Polys online voting system

Whitepaper
Version 2.0

kaspersky BRING ON THE FUTURE

Learn more on
kaspersky.com

Abstract

Online voting is a trend that's gaining momentum in modern society. It has great potential to decrease organizational costs and increase voter turnout. It eliminates the need to print ballot papers or open polling stations – voters can cast their vote from wherever there is an internet connection. Despite these benefits, online voting solutions are viewed with a great deal of caution because they introduce new threats, and a single vulnerability can lead to large-scale manipulations of votes.

The concept of online voting has been studied for decades and many techniques have been developed that, to a certain extent, solve the problems of building online voting systems. However, most solutions offer a trade-off between the desired properties and the important task of building a system that works transparently for everyone, where every process can be audited and verified. Our voting system solution uses state-of-the-art techniques for achieving the desired properties and relies on blockchain technology for providing maximum verifiability for all system users.

Problems and solutions of building online voting systems

Whether we are talking about traditional paper-based voting, voting via digital voting machines or an online voting system, several conditions need to be satisfied:

Eligibility
Unreusability
Privacy
Fairness
Soundness
Completeness

- Eligibility: only legitimate voters should be able to take part in voting;
- Unreusability: each voter can vote only once;
- Privacy: no one except the voter can obtain information about the voter's choice;
- Fairness: no one can obtain intermediate voting results;
- Soundness: invalid ballots should be detected and not taken into account during tallying;
- Completeness: all valid ballots should be tallied correctly.

Below we give a brief overview of the solutions for satisfying these properties in online voting systems.

The solution to the issue of eligibility is rather obvious: in order to take part in online voting, voters need to identify themselves using a recognized identification system, and the identifiers of all legitimate voters need to be added to the list of participants. But there are threats: firstly, all modifications made to the participation list need to be checked so that no illegitimate voters can be added, and secondly, the identification system should be both trusted and secure, so that a voter's account cannot be stolen or used by an intruder. Building such an identification system is a complex task in itself. However, because this sort of system is necessary in a wide range of other contexts, especially related to digital government services, we believe it is best to use an existing identification system, and the question of creating one is beyond the scope of our work.

At first glance, implementing unreusability may seem straightforward – when a voter casts his/her vote, all that needs to be done is to place a mark in participation list, and don't allow him/her to vote a second time. But privacy needs to be taken into consideration, so providing both unreusability and voter anonymity is tricky. Moreover, it may be necessary to allow the voter to re-vote, and this makes the task even more complex. A brief overview of unreusability techniques will be provided below in conjunction with the overview on implementing privacy.

Blind signature is a method of signing data when the signer doesn't know what exactly he/she is signing.

Privacy in the context of online voting means that no one except the voter knows how a participant has voted. Achieving this property mostly relies on one (or more) of the following techniques: blind signatures, homomorphic encryption and mix-networks. Blind signature is a method of signing data when the signer doesn't know what exactly he/she is signing. This is achieved by using a blinding function so that blinding and signing functions are commutative – $Blind(Sign(message)) = Sign(Blind(message))$. The requester blinds (applies blinding function to) his/her message and sends it for signing. After obtaining a signature for a blinded message, he/she uses his/her knowledge of blinding parameters to derive a signature for an unblinded message. Blind signatures mathematically prevent anyone except the requester from linking a blinded message and a corresponding signature pair with an unblinded one.

Many online voting protocols have evolved from scheme proposed by Fujioka, Okamoto and Ohta in 1992.

The voting scheme proposed by Fujioka, Okamoto and Ohta in 1992, uses blind signature as follows: an eligible voter blinds his ballot and sends it to the validator. The validator verifies that the voter is allowed to take part in voting, signs the blinded ballot and returns it to the voter. The voter then derives a signature for the unblinded ballot and sends it to the tallier, and the tallier verifies the signature of the validator before accepting the ballot.

Many online voting protocols have evolved from this scheme, improving usability (in the original scheme the voter had to wait till the end of the election and send a ballot decryption key), allowing re-voting or implementing coercion resistance. The main threat here is the power of the signer: there must be a verifiable log of all emitted signatures; this information logically corresponds to the receiving of a ballot by the voter, so it should be verified that only eligible voters receive signatures from the signer. It should be also verifiable that accounts of voters who are permitted to vote but have not taken part in voting are not utilized by an intruder. To truly break the link between voter and ballot, the ballot along with the signature needs to be sent through an anonymous channel.

Homomorphic encryption is a form of encryption that allows mathematical operations to be performed on encrypted data without decryption.

Homomorphic encryption is a form of encryption that allows mathematical operations to be performed on encrypted data without decryption, for example, addition $Enc(a) + Enc(b) = Enc(a + b)$; or multiplication $Enc(a) * Enc(b) = Enc(a * b)$. In the context of online voting, additive homomorphic encryption allows us to calculate the sum of all the voters' choices before decryption.

It is worth mentioning here that multiplicative homomorphic encryption can generally be used as additive. For example, if we have choices x and y and multiplicative homomorphic encryption, we can select a value g and encrypt exponentiation: $Enc(g^x) * Enc(g^y) = Enc(g^{(x+y)})$.

Homomorphic encryption can be used to obtain various properties necessary in an online voting system; with regards to privacy, it is used so that only the sum of all the choices is decrypted, and never each voter's choice by itself. Using homomorphic encryption for privacy implies that decryption is performed by several authorities so that no one can obtain the decryption key; otherwise, privacy will be violated.

