KANTPOLL E-VOTING PROTOCOL

A PREPRINT

Thiago G. Quinália
Department of Mathematics
University of Brasília

Daniele N. Sobrinho
Department of Mathematics
University of Brasília

August 9, 2019

ABSTRACT

Write the abstract here

Keywords First keyword · Second keyword · More

Contents

1 Introduction

Kantpoll is a voter-verifiable e-voting protocol that allows elections to be held respecting the secrecy of votes. This Protocol, along with the Onion Service Protocol[**?**], ensures that votes cannot be traced back to voters, i.e., they remain secret. This is possible through the use of ring signatures, as will be explained in more detail further on.

Daniel Ziblatt and Steven Levitsky show that democratic institutions are not immune to corruption. They point out that a culture of respect and defense of democratic values is essential to prevent the weakening of the democratic process [**?**]. This being said, the central purpose of this Protocol is to offer a tool for governments and other institutions to reassure their commitment to voting secrecy and electoral transparency, two important principles of healthy democracies.

Electoral transparency here means that all votes are counted correctly and all votes counted come from registered voters. In order to achieve this, three types of verification are provided after the counting of votes:

1. A voter can verify that her vote was cast appropriately.
2. Anyone can check the validity of each vote signature.
3. Anyone with access to the *blockchain* of the campaign can verify if each vote is referencing the originally chosen candidate, i.e., if there was no vote manipulation.

This protocol does not meet the requirement of *receipt-freeness*, i.e., the absence of information (receipt) to prove to a coercer the way one has voted[1], for the voter holds the secret key used to generate votes and cannot be prevented from disclosing his keys. Despite this, the issue of voter coercion is partially addressed in the vote cancellation stage.

2 Kantpoll Protocol: high level specification

2.1 Entities

- **Certificate Authority** is the web service[1] responsible to authenticate users. Ideally, it should guarantee that:
  1. only one user controls an identity; and
  2. one user controls only one identity.
- **User** is any person who interacts with a campaign, within which she can play two roles, voter and campaign creator.
- **Voter** is the user who can cast votes and be part of groups. These actions are only possible after joining a campaign.
- **Campaign creator** is the user who create and manage campaigns.
- **Group** is a fixed amount of voters who cast unidentifiable votes. Or rather, votes are cast on behalf of the group.
- **Group chairperson** is a representative of the campaign creator that provides voters with information about the campaign, and receives and stores votes.
- **Campaign** is a smart contract[2] that defines a set of rules and procedures for the selection of candidates.

[1]A Web service is a software service used to communicate between two devices on a network. - https://www.techopedia.com/definition/25301/web-service

[2]A smart contract is a computer code running on top of a blockchain containing a set of rules under which the parties to that smart contract agree to interact with each other. - https://blockchainhub.net/smart-contracts/

- **Observer** is an independent *Onion service*[3] that provides voters with access to the blockchain.
- **Server** is an onion (and local) service that works as an interface between users and the campaign.

2.2 Phases

1. Create a campaign
2. Join a campaign
3. Send a vote

3 Phase 0. Vault Creation

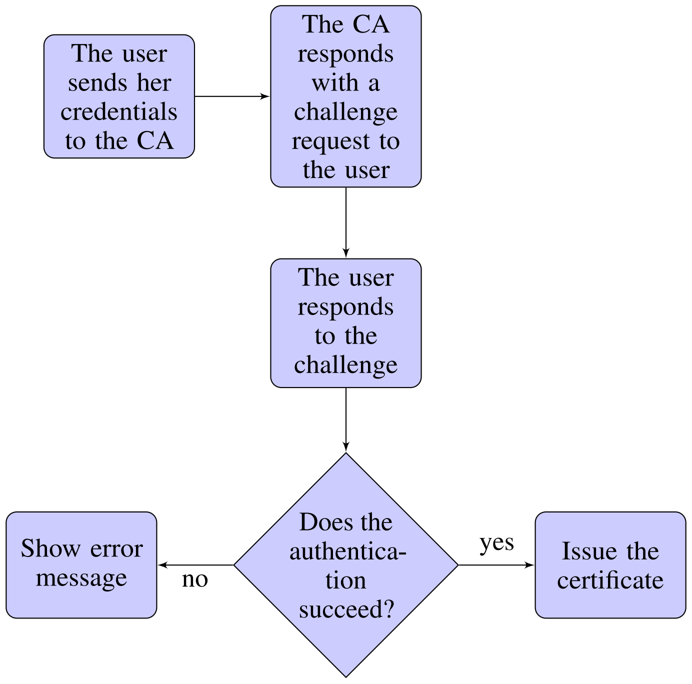
A vault consists in a computer file containing information needed to interact with campaigns, such as information to validate a password and generate cryptographic key pairs. Vaults may also contain a certificate issued by a Certificate Authority (CA). The purpose of this certificate is to assure campaign creators that an *ECDSA address*[4] belongs to a particular voter.

