update transparency via collectively signed skipchains and verified builds. 2017.

[7] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985.

[8] Daniel J Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. Twisted edwards curves. In *International Conference on Cryptology in Africa*, pages 389–405. Springer, 2008.

[9] Daniel J Bernstein. Curve25519: new diffie-hellman speed records. In *International Workshop on Public Key Cryptography*, pages 207–228. Springer, 2006.

[10] Signal. https://signal.org/docs/, 2017.

[11] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 186–194. Springer, 1986.

[12] David Chaum and Torben P Pedersen. Wallet databases with observers. In *Crypto*, volume 92, pages 89–105. Springer, 1992.

[13] Ewa Syta, Iulia Tamas, Dylan Visher, David Isaac Wolinsky, and Bryan Ford. Decentralizing authorities into scalable strongest-link cothorities. *CoRR, abs/1503.08768*, 2015.

[14] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. In *Eurocrypt*, volume 99, pages 295–310. Springer, 1999.

[15] Torben P Pedersen et al. Non-interactive and information-theoretic secure verifiable secret sharing. In *Crypto*, volume 91, pages 129–140. Springer, 1991.

[16] Kyber, dedis advanced crypto library. https://github.com/dedis/kyber, 2017.

[17] The go programming language. https://golang.org/, 2017.

[18] Protocol buffers. https://developers.google.com/protocol-buffers/, 2017.

[19] Node.js. https://nodejs.org/en/, 2017.

[20] Tequila identity management system. https://tequila.epfl.ch/, 2017.

[21] Evoting epfl - frontend and authentication. https://github.com/dedis/student_17_evoting_frontend, 2017.