CSCI 1515: Vote Extension

Ayman Benjelloun Touimi
ayman_benjelloun_touimi@brown.edu
Brown University
Providence, Rhode Island, USA

Patrick Peng
patrick_peng@brown.edu
Brown University
Providence, Rhode Island, USA

1 INTRODUCTION

Online voting is extremely convenient. It makes the process significantly more accessible to voters, ultimately improving participation and thus, representation. A major challenge with online voting resides in its security: it is difficult to ensure the validity and confidentiality of every party's vote.

Implementing the Vote project this year has taught us how to combine different cryptographic protocols to put together a valid, cryptographically secure, and anonymous voting protocol based on the widely used Helios [1] protocol. However, the project only allows voting for one of two candidates.

We describe in this final report our extension to the Vote project to support voting for multiple candidates. Specifically, our new extended protocol enables voters to vote for exactly k candidates, and incorporates cryptographic mechanisms to enforce the rule.

2 BACKGROUND KNOWLEDGE

Our project is built off of a working solution for CSCI1515's Vote Project [2], as offered in the Spring 2024 iteration of the course. In this section, we will explore fundamental cryptographic protocols we applied in this project. We will not provide extensive details on each of these protocols; the linked specification explains these in more depth.

2.1 Additively Homomorphic Encryption

Usually, the ability to change the value of encrypted data is undesirable. This property, **malleability**, can be useful, though, as computations can be performed over data without the requirement of input from other parties or knowledge of the decrypted values. When a scheme allows for computation over ciphertexts, it is a **homomorphic encryption** scheme; when it allows addition specifically, it is **additively** homomorphic.

In this project, we utilize an additively homomorphic scheme based on ElGamal encryption, which gives us the ability to combine two ciphertexts and obtain a ciphertext that decrypts to their sum. Although the decryption in this scheme requires solving the discrete logarithm problem, there is only a small range of possible values for our application ($[0, n]$ where n is the number of voters), so it suffices to check all of them by brute force.

2.2 Threshold Encryption

Now that we have homomorphic encryption, we have a secure way for voters to add votes to publicly available values. However, anyone with access to the decryption key is able to check the value at any time, which nullifies the security benefits–they can simply check whenever a vote is added.

To solve this, we extend ElGamal encryption so that each party can only partially decrypt the ciphertext, and a full decryption requires partial decryptions from each party.

2.3 Zero-Knowledge Proofs (ZKPs)

There are also many conditions we must uphold in this voting scheme. We must ensure that voters are actually submitting a value in $[0, 1]$ for their vote, that partial decryptions are correct, that the vote sum is the right amount, and so on. This is accomplished with zero-knowledge proofs, which reveal some condition about the underlying information without revealing the underlying information. We extend these to be non-interactive with hashes of the transcript, so that they do not require a challenge from an independent oracle.

2.4 Blind Signatures

In the voting scheme, we'd like the voters to be anonymous. How, then, can we ensure that each voter only votes once? Someone along the way must know the identity of the voter, but only enough to prevent them from voting again, so there must be no way for this party to know the vote of the voter. To accomplish this, we can use blind signatures. Under this scheme, the voter can first blind their message, making it look random; we can then have a party sign this such that it beceomes a valid signature on the original message after being unblinded.

In this project, we adapt RSA to achieve this.

2.5 Different Parties in *Vote*

In the Vote project, we have a few independent parties that conduct the election together.

2.5.1 Voters. Voters begin by sending their blinded votes with their ID to the registrar for signing. Then, they unblind the votes and send them, along with the registrar's signature and vote/vote-sum ZKP(s), to the tallyer. At the end, the voter can collect partial decryptions and check the results of the vote.

2.5.2 Registrar. The registrar is responsible for performing the blind signature on the voter's blinded message. Here, they can check if the voter's ID has been recorded in the database already.

2.5.3 Tallyer. The tallyer receives votes from the voter after the registrar has signed them. The tallyer verifies the registrar's signature, as well as each individual vote ZKP and the ZKP for the sum of the votes. These votes are then stored into the database.