It is usually implemented with a threshold decryption scheme. For instance, let's say that we have n authorities. To decrypt a result we need t of them, $t <= n$. The protocol assumes that each authority applies its part of the key to the sum of the encrypted choices, and after t authorities perform this operation, we get the decrypted total sum of choices. In contrast to the blind signature scheme, no anonymous channel between voter and system is needed, but privacy relies on trust in the authorities: if a malicious agreement is reached, all voters can be deanonymized.

Mix-networks also rely on the distribution of trust, but in another way. The idea behind a mix-network is that voters' choices go through several mix-servers that shuffle them and perform an action – either decryption or re-encryption, depending on mix-network type. In a decryption mix-network each mixing server has its own key, and the voter encrypts his/her choice like an onion, so that each server will unwrap its own layer of decryption. In re-encryption mix-networks each mix server re-encrypts the voters' choices.

There are a lot of mix-network proposals, and reviewing all their properties is beyond the scope of this paper. The main point regarding privacy here is that, in theory, if at least one mix-server performs an honest shuffle, then privacy is preserved. This is a bit different from privacy based on homomorphic encryption, where we make assumptions about the number of malicious authorities. Also, the idea behind mix-networks can be used to build anonymous channels required by other techniques.

Fairness, in terms of no one being able to obtain intermediate results, is achieved in a straightforward way: voters encrypt their choices before sending, and those choices are decrypted at the end of the voting process. The important thing to remember here is that if someone owns a decryption key with access to encrypted choices, they can obtain intermediate results. This problem is solved by distributing the key among several key holders. A system where all the key holders are required for decryption is unreliable – if one of the key holders does not participate, decryption cannot be performed. Therefore, threshold schemes are used whereby a specific number of key holders are required to perform decryption. There are two main approaches for distributing the key: secret sharing, where a trusted dealer divides the generated key into parts and distributes them among key holders (e.g. Shamir's Secret Sharing protocol); and distributed key generation, where no trusted dealer is needed, and all parties contribute to the calculation of the key (for example, Pedersen's Distributed Key Generation protocol).

The zero-knowledge proof is a cryptographic method of proving a statement about the value without disclosing the value itself.

On the face of it, the completeness and soundness properties seem rather straightforward, but realizing them can be problematic depending on the protocol. If ballots are decrypted one by one, it is easy to distinguish between valid and invalid ones, but when it comes to homomorphic encryption things become more complicated. As a single ballot is never decrypted, the decryption result will not show if more than one option was chosen, or if the ballot was formed so that it was treated like 10 choices (or a million) at once. So, we need to prove that the encrypted data meets the properties of a valid ballot, without compromising any information that can help determine how the vote was cast. This task is solved by zero-knowledge proof. By definition this is a cryptographic method of proving a statement about the value without disclosing the value itself. More specifically, in such cases range proofs are used to prove that a certain value belongs to a particular set.

The properties described above are the bare minimum for any voting solution. But all the aforementioned technologies are useless if there is no trust in the system itself. To earn this trust, a voting system needs to be fully verifiable, i.e., everyone involved can ensure that the system complies with the stated properties. Ensuring verifiability can be split into two tasks: personal, when the voter can verify that his/her own ballot is correctly recorded and tallied; and universal, when everyone can verify that the system as a whole works correctly. This entails the inputs and outputs of the voting protocol stages being published along with proof of correct execution. For example, mix-networks rely on proof of correct shuffling (a type of zero-knowledge proof), while proof of correct decryption is also used in mix-networks and for threshold decryption. Obviously, the more processes that are open to public scrutiny, the more verifiable the system is. However, online voting makes extensive use of cryptography and the more complex the cryptography, the more obscure it is for the majority of system users. It may take a considerable amount of time to study the protocol, and even more to identify any vulnerabilities or backdoors. And even if the entire system is carefully studied, there is no guarantee that exactly the same code is used in real time.

The one of the desired properties of an online voting system is coercion resistance.

Last but not least are problems associated with coercion and vote buying. Online voting brings these problems to the next level – as ballots are cast remotely from an uncontrolled environment, coercers and vote buyers can operate on a large scale. That's why one of the desired properties of an online voting system is coercion resistance. It is called resistance because nothing can stop the coercer from standing behind the voter and controlling the voter's actions. The point here is to do as much as possible to lower the risk of mass interference. Both kinds of malefactors – coercers and vote buyers – demand proof of how a voter voted. That's why many types of coercion resistance voting schemes introduce the concept of receipt-freeness: the voter cannot create a receipt that proves how he/she voted. The approaches to implementing receipt-freeness generally rely on a trusted party – either a system or device that hides the unique parameters used to form a ballot from the voter, so the voter cannot prove that a particular ballot belongs to him/her. The reverse side of this approach is that if a voter claims that his/her ballot is recorded or tallied incorrectly, he/she simply cannot prove it due to a lack of evidence.

In our brief overview of technologies used to meet the necessary properties of online voting systems, we deliberately considered the properties separately: when it comes to assembling the whole protocol, most solutions introduce a trade-off. For example, as we noted for the blind signature, there is a risk that non-eligible voters will vote, receipt-freeness contradicts with verifiability, a more complex protocol can dramatically reduce usability, and so on. Moreover, we discussed the basic concepts of building the solution, but in a real-world system many more factors need to be taken into account: security and reliability of the communication protocols, system deployment procedure, access to system components. At the present time, no protocol satisfies all the desired properties and, therefore, no 100% truly robust online voting system exists.

