Electis.app White Paper

Electis.io

Abstract. The Electis voting App (Electis.app) is a web application built using Django and ElectionGuard SDK). The latter comes with homomorphic encryption and end-to-end verifiable proof of ballots and tally (initially designed for US election machines. In addition, Electis.app relies on the Tezos blockchain to generate proof of the election via a smart contract. Finally, it uses IPFS decentralized storage to share the proof and ballots with voters to allow them to verify the election was not violated. This document dives into the overall architecture of the e-voting platform and discusses the application'sapplication's key features and how the election is decentralized.

1 Software Architecture

The Electis e-voting platform is separated into three core apps, electeez_auth, djelectionguard, and djectionguard_tezos. All user-related authentication and account management for the e-voting platform is handled within the ""electeez_auth"" app. The contest-related functionalities are managed in the ""djelectionguard"" app, and finally, all Tezos-related functionalities are in the ""djelectionguard_tezos"" app[1].

1.1 User Authentication

Django's inbuilt user authentication is utilized for the user account management and authentication system. The user model is extended from Django's AbstractUser to include the username and email, and a token model was added for verification (to verify unique users). To sign up, a user needs to do is to insert their email and password. Then, A user object is created in the database, and a token is generated for the user, set to expire after 30 days. A URL-safe token is generated and stored for later verification using python's inbuilt secrets library. When a user activates a verification link with the generated token, a crosscheck occurs in the token database. If not expired, the user is logged in, and the account is marked as active. Users can also reset their passwords, which is done by using the provided functionalities from Django.contrib.auth.urls. An extended feature in Electis.app is the ability to log in using One-Time Passwords (OTP).

During contest creation, when the moderator of the election shares the voters' email list, an account is created for any email address that does not have an account registered. An OTP is shared with users via email to use and log in to the platform and participate in the contest they were invited to vote in. In addition, a second level of identification through SMS is also available, requiring the voter to enter their phone number after

[1] Tezos. “Tezos White Paper”. Tezos Agora Wiki (Link)

signing up or receiving the OTP and then confirming the second password they receive via SMS.

1.2 ElectionGuard extension

ElectionGuard[2], which was designed for voting machines (within the existing infrastructure used in elections), is extended in Electis.app to organize a secure remote e-election. The following sections in this chapter will explain the entire flow of the extension.

Configuring an Election

In step 1, the user shares the basic details of the election, and a manifest is created in JSON format and parsed into an election description. Election Builder is instantiated and generates the public-private key pairs, readying for the key ceremony. The manifest includes contest- related information, such as name, ballot details, candidates, etc.

Key Ceremony

The key ceremony is the process of sharing the encryption keys for the contest. Before the election is opened, a fixed number of guardians pre-determined by the mediators must hold the private keys to decrypt the election results later. To account for potentially missing guardians, the quorum count can be less than the total number of guardians.

Each guardian has a unique id and sequence to generate their public-private key pair, where they will hold their private and all guardians' public keys. They will need to verify the key backup with all the keys they hold. If the verification fails, the guardian whose key doesn't match will perform a key challenge and share the unencrypted key with all guardians to verify. If it fails to verify, the guardian with the corrupted key is replaced with a new one (Electionguard Python). Once verified, the joint public keys are published for the election. Although ElectionGuard can organize elections without the key ceremony[3], it is recommended to increase the security of the election.

Encrypted Ballots

Once the election is opened for participants to share their vote, a client-side encrypted ballot is created; it first verifies that the ballot is well formed against the Election Metadata and generates a master nonce value as a secret when encrypting the ballot as CiphertextBallot.

As mentioned before, ElectionGuard uses homomorphic encryption and Non-Interactive Zero-Knowledge proof (Electionguard Python Documentation[4]). The proof is used to show that the encryption is either an encryption of zero or one for each selection on the ballot, or the sum of all encrypted ballots is equal to the selection limit

---

2 ElectionGuard Structure (Link)

3 Electionguard Python Documentation. "Key Ceremony." *GitHub*. (Link)

4 Electionguard Python Documentation. "Encrypt Ballots." *GitHub*. (Link)

on the contest. A verification code is then generated to share with the voters that they can use later to verify the tallying. The homomorphic property allows the encrypted votes to be combined to form encrypted tallies. First, the public guardian keys are combined into a single public key to encrypt the ballots. Then, at the end of the election, ideally, the guardians use their private key to decrypt each tally partially. In the end, the partial decryption is combined to form verifiable decryptions of the tallies (Electionguard Python Documentation).

Homomorphic Properties

The votes are encrypted using an exponential form of the ElGamal cryptosystem by selecting a random nonce and forming a pair. Then, the secret key's guardians, or multiple guardians, can decrypt the message (ElectionGuard). This allows it to have additively homomorphic properties.

Component-wise product of two encrypted messages would be the encryption of the sum of the two messages. Hence, all the encryptions of a single option across ballots can be multiplied to form the encryption of the sum of the individual values. The individual values are on ballots that select that option and zero. Otherwise, the sum is the tally of votes for that option, and the product of the individuation encryptions is an encryption of the tally (ElectionGuard).