Users need to have a vault before creating or joining a campaign, since there is no web service where users' private keys are stored. Users only rely on vaults for this.

Users can create a vault using a static web page (the code is available on *Github* **??**). On this page, they can choose to get a certificate and add it to the vault.

3.1 Certificate Creation

The certificate creation process begins with users sending credentials, an ECDSA address and an identification, to the Certificate Authority. The Certificate Authority then authenticates each user using some authentication method, such as the SMS One Time Password (OTP). If the authentication is successful, a certificate is issued.


The user sends her credentials to the CA
The CA responds with a challenge request to the user
The user responds to the challenge
Does the authentication succeed?
no
Show error message
yes
Issue the certificate

[3]Onion services anonymous network services that are exposed over the Tor network. In contrast to conventional Internet services, onion services are private, generally not indexed by search engines, and use self-certifying domain names that are long and difficult for humans to read. - https://nymity.ch/onion-services/pdf/sec18-onion-services.pdf

[4]ECDSA address...

3.2 Vault composition

Vaults are intended to increase user privacy, since they allow users to keep their private keys with them, while providing a two-factor mechanism for users to generate key pairs. These two factors comprise information stored in the vault (a mnemonic sentence) and information kept in voters' brains (the user name and a password). Besides the mnemonic sentence, a vault also contains:

- A validation hash code to inform the user if his password is valid.
- A certificate (optional)
- A RSA key pair (only used by campaign creators to receive voter requests)

$$vault(x) = \langle mnemonicsentence(x), certificate(x), hash(passwordvalidation(x)), RSAkeys(x)\rangle \quad (1)$$

3.2.1 Mnemonic sentence

According to the Bitcoin Improvement Standard 39 (BIP39) **??**, a mnemonic sentence is a group of easy to remember words for the generation of deterministic wallets. This is considered a more human-friendly approach to generate these wallets, which are, in turn, defined as systems of deriving keys from a single starting point known as a seed. **??**

A vault contains a mnemonic sentence (mnsentence) made up of 12 words (from a list of 2048 words), what provides 128 bits of entropy **??**. In order to generate the key pairs, we use the Javascript library Ethers.js **??**, version 3.0.25. The function called is `ethers.HDNode.mnemonicToSeed`, which we sugar to `mnemonicToSeed`, and the parameter is a string consisting of the 12 words, the user name and a password, separated by spaces.

$$seed(x) = \texttt{mnemonicToSeed}(mnsentence(x), username(x), password(x)) \quad (2)$$

3.2.2 Validation hash code

The validation hash code verifies if the user entered the correct password. The hash algorithm used is the Keccak-224[**?**], and the parameter is a string consisting of the JSON of the key corresponding to the user's ECDSA address, and the user name.

$$hashcode(x) = \langle mainkey(x), username(x)\rangle \quad (3)$$

3.2.3 Certificate (optional)

The certificate used to authenticate users was loosely based on the X.509 standard, version 3 **??**. It is composed of:

User certificate		
Key	Description	Example value
version_number	The version of the certificate	1
signature_algorithm_id	The algorithm used to sign the certificate	id-ecPublicKey
issuer	The Certificate Authority	C=BR, O=Kantpoll Project
issuer_address	An ECDSA address representing the Certificate Authority	0x10dfE5c1DE2e276...
validity	A JSON object composed of not_before and not_after	
not_before	The moment representing the beginning of the validity in Unix time	1552999153193
not_after	The moment representing the ending of the validity in Unix time (0 = forever)	0
subject	The user identification	55-61912341234
subject_public_key_info	A JSON object composed of public_key_algorithm, address and certificate_signature	
public_key_algorithm	The algorithm which the user uses to sign her transactions	id-ecPublicKey
address	An ECDSA address representing the user	0x5Afb09b4811179100...
certificate_signature	The signature to validate the certificate	0x797d0c967d9745de...

The certificate signature is generated by signing a message composed of the user's identification and ECDSA address, and the not_before timestamp, separated with commas, using the issuer ECDSA private key.

$$certificate\_signature(user, certificate\_authority) = \langle subject, address, not\_before, issuer\_private\_key\rangle \tag{4}$$

3.2.4 RSA key pair

The RSA key pair is employed by users to protect the contents of their requests to the server. The reason for using it is to secure localhost traffic, more specifically, the communication between campaign creators and the server, on the port 1985, and between voters and the Orbot, a free Android proxy application **??**, on the port 9050.

In addition to the option above, i.e., to interact with a campaign through an Android application, voters have two other options. They can access the static web page using the Tor browser **??**, which sends requests over the Tor network, or they can access the static web page using a common web browser (by common web browser we mean a browser with no access to the Tor network) and a Tor2web proxy. In the latter case, the RSA key pair can also be useful to hide the contents of a request from the Tor2web proxy provider.

This key pair is not derived from the seed (2). Instead, it is generated with the generateKey method of the SubtleCrypto interface **??**. It is stored in the vault in order to be portable to the user. Below, the algorithm-specific parameters applied to generate the key pair:

```
const ENCRYPT_ALGORITHM_RSA = {
    name: "RSA-OAEP",
    modulusLength: 1024,
    publicExponent: new Uint8Array([1, 0, 1]),
    extractable: false,
    hash: {
        name: "SHA-256"
    }
};
```

3.3 Key pairs derived from the seed

The seed (2) originates three ECDSA key pairs:

1. The *participant key pair*, used to join campaigns and groups, and to manage campaigns.
2. The *voter key pair*, used to send votes.
3. The *pre-voter key pair*, used to send pre-votes and cancellations.