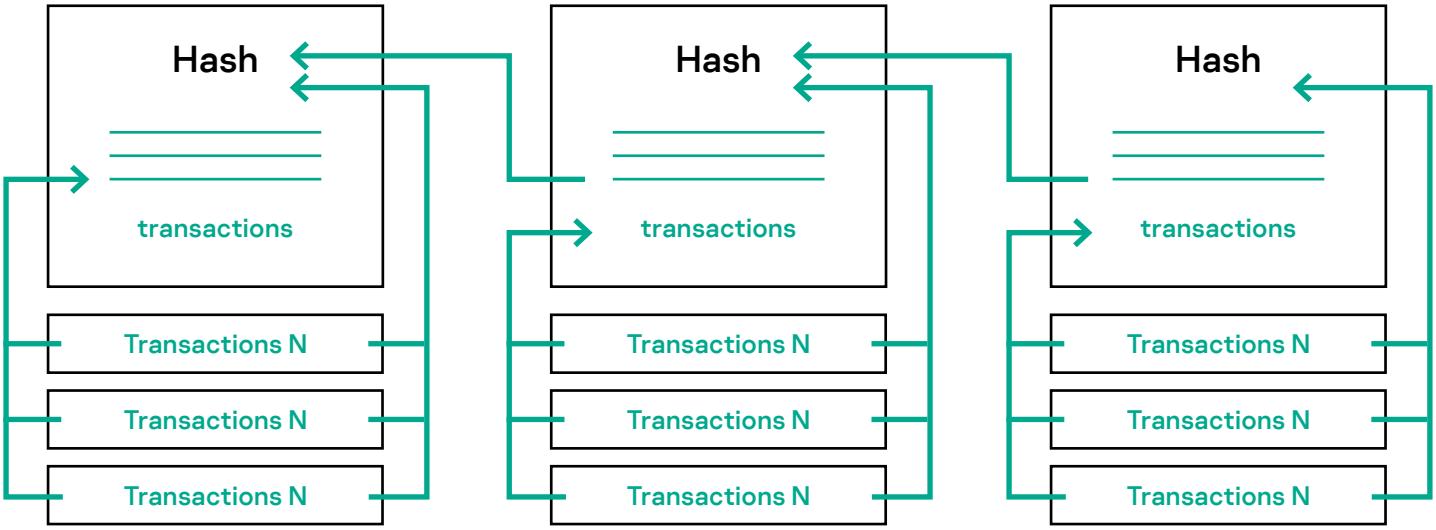
What we call a blockchain today is a set of technologies combined together: the blockchain data structure itself, distributed consensus algorithm, public key cryptography, and smart contracts.

Blockchain technology

The things that come to mind first when blockchain is mentioned are cryptocurrencies and smart contracts, due to the widely known Bitcoin and Ethereum projects. Bitcoin was the first cryptocurrency solution that introduced a blockchain data structure, and Ethereum, while providing a cryptocurrency solution similar to Bitcoin, introduced smart contracts that use the power of blockchain immutability and distributed consensus. The concept of smart contracts was introduced much earlier by Nick Szabo in the 1990s, and is described as "a set of promises, specified in digital form, including protocols within which the parties perform on these promises". In Ethereum, a smart contract is a piece of code that is deployed to the network so that everyone has access to it, and the result of executing this code is verified by a consensus mechanism and by every member of the network as a whole.

What we call a blockchain today is a set of technologies combined together: the blockchain data structure itself, distributed consensus algorithm, public key cryptography, and smart contracts. Below we describe these technologies in more detail.

As was already mentioned, the blockchain itself is the name for the data structure. All the written data are divided into blocks, and each block contains a hash of all the data from the previous block as part of its data. The aim of using such a data structure is to achieve provable immutability: if a piece of data is changed, the hash of the block containing this piece needs to be recalculated, and the hashes of all subsequent blocks also need to be recalculated. This means only the hash of the latest block has to be used to guarantee that all the data remains unchanged. In blockchain solutions, data that are stored in blocks are formed from all the transactions that were validated during their creation, which means no one can insert, delete or alter transactions in an already validated block without it being noticed. The initial zero-block, called the "genesis block", usually contains some network settings, for example, the initial set of validators (those who issue blocks).



Blockchain solutions are developed to be used in a distributed environment. It is assumed that nodes contain identical data and form a peer-to-peer network, so that there is no central authority. To reach agreement on blockchain data, a consensus algorithm is used that is fault tolerant in the presence of malicious actors. Such consensus is called Byzantine fault tolerance, named after the Byzantine Generals' Problem. Blockchain solutions use different BFT consensus algorithms: those that are intended to be used in fully decentralized self-organizing networks, like cryptocurrency platforms, use algorithms like Proof-of-Work or Proof-of-Stake, where validators are chosen by algorithm so that it is economically profitable for them to act honestly. When it is not necessary for the network to be self-organized, validators can be chosen at the network setup stage. The point is that all validators execute all incoming transactions, and come to an agreement on executing results in such a way that more than two-thirds of honest validators need to agree on the outcome.

Public key cryptography is used mainly for two purposes: firstly, all validators own their personal keypairs that are used to sign consensus messages and, secondly, all incoming transactions (requests to modify blockchain data) have to be signed to determine the requester. Anonymity in the context of a blockchain relates to the fact that anyone wanting to use cryptocurrencies just needs to generate a random keypair and use it to control a wallet linked to a public key. The blockchain solution guarantees that only the keypair owner can control the funds in the wallet, and this property is verifiable. As for online voting, ballots need to be accepted anonymously, but only from eligible voters, so a blockchain by itself definitely can't solve the issue of voter privacy.