2.5.4 Arbiters. Arbiters are responsible for generating the result of the election. Each arbiter holds a different partial key, which generates a partial decryption of the ciphertext; threshold encryption only allows the result to be seen when all the arbiters have provided their partial decryptions. Each arbiter will first verify the signatures and ZKPs for the votes before generating their partial decryption.

3 VOTING FOR EXACTLY k CANDIDATES

3.1 Premise

We propose that each voter i casts a vote of either 0 or 1 for each candidate j out of n candidates. 1 means they'd like to vote for the candidate, 0 means they do not. At the end, each voter will have n ciphertexts corresponding to the encryption of their vote for each candidate. Along with each ciphertext, the voter provides a ZKP that the ciphertext is an encryption of either 0 or 1, using the protocol used in the basic version of the assignment.

3.2 ZKP Proving that the Sum of Votes is k

In order to enforce that each voter votes for exactly k candidates, we leverage the power of ZKP to prove, in zero-knowledge, that the sum of a voter's ciphertexts is k.

3.2.1 ZKP Generation and Verification. The protocol is as follows.

(1) The voter, who is also the prover here, obtains a homomorphic sum of their votes with the protocol described in section 2.1. $c = (c_1, c_2)$. Specifically, $c_1 = g^{\sum_i^v r_i} = g^r$ and $c_2 = \mathsf{pk}^{\sum_i^v r_i} \cdot g^{\sum_i^v v_i} = \mathsf{pk}^r \cdot g^s$. Here, we should have $s = k$.
(2) The prover samples a random r' from $\mathbb{Z}_q$, and computes a pair (A, B) with $A = g^{r'}$ and $B = \mathsf{pk}^{r'}$. They compute a challenge $\sigma = H(\mathsf{pk}, c_1, c_2, A, B)$. They then compute $r'' = r' + \sigma \cdot r$. They send/make available to the verifier A, B, c_1, c_2 and r''.
(3) The verifier derives $\sigma = H(\mathsf{pk}, c_1, c_2, A, B)$. They then verify that $g^{r''} = A \cdot c_1^{\sigma}$ and $\mathsf{pk}^{r''} = B \cdot \left(\frac{c_2}{g^k}\right)^{\sigma}$.

3.2.2 Explanation of Correctness. This ZKP is essentially the same as the ZKP for proving correctness from the Vote handout [2]. However, we have modified the r used for c_1, c_2 to be the sum of all the r_i of the individual votes instead of a randomly sampled number, as this allows for the verifier to independently verify the ciphertext provided is the sum of the votes by multiplying the individual vote ciphertexts together. This quantity is still effectively random, and gives the verifier no extra information, while working with our scheme.

3.2.3 Implementing the Sum ZKP in Practice. We created a new message, `VoteSumZKP_Struct`, to hold the values of A, B, c_1, c_2 and r. `VoterClient` was modified to hold, as a field, such a struct and a two new functions, `SaveVoteSumZKP` and `LoadVoteSumZKP` to save and load the ZKP into a file. `VoteRow` was also changed to include a `VoteSumZKP`. We also adjusted the tallyer code to include, by concatenating it, the `VoteSumZKP` as part of the message it signs. This way, we ensure the integrity of the `VoteSumZKP` on the database and in individual files.

3.3 Design Decisions

3.3.1 Changes to the Common Configuration. We modified the common configuration file, `common_config.json`, to include two new parameters, `num_candidates` and `num_votes` .They respectively define the total number of candidates taking part in the election, and k, the number of votes per candidate. We have changed all classes using `CommonConfig` accordingly, to store this information as fields.

3.3.2 Changes to Messages. We modified five messages (register message from voter to registrar, blind signature message from registrar to voter, vote message from voter to tallyer, `VoteRow`, and `PartialDecryptionRow`) to include a new `CryptoPP::Integer` field, corresponding to the candidate for which a given vote was casted. This information is crucial later to reorganize votes in order to combine them, but also to ensure votes are tallied properly in case they are not sent exactly in order.

