CS1515 Final Project: Voting with Multiple Candidates

Anand Advani
anand_advani@brown.edu
Brown University
Providence, RI, USA

George Chemmala
george_chemmala@brown.edu
Brown University
Providence, RI, USA

Abstract

For our Applied Cryptography final project, we have implemented an extension of the Vote project in which each voter can vote for multiple candidates. We enforced the condition that each voter votes for exactly k out of n total candidates, where k and n are global constants (e.g. $k = 2, n = 6$). In order to do this, we modified the project infrastructure to accommodate each voter voting for multiple candidates and each candidate's votes being totaled separately. We also included a zero-knowledge proof (ZKP) to ensure that each voter voted for exactly k candidates.

1 DCP Background

1.1 Homomorphic Encryption

Definition 1.1 (Homomorphic Encryption). Homomorphic encryption is a form of encryption such that for and operation $\oplus$ on plaintexts, there exists an operation $\otimes$ on ciphertexts such that:

$$\text{Enc}(m_1) \otimes \text{Enc}(m_2) = \text{Enc}(m_1 \oplus m_2)$$

[5]

This is a handy trick that allows a agent to perform operations on our data as ciphertexts without being able to directly read it. And, we can see how this is useful in the context of voting since we preforming a tally of votes while keeping the votes private.

Example 1.2 (ElGamal Homomorphic Encryption). The encryption of a message m is given by:

$$\text{Enc}(m) = (g^r, pk^r \cdot g^m)$$

where g is a generator, r is a random number, $pk = g^{sk}$ is the public key, and sk is the secret key. The ciphertext can be added or multiplied to yield the same result as adding or multiplying the plaintexts. For example, if we have two ciphertexts $c_1 = (g^{r_1}, pk^{r_1} \cdot g^{m_1})$ and $c_2 = (g^{r_2}, pk^{r_2} \cdot g^{m_2})$, we can define the addition of ciphertexts as:

$$\begin{aligned} c_1 + c_2 &= (g^{r_1} \cdot g^{r_2}, pk^{r_1} \cdot g^{m_1} \cdot pk^{r_2} \cdot g^{m_2}) \\ &= (g^{r_1+r_2}, pk^{r_1+r_2} \cdot g^{m_1+m_2}) \\ &= \text{Enc}(m_1 + m_2) \end{aligned}$$

where the addition of the ciphertexts is component-wise multiplication. [3]

1.2 Threshold Encryption

In addition to homomorphic encryption, we also want to be able to split the decryption key among multiple parties. Not only will this prevent a single party from calling the election whenever they want, but it also allows for a more secure system since one party cannot decrypt the votes by itself. One attack on the system is that a malicious party could "call the election" throughout the election process and decrypt the votes examining the votes and potentially tracking the votes back to the voters.

This strategy of splitting the decryption key among multiple parties is known as threshold encryption [2]. In threshold encryption, a ciphertext can only be decrypted by a subset of parties, and the decryption key is shared among those parties. This allows for a more secure system, as no single party has access to the decryption key.

Example 1.3 (ElGamal Threshold Encryption). In ElGamal threshold encryption, n parties (arbiters) collaborate to generate a shared public key and distribute the decryption process. Each party i generates a keypair (sk_i, pk_i), where the public key is computed as: $pk_i = g^{sk_i}$, and sk_i is kept private. Each party publishes pk_i, and the combined public key is computed as:

$$pk = \prod_{i=1}^{n} pk_i = g^{\sum_{i=1}^{n} sk_i}.$$

This combined public key pk is used for encryption, while the corresponding secret key $sk = \sum_{i=1}^{n} sk_i$ is secret-shared among the n parties.

To encrypt a message m, the ciphertext is computed as $c = (c_1, c_2) = (g^r, pk^r \cdot g^m)$, where r is a random value.