Despite these issues, online voting systems can still benefit from applying blockchain technologies for two main reasons: immutability of data and, notably, verification of smart contract execution.

Smart contracts breathed new life into blockchain solutions. They stimulated the application of blockchain technology in efforts to improve numerous spheres. A smart contract itself is nothing more than a piece of logic written in code, but in conjunction with the immutability provided by a blockchain data structure and distributed consensus it can act as an unconditionally trusted third party. Once written, it cannot be altered, and all steps are verified by all the network participants. The really great thing about smart contracts is that anybody who can set up a blockchain node can verify its outcome.

As is the case with any other technology, blockchain technology has its drawbacks. Unlike other distributed solutions, a blockchain is really hard to scale: an increasing number of nodes does not improve network performance because, by definition, every node needs to execute all transactions, and this process is not shared among the nodes. Moreover, increasing the number of validators impacts performance because it implies a more intensive exchange of messages during consensus. For the same reason, blockchain solutions are vulnerable to various denial-of-service attacks. If a blockchain allows anyone to publish smart contracts in a network, then the operation of the entire network can be disabled by simply putting an infinite loop in a smart contract. A network can also be attacked by simply sending a huge amount of transactions: at some point the system will refuse to receive anything else. In cryptocurrency solutions all transactions have an execution cost: the more resources a transaction utilizes, the more expensive it will be, and there is a cost threshold, with transactions exceeding the threshold being discarded. In private blockchain networks this problem is solved depending on how the network is implemented – via the same mechanism of transaction cost, access control, or something else more suited to the specific context.

Despite these issues, online voting systems can still benefit from applying blockchain technologies for two main reasons: immutability of data and, notably, verification of smart contract execution.

Polys description

Introduction to the solution

Polys, our online voting solution, relies on blockchain technology to provide maximum verifiability at all steps of the voting process. Our system is designed for a variety of election types – municipal, political party, student organization, participatory budgeting, and so on. Different use cases imply different requirements, so in order to satisfy all those needs we made our system modular. There are various ballot types: single choice, multiple choice, distribution of scores or referendum ballots. In some cases, anonymity is just not needed, in others each voter has his/her own vote weight (for example, shareholder voting). We allow the organizer to choose the necessary options for upcoming voting events, and are open to developing other desired options in addition to those of our base platform.

To achieve system modularity, we isolate several entities and processes in the voting domain. The entities are:

- **Account (two types):** the organizer account for setting up and controlling votes, and the voter account for participating in votes;
- **Access Control List (ACL):** the list of voters eligible to vote. ACLs can be reused, enabling several votes to be organized at a voting event;
- **Vote:** contains general properties such as voting agenda, available options, start and finish dates, ACL, and settings controlling the participation process, ballot type and counting method;
- **Participation List:** controls ballot issuance and ballot acceptance processes for a specific vote. For example, if anonymization with a blind signature is used, for ballot issuance a blinded signature is stored as a sign the voter received the ballot, and for ballot acceptance the signature sent with the ballot is verified. If anonymization is not necessary, then ballot issuance is skipped, and voter eligibility is checked at the ballot acceptance stage;
- **Ballot Storage:** defines the ballot type used. Encrypted or not, single choice, multiple choice, with score distribution, and so on;
- **Tally storage:** defines the way ballots are treated to count the results. Voter choices are usually just added up, but there are also methods such as ranked-choice voting where defining the winner is not so trivial.

Not all combinations of the entities are allowed. For example, access control lists with vote weights cannot be directly combined with blind signature anonymization: such ACLs require special treatment, and ranked-choice voting is meaningless with single-choice ballots. But where it is possible, we permit a combination of entities to meet the needs of system users.

Not all combinations of the entities are allowed. But where it is possible, we permit a combination of entities to meet the needs of system users.

The Polys solution doesn't just consist of a blockchain platform. The key components of our system are:

- blockchain platform
- service layer
- polys-protocol library

The Polys solution doesn't just consist of a blockchain platform. The key components of our system are:

- blockchain platform: incorporates smart contracts defining voting process logic;
- service layer: responsible for user authentication, signing blinded messages, and infrastructural tasks like sending notifications;
- polys-protocol library: provides abstraction layer over blockchain communication protocol, and permits operations with high-level entities (mentioned above).

Optional, but also vital components are:

- organizer panel: developed as web application that relies on polys-protocol library, provides graphical interface for vote organizers;
- voter panel: developed as web application that relies on polys-protocol library, provides graphical interface for voters;
- observer software package. Consists of three parts: auditor blockchain node (fully operational blockchain node that cannot issue blocks), server part that parses data from blockchain and provides HTTP API for convenient retrieval, and web-based graphical interface presenting that data.

The key generation process is not in the core of the system – by default, the encryption keypair is generated in the organizer's application and stored with the organizer's data. Upon request we provide an application for key sharing or services for distributed key generation setup.

Blockchain platform

As we mentioned above, we use blockchain technology as a building block to provide voting system verifiability due to the immutability of blockchain data and the ability to verify the logic of all processes defined in smart contracts. Our solution is built on top of the Exonum blockchain framework, and below we note the reasons for our choice.

Anchoring is the process of periodically publishing a hash of the latest block to a public blockchain network like Bitcoin or Ethereum.

