Secret Key Sharing Dynamically

Inbal Mishal
Hebrew University of Jerusalem
inbal.mishal@mail.huji.ac.il

Reut Ora Oelbaum
Hebrew University of Jerusalem
reutora.oelbaum@mail.huji.ac.il

Amit Roth
Hebrew University of Jerusalem
amit.roth@mail.huji.ac.il

***Abstract*—In our project, we implemented a verifiable and dynamic secret key sharing system based on Shamir secret key sharing [3]. our system supports secret key sharing, dealer-free dynamic threshold [2] and dealer verification [1].**

I. Introduction

Our system will be capable of supporting threshold modification dynamically without the need of a dealer regenerating and distribution of points. In order to support this capability we uses the analyses published by [2]. The system also supports dealer verification, to prevent distribution of false points, implemented using Feldman's scheme for VSS (Verifiable secret sharing) [1]. The system implemented in python and the connection is made by sockets. You can connect to the entities using the CLI or add your own code to the main.

II. The Attacker

We assume that all the members are friendly and does not have any malicious intentions. We assume the presence of an attacker in our network and the ability of occasionally revealing a C_v which is the share of information which each member poses. We assume that the only data that can be leaked from each member is the C_v, this assumption is reasonable because the points only made for self calculation and the only data that the member shares with the environment is the encrypted C_v. When the threshold changes, the member calculates C_v according to the new threshold. We doubt the reliability of the dealer and because of that we verify the dealer using Feldman's scheme as will be explained below.

III. The Solution

Our project is the implementation of 2 articles proposing methods to deal with our problems: dealer-free dynamic threshold change and verification of the dealer. We will describe shortly on each solution, full analysis can be found in the original articles.

A. Dealer-Free Dynamic Threshold

For n users and threshold t, the dealer chooses r polynomials with degree $t-1$. Each user has a corresponding x_i and he gets list of points from each polynomial at point x_i. For a current threshold l each user calculates his own share of information:

$$C_v = \sum_{i=1}^{r} a_i h_i(x_{w_v}) \Pi_{j=1, j\neq v}^{l} \frac{i - x_{w_j}}{x_{w_v} - x_{w_j}} modp$$

Whenever the members chooses to increase the threshold, each user deletes his own C_v and calculates the new C_v.

B. Verifiable Dealer

Feldman's scheme makes it possible to verify the credibility of the dealer. In general, the dealer can give some users incorrect points - which later will not allow the secret to be properly restored. Feldman's [1] scheme brings an interface that allows the user to verify the correctness of his points. This is done using the following calculation:

$$C_i = (g^{a_0})(g^{a_1})^{x_i} \dots (g^{a_{t-1}})^{x_i^{t-1}} = g^{y_i}$$

G is the power of the corresponding polynomial coefficient and it is sent modulo Q. In addition G, P, Q were selected such that according to the discrete log problem the coefficients of the polynomial are sufficiently difficult to reconstruct. In our implementation we used an object called g_matrix - a matrix in which each row corresponds to the g in power of the coefficients of one of the polynomials. g, p, q are constants who maintain the following connections: p, q primes such: $p|q-1$ g is a generator of a cyclic group G of prime order q. The group G chosen such that computing discrete logarithms is hard in this group.

IV. API

We implemented a rich API for the members, dealer and validator using python sockets. In order to communicate each server can use the CLI as well as calling the commands directly from a main file.

A. Member

1) Client Side: When member connects to our system, he has several actions he can choose from (by choosing a number):

- Get data from dealer - Here, the member connects to the dealer and gets the public parameters: t, n, a_coeff and g_matrix. He gets his shares also. The dealer save the member details in order to send them to the other members later.
- Send a voting requests to others
- Increase the threshold
- Check g_matrix - Each member can compare the g_matrix he got with other members (after everyone connected to the dealer).

2) Server Side: Each member listening to other entities. As a result, he performs action according to the request:

- Save members details

- Vote - The member gets voting request and he votes [y/n] (yes or no).
- Wait for votes - After the member sent a vote request, he waits for others votes.
- Increase threshold - increase the threshold according to others request.
- Send g_matrix - The member sends his own g_matrix to other member.

B. *Dealer*

The dealer generate a_coeff and g_matrix. He also sends the secret hash to the validator. In addition, he generates shares for each member and send the validator's public key to the members.

Once he sends all of the details, the system doesn't use the dealer anymore.

C. *Validator*

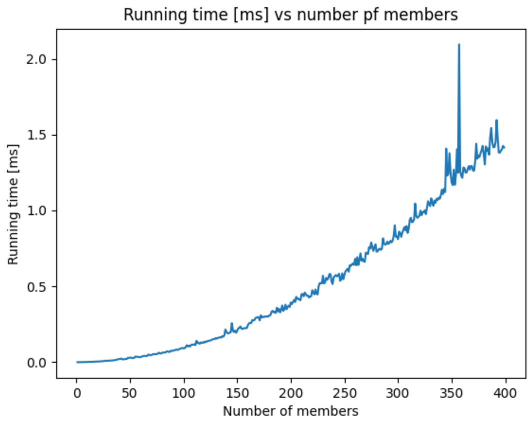
At the beginning, the validator connects to the dealer and send him his public key, IP and port. He gets the secret's hash from the dealer. After that, he listens to the members and waits for secret validation request. When he gets this request, he decrypts the C_v 's using it's private key, calculate the secret and compare the secret's hash it to the original secret's hash he has.