To decrypt the ciphertext $c = (c_1, c_2)$, each party computes a partial decryption. Specifically, party i computes $d_i = c_1^{sk_i}$. The partial decryptions are then combined by multiplying them together:

$$\prod_{i=1}^{n} d_i = \prod_{i=1}^{n} c_1^{sk_i} = c_1^{\sum_{i=1}^{n} sk_i} = c_1^{sk}.$$

This result, c_1^{sk}, is used to decrypt the second component of the ciphertext:

$$g^m = \frac{c_2}{c_1^{sk}}.$$

[1]

2 Zero-Knowledge Proofs

A zero-knowledge proof (ZKP) is a cryptographic protocol that allows one party (the prover) to prove to another party (the verifier) that they know a value w without revealing any information about w itself [4]. A ZKP of knowledge has these 5 properties:

- **Completeness:** If the statement is true, there exists a proof that proves it is true.
- **Soundness:** If the statement is false, any proof cannot prove it is true.
- **Proof of Knowledge:** If a prover P^* can prove, then they must know w.
- **Honest-Verifier Zero-Knowledge (HVZK):** An honest verifier does not learn anything about w.

- **Zero-Knowledge:** A malicious verifier does not learn anything about w.

This protocol will be useful in our voting system to ensure that each voter votes either 0 or 1 for each candidate, and that they vote for exactly k candidates. The ZKP will allow the voter to prove to the verifier that they have voted for exactly k candidates without revealing which candidates they voted for.

However, one of the limitations of ZKPs is that they can be interactive, meaning that the prover and verifier must communicate back and forth to complete the proof. This can be a problem in some scenarios, such as in our voting system where we want to ensure that the votes are private and cannot be traced back to the voter. If we did not have a way to make the ZKP non-interactive, each of the agents in out problem would have to communicate with the orgional voter to verify the proof. This would be a problem in our voting system, as we want to ensure that the votes are private and cannot be traced back to the voter.

2.1 Non-Interactive Zero-Knowledge Proofs

Definition 2.1 (Random Oracle Model). The random oracle model (ROM) is a theoretical model in which a hash function is treated as a random oracle. In this model, the hash function behaves like a random function, meaning that it produces random outputs for each unique input. The prover and verifier have access to this hash function, and the security of the protocol relies on the assumption that the hash function behaves randomly.

We can use the Fiat-Shamir heuristic and the random oracle model to transform a three-round sigma protocol into a non-interactive zero-knowledge proof (NIZK) [6]

In a three-round sigma protocol,

(1) The prover sends a message to the verifier setting up the proof.
(2) The verifier sends a challenge to the prover.
(3) The prover sends a response to the challenge.
(4) The verifier checks the proof using the challenge and response.

This protocol can be transformed into a NIZK by replacing the verifier's challenge with a hash of the first two messages.

Now our protocol looks like this:

(1) The prover sends a message to the verifier setting up the proof.
(2) The prover computes the challenge as the hash of the first message and sends it to the verifier.
(3) The prover sends a response to the challenge.
(4) The verifier checks the proof using the challenge and response.

This insures that the prover does not have to interact with the verifier, and the verifier does not have to interact with the prover. Moreover, the proof is secure against a malicious prover and verifier. The prover cannot predict the challenge, and the verifier cannot influence the challenge to bias the proof in their favor. This ensures that the proof is secure against a malicious verifier and maintains the zero-knowledge property.

3 k-candidate ZKP

The votes that the voter sends to the tallyer are of the form:

$$(g^{sk}, g^{r_1}, g^{sk\cdot r_1+v_1})$$
$$(g^{sk}, g^{r_2}, g^{sk\cdot r_2+v_2})$$
$$\vdots$$
$$(g^{sk}, g^{r_n}, g^{sk\cdot r_n+v_n})$$

where n is the number of candidates, v_i is the vote for candidate i, and r_i is a random number.

Therefore, when we multiply the votes together componentwise and preseve first component, we get

$$(g^{sk}, g^{r}, g^{sk\cdot r+\sum_{i=1}^{n} v_i})$$

where $r = \sum_{i=1}^{n} r_i$

This now looks like a single ElGamal ciphertext but we have the extra information about the sum of the votes. Now we can divide the last component by g^k, where k is the number of candidates that we require the voter to vote for, and check if the result is an Elgamal ciphertext to verify that the voter voted for exactly k candidates.

We can use a Non-Interactive Zero-Knowledge Proof (NIZK) to prove that the voter voted for exactly k candidates. This will mirror the same ZKP that we used to prove that a vote is an encryption of a 0 or 1.

