Sharing Responsibilities through Distributed Key Generation in the Chirotonia e-Voting Framework

A. M. Calabria[1,†], A. Piscitelli[1,†], S. P. Romano[1,*,†] and A. Russo[1,†]

[1]*University of Napoli Federico II, Naples, Italy*

Abstract

Electronic voting systems are the subject of this paper. In particular, we focus on the *Chirotonia* platform, a secure and scalable voting framework based on blockchain and linkable ring signatures. One of the missing points of Chirotonia concerns the ownership of Confidentiality and in particular the use of a mechanism that allows to manage, in a distributed manner, the generation of the cryptographic material necessary to encrypt and decrypt ballots during and at the end of the voting process. The integration of a protocol for the *distributed generation of cryptographic keys* within the relevant web administration portal is hence appropriate. We will present in the paper a blockchain-based approach for the seamless integration of such a protocol inside Chirotonia.

Keywords

E-voting Frameworks, Distributed Key Generation, Ring Signatures, Smart Contracts

1. Introduction

Electronic voting systems can be seen as tools aimed at improving traditional voting, by facilitating and speeding up the process of counting ballots, as well as removing possible human errors that could affect the accuracy of the results. On the other hand, electronic voting architectures do represent critical systems from the security perspective and therefore require early identification of the requirements and call for security checks based on the analysis of potential vulnerabilities, threats, attacks and associated risks.

In order to function properly, it is assumed that for each voting session, a predefined authority configures the system and distributes the cryptographic material to all of the involved actors, including the voters (with different material for each of them). A key requirement, in order to guarantee all the security properties of the system, is that the authority in question does not share the secret information it has generated in the name of the various participants. Indeed, even if the computation of information can be verified, it is not possible to prevent a malicious authority from sharing voters' secret information with other parties, which would allow them to vote on someone else's behalf.

ITASEC 2023: The Italian Conference on CyberSecurity, May 03–05, 2023, Bari, Italy

*Corresponding author.

†These authors contributed equally.

alfo.calabria@studenti.unina.it (A. M. Calabria); anna.piscitelli5@studenti.unina.it (A. Piscitelli); spromano@unina.it (S. P. Romano); antonio.russo74@studenti.unina.it (A. Russo)

0000-0002-5876-0382 (S. P. Romano)

The rest of the paper is structured as follows. In Section 2, a quick overview of the Chirotonia e-voting platform is provided, by focusing on the overall architecture, as well as highlighting the need for improving it with the addition of distributed key generation functionality. Section 3 briefly describes how we are currently leveraging Chirotonia in order to support e-voting sessions at the University of Napoli Federico II. Section 4 presents the distributed key generation protocols available in the literature, with particular attention to the Ethereum Distributed Key Generation (ETHDKG) protocol. Section 5 goes deeper into the details of how we integrated the ETHDKG protocol inside Chirotonia, by focusing on the high level architecture and briefly describing the various phases involved. Implementation of the integrated architecture is discussed in Section 6. Finally, Section 7 draws some conclusions by summarizing the main results achieved with our work.

2. The Chirotonia e-voting framework

Chirotonia (from the Greek $\chi\epsilon\iota\rho o\tau o\nu\acute{\iota}\alpha$, a voting system based on show of hands) is an architecture proposed for implementing a secure electronic voting system, based on the use of *blockchain*, *smart contracts* and *linkable ring signatures* [1].

The voting protocol proposed by Chirotonia is divided into the following phases: (i) organization of the voting session, (ii) registration, (iii) opening of the voting session, (iv) voting in progress, (v) closing the voting session and (vi) counting. In each such phase different actors are involved: the *voter*, the *organizer*, the *identity manager* and the *confidentiality manager*.

In particular, the Confidentiality Manager is responsible for storing and distributing (when required) the information necessary to encrypt and decrypt the votes. There are two other components that are used in the different phases of the process: the *ID Storage*, that is the bulletin board containing all the identities of the voters who have the right to vote and the *Ballot Box*, that is the archive of valid voting cards.

When the voting phase is over, it is up to the Confidentiality Manager to release the secret key (paired with the public key used to encrypt the votes) so that anyone, thanks to the information publicly available on the blockchain, can decrypt the votes and calculate the final result.

Chirotonia uses a scheme of connectable ring signatures based on *ECC* (*Elliptic Curve Cryptography*). This is done in order to obtain cryptographic keys and digital signatures of reduced dimensions.

2.1. Security requirements achieved

An e-voting system, to be a secure system, must meet certain requirements. Below we will focus just on Confidentiality, that is the property of interest for the scope of this paper.