Non-Interactive Zero-Knowledge (NIZK) Proofs

Four techniques are used in ElectionGuard to provide numerous proofs about encryption keys, ballots, and tallies. These techniques ensure keys are correctly chosen, the ballots are properly formed, and finally, the decrypted tally matches the claimed values.

- A Schnorr proof - allows a holder of an ElGamal secret key to interactively prove possession of the secret key without revealing it.
- A Chaum-Pedersen proof - allows ElGamal encryption to be interactively proven to decrypt to a particular value without revealing the nonce used for encryption or the secret decryption key.
- The Cramer-Damgard-Schoenmakers technique - enables a disjunction to be interactively proven without revealing which disjunct is true.
- The Flat-Shamir heuristic - converts interactive proofs into non-interactive ones. (ElectionGuard)

Cast and Spoiled Ballots

A key feature to be implemented soon in the Electis.app development roadmap is the "Benaloh Challenge" [1]. When voters create the ballots, they must be either cast or spoiled. When each ballot is loaded into the memory and verified to be correct using the proofs mentioned above, the ballot is submitted and can be either identified as cast or spoiled. The cast ballot is combined into CiphertextTally, whereas spoiled ballots are cached for later decryption (Electionguard Python Documentation[5]).

---

[5] Electionguard Python Documentation. "Cast and Spoil Ballots." *GitHub*. (Link)

Decrypting the Tally

When the election is closed, the encrypted ballots and proofs that the ballots are well formed are shared as artifacts. Each option's encryptions are homomorphically combined to form encryption of the total number of times that each option was selected. Finally, the combined encryption is decrypted to generate the election tally. No individual cast ballots are decrypted. To decrypt the combined encryption, a specific decryption share of the decryption is computed for each guardian. During this process, the spoiled ballots are also decrypted and shared, as it is verifiable in the same way that the aggregate ballot of tallies is decrypted. This allows voters to explicitly generate challenge ballots which they can later use to verify the authenticity of the election (Electionguard Python Documentation[6]).

1.3 Voter Participation Tracking

Electis.app can let election moderators track voter participation. It stores and checks whether the voter received the email with the OTP link or not and whether the voter cast it. If required, moderators can also request new OTP links for voters and share that with the voter manually.

1.4 Decentralization of the election

In the beginning, when the contest is set up, a smart contract for the election is created that holds all the election-related information, including the link to the InterPlanetary File System (IPFS) network. The published ballots by ElectionGuard at the end are published on IPFS storage. The smart contract created should be managed by the moderator of the election and should have its unique wallet. The smart contract holds all information such as, but not limited to, when and what time the contest was opened and closed, what it was about, the election manifesto, public keys, who the candidates were, the moderators, the guardians, etc.

IPFS Storage

In the last stage of the election, the artifacts are ready to be published when the contest has been tallied and decrypted. Hence, the moderator can initiate the action to publish all artifacts created by ElectionGuard and upload them on IPFS decentralized storage. The uploaded files include the encrypted casted ballots, spoiled ballots, proofs, and the encrypted and decrypted tally. Once the file is uploaded, the cryptographic hash is received, and later, anyone can look up the file using the unique fingerprint to download and verify the election. A transaction is made in the smart contract to update the storage to include the fingerprint and store the actual close time and final election tally.

What Smart Contracts Enable

---

[6] Electionguard Python Documentation. "Decryption." *GitHub* (Link)

While it is shared, the smart contract can function as a legal document to prove the election has met all the requirements. In addition, it also enables it to trigger actions depending on the output of the election. Actions based on the election result can be automated while keeping it transparent and secure.

1.5 Universal and Personal Verification

Elections should provide both privacy and integrity, i.e., enable everyone to audit the election results while not having to go transparent on their votes and forgetting about anonymity. We already know that with threshold encryption, there would be no way to decrypt the individual ballots as it would require combining all the private keys the guardians hold to run the decryption protocol. Along with the proofs prepared by the ElectionGuard SDK, Electis.app also uses the Fiat-Shamir Heuristic to verify the final tally. The process for the proof is public and verifiable.

With the proof of the results, voters receive their code (bulletin number), which they can use to verify that their casted ballot is counted in the final tally. All encrypted ballots are uploaded to the IPFS decentralized storage. At the end of a vote, the unique ID of their ballot and the unique ID of the election is shared with the voter. We use the email/login info as a key to present the election details and allow the user to check if their vote was considered. A hash of the encrypted ballot is provided to the voter after casting their vote, which the voter can compare to the hash of the encrypted ballot stored on IPFS under the correspondent bulletin ID. This double-check (bulletin ID and hash) provides a full personal identification.

Additionally, any voter can redo the tallying once the Guardians share the decryption keys publicly. This universal verification feature is a priority in our development roadmap and will soon be implemented. This, in combination with the "Benaloh challenge", offers full end-to- end verifiability. Trust in the central server's operations is then guaranteed by the trust in the in-flows (through the Benaloh challenge) and out-flows (through personal and universal verification) to and from the server.

2 Bibliography

1. Benaloh, J.: Simple verifiable elections. In: EVT 6. p. 5 (2006).