(1) Our voter sends all of their votes to the tallyer.
(2) The voter adds a ZKP to prove that they voted for exactly k candidates:
  (a) The voter creates a random number $r' \leftarrow \mathbb{Z}_q$
  (b) The voter sends $(g^{r'}, pk^{r'})$ to set up the zkp challenge.
  (c) The voter computes the challenge as
  $$\sigma = H(g^{r}, pk^{r}, g^{r'}, pk^{r'})$$
  where H is a hash function
  (d) The voter sends $r'' = r' + \sigma \cdot r$ to the tallyer
(3) Now the tallyer can verify the proof:
  (a) The tallyer computes $g^{r'} \cdot (g^{r})^{\sigma}$ and checks if it is equal to $g^{r''}$
  (b) The tallyer computes $pk^{r'} \cdot (pk^{r})^{\sigma}$ and checks if it is equal to $pk^{r''}$
  (c) If both checks pass, the tallyer can be sure that the voter voted for exactly k candidates.
(4) The tallyer can now strip the signature and continue with the voting process.

4 Vote Architecture Summary

Since the Vote project is already a cryptographically secure voting platform, we will be building on top of the existing Vote project.

We will build off of this existing architecture and modify it to allow for multiple candidates. The main changes we will make are that each vote will contian subvotes for each candidate which will be tagged with the candidate's index. Essentially, we will be running n elections - one for each candidate.

(1) **Registrar:** The registrar will check that all voters are registered to vote only once. For each voter, they will issue a certificate for their verification key for each candidate. The

registrar will also check that each voter submits a vote for all n candidates.

(2) **Tallyer:** The tallyer will post votes on a public bulletin board. They will check that the signature is valid and strip the signature from the vote. It will do this for each subvote for each candidate. The tallyer will also check that the ZKP is valid and that the voter submits a vote for all n candidates.
(3) **Arbiters:** The arbiters will generate the election parameters and decrypt the final result. They will generate the threshold encryption keys. There will be t arbiters and each will have their (pk_i, sk_i). They all reveal pk_i to the public, so that everyone can compute the full public key pk.
(4) **Voters:** The voters will be able to vote, view, and verify the final result. They will encrypt their votes using the public key. The voter will sign this vote using their signing key. They will send this vote to the tallyer. The voter will also generate a ZKP to show that they voted for exactly k candidates.

5 Project Overview

In this project, we extended the Vote project to allow voters to vote for multiple candidates while ensuring that each voter votes for exactly k out of n candidates. To achieve this, we modified the existing cryptographic voting infrastructure to handle multiple subvotes per voter and incorporated zero-knowledge proofs (ZKPs) to enforce vote validity.

The key components of our implementation are:

- **Voting Mechanism:** Each voter submits a vector of encrypted votes, where each entry corresponds to a candidate and is either 0 or 1. We used homomorphic encryption to preserve vote privacy while enabling the tallying process.
- **Zero-Knowledge Proofs:** Voters generate ZKPs to prove that their votes are valid (i.e., each vote is either 0 or 1) and that they voted for exactly k candidates, without revealing which candidates they selected.
- **Threshold Encryption:** To enhance security, the decryption key is split among multiple arbiters. This ensures that no single party can decrypt the votes, and the final tally is decrypted collaboratively.

This project highlights how cryptographic techniques like homomorphic encryption, threshold encryption, and zero-knowledge proofs can be combined to build a secure and privacy-preserving voting system. The system ensures that votes remain private, voters cannot cheat by voting for more than k candidates, and the final results are verifiable by all participants.

6 Design Decisions and Difficulties

In designing the extension of the Vote project to support multiple candidates, several key decisions had to be made to ensure correctness, security, and maintainability. Below, we outline the major design choices and the challenges encountered during implementation.

6.1 Design Decisions

6.1.1 Vote Representation. We decided to represent each voter's submission as a group of encrypted subvotes, where each subvote corresponds to a candidate. This approach allowed us to leverage homomorphic encryption to tally votes for each candidate independently. Additionally, we included a zero-knowledge proof (ZKP) for each vote to ensure that it was either 0 or 1. A separate ZKP was used to prove that the sum of the votes in the vector equaled k, enforcing the constraint that each voter votes for exactly k candidates.