To archive valid ballot papers, Chirotonia uses a Ballot Box built on the blockchain. Votes are collected gradually during the voting phase. Therefore, a mechanism is needed to avoid early disclosure of results, as this could affect those who have not yet cast their vote. To do this, there are several mechanisms that can be used. One option is to leverage an ad hoc component called *Confidentiality Manager*, that manages the information needed to encrypt/decrypt the votes. Indeed, the task of generating the necessary keys should be shared among different entities, to avoid the presence of a single point of failure, as an attacker might compromise it

and invalidate the election results. To reach such a goal, one might think of integrating the *ETHDKG (Ethereum Distributed Key Generation)* [2] protocol into the voting platform. ETHDKG allows different entities to generate and share a pair of keys, using a K-out-of-N threshold mechanism for rebuilding the private key.

As an alternative to the distributed generation of keys, a two-step procedure might be used, in which, during the first phase (*Submit Step*), the voters express their preference, but do not send their vote to the ballot box. They rather store it aside and send the ring signature of the hash string of the vote. Subsequently, during the *Redeem* phase, the voters submit their vote. Only votes that match previously stored hashes will be accepted. This turns out to be a good solution to guarantee confidentiality. Though, there are disadvantages to consider. As an example, the storage of the vote during the first phase must take place on a device or system that has to be kept safe.

We opted for the ETHDKG approach as the preferred one for Chirotonia.

3. Real use case scenario: university elections

Chirotonia is currently the official e-voting platform of the University of Napoli Federico II in Italy. During the last year it has been used for a number of elections of university officials and representing faculty members. The platform has also become one of the official use cases of the Italian Blockchain Service Infrastructure (IBSI).

The overall architecture makes use of a blockchain, a blockchain explorer, a web server that acts as an Identity Manager, a web server that acts as an anonymization proxy for submitting votes, an administration portal and a web application used by voters to interact with the system.

Before voting begins, voters are required to register with the system and generate an *electoral card* which includes: a pair of cryptographic keys, the email of the voter in question and a string, used to generate the keys. The secret key is encrypted with a password and the public key is sent to the Identity Manager. The identity of the voter is verified by the university system, through the use of the SAML 2.0 protocol. Following verification, the voter can either save their card in the browser or download and store it on a device.

During the voting phase, voters can select their preferences, load the voting card in the browser (if it was not previously stored) and enter the password to decrypt the private key. The browser, in turn, calculates the connectable ring signature and encrypts the voting choice, sending both results to the anonymization proxy, which redirects the data to the blockchain, returning the hash of the transaction to the voter and a link to the explorer of the blockchain, valid as a receipt.

When the voting phase is over, the members of the electoral committee, through the administration portal, request decoding of the votes. Since ETHDKG has not yet been integrated into the portal, the committee generates a pair of RSA keys, publishing on the Ballot Box the public key used to encrypt the votes during the voting phase and securely storing the associated secret key needed to decrypt them. Such a key is used right after closure of the voting phase in order to enable proper decryption of the submitted ballots.

The mentioned procedure is far from optimal. That's why we have recently worked on improving it through the integration of ETHDKG inside Chirotonia.

4. Distributed Key Generation

A distributed key generation (DKG) protocol is a fundamental tool for threshold-based cryptographic systems. Its main goal is to generate a pair of keys, public and private, so that the private key (the secret) is shared between n participants. Only through the collaboration of $t + 1 \leq n$ nodes can the generated secret key be revealed or used. Subsets of lower dimension cannot succeed with the above task.

At the basis of a DKG protocol are the protocols used for the fair sharing of a secret. Such protocols rely on a trusted third party (or dealer) who has the task of generating and distributing the secret key among the various members.
Distributed key generation wants to remove this single point of failure, so the key pair is calculated thanks to the cooperation among the participants.

By using a threshold scheme (k, n), where $n = 2k - 1$, a robust key management procedure can be obtained. In fact, this mechanism allows to recover the key even when half of the fragments have been destroyed or lost. Furthermore, even in case a security breach exposes all but one remaining fragments, it is impossible for an attacker to be able to reconstruct the key.

One of the first proposed threshold schemes is that of Adi Shamir and is based on *polynomial interpolation* [3]

4.1. Distributed key generation: state of the art

Distributed key generation is a topic that has been studied for over two decades and there are several protocols proposed in the literature.

The first protocol for DKG was introduced in 1991 by Pedersen [4], whose study aimed to implement a scheme for sharing a secret among members of an organization, without the help of a trusted third party that selects a secret key, splits it and distributes each of the resulting fragments to the participants. Rather, it is the members who select the secret and verifiably distribute it between each other, so that each member of the group can verify that the sharing is correct. The verifiability property is important because the different fragments are no longer calculated by a trusted party, and therefore they are not necessarily calculated by everyone correctly. The problem with the solution proposed by Pedersen is that the information necessary for the calculation of the public key is shared before the different parties have agreed on the fragments to be used to create the master secret key. By doing so, an attacker could influence the final result (*biasing attack*) having previously bribed parts of the organization.