Exonum is designed primarily for organizing permissioned blockchain networks, which are best suited for online voting. We cannot automatically make assumptions about who is most interested in the honest operation of the network as is the case for public blockchains, nor can we restrict the system to everyone but the vote organizers – this will remove all the benefits of the blockchain. So, several authorities need to be set up that are authorized to participate in achieving a consensus and issuing blocks, while all other members are allowed to obtain all the blockchain data. To rule out the possibility of authorities colluding to overwrite blockchain history, an anchoring mechanism can be used. Anchoring is the process of periodically publishing a hash of the latest block to a public blockchain network like Bitcoin or Ethereum. When anchoring is used, even if all the authorities collude and rewrite the blockchain history, it can be easily detected because the hashes of the rewritten blockchain will differ from the anchored hashes.

Exonum is written in the Rust programming language and allows the development of smart contract logic in Rust, which is a great choice for building reliable and fast applications. Public blockchains use various methods to prevent execution of network-breaking logic. Ethereum, for example, uses Ethereum Virtual Machine, which provides a Turing-complete assembly language (the Solidity language is built on top of it), but the virtual machine itself allows you to predict function complexity and discard operations that are too heavy. This can be considered a custom solution to the halting problem. Other blockchains provide Turing-incomplete language for smart contracts, which by design does not permit the creation of an infinite loop. Exonum has no such restrictions, meaning smart contract logic needs to be carefully tested, but this is offset by the fact that contracts cannot be published by everyone: smart contracts can only be deployed by authorities and only after they reach agreement on deployment via consensus. This solution for creating smart contracts also produces the highest performance among all the different blockchain solutions – up to 5000 transactions per second, and in some cases even more.

Exonum uses a customized consensus algorithm based on the PBFT. It also utilizes some ideas from Tendermint, but with some distinguishing features. Like any other BFT-based algorithm, it acquires the property of finality so that it excludes the possibility of blockchain forks. Issuing a block requires more than just approval of the round leader, as is the case with the Parity Aura algorithm – it only occurs when more than two-thirds of the validators agree on it. Exonum does not require mining or economic rewards. Exonum also provides tools that allow anyone to get cryptographic proof of the presence of certain data in the blockchain and the fact that the data have been added as a result of a transaction that achieved consensus among known validators. These tools allow you to provide all network participants with reliable evidence of the correct logic in real time, including for clients with limited computing resources (in particular, the organizer and voter web applications), providing greater system transparency.

Voting process in detail

As we stated above, our solution is intended to be adaptable to different requirements, and we will not try to cover all the possible use cases in this paper. Instead, we provide a step-by-step description of the voting process by default, with explanations of what can be verified at each step and how, as well as commentary on possible modifications.

Step 0. Initial deployment

The core of the system is a blockchain network. To deploy it, authorities need to be selected. Each authority needs to generate their own keypair for signing messages, and public keys should be exchanged and added to node configurations as validators. The validators list is written to the genesis block of the blockchain database, so that once set it cannot be changed by any means other than through a consensus mechanism. Public keys of nodes should be made publicly available in order to verify the cryptographic proofs provided by the blockchain framework. These proofs are used not only for retrospective verification of the system but also to ensure that users don't communicate with a malicious man-in-the-middle entity instead of the real network. If anchoring is envisaged, it should be configured at this step. Exonum provides anchoring to a Bitcoin network by default, but anchoring to any other system can easily be implemented.

After the network is configured and launched, the service configuration needs to be applied. As well as information on the service versions used, it contains public keys of service layer accounts. This is the only place these accounts are authorized to perform operations. These operations are related with user account representation in the blockchain: Polys is intended to be used with an external identification system, so the user needs to first authenticate in it, then make a request to the service layer to link his/her randomly created transaction signing keypair with his/her account in the blockchain. The service layer contains the necessary information to verify user authentication via an external identification service and mapping of an external user ID to the internal one that is used in the blockchain. This means that only the service layer contains sensitive personal information, and the user doesn't need to remember keypairs or his/her seed phrases – he/she can have an unlimited number of them. For more details about this process, see "Step 2: Voter authentication". The authentication process for the organizer is similar.

A blockchain service keypair needs to be configured for the service layer, along with settings like database connection properties or mailing service parameters. As noted above, the owner of this keypair is authorized to link accounts from an external identification service with blockchain accounts, so this keypair should be treated with care. Also, a keypair for blind signatures needs to be configured. By default, Polys uses a single keypair, but this can be configured to use threshold multisignature, and the signing service is detachable from other services.

The organizer and voter applications and observer package have no settings that require special care – they use only publicly available information like blockchain endpoint settings, validator public keys and so on.

Step 1. Vote setup

The vote setup process starts by defining the access control list (ACL) with the eligible voters' identifiers. The parameters of the ACL are: voter authentication type, whether voters have different vote weights or not, and whether the list of voters is predefined (the organizer provides entries) or voters are allowed to add themselves, i.e., for a public vote. Depending on which identification system is used (by default, Polys provides identification via email, phone number and a generic module for the OAuth 2.0 protocol), the organizer prepares the list of corresponding identifiers and makes an ACL creation request to the service layer. The ACL creation service assigns an internal identifier to each voter and stores the link to the external ID in the service database. The service sends an ACL creation transaction, and then registers the list of internal identifiers. Note that only the ACL creation service is authorized to create ACLs and add entries to them, due to the fact that this process involves working with sensitive voter data. As the ACL is created in a blockchain, it can be reviewed by anyone, but public users see only the list of internal identifiers and registration status. For predefined lists, the service also sends a transaction that stops registration, thus preventing the addition of new entries, even by the service. If a list or real identifiers need to be audited, for example, by observers, access to view the service database can be provided. Also, there is an option to store the hash of the original list (with external identifiers) in ACL data in the blockchain as a guarantee of list immutability.