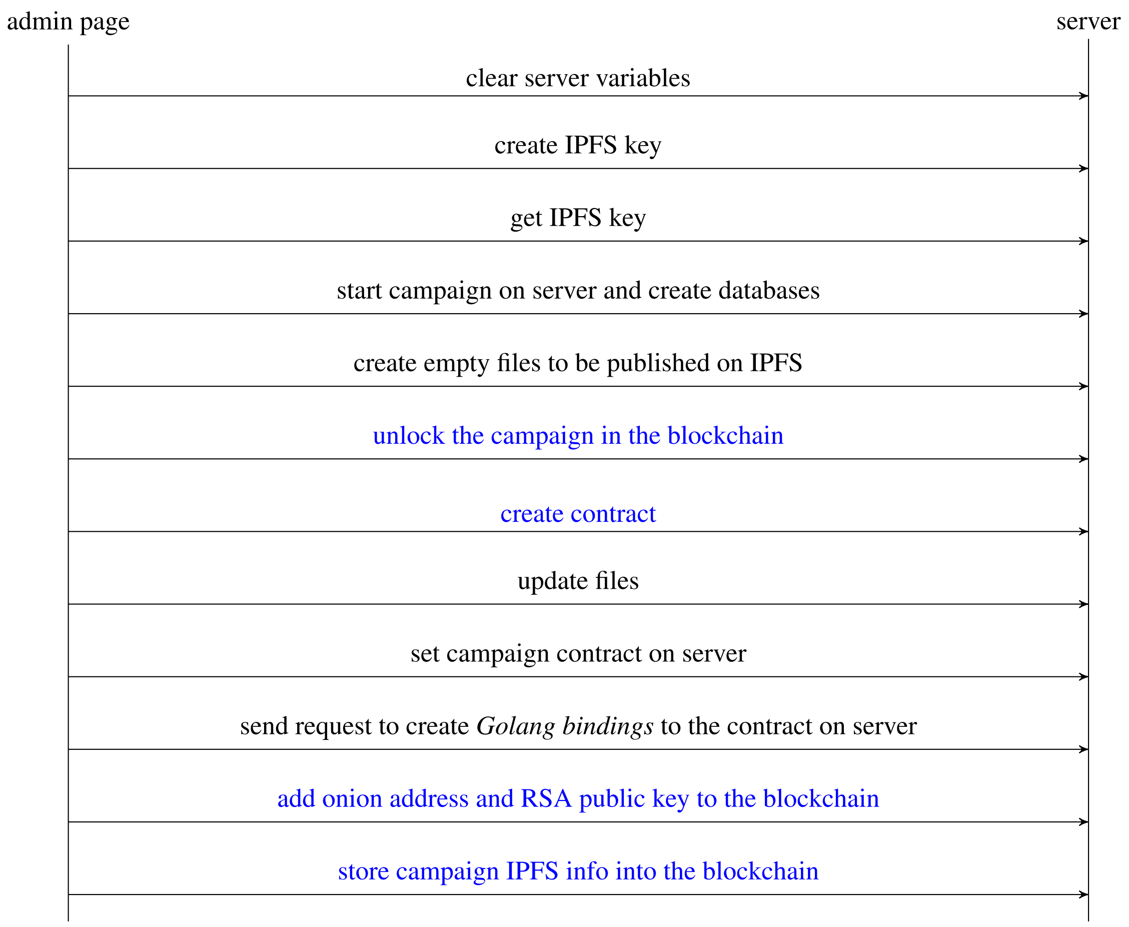
It is extremely important that the voter key pair be different from the pre-voter key pair. In addition, these keys should not be guessed from each other, at the risk of revealing the voter's choice.

4 Phase 1: Creating a campaign

The schema below shows the requests made by the static page (admin page), used by the campaign creator, to the server. These requests can be sent directly on the port 1985 or via a web3 HTTP provider on the port 8545. In this case, the campaign creator modifies the state of the blockchain by sending a transaction signed with her *participant private key*.

It should be noted that all but the first request between the admin page and the server, on the port 1985, are encrypted with an cipher text, using the AES algorithm. The first request (queryNewSession) informs the server of the campaign creator's *RSA public key*, to which the server responds with a cipher. All subsequent requests received by the server are decrypted with this cipher until the server is restarted. If a malicious software sends the queryNewSession request to the server before the campaign creator, it would be readily noted by the campaign creator, because none of her interactions with the server would succeed.

Requests from the admin page to the server (in blue are the requests sent via Web3 HTTP provider):


admin page
server
clear server variables
create IPFS key
get IPFS key
start campaign on server and create databases
create empty files to be published on IPFS
unlock the campaign in the blockchain
create contract
update files
set campaign contract on server
send request to create Golang bindings to the contract on server
add onion address and RSA public key to the blockchain
store campaign IPFS info into the blockchain

4.1 Ways to interact with the blockchain

In this protocol, the two most important ways to interact with the blockchain are:

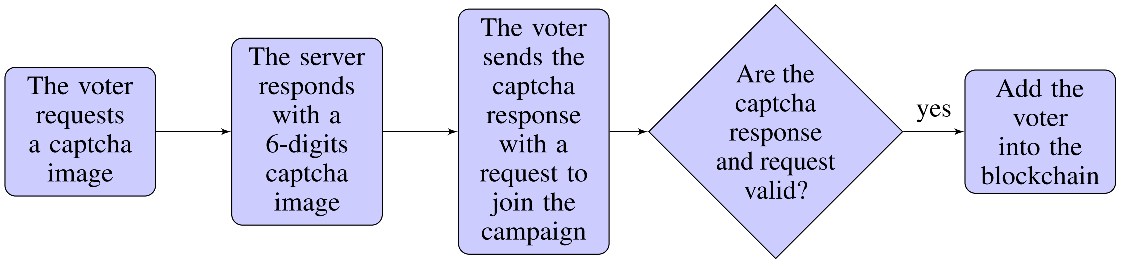
- Via the web3.js Javascript API. It is used mainly by the campaign creator so that she can manage the campaign. This library allows the campaign creator to connect to a local Ethereum node using an HTTP connection **??**.
- Via *Golang bindings*. This method is used by voters, who interact with the blockchain indirectly, through the server. *Golang bindings* are automatically generated Golang files that allows a Golang software to interact with an Ethereum smart contract **??**.

5 Phase 2: Joining a campaign

First of all, the voter's static web page checks if she meets the requirement to join a campaign. It basically means that her user name is validated with a regular expression defined by the campaign creator during the process of creating the campaign. This validation is performed in the static web page and consists in verifying if the user name starts with a predefined prefix, for example, a telephone dialing prefix, or if it ends with a predefined suffix, for instance, an email domain. [5]

After the aforementioned validation, the static web page starts communicating with the server to register the voter in the blockchain, as illustrated below:

[5]Notice that it is possible that the campaign creator herself adds each voter directly into the blockchain without validating his or her user name. In this case, the voter does not need to send a request to join the campaign.


The voter requests a captcha image
The server responds with a 6-digits captcha image
The voter sends the captcha response with a request to join the campaign
Are the captcha response and request valid?
yes
Add the voter into the blockchain

5.1 Request composition

Below is the type (in Golang) of a message to register a voter.

```go
type RegisterVoterMessage struct {
  User                 *string `json:"user"`
  Address              *string `json:"address"`
  Pubkey               *string `json:"pubkey"`
  Signature            *string `json:"signature"`
  NotBefore            *string `json:"not_before"`
  CertificateSignature *string `json:"certificate_signature"`
}
```

5.2 Request validation

The server performs four checks on a request to join a campaign:

- Verify that voter has signed a message containing the *campaign id* with her *participant private key*.
- Verify that the certificate is valid, validating a message composed of user, *participant address* and *not_before* using the certificate authority address and the certificate signature.
- Verify that the voter meets the user name requirement.
- Verify that the voter was not yet registered in the blockchain.

Voter user names are stored in the blockchain in order to prevent a voter from having multiple *participant and voter addresses*. These names are not stored in clear text to safeguard their identities. Instead, we store a hashcode (sha256) generated from a predefined prefix plus the user name of the voter.

5.3 Giving the right to vote

After all the above verifications, the user is given the right to vote. In order to do this, the server calls the *giveRightToVote* function of the campaign (remember that the campaign is a smart contract), passing as parameters a *participant address*, a *voter public key* (split into two variables, *prefix* and *pubkey*, due to length constraints of variables in the Solidity programming language), and a hashcode representing a voter. This function can only be called by the campaign creator. Only he or she can give the right to vote to some user.

6 Phase 3: Joining a group

All votes are signed with an Unique Ring Signature (URS)**??** before they are cast. In order for the voter to know which public keys she should use to compose her signature, she must be part of a group of voters beforehand. Being part of a group of voters allows her to cast a vote on behalf of a group, without revealing her identity.

Groups have an attribute called category that is meant to provide candidates with relevant information about their voters. For example, there can be groups of women, men and agender people. After the counting of votes, candidates may be able to tell how many votes they received from each division of the electorate.

