INF-3315 Assignment 2 - Shamir's Secret Sharing Algorithm

Kevin Mathias Bergan

October 7, 2025

1 Introduction

This report presents the implementation of a privacy-preserving voting protocol based on the additive homomorphism of Shamir's Secret Sharing (SSS) scheme [Sha79]. The objective of the assignment was to simulate a secure election process in which individual votes remain confidential while still allowing a group of authorities to collaboratively compute the final tally.

The protocol ensures that no single authority can access any voter's private choice but instead each vote is divided into multiple shares distributed among the authorities. Only when a sufficient number of shares are combined can the total result be reconstructed. This approach takes advantage of the mathematical properties of secret sharing to maintain both privacy and integrity of the election outcome. The scheme exploits the Lagrange interpolation theorem, specifically that k points on the polynomial uniquely determines a polynomial of degree less than or equal to $k-1$. For instance, 2 points are sufficient to define a line, 3 points are sufficient to define a parabola, 4 points to define a cubic curve and so forth.

My implementation is developed in Python, using the Galois library to perform finite-field arithmetic. With this implementation there is also a unit-test file that checks certain requirements my program must meet which is described in the assignment text.

1.1 Example

> A company needs to secure their vault. If a single person knows the code to the vault, the code might be lost or unavailable when the vault needs to be opened. If there are several people who know the code, they may not trust each other to always act honestly.
>
> SSS can be used in this situation to generate shares of the vault's code which are distributed to authorized individuals in the company. The minimum threshold and number of shares given to each individual can be selected such that the vault is accessible only by (groups of) authorized individuals. If fewer shares than the threshold are presented, the vault cannot be opened.
>
> By accident, coercion or as an act of opposition, some individuals might present incorrect information for their shares. If the total of correct shares fails to meet the minimum threshold, the vault remains locked. (Example from Wikipedia [Sha79]).

2 Implementation

My implementation consists of 2 main functions *cast_vote* and *tally_votes* and one initialization call. My implementation has some assumptions such as:

- A vote can't be traced back to a voter ID, which keeps the votes private.
- A voter also can't prove how they voted.
- It's impossible to verify individual votes.

2.1 Example Round

To start of a round of my voting protocol you first have to start be agreeing on some factors. The first element in this protocol is to choose a large prime number. This is to ensure that the finite field is large enough such that the sample space, which the shares for each vote live, does not overlap. In mathematical terms, this means that we need to choose a finite q such that $q = p^r$ where p is our prime and where $q > n$. I have chosen the 11th Mersenne prime which has the value $2^{127} - 1$.

Next, we need to agree on how many authorities we trust. In our example, we will go for 3 trusted authorities. These authorities are entities that can count the votes each participant, which are not an authority, in the voting round. Lastly, we need to choose how many participants we would like to have in our voting round. In our example, we will go for 5 participants.

Generate Authority Keys: We start by generating keys that are public keys for each authority. My implementation does this pretty naively, where each key is just an integer incremented by one for each authority.

Participants Cast Votes Now all participants encode their vote (either 1 or 0) using a SSS polynomial p_n of degree $k - 1$. The polynomial is created by the function *generate_polynomial*. In our case, the function creates a polynomial with three parts, a quadratic term, a linear term, and a constant term ($k - 1 = 2$). The constant term is the actual vote itself. This polynomial is then used with the authority key to generate a share for that particular authority.

All calculations involving the polynomial must also be calculated over the field instead of over the integers. Both the choice of the field and the mapping of the secret to a value in this field are considered to be publicly known [Sha79].

Tally Votes Final step is to gather all votes and distribute each vote associated with each of the authorities. Then they must sum the shares to create a total. Since we are working over a field, they need to take the modulo given the prime they agreed on (which in our case is $2^{127} - 1$). Then we use the Lagrange interpolation theorem to reconstruct the polynomial $p(x)$ that contains the number of votes represented as the constant term.

3 Discussion

The protocol works as long as not all of the k authorities are corrupt. If they all were, they could collaborate to reconstruct each voter's polynomial $P(x)$ and possibly even alter the votes. The protocol only needs at least $t + 1$ honest authorities to function correctly. Therefore, when there are $N > t + 1$ authorities, up to $N - t - 1$ of them can be corrupted without breaking the system. This gives the protocol a certain level of robustness and fault tolerance [Sch99].

Each vote is submitted along with a voter ID, allowing the protocol to verify that only legitimate voters have voted. The protocol also prevents corruption of votes, since each authority only has a small share of the data and doesn't know how changing their share would affect the final result, so there's no real reason or incentive to tamper with it [Sch99].

3.1 Vulnerabilities / Weaknesses

A voter can't be totally sure their vote was recorded correctly. Likewise, the authorities can't be certain all votes are legal and fair, for instance, a voter could pick a value that isn't a valid option, like 50 or 100, which could skew the results in their favor [Sha79]. SSS also doesn't provide verifiable secret sharing, meaning there's no built-in way to check that each share is correct during reconstruction. This opens the door for dishonest participants to submit fake shares without getting caught. On top of that, SSS has a single point of failure: the secret has to exist in one place when it's split into shares and again when it's put back together. These points can be targeted by attacks, whereas other schemes, like multisignature protocols, can at least remove one of these vulnerable spots.

References

[Sch99] Berry Schoenmakers. A simple publicly verifiable secret sharing scheme and its application to electronic voting. 1999.

[Sha79] Adi Shamir. Shamir's secret sharing. 1979.