Then it comes to creating the vote itself. The organizer specifies the ACL identifier to use, a vote description, the list of options to vote for, and the start and finish dates. The organizer also sets up the voter participation method (there are currently two: deanonymized access, and access with anonymization using a blind signature), ballot type

(single choice, multiple choice, encrypted or not, etc.) and the tallying method. In this case we are using the default setup with blind signature anonymization and encrypted ballots.

In order to use encrypted ballots, an encryption keypair needs to be generated and a public key stored in the blockchain. At this point it's worth saying a few words about the cryptography we use.

Both blind signature and ballot encryption use elliptic curve cryptography. We use Curve25519, along with the **Ristretto technique** to ensure fast, safe cryptographic operations. The specific implementations used are **curve25519-dalek** for cryptographic operations in smart contracts, and the **JS implementation of ristretto255** for client applications (organizer and voter panels). The blind signature protocol will be covered in detail in the sections below about acquiring an electronic ballot and casting a vote. We use the Elliptic Curve ElGamal encryption scheme for ballot encryption. It is homomorphic by addition, thus making it possible to calculate the sum of the votes before decryption to speed up the tallying process. It also allows to construct efficient zero-knowledge range proofs to verify the ballots. The cryptographic protocol is based on the scheme described in [1]. The stages of ballot encryption and the tallying process will be covered in more detail in the sections on casting a vote and result tallying.

Encryption keypair generation is a very important process: there needs to be a sufficient level of randomness, and the private key should never be accessible to intruders. But in reality, not everyone needs this process to be performed in a sealed room with no connection to the outside world: different levels of elections imply different levels of adversary. By default, for the sake of simplicity and user convenience, the keypair is generated in the organizer application and stored along with organizer data in our system. We provide an option for the organizer not to store the keypair in the system and download it, or generate it outside of our application and provide the public key for vote creation (and a private key just before the tallying). We also provide an application for the key sharing protocol (Shamir's Secret Sharing), or Distributed Key Generation network can be set up upon request.

As well as a keypair, the necessary parameters are the generator of an elliptic curve field and points representing voter choice. For the current set of ballot types, two points are selected: one representing the choice (choice point), and one representing an absence of choice (blank point). These parameters are hardcoded in the blockchain application. The generator is part of the elliptic curve domain parameters, and the method of choosing points will be covered in detail in the section on tallying results.

Step 2. Voter authentication

Polys implies the use of an external identification service. By default, we provide identification via emails, phone numbers, and a module for using any system compliant with OAuth 2.0 protocol, though any other option can easily be added. Regardless of the identification service used, the authentication protocol is always the same.

After the authentication process, users mostly communicate with the blockchain directly. For any user an entity called 'Alias' is assigned in the blockchain that contains the user internal identifier and list of public keys belonging to the user. Mapping of external identifiers to internal identifiers is stored in the authentication service database. So, in the first step of authentication the user makes a request to the authentication service claiming that he/she owns a specific external account. Depending on the authentication method, a specific strategy is selected. For example, for email and phone, a verification code is sent, and for OAuth 2.0 authentication the user is redirected to a signup page of the identification service. After the user verifies external account ownership, the authentication service finds or creates an internal ID belonging to the user and generates a random secret. The hash of the secret with the internal ID is sent to the blockchain. This tuple is called a "proof request", and the secret itself is sent back to the user. The user, in turn, generates a random blockchain keypair, and signs the transaction containing the secret with it. The smart contract checks the secret and if the hashes are equal, links the public key derived from the transaction signature with the account specified in the proof request. All further transactions signed by this keypair are deemed to have been sent by that specific user.

Step 3. Acquiring an electronic ballot

In order to achieve voter privacy, we use a blind signature scheme. The scheme we use is described in [2]. This scheme satisfies the necessary properties of unforgeability (under the assumption of EC DLP; no one can sign except the authorized signer), and blindness (no link can be set between blinded and unblinded values by the signer). The protocol works as follows.

At the first step, the voter needs to obtain blind signature domain parameters – elliptic curve generator G, signing public key Q and random point R ($R = k * G$, the signer should know k to sign the blinded value). Keypairs are stored in the signing service, and their public parts are published to the blockchain, so the voter obtains the parameters directly from there.

We call the value that is signed blindly the "ballot authorization token". To create this token, the voter first needs to generate a random blockchain keypair (not linked to his/her account) that will be used for signing the transaction with the ballot. The hash of the public key for this keypair will serve as the ballot authorization token. This keypair is not linked to the voter's account, so doesn't disclose any personal information, and the link between the public key and ballot authorization token makes it possible to check that the person who used the token is the same person that owns the transaction signing keypair. This technique prevents man-in-the-middle attacks on the ballot sending channel: if intruders intercept a transaction, they still won't be able to use the ballot authorization token to send the ballot because they don't know the private key for signing the transaction, and the smart contract checks that the hash of the public key derived from the signature corresponds to an authorization token.

At the next step the voter chooses three random values as blinding factors – t_1, t_2, t_3 – and blinds the authorization token (d is the signer's private key):
$X = (t_1R + t_2G + t_3Q) = (t_1k + t_2 + t_3d)G$
$m' = x_r t_1 (m^{-1} + t_3)^{-1}$
where m' is the blinded message, and X will be sent as part of the signature.

At this point all generated parameters – signing keypair and blinding factors – can be saved to a reliable storage. In the event of a network failure or other process interruption, it will allow the voter to restore the process.