3.3.3 Changes to VoterClient. We first modified the behavior of the Voter CLI to take exactly k votes. This way, we can specify k integers from 0 to *num_candidates* – 1 when registering with the registrar. We adjusted the `vote`, `vote_zkp`, `registrar_signatures`, and `blind` fields accordingly to hold arrays. To save vote information, we've also extended the `Save` helpers to take arrays as input, and store each element individually in a directory it creates in the input path. We've also updated the `Load` macros accordingly. Remaining functions which involve communication with other parties in the project were modified to account for multiple votes by sending and receiving a separate message for each vote. Finally `DoVerify` was adjusted to handle multiple votes iteratively and properly set up vectors as inputs to `CombineVotes` and `CombineResults`.

3.3.4 Changes to ElectionClient. We extended methods related to generating votes and verifying them to be able to handle multiple votes, mostly by having them take and return vectors of the initial inputs and outputs. More specifically:

- `GenerateVote` now takes a vector of k votes, from 0 to `num_candidates-1`, and generates a vector of `num_candidates` ciphertexts of either 0 or 1, based on whether the candidate number was present in the input vector. A second vector is returned with `num_candidates` ZKPs, each proving that their corresponding ciphertext is an encryption of either 0 or 1. Along with this, a third argument is returned - a `VoteSumZKP_Struct` - proving that the sum of all ciphertexts is k, as described in section 3.2.
- `VerifyVoteZKP` now takes a vector of ciphertexts, ZKPs, and a `VoteSumZKP_Struct`. By iterating through each ciphertext and ZKP, we are able to verify each ZKP individually as was done in the base version of the project. To verify the ZKP for the sum, we recompute the homomorphic sum using all ciphertexts, verify that it is consistent with the data in the sum ZKP, and use that as outlined in 3.2 to verify the proof.
- `CombineVotes` is modified to receive a $v \times n$ vector, with v the number of voters and n the number of candidates, and `vec[i][j]` corresponding to voter i's vote for candidate j. As an output, the method returns a length n vector, with each element j corresponding to the summed ciphertext for candidate j.
- `CombineResults` is modified to take in two vectors: a vector of `Vote_Ciphertext`, corresponding to the combined votes, and a 2D, $n \times v$ vector of partial decryptions, where `vec[i]` is the list of partial decryptions to decrypt the combined vote for candidate i.

3.3.5 Changes to ArbiterClient, RegistrarClient, TallyerClient. These classes have not undergone any significant change in logic, apart

from iterating through vectors and receiving or sending multiple messages (one per vote). The most notable change is the concatenation of the candidate number for the signature in `TallyerClient`, explained in more detail in section 3.2.2.

4 CHALLENGES

4.1 Design Challenges

4.1.1 Sending Multiple Messages. We were initially thinking of modifying data structures involving vote data to include vectors of votes. This way, we would avoid adding the `candidate_num` field and would track information based on the position of the vote in the array. We decided to go with the latter option, however, due to the increased difficulty of adjusting our serialization to fit vectors in the database and in files. Additionally, this design choice makes it easy to re-send individual votes, in case of a network communication error.

4.1.2 Keeping Track of Votes for Verification. A major challenge we encountered was to organize vote data properly to prepare inputs for `VerifyVoteZKP`. The function must take a 2D Vector of `VoteRow`, where each row corresponds to all votes from a single voter. However, our `VoteRow` structure does not include a field identifying the voter ID. We leveraged instead the uniqueness of the randomness r of each `VoteSumZKP` structure to solve this problem. Because the randomness r corresponds to the sum of each individual randomness r_i for a given voter, and given the very large range of values r_i can take, r would be extremely likely to be a unique identifier for voters. We've gathered votes into individual lists per candidate by using a `std::map` mapping from an r value to a list of `VoteRows`.

4.2 Implementation Challenges

4.2.1 Dealing with 2D Vectors. We ran into challenging bugs (segmentation faults) related to incorrect initialization of two-dimensional `std::vectors`. These bugs were addressed through proper use of the `reserve` and `resize` methods, allowing memory reservation and determination of a vector's size before elements are added to it.