6.1.2 Candidate Tracking. To track votes for individual candidates, we added a candidate identifier field to the database for both the Vote and Partial Decryption tables. This allowed us to associate each subvote with its corresponding candidate and ensured that the tallying and decryption processes could handle votes for multiple candidates seamlessly.

6.1.3 Message Serialization. We modified the message serialization format to include the candidate identifier alongside each vote. This ensured that the tallyer and arbiters could correctly process votes for different candidates without ambiguity. The serialization format was updated to handle vectors of votes and ZKPs, rather than single votes.

6.2 Challenges and Bugs

6.2.1 Refactoring. One of the most monotonous aspects was updating the voter code to handle vectors of votes and ZKPs. Initially, there were several instances where the code still referenced single votes (e.g., `this->vote` instead of `this->votes`). These references caused votes to be overwritten and led to errors during the unblinding process.

6.2.2 Serialization Errors. During the implementation, we encountered issues with serializing and deserializing the candidate identifier. Specifically, the use of `put_integer()` and `get_integer()` functions, which operate on `CryptoPP::Integer` types, led to subtle bugs. These were resolved by ensuring consistent handling of candidate identifiers across all serialization and deserialization routines.

6.2.3 Database Schema Updates. Initially, the database schema did not include fields for candidate identifiers in the Vote and Partial Decryption tables. As a result, the tallyer correctly inserted votes for multiple candidates, but the arbiter read all votes as belonging to the same candidate. Adding candidate fields to the schema and updating the corresponding database functions resolved this issue.

6.2.4 ZKP Verification. The ZKP for the k-candidate constraint required careful implementation to ensure that the sum of the votes was checked without revealing individual votes.

6.3 Lessons Learned

This project highlighted the importance of modular design and rigorous testing when extending cryptographic protocols. By isolating changes to specific components (e.g., vote representation, database schema, ZKP verification), we were able to incrementally build and debug the system. Additionally, the use of logging to the console and database allowed us to trace the flow of data and identify issues more effectively.

7 Results and Conclusions

We successfully extended the Vote project to support multiple candidates while ensuring that each voter votes for exactly k candidates. The system is built on a foundation of homomorphic encryption, threshold encryption, and zero-knowledge proofs, providing a secure and privacy-preserving voting mechanism.

The final implementation allows voters to submit encrypted votes for multiple candidates, with ZKPs ensuring the validity of each vote and the total number of votes. The use of threshold encryption ensures that no single party can decrypt the votes, enhancing the security of the system. The project demonstrated the feasibility of combining cryptographic techniques to build a secure voting system that can handle multiple candidates and enforce constraints on voter behavior.

Acknowledgments

Thank you to Peihan for a wonderful semester, and to the TAs for all the grading and feedback!

Also, thank you to our goats Aryan Singh and Siddu Sitaraman!

References

[1] Alfredo De Santis, Yvo Desmedt, Yair Frankel, and Moti Yung. 1994. How to share a function securely. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing* (Montreal, Quebec, Canada) *(STOC '94)*. Association for Computing Machinery, New York, NY, USA, 522–533. doi:10.1145/195058.195405

[2] Yvo Desmedt and Yair Frankel. 1990. Threshold cryptosystems. In *Advances in Cryptology — CRYPTO' 89 Proceedings*, Gilles Brassard (Ed.). Springer New York, New York, NY, 307–315.

[3] T. Elgamal. 1985. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory* 31, 4 (1985), 469–472. doi:10.1109/TIT.1985.1057074

[4] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. 1989. The Knowledge Complexity of Interactive Proof Systems. *SIAM J. Comput.* 18, 1 (1989), 186–208. doi:10.1137/0218012 arXiv:https://doi.org/10.1137/0218012

[5] Ronald L Rivest, Len Adleman, Michael L Dertouzos, et al. 1978. On data banks and privacy homomorphisms. *Foundations of secure computation* 4, 11 (1978), 169–180.

[6] Huixin Wu and Feng Wang. 2014. A Survey of Noninteractive Zero Knowledge Proof System and Its Applications. *The Scientific World Journal* 2014, 1 (2014), 560484. doi:10.1155/2014/560484 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1155/2014/560484

Received 9 May 2025