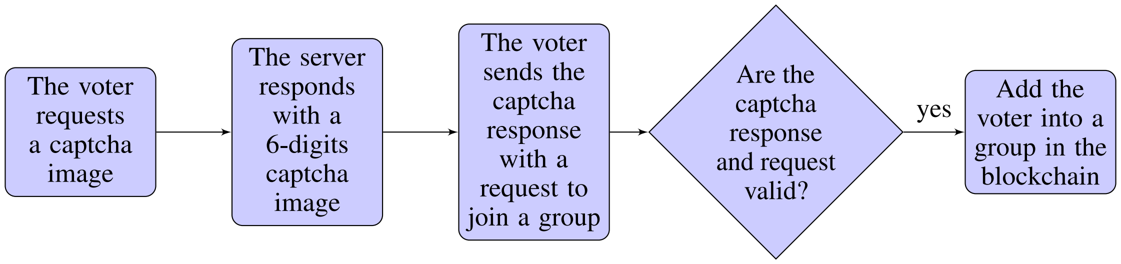
The server provides voters with information they need to interact with a campaign. For instance, voters need to know the categories of the groups in some campaign to choose the one they most identify with. At the end of this section, we will explain in detail how this type of information is provided to users.

6.1 The request

Below is the type (in Golang) of a message to add a voter into a group.

```go
type EnterGroupMessage struct {
  Address   *string `json:"address"`
  Category  *int    `json:"category"`
  Signature *string `json:"signature"`
}
```

The request to join a group follows the same path as the request to join a campaign.


The voter requests a captcha image
The server responds with a 6-digits captcha image
The voter sends the captcha response with a request to join a group
Are the captcha response and request valid?
yes
Add the voter into a group in the blockchain

6.2 Request validation

The process of validating a voter before inserting her into a group is simpler than that used to register a voter. The server simply checks if she has signed a standard message with her *participant private key* (whose *address* was previously registered in the blockchain). Then, the server finds a group among those of the chosen category and adds the voter into it. If she was already part of group, the smart contract of the campaign will block the attempt.

6.3 Information provided by the server

After connecting to a campaign, the static web page periodically sends requests to the server in order to obtain four types of information, which are:

- The index of the group the voter is part of. The voter must inform her *ECDSA participant address* along with the request. If the voter has not joined the campaign, the server returns "-1". If she has joined the campaign, but she has not joined a group, the server returns "-2".
- Information about a requested group (below is the type in Golang):

```go
type GroupInfo struct {
  Index          int            `json:"index"`
  Chairperson    string         `json:"chairperson"`
  ChairpersonTor ChairpersonTor `json:"chairperson_tor"`
  Category       string         `json:"category"`
  Size           int            `json:"size"`
  Voters         []string       `json:"voters"`
  Pubkeys        PublicKeys     `json:"pubkeys"`
  Ballots        []GroupBallot  `json:"ballots"`
  Signature      string         `json:"signature"`
}
```

- Information about a campaign that is only stored in the blockchain (type 1). This information corresponds to variables of the campaign (smart contract), which are relevant only during the voting process. See below the type in Golang:

```go
type CampaignInfo struct {
  Chairperson                       string        `json:"chairperson"`
  Mgz                               int           `json:"mgz"`
  Rounds                            int           `json:"rounds"`
  CurrentBallot                     int           `json:"current_ballot"`
  CurrentMessage                    string        `json:"current_m..."`
  HowManyBallots                    int           `json:"how_many_b..."`
  HowManyGroups                      int             `json:"how_many_g..."`
  CanCancel                          bool            `json:"can_cancel"`
  StoppingAccessionToGroups          string          `json:"stopping_a..."`
  DisableCandidateLink               bool            `json:"disable_c..."`
  ChairpersonTor                     ChairpersonTor  `json:"chairperson..."`
  Ballots                            []Ballot        `json:"ballots"`
  VotesPerBallotCandidateCategory    [][]string      `json:"votes_p..."`
  Signature                          string          `json:"signature"`
}
```