A blinded message is sent to the signing service. Once authenticated, the voter sends this request and the signing service first checks whether or not the voter has acquired the signature yet. This check is delegated to smart contract logic in the blockchain: if the service signs the blinded value for the voter, it stores the resulting signature in the blockchain as a mark that this voter received it, and only then sends it to the voter. This prevents double voting because the voter should never receive a second signature. With subsequent attempts to request the signature the voter will receive the one stored in the blockchain, so in the event of a process interruption it will not be lost.

Singing is performed as follows:
$s' = dx_r + km'$

After the voter receives the signature for the blinded authorization token, he/she uses blinding factors to produce a signature for the unblinded token:
$s = m(t_1 s' m'^{-1} + t_2)$
signature is pair (X, s)

The signature can be verified by checking that the following equation holds true:
$sG = mX + Q$

The resulting signature with ballot authorization token will be sent with the ballot via an anonymous channel. Note that, in fact, it is not necessary to rely on our voter application for this process: with all the parameters that the voter has, he/she can produce a signature for unblinded token with any other software (or custom-made script) that he/she trusts, as well as form the ballot itself, and send the transaction from any place directly to the blockchain. In other words, after the voter receives the signature for the blinded token, the process may be safely interrupted and resumed at any time from any other device without the need to authenticate.

Step 4. Casting a vote

To encrypt the choice, the voter obtains the elliptic curve domain parameters, public key and points used to encode the choice (choice point and blank point) from the blockchain, as well as the available options list. Ballot construction methods differ slightly depending on ballot type. We will cover single-choice, multiple-choice and cumulative (with score distribution) ballots:

1. Single-choice ballot

In this type of ballot, the voter is allowed to select only one option from those provided. For each option from the list, the voter encodes a blank point if the option is not chosen, and a choice point for the chosen option:
$(A, B) = (r * G, blankPoint + r * Q)$
or

$(A,B) = (r * G, choicePoint + r * Q)$
where (A,B) is the resulting ciphertext, G is the elliptic curve generator, r is the randomization parameter generated by the voter and Q is the encryption public key. For each encoding, a different session key (randomization factor) needs to be chosen; otherwise, all the options that are not selected will have an identical encryption result.

Zero-knowledge range proofs then need to be constructed to prove that the resulting set contains only one choice. The apparatus for constructing proofs works as follows:

The prover needs to prove that choice $C_p = (A_p,B_p)$ belongs to the set $\{M_1,M_2,...,M_m\}$.
The prover generates points A_k,B_k, for $1 <= k <= m$:
$A_k = w_k * G + u_k * A_p, \forall k \neq c_p$
$A_{cp} = s * G$
$B_k = w_k * Q + u_k * (B_p - M_k), \forall k \neq c_p$
$B_{cp} = s * Q$
$challenge = hash(A_1,A_2,...A_m,B_1,B_2,...B_m),$
$u_{cp} = challenge - \sum_{k \neq cp} u_k,$
$w_{cp} = s - u_{cp} r_p$
where w_k,u_k,s are random values from a group. The prover sends values A_k,B_k,u_k,w_k to the verifier. The verifier checks that
$A_k = w_k * G + u_k * A_p, \forall k \in [1,m]$
$B_k = w_k * Q + u_k * (B_p - M_k), \forall k \in [1,m]$
$challenge = \sum_{k=1}^{m} u_k$
with challenge computed as $hash(A_1,A_2,...A_m,B_1,B_2,...B_m)$

In the case of a single-choice ballot, we need to prove that the sum of all encrypted options is exactly $choicePoint + (n - 1) * blankPoint$, where n is the total amount of options.

2. Multiple-choice ballot

In this type of ballot the voter is allowed to choose several options – maximum and minimum values are defined during voting construction and stored in the blockchain. Encoding is the same as for a single-choice ballot, keeping in mind that several options may be chosen. The zero-knowledge proof is a bit trickier.

First, the voter needs to prove that he/she selected no less than the minimum and no more than the maximum for the permitted amount of options. So, the permitted ZKP set for the sum is as follows (*cP* stands for *choicePoint* and *bP* stands for *blankPoint*, *min* and *max* for minimum and maximum amounts of options permitted for selection):
$\{min * cP + (max - min) * bP, (min + 1) * cP + (max - min - 1) * bP,...,max * cP\}$

But that is not enough – because the voter can choose more than one option, he/she can in theory construct a ballot like $max * cP$ for one option, and encode blank options for all the other options. This behavior is definitely not desired, so for each encoded option we need to construct proofs with the statement that only *cP* or *bP* is encoded in it.

3. Cumulative ballot

In this type of ballot the voter needs to distribute scores among the available options. The scores can all be set for one option, or distributed in any manner desired by the voter. Minimum and maximum numbers of options are defined during voting configuration. For each option the voter constructs the value for encoding as follows:
$scores * cP + (max - scores) * bP$
where *scores* is the amount of scores distributed for this option. The point of encoding each option like this is that we need to have a set 'amount' of points encoded – the sum of choice and blank point multipliers must be equal to the maximum amount of scores the voter can distribute. The reason for this will be described in detail in the section on tallying.

For zero-knowledge proof of sum, the range is defined as follows (*n* is the total amount of options):
$\{min*cP +(max*n-min)*bP, (min+1)*cP +(max*n-min-1) *bP,...,max*cP +(max*n-max)*bP\}$

Besides checking the sum, we need to check that each option encoding contains from zero up to the maximum amount of scores:
$\{max * bP,cP + (max - 1) * bP, 2 * cP + (max - 2) * bP,...,max * cP\}$

The resulting ballot is constructed from the list of encrypted values for each option, the ZKP for the sum of encrypted values, the list of ZKPs for each option if necessary, and the ballot authorization token with signature that was obtained during the previous step. All this data is packed in the transaction and sent directly to the blockchain via an anonymous channel.