The *Joint-Feldman* protocol, proposed by Gennaro et al. [5], is an alternative to that of Perdersen. In both works, the basic idea is that each participant executes Feldman's *VSS* (Verifiable Secret Sharing) protocol for randomly sharing a (previously chosen) secret among all parties. Subsequently, the public key can be obtained using the *commitments* published during the first phase of the Feldman protocol. Instead, the corresponding private key can be obtained with the collaboration among the parties. In addition, several mitigation strategies are presented in the Joint-Feldman protocol for attacks that can manipulate the final output of the process. However, this approach adds complexity to Pedersen's initial protocol as it involves an additional sharing step using the VSS scheme.

Most of the solutions proposed in the literature to improve Pedersen's DKG protocol have several disadvantages: they use private communication channels, require participants to reveal their secret fragments to prove their honesty, use bugged mechanisms to recognize dishonest participants, and make use of complex and computationally expensive calculations. This makes them not easy to use in practical situations. For this reason, Neji et al. [6] propose further countermeasures to biasing attacks and remove the double step of sharing the secret. In addition, they first present an extended version of the DKG Joint-Feldman protocol, which ensures a uniform distribution of the generated keys. Subsequently, they devise a DKG protocol that makes use of public communication channels and introduces a new strategy to identify dishonest participants who share invalid fragments. This is achieved by performing a *dispute phase* to disqualify them.

Schindler et al., combining the proposals of [5] and [6], present their *ETHDKG* [2] protocol (described in detail in the next section). Specifically, they provide a more efficient mechanism for handling disputes during protocol execution. Furthermore, the presented protocol is generally described for any encryption system based on discrete logarithms.

The authors start by considering a set P of n participants who intend to jointly generate a pair of master keys (one public and one secret), in the form $mpk = g^{msk}$. The public key mpk is the protocol output; instead, the corresponding private key msk is shared among the participants and can be obtained by putting together fragments of at least $t + 1$ parts.
It is assumed that all participants can monitor an authenticated communication channel and use it to broadcast all the messages envisaged in each phase of the protocol. In this regard, a *synchronous* model is considered, in the sense that all messages sent by a participant during the execution of a phase are received by all the others before the start of the next phase. The authenticated public communication channel that the authors refer to in their implementation is the Ethereum blockchain, hence the name of the protocol.

4.1.1. ETHDKG phases

The *Ethereum distributed key generation* protocol has three main phases:

1. *sharing phase*, in which each participant $P_i \in P = \{P_1, ..., P_n\}$ selects a random secret and uses Feldman's VSS protocol to share it with others, in such a way that $t + 1$ parts can rebuild it in case a malicious part holds its piece in the derivation phase;
2. *dispute phase*, in which any participant who has received invalid fragments can prove to the other participants that someone has violated the protocol;
3. *key derivation phase*, in which a $Q \subseteq P$ set of P_i parts is composed, which contribute to the calculation of the $\langle mpk, msk \rangle$ key pair.

Before the start of the protocol, each participant $P_i \in P = \{P_1, ..., P_n\}$ has the task of generating a key pair $\langle sk_i, pk_i \rangle$ in the form $pk_i = g^{sk_i}$. The latter must be shared with all other participants. These keys are used to calculate other cryptographic keys that will be used for the symmetric cryptography algorithm used to guarantee the secrecy of the broadcast messages exchanged during the sharing phase of the protocol.

Again, if a participant P_i does not share their secret s_i, it can be obtained with the above formula, as each $P_i \in Q$ has correctly shared the fragments of their secret during the initial

phase of the protocol. In some application scenarios it may not even be necessary to calculate the private key.

The security properties that a distributed key generation protocol must aim for are: *correctness*, *secrecy*, *uniformity*, *robustness*, *liveness*. The authors of ETHDKG [2] report a definition and a security proof for each of these.

5. ETHDKG integration into Chirotonia

As we discussed previously, the *confidentiality* property is of fundamental importance for any e-voting platform in order to guarantee the total correctness and neutrality of the entire voting process.

In this regard, the use of the *DKG* protocol is suggested, which allows different entities within a group to generate and share a pair of keys, through a *k-out-of-n threshold* mechanism.

In this section we discuss how we integrated the above mechanism within the administration portal of Chirotonia for the management of cryptographic keys. The execution of the DKG protocol is foreseen before the voting phase is open, in order to generate the master public key used to encrypt the ballot papers. When the voting is over, the secret key is calculated and revealed in order to carry out the counting phase. In our case, the Ethereum blockchain is used as a communication channel among the parties.