- Information about the campaign that is stored in the blockchain and is also available on an IPFS web site (type 2). Here, there is data that is complementary to the voting process. For example, the description of the campaign and the party of each candidate. This information is rarely modified and is intended to be preserved for posterity. Below is a JSON containing sample data from a campaign:

```
{"profile":{
  "id":"15572483971",
  "address":"0xF58165984F8A7785Dd372Ab9E4720A250A0dA133",
  "name":"campaign 1",
  "description":"A campaign",
  "image":"",
  "link":"",
  "country":"AD",
  "certificate_authority":"0x10dfE5c1DE2e2767Bf29C8fB69bF9De0F2827f9E",
  "period":"08/05/2019 - 29/05/2019",
  "rounds":"5",
  "onion":"jmg763pxnzqw7vtl.onion",
  "mgz":"3",
  "regexp":"$",
  "ipns":"QmPG7z1vfYiqVCn5Ea25QaZxU9DP99G45WsqtNrwCzS5Si"},
  "parties":{
    "party0":{
      "url":"https://www.party0.com",
      "photo":"https://kantpoll.com/imgs/no-image-icon.png"
    },
    "party1":{
      "url":"https://www.party1.com",
      "photo":"https://kantpoll.com/imgs/no-image-icon.png"
    },
  },
  "gcategories":{
    "ME":{
      "content":"Men"
    },
    "WO":{
      "content":"Women"
    }
  },
  "candidates0":{
    "ballot":"0",
    "data":{
      "https://www.yussif.com":{
        "name":"Yussif",
        "party":"party0",
        "photo":"https://kantpoll.com/imgs/no-image-icon.png"
      },
      "https://www.natasha.com":{
        "name":"Natasha",
        "party":"party1","photo":"https://kantpoll.com/imgs/
            no-image-icon.png"
      }
  "signature":"0
      x2120174a7d6fd368d087ce5e9b4c3ddc61fd1037eb24ff144421b132155b88..."
}
```

These requests are encrypted using a cipher and the AES algorithm. This cipher is transmitted to the server encrypted with the *RSA public key* of the campaign creator.

The index of the first item is encrypted with the same cipher of the request, and then it is sent to the voter on the server response. The information of the other items are not encrypted, since they do not contain personal data.

Note that the information of the last three items contain a signature. This signature allows voters to confirm the origin of each response.

The last item, the information about the campaign (type 2), is signed by the campaign creator. This signature may be verified using her *ECDSA participant address*, which can be found in the *link of the campaign*.

6.4 Chain of trust

The trust anchor of a campaign is its *link*. See below an example of it:

```
https://kantpoll.com/home?ipns=QmNXcBVYv6hgEtj1U5GMmCbkAE6gopZKjN2tXYHNk111Qz&
    onion=jmg763pxnzqw7vtl.onion&address=0
    xF58165984F8A7785Dd372Ab9E4720A250A0dA133
```

When a voter opens the link of a campaign (using the Tor browser, for example) the static web page tries to connect to a *Onion Service* through the informed onion address. It requests the two types of information about the campaign, i. e., the information stored only in the blockchain (type 1) and the information stored in the blockchain and possibly on an IPFS web site (type 2). If the voter has already joined the campaign or has already been registered by the campaign creator, it may also request information about her group.

All of these information are signed by the campaign creator, and the signatures may be verified using the *ECDSA address* present in the link of the campaign. See below the composition of this link:

- An IPFS address. This address is the identification and location of a campaign on the InterPlanetary File System network. There, anyone can find information about the candidates, parties, and group categories of a campaign. If the campaign creator wants, she can also publish all the received votes there for further scrutiny.
- An ECDSA address. With this address, any user can verify the authenticity of the information provided by the campaign creator.
- An Onion address. It represents the location of the campaign on the Tor network.

Besides that, the link of a campaign can contain:

- An Onion address for each *observer* of the campaign.
- The hash of the RSA public key of each *observer* of the campaign.

The importance of these observers will be explained later.