To find the transaction in the blockchain, the voter simply needs the transaction hash, which he/she can easily calculate. The Exonum blockchain provides cryptographic proof of the fact that the transaction passed through consensus among known validators, and this proof can be used in the event of any dispute. The hash is also used as an internal ballot identifier, and the blockchain platform provides a method for obtaining all the information about it: how the data is stored and the verification results. After voting has finished, the voter can use the published private key and decrypt the ballot to test whether it contains his/her choice.

Step 5. Results tallying

To start tallying results, voting first needs to be stopped and the addition of more ballots blocked. Then the decryption key is published to the blockchain by the organizer(s), and the decryption process is started.

After verification of each ballot's authorization token and zero-knowledge proofs, and if everything is correct, the encrypted values of each option are summed up one by one with those already stored: it means that for each option a separate sum is stored, and each value from the ballot's encrypted list of choices is added to the corresponding sum.

The value in this sum is as follows: $a*cP + x*bP$, where a is the number of chosen options, and x is the number of blank points. To retrieve the amount of chosen options, a bounded discrete logarithm should be solved. [1] suggests solving this task as a knapsack problem using the meet-in-the-middle algorithm with precomputed table. Due to the way we encrypt the choices, we can use a slightly modified approach. In the case of voting with single-choice ballots, the resulting sum will be $a*cP + (n-a) *bP$, where n is the amount of stored ballots. So, to compute a, we can simply calculate the table with all possible combinations with the values: $\{n * bP, cP + (n - 1) * bP, (2 * cP) + (n - 2) * bP, \ldots, n * cP\}$ and simply perform a lookup of the sum in this table to determine the amount of choices. For large-scale elections computing a table like this can take a significant amount of time, but we can speed things up by precomputing it. As we do not know the number of voters that will send ballots, we define the size of the votes package, for example, 100,000, and precompute a table for this amount. During the vote the pack is filled and if it reaches the set size, another pack is started. To obtain the results, all the packs are decrypted and the values representing the number of choices for each pack are added up.

If the number of ballots is less than the pack size, then we need the values in our table where this amount is used instead of n. If we precompute all possible conditions, the size of the resulting table will be $n + (n - 1) + (n - 2) + \ldots + 1$, and the size dramatically increases while incrementing the n, so the effectiveness of this solution is debatable. Instead, for incomplete packs we can add some padding with encoded blank points like $(n - s) * bP$, where s stands for incomplete pack size. That allows us to compute a table of size n.

As for a cumulative ballot, each option has several "numbers" of points, defined by the maximum scores a voter can distribute. So, if we use a package of size n, for cumulative voting one package can store $n/maxScores$ ballots. If n is divided by $maxScores$ with the remainder, each filled package should be padded with $remainder * bP$.

The selection of choice and blank points depends on the selected package size. The base point P of the curve serves as the choice point, and the blank point is selected as $(n + 1) * P$, so that any number of choice points cannot be treated as one blank point. As the choice points and blank points, as well as package size, are the same for all votes, we precompute the table for decryption while compiling the blockchain node code.

Due to the fact that the decryption key is published to the blockchain, anyone can verify the whole process step by step using any relevant tools, and check the accuracy of the results.

Note on coercion resistance

We haven't mentioned coercion resistance in the overview of the protocol because tools for coercion resistance are not implemented at the core of the system. The protocol design is focused on maximum verifiability, and as we have already stated, techniques for implementing coercion resistance inevitably lower the verifiability level. However, it is possible to change some parts of the protocol to achieve coercion resistance to a certain extent. For example, a private key doesn't have to be published to blockchain, implying that decryption is performed by the key holders and the total decryption result with proof of correctness is published to the blockchain, and in conjunction with a trusted party (either service or device) that hides the randomness from the voter. The coercer will have no confidence in what is encrypted. But there are other vulnerable points such as authentication and ballot issuance processes.

Note on tooling for vote observation

The direct interaction of users with blockchain and cryptographic proof of transaction processing can quickly assure anyone that their request will not be lost. Smart contracts by their very nature provide verifiable logic execution. And the anchoring technique can further increase confidence in data immutability. Moreover, cryptographic proofs can be used to prove certain claims in the case of a dispute. As can be seen from the protocol, every step of the process is observable through the blockchain. We mentioned our observer software that provides the user interface to easily view the state of the system, as well as the API interface for automatic data processing. We will continue to develop observer functionality as well as tooling aimed to verify complex processes like ballot decryption.

Conclusion

This is definitely not where we stop – there's still much work to be done. We are constantly researching ways to improve the protocol and our goal is to develop a robust, fully auditable voting solution that suits all voting scenarios.

References

[1] M.A. Cervero, V. Mateu, J.M. Miret, F. Sebe', J. Valera: "An Elliptic Curve Based Homomorphic Remote Voting System" (2014)
[2] Hossein Hosseini, Behnam Bahrak and Farzad Hessar: "A GOST-like Blind Signature Scheme Based on Elliptic Curve Discrete Logarithm Problem" (2013)

Cyber Threats News: www.securelist.com
IT Security News: business.kaspersky.com/

www.kaspersky.com

kaspersky BRING ON THE FUTURE

GBD-7262 Q1/21 V1