5.1. General Architecture

As said, the ETHDKG protocol has been integrated into the Chirotonia web administration portal, which currently uses a pair of RSA keys to encrypt and decrypt the ballot papers with the votes received. In particular, the public key is used to encrypt the cards before sending them to the system; the corresponding private key is released after voting by the Confidentiality Manager and used by the Organizer to decrypt the votes and calculate the result of the election. By doing so, the Confidentiality Manager has the possibility to completely invalidate a voting session, as the disclosure of the secret key depends exclusively on that component. For this reason, the responsibility for generating the key pair is attributed to a group of cooperating entities, effectively applying the distributed key generation protocol.

This section presents the architecture adopted for the integration of ETHDKG in Chirotonia. Within the *Chirotonia administration portal* there is a *"Committee"* section, in which the members of the committee of a specific election can personally carry out the various phases of the protocol and check that they have been completed correctly by others as well.

In ETHDKG it is assumed that all participants actively monitor a public and authenticated communication channel and, moreover, a synchronicity hypothesis is made in order to separate the different phases. The *Ethereum blockchain* was also chosen as the communication channel for our implementation. Specifically, the client looks at all the transactions made to the address of the previously distributed DKG smart contract. A message is sent through the issuance of an Ethereum transaction, which effectively performs a function defined within the smart contract. If the call is made correctly, the contract generates an Ethereum event, which is then processed on the client side, in our case from the administration portal.

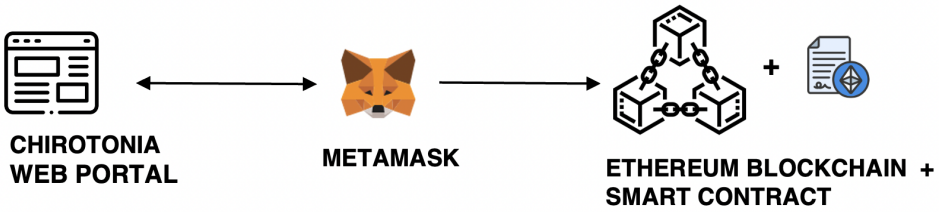

Figure 1: Architecture for ETHDKG integration in Chirotonia.

The assumptions of synchronicity are guaranteed by specifying in the contract the beginning and the end of each phase of the protocol on the basis of the height of the produced blocks reached within the chain. So, on the client side, after receiving a new event, one has to wait for a certain number of blocks before they can proceed with the subsequent steps of the protocol.

In Figure 1 the architecture of the system is shown at a high level. In particular, the central element *MetaMask* is inserted to simplify user interaction with the blockchain. The latter is a middleware that has nodes synchronized with the blockchain and, through the *Web3* interface, allows the client to request data or carry out transactions on it.

5.1.1. Chirotonia web portal

The web portal of the Chirotonia electoral committee is used by members of an electoral commission to control the voting process. In particular, thanks to it, it is possible to view the list of voters and candidates for a specific voting session and the relative results after the ballot has taken place.

5.1.2. Ethereum: blockchain and smart contracts

Ethereum [7] is a technology that aims to create an alternative protocol for the construction of decentralized applications, through a blockchain with an embedded Turing-complete programming language. It allows everyone to establish, through the use of smart contracts, their own rules for the properties, transaction formats and state transition functions necessary for the execution of an application [8].

Thus, Ethereum can be formally described as a state machine, based on ordered transactions, in which the state is made up of objects called *accounts*, each of which is associated with a 20-byte address. Status can include information such as account balances, reputations, trust agreements, physical world information data: in short, anything that can currently be represented by a computer. There are two types of accounts in Ethereum:

1. **Externally owned accounts:** there is a pair of cryptographic keys (public and secret) used by a user to sign and send transactions within the network. Specifically, the secret key is used to sign transactions and the public key is used to verify the validity of the signature.
2. **Contract account:** there is a smart contract, controlled by its own code, which is deployed from an account owned externally by a user. In this case there is no private key associated

with the account, but the latter stores the code defined within the contract, which decides the flow of *ethers* (ETH – ethereum's native currency) between the accounts. Obviously, the smart contract cannot execute code by itself, but must interact with an account in order to perform the functions defined within it. In fact, any account of type 1 can send transactions to the contract address.

Both accounts can accumulate and spend a certain number of ethers to transact or use computational resources on the network.

As mentioned, the status inside the Ethereum machine consists of the information associated with an account. Transactions, on the other hand, represent an arc between two states and therefore, if valid, their submission entails a valid state change in the network. Transactions are collected within blocks, linked together through cryptographic hashes.