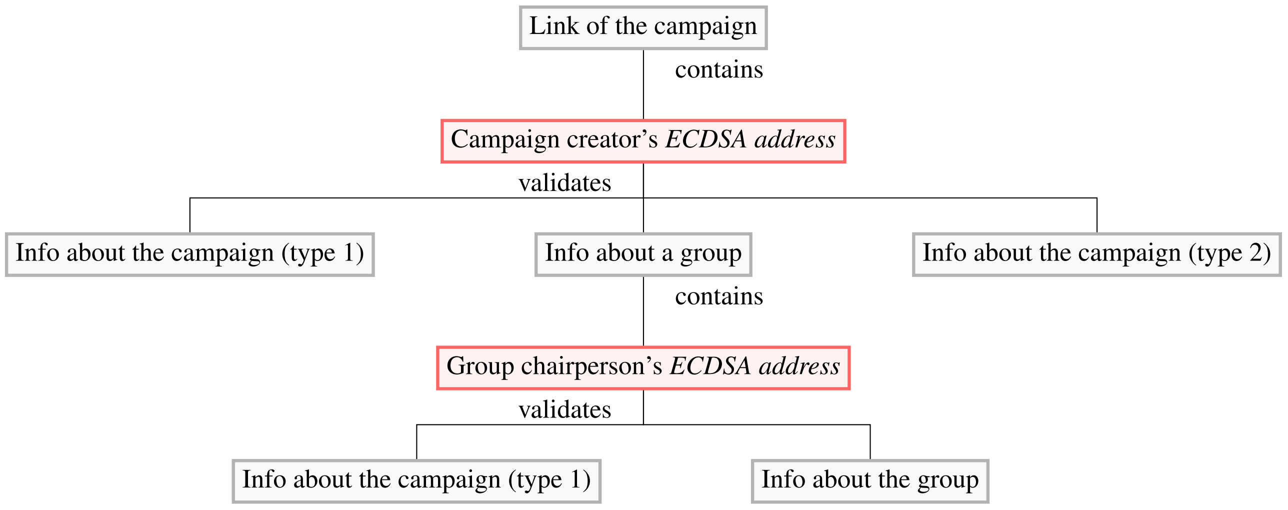
As well as the campaign, the groups have a chairperson. He is used to scale up campaigns by providing voters with the three types of information mentioned above. In doing so, he frees the campaign creator to receive requests from new voters only.

When the voter interacts with the campaign for the first time, she uses only one *ECDSA address* (that present in the link of the campaign) to verify the information she receives. After joining a group, she marks the group chairperson *ECDSA address* as trustworthy.

The group chairperson provides an *Onion Service* for the voters to request information and cast their votes. The Onion address that voters must use to connect to this Onion Service is provided, along with a *ECDSA address*, by the campaign creator in the information about a group (see the Golang type GroupInfo).

The campaign creator may also be a (or the only) group chairperson. This is more appropriate for small campaigns, when the number of voters connected at the same time is not a concern.

Graphic representation of a campaign's chain of trust:


Link of the campaign
contains
Campaign creator's *ECDSA address*
validates
Info about the campaign (type 1)
Info about a group
Info about the campaign (type 2)
contains
Group chairperson's *ECDSA address*
validates
Info about the campaign (type 1)
Info about the group

Note that the information about the campaign (type 2) is signed only by the campaign creator. She stores this information in the blockchain. After that, group chairpersons send this information unchanged to voters, i. e., with the same signature.

7 Phase 4. Vote

This step ensures the secrecy of the votes and the transparency of the results, but it guarantees neither coercion-resistance nor receipt-freeness. On the contrary, each voter receives a receipt of his vote and can verify if it was cast appropriately.

The campaign creator can not insert fraudulent votes into the election results records without being detected Groups of voters are composed of a fixed number of public keys. Only voters who possess the private key related to one of these public keys can sign a vote on behalf of their group.

Remember that the campaign creator is the responsible for giving the right to vote to voters, which basically means that she adds the *voter* public key of some user into a group of voters. Taking advantage of her power, a dishonest campaign creator can impersonate a legitimate voter and thus rig the election. We can mitigate this risk by releasing a list with the authorized users before the election (perhaps this may be done by an independent certificate authority).

8 Phase 5. Vote Cancellation (optional)

References

[1] Stéphanie Delaune, Steve Kremer, and Mark Ryan. Coercion-resistance and receipt-freeness in electronic voting. In *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006), 5-7 July 2006, Venice, Italy*, pages 28–42. IEEE Computer Society, 2006.