4.2.2 Modifying Database Tables. With the addition of multiple votes for each voter, we had to edit the tables to have multiple primary keys, and modify the queries and insertions made. To assist with this, we looked up SQL tutorials and used a program called *DB Browser for SQLite* [3] to view the contents of the database.

5 EXPERIMENTS

5.1 Process

To conduct experiments, we followed the process outlined below:

(1) `cd` into the `build` file, and run "`cmake ..`". Then, run `make` to compile the binaries. Finally, in the root directory, run `mkdir disk`.
(2) Modify `config/common_config.json` to include the number of candidates (`num_candidates`) and maximum number of votes (`num_votes`) we want to experiment with.
(3) Run the `vote_registrar` binary on port 5000, with `config/registrar_config.json` and `config/common_config.json`.
(4) Run the `vote_tallyer` binary on port 6000, with `config/tallyer_config.json` and `config/common_config.json`.
(5) Run the `vote_arbiter` binary with `config/arbiter0_config.json` and `config/common_config.json`. Inside the CLI, run the command `keygen`. This will exit with an error.
(6) Run the `vote_arbiter` binary with `config/arbiter1_config.json` and `config/common_config.json`. Inside the CLI, run `keygen`.
(7) Exit the `vote_registrar` and `vote_tallyer` REPLs using `exit`, and repeat steps 3 and 4.
(8) Run the `vote_voter` binary with `config/voter0_config.json` and `config/common_config.json`.
(9) In the voter CLI, register with the registrar using `register localhost 5000 <your votes>`. The placeholder <your votes> should correspond to a space-separated string of candidate numbers, from 0 to `num_candidates-1`, for which you wish to vote. Then, send your vote to the tallyer using `vote localhost 6000`.
(10) Repeat step 9 with as many voters as you want.Make sure to use a different configuration file every time.
(11) Once voting is done, run the `vote_arbiter` binary with two different config files, as described in steps 5 and 6 (without running `keygen`). Then, for each arbiter, run the command `adjudicate`, which should combine votes and results, and publish them to the database.
(12) Finally, using the `vote_voter` binary with any configuration, run the `verify` command to verify the election. The total number of votes for each candidate should be printed in the console.

5.2 Valid Scenarios

5.2.1 Three voters, two candidates, one vote. As a sanity check, we ran this scenario to ensure the basic functionality was still there after making our changes. This (kind of) recreates the original scenario from the Vote project.

5.2.2 Three voters, ten candidates, five votes. This was a larger test of our changes. We tested this to make sure multiple votes for candidates could be counted, to ensure the votes were going to the correct candidates, and to make sure all the ZKPs and other data were being generated, stored, and verified properly.

5.2.3 Invalid Scenarios. Our project has checks at multiple points to ensure that invalid set of votes, for instance multiple votes for the same candidate, are rejected.

6 FUTURE WORK

A further extension could be to add the capability to require voting for at most k candidates, which requires a more complicated ZKP. Specifically, this involves implementing a ZKP to prove that a given ciphertext is either an encryption of $0, 1, 2, \ldots$ up to k. If we were to implement this, we would do so in addition to our current protocol for exactly k candidates, and we would include a parameter/option to choose between both for a given election.

We could also focus more on robustness in real world use cases; this would likely involve more responsiveness and adaptability to malicious actors/erroneously submitted votes. For example, we

might want to have an option for users to resubmit different votes before the election has finished, or clear their data from the database.

7 RESOURCES USED

We used `CryptoPP` and other libraries involved in the implementation of the drivers and other files from the Vote project. We also used DB Browser for SQLite[3] to view the database.

ACKNOWLEDGMENTS

A huge thank you to Professor Miao and the TAs for the course; the projects presented interesting concepts well and the final project especially was a very satisfying challenge.

REFERENCES

[1] David Bernhard and Bogdan Warinschi. 2016. Cryptographic Voting - A Gentle Introduction. *International Association for Cryptologic Research* (2016).
[2] CSCI1515 - Vote Spring 2024. https://cs.brown.edu/courses/csci1515/spring-2024/static/latex/projects/vote.pdf
[3] DB Browser for SQLite Accessed May 2024. https://sqlitebrowser.org/