The model provided by Ethereum is highly customizable and allows the creation and execution of different types of applications, thanks also to the use of smart contracts. A smart contract contains code within it that allows the transfer of digital assets between the different participants of the network (by actually making a transaction or a call) based on certain rules defined within the contract itself. The participants in the network, who do not trust each other, can therefore safely "negotiate" thanks to the rules defined in the contract.

A smart contract in Ethereum can be seen as an "autonomous agent" that lives within the execution environment and that calls a specific portion of code when the latter is activated by a message or a transaction. Thanks to this mechanism it is possible to execute any program, regardless of the programming language used. This is possible thanks to the main innovation introduced by Ethereum, the *Ethereum Virtual Machine*. The EVM is a quasi-complete Turing machine, where the quasi is due to the fact that the calculation operations are intrinsically limited by a parameter, namely the *gas*. The architecture of the EVM is the classic *stack-based* architecture. It foresees a word of 256 bits (this to facilitate the cryptographic operations on elliptic curves, for example calculation of the Keccak-256 hash function). Memory is word-addressable and the stack has a depth of 1024 words. All operations are performed on the stack [7].

Before being able to execute the instructions of a smart contract, the latter is compiled by generating a bytecode, in which each byte represents an operational code among those belonging to the set of operational codes recognized by the EVM. An operational code, when interpreted, is executed and consequently updates the state of the blockchain.

5.1.3. MetaMask

MetaMask [9] is a browser extension designed to access distributed applications (Dapps) that leverage the blockchain. In particular, it manages user accounts and their connections to the distributed network. When a Dapp wants to execute a transaction and write to the blockchain, the user, through a secure interface, can review the transaction and, consequently, approve or reject it.

5.2. Interaction with the platform

The ETHDKG protocol, as described in the previous section, includes several phases. Below, they are briefly described and for each of them the interactions of the members with the whole system are shown.

5.2.1. Registration phase

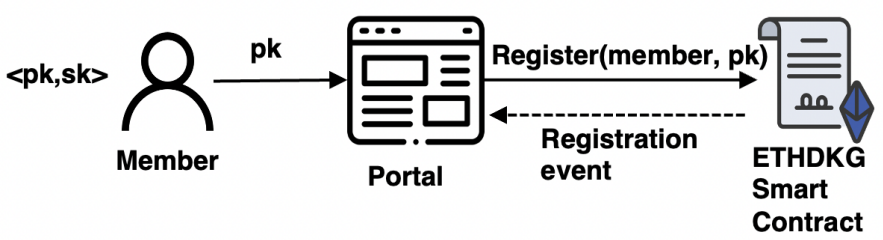

Figure 2: Registration phase: high-level interaction diagram.

Before the start of the protocol, each member of the committee is expected to register by communicating their public key which will be used for the calculation of the shared keys. Furthermore, this procedure allows to implement a dynamic participation strategy (thanks to the use of the smart contract): only those who have completed the registration both on time and successfully can take part in the subsequent phases.

Upon authentication of a committee member, as it can be seen from Figure 2, the latter accesses the portal and registers through a special section. This implies that the cryptographic key pair is generated and the transaction is submitted to the smart contract address on the blockchain. If the registration phase is still open and if the various checks provided are passed, then the user is registered on the blockchain with the corresponding public key associated with their address and the *Registration(member_address, member_pk)* event is emitted, which can be handled on the client side.

5.2.2. Secret generation and exchange phase

The first phase of the protocol is the secret generation and exchange phase, in which each member generates their starting secret and divides it into n fragments to be shared with all the others. Also in this case, the member of the committee accesses the portal and by clicking on the *"Generate and Exchange"* button the interaction with the system starts. First of all, on the client side, it is checked that the user can actually take part in the protocol and that a registration event associated with their address has therefore been generated. Subsequently, if the member has not already completed this phase, the secret is generated. The latter is appropriately stored in the member's device and is then divided into parts, whose commitments are also calculated. After the generation and splitting of the secret has taken place, we move on to the transmission phase of the various fragments obtained.

The exchange takes place through the communication channel, therefore the blockchain. To do this, the various fragments must be encrypted. In particular, this is done through a

properly computed shared key. In this regard, for each member of the committee it is checked whether the latter has completed the registration phase, consequently the associated public key is taken and the key k_{ij} is calculated between the user who generated the fragments and all the others. The shares are sequentially encrypted and the transaction is submitted via the *distribute_shares* function defined in the contract with the carrier of encrypted shares and the related commitments (see Figure 3).
Finally, if the contract is in the distribution phase, after the appropriate checks, the event *ShareDistribution(member_address, enc_shares, commitments)* associated with the distribution of the encrypted fragments and commitments is issued by a specific member of the commission.

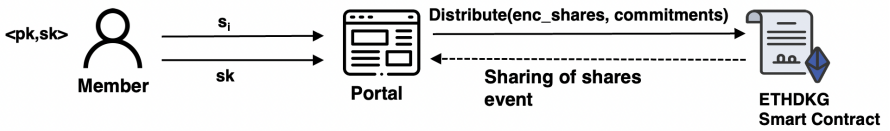

Figure 3: Generate and exchange phase: high-level interaction diagram.

5.2.3. Dispute phase

A dispute against the i^{th} participant in the protocol is issued when the j^{th} participant realizes that they have received an invalid fragment. Through the verification section provided within the portal, the user is first of all required to upload the keys shared with the other members in order to decrypt the fragments received. Specifically, for each member the transmission of the *ShareDistribution* event is checked, in order to assess whether or not they have already shared their own fragments. If this last condition is satisfied, then we pass to the verification of the j^{th} share, which is first decrypted with the corresponding key k_{ij} and then checked for validity. If the verification is successful and there are no errors, no dispute is submitted. Otherwise, the correctness proof of the k_{ij} key used to decrypt is calculated using the mandated "dleq-verify" [2] procedure and, subsequently, a dispute is issued by the j^{th} member against the i^{th} participant.

On the blockchain side, after verifying that the dispute phase is open and after several checks, the validity of the k_{ij} key provided through the mentioned dleq-verify procedure is first checked. If the key provided is incorrect, then the dispute is aborted; otherwise, the share is verified. If the latter is wrong, then the dispute for member i is submitted and the corresponding event *Dispute(member_i, member_j, k_{ij}, proof_k_{ij})* is raised, as shown in Figure 4.

5.2.4. Keys derivation phase

The output of the distributed key generation protocol is, precisely, the calculation of the *Master Public Key* shared among the parties. As anticipated before, the participants in this phase are those who have correctly shared their starting secret and against whom no dispute has been issued. So, after verifying that the member requesting the calculation of the keys is qualified, we proceed with the steps shown, at a high logical level, in Figure 5. Specifically, each participant $P \in Q$ is asked to upload their own starting secret in order to calculate the mpk according to

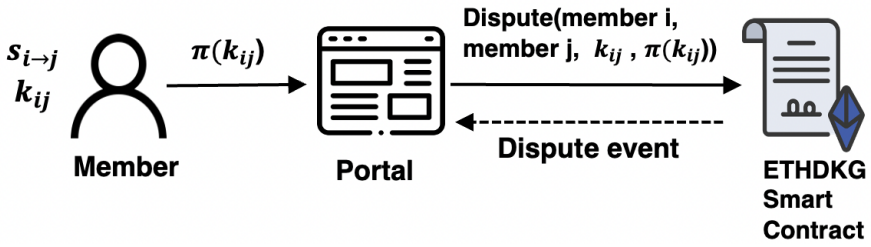

Figure 4: Dispute phase: high-level interaction diagram.

the equation mandated in [2]. In fact, for each admitted participant, the web portal calculates the partial public key, evaluating the quantity h^{s_i} and the relative validity test using the already mentioned DLEQ procedure.

Subsequently, the computed partial key is submitted to the smart contract, which carries out appropriate checks, verifying that the member in question has not already submitted their part and that the shared key is correct. In this case, the *KeyShareSubmission* event for the member, the related partial key, and proof of validity are raised.

Finally, each participant admitted to this phase, by checking the *KeyShareSubmission* events of all the other participants, can fetch the single shared h^{s_i} values, calculate the mpk and submit it to the smart contract. The latter can recalculate it, verify its validity and store it.

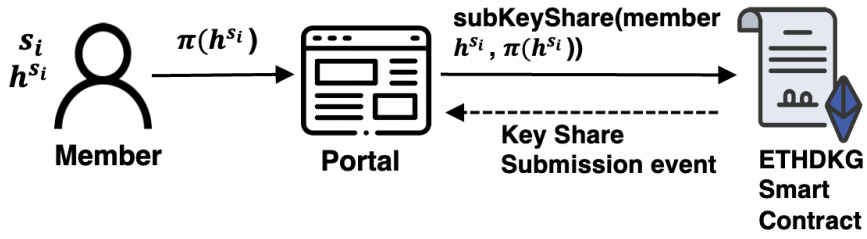

Figure 5: Keys derivation phase: high-level interaction diagram.

As an alternative to the above described procedure, the calculation of the *master secret key* can take place *off-chain*, after the closing of the voting phase, carrying out appropriate checks on the client side and making sure to retrieve the starting secrets of all qualified members.

6. Implementation

Essentially two phases were followed for the implementation of the whole system. In the first phase, all the functions useful for the computations to be carried out on the client side and therefore on the Chirotonia web portal were implemented. In a second phase, the smart contract made publicly available [10] by the authors of the protocol proposed in [2] was used to integrate the ETHDKG protocol into the platform using the Ethereum blockchain as a communication channel.

6.1. Client side

The integration of the distributed key generation protocol requires, on the client side, that any member of the committee, using the Chirotonia web portal, is able to complete the various phases we described above.

In this regard, various functions have been created using the *JavaScript* programming language in order to carry out the preliminary operations for submitting a transaction to the address where the smart contract is deployed. It is possible to deep-dive into the implementation details of the aforementioned functions by checking the ad hoc created Github [11] repository.

6.1.1. Registration phase

During the registration phase it is the participating member who initiates the interaction with the web portal, which in turn proceeds with the generation of the public and private key pair. For this reason the *registration()* function has been created, in which the private key is a random integer generated through the use of the JavaScript *crypto* library. The public key is obtained by multiplying the corresponding private key by the generator of the elliptic curve *alt_bn-128*, thus obtaining a point of the latter. Operations on elliptic curves have been managed through the *elliptic* JavaScript library. Registration is then done by sending a transaction to the contract, via MetaMask, through the standard interface provided by the *Web3* API.

6.1.2. Sharing phase

The sharing phase includes three sub-phases: the secret generation phase, the fragment generation phase and the sharing phase. After checking that the participating member has previously registered and has not already shared their fragments, the initial secret (s_i) is created through the *initial_random_secret ()* function. The latter through the *crypto* library generates a random integer. The secret is then stored in the browser and there is also the possibility of downloading it to store it on one's own device, so that it can be recovered for the subsequent phases. Once this is done, through the function *share_secret(s_i, n, threshold)*, the coefficients of the polynomial $f(x)$ [2] are randomly generated and used for the calculation of the different fragments. Finally, the commitments, necessary to verify the validity of the individual fragments, are also computed.

Once the fragments generation phase is over, fragments must be shared with the other participants. First of all, by checking the event logs generated by the contract, through the *Web3* interface, it is possible to verify that the other members have completed the registration phase and consequently take the corresponding public key to calculate the shared key between any two participants. The shared key is computed using the *sharedKey(sk_i, pk_j)* function, according to the equation in [2], thus also resulting in a point of the considered elliptic curve. Finally, before submitting the transaction to the contract for the distribution of the shares, the latter are encrypted using the *encrypt_share($shares_i$)* function, which performs an XOR operation between the fragment itself and a hash calculated using the hash function *SHA-3* provided by Web3. The latter corresponds to the hash function *keccak-256* available in *Solidity* for use within the contract.

6.1.3. Dispute phase

During the dispute phase, a member who realizes that they have received a wrong fragment can issue a dispute against the sender, providing both the shared key used to decrypt it, and proof of the key's validity.

The verification of the fragment is carried out with the *verify_share(x, share, commitments)* function, which, based on the parameters received in input and therefore the fragment, the commitments and the index used to obtain that fragment, checks its validity. Finally, the *dleq* procedure has been implemented to calculate the correct proof of the shared key between the two members, always making use of operations on elliptic curves and leveraging the crypto library as well. After having made the necessary preliminary operations, the portal, through MetaMask, submits the dispute to the smart contract.

6.1.4. Keys derivation phase

To obtain the protocol output and therefore the master public key, each member is required to upload their starting secret s_i in order to calculate the partial public key to be submitted to the blockchain, so that the mpk can be derived. The partial public key is computed using the *partialPublicKey(s_i)* function, which takes the starting secret s_i as input parameter and multiplies it by the h generator of the considered elliptic curve *alt_bn-128*, hence obtaining the value h^{s_i}. Of the latter, the proof of correctness $\pi(h^{s_i})$ is evaluated using the *dleq* procedure. Finally, each participating member receives all the fragments shared by the other participants through the portal (which listens, via the interface provided by Web3, to the events generated for the submission of the individual partial keys) and can eventually compute the master public key. For this final phase, we have envisaged two alternative approaches:

1. Introduce a database into the architecture and store all of the starting secrets inside it (once the voting is over), so that each member can verify that all qualified participants have shared their secret;
2. Implement a function within the smart contract that allows qualified members to upload their own starting secret and generate a *SecretSubmission* event after submission. This makes everyone able to compute the master secret key as in point 1 above, by catching the events generated and associated with actions undertaken by the other participants.

6.2. Blockchain side

Inside the ETHDKG [10] smart contract, two constants are first defined:

- the number of blocks to wait for in order to ensure that a transaction with adequate fees is included in a block;
- the number of confirmations to wait for in order to ensure that a transaction cannot be revoked.

Further events are defined, which, together with the previously mentioned constants, are leveraged to separate the different phases of the protocol and respect the assumption of synchronicity among the parts.

The events defined are the following:

1. *Registration:* the generation of this event indicates that a member has successfully completed the registration phase, sharing their public key. The event shows the correspondence between the address of the sender (who carried out the transaction) and the shared public key;
2. *Share Distribution:* the generation of this event indicates that a member has correctly shared the fragments of their secret and the related commitments, thus making them available to the other participants in the protocol;
3. *Dispute:* the generation of this event implies that a dispute has been enacted against a participating member, who is then de facto excluded from the subsequent phases of the protocol. The event indicates the issuer of the dispute, the accused, the incorrect fragment and the shared key with the relevant proof of validity for verifying the fragment;
4. *Key Share Submission:* the generation of this event indicates the fact that a participating member, who has not been excluded from the key derivation phase, has correctly shared their fragment, which can thus be used for calculating the mpk.

Events are managed and controlled on the client side through the communication interface provided by Web3 for MetaMask, subsequently carrying out the appropriate operations to complete each single step of the process. Within the smart contract, all the functions necessary to manage the various phases of the protocol (registration, sharing of the secret, dispute, derivation of keys) are defined, carrying out the appropriate validity checks before issuing the corresponding completion event.

7. Conclusions

In this paper we have presented an approach for improving the confidentiality level of Chirotonia, a secure and scalable e-voting framework based on blockchain and linkable ring signatures. In particular, we have designed, implemented and integrated inside Chirotonia a mechanism that allows to manage, in a fully distributed fashion, the generation of the cryptographic material necessary to both encrypt and decrypt ballots during and at the end of the voting process.

This has been achieved through the integration of a protocol for the distributed generation of cryptographic keys within the already available Chirotonia web administration portal. The proposed approach is based on the use of the Ethereum Distributed Key Generation protocol, deployed as a smart contract. Interaction between the client-side web application and the Ethereum blockchain is instead implemented through the use of the standard Web3 application programming interface.

The rationale behind our contribution has been discussed in depth and a few implementation hints have been provided.

References

[1] A. Russo, A. F. Anta, M. I. G. Vasco, S. P. Romano, Chirotonia: A scalable and secure e-voting framework based on blockchains and linkable ring signatures, in: 2021 IEEE International Conference on Blockchain (Blockchain), 2021, pp. 417–424. doi:10.1109/Blockchain53845.2021.00065.

[2] P. Schindler, A. Judmayer, N. Stifter, E. Weippl, ETHDKG: Distributed Key Generation with Ethereum Smart Contracts, Cryptology ePrint Archive, Report 2019/985, 2019. https://ia.cr/2019/985.

[3] A. Shamir, How to Share a Secret 22 (1979) 612–613. URL: https://doi.org/10.1145/359168.359176. doi:10.1145/359168.359176.

[4] T. P. Pedersen, A Threshold Cryptosystem without a Trusted Party, in: D. W. Davies (Ed.), Advances in Cryptology — EUROCRYPT '91, Springer Berlin Heidelberg, Berlin, Heidelberg, 1991, pp. 522–526.

[5] R. Gennaro, S. Jarecki, H. Krawczyk, T. Rabin, Secure Distributed Key Generation for Discrete-Log Based Cryptosystems, Journal of Cryptology 20 (2007) 51–83. doi:10.1007/s00145-006-0347-3.

[6] W. Neji, K. Blibech, N. Ben Rajeb, Distributed key generation protocol with a new complaint management strategy, Security and Communication Networks 9 (2016) 4585–4595. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/sec.1651. doi:https://doi.org/10.1002/sec.1651. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/sec.1651.

[7] G. Wood, ETHEREUM: A Secure Decentralised Generalised Transaction Ledge, https://ethereum.github.io/yellowpaper/paper.pdf, BERLIN VERSION b8ffc51 – 2022-02-2.

[8] V. Buterin, A Next Generation Smart Contract and Decentralized Application Platform, 2014.

[9] MetaMask, https://metamask.io/, ????

[10] P. Schindler, A. Judmayer, N. Stifter, E. Weippl, ETHDKG Smart Contract, https://github.com/PhilippSchindler/EthDKG/blob/master/contracts/ETHDKG.sol, 2019.

[11] A. M. Calabria, A. Piscitelli, Repository DKG functions, https://github.com/alfonsomcalabria/DistributedKeyGeneration, 2021.