V. BENCHMARKS

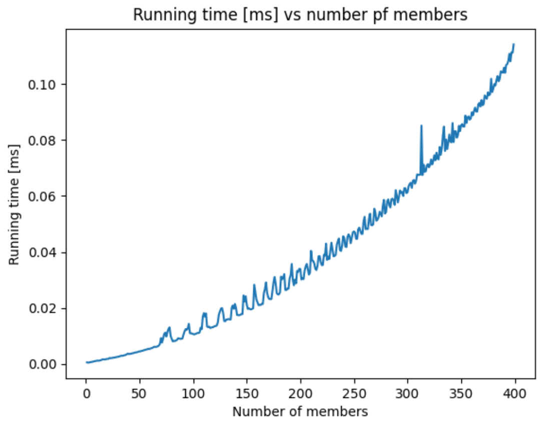
The benchmarks preformed without the python sockets interface and all entities were run from the same machine. The code can be found at the project files in Github. We will test the performance of the system while creating points by the dealer, calculating C_v's and reconstructing the secret

A. *Changing Members*

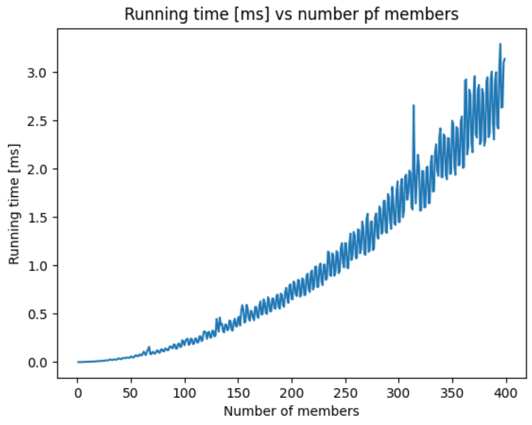
For q, p 10^6 We will get the results in order to see the scalability of the algorithm. For fixed number of threshold $t = 2$



For changing threshold $t = \frac{n}{2}$

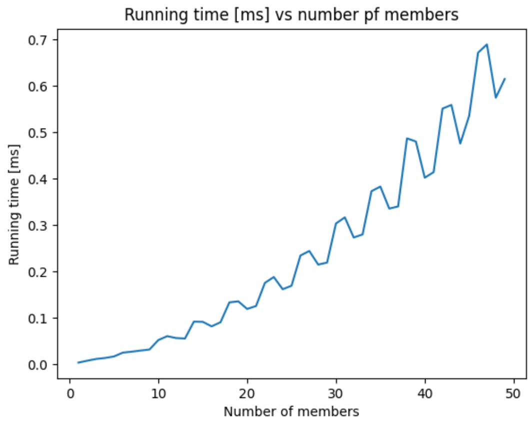


For changing threshold $t = \frac{n}{4}$ and new threshold $t = \frac{n}{2}$



B. *Reconstructing The Secret*

We will measure the time to construct a real world secret of 256 bits (10^{77}).



VI. Summary

In our project we implemented Shamir's schemes with some extensions. The first extension is a mechanism that allows the threshold to be raised dynamically. This mechanism allows each of the users to raise the threshold - and thus actually raise the degree of consent required to reconstruct the secret. Another extension is the use of Feldman's scheme - which makes it possible to each member to verify the points received by the dealer. As we saw at the benchmarks, our system achieves impressive results in real world applications, scalable and reliable.

References

[1] Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, pages 427–438, 1987.

[2] Chingfang Hsu, Lein Harn, Zhe Xia, and Maoyuan Zhang. Non-interactive dealer-free dynamic threshold secret sharing based on standard shamir's ss for 5g networks. *IEEE Access*, 8:203965–203971, 2020.

[3] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

VII. Appendix I - Run Project

There are 2 ways to interact with our system. One is deploying with the help of the sockets interface, and the second is using it without the sockets as a python package. Note that anyway you choose you need to supply p, q, and g in $settings.py$ and t and n as parameters for the Dealer.

A. Run Using Python Sockets

The entities in the system are: n members, dealer and validator. You need to run the files in the next order: dealer - validator - members. The dealer needs to run first because when you run the validator he connect to the dealer and send him his details. Then, the members need to create connection with the dealer by pressing (1). After that, you can use the system according to the API.

B. Run As a Package

You can use the code as a python package and ignore the sockets interface. Create a Dealer object

```
dealer = Dealer(t, n)
dealer.generate_a_coeff_list()
dealer.generate_polynomial_list_and_g_matrix()
```

and you can also call

```
dealer.share_generation()
```

In order to get the share before the shares distribution.

In order to create a Member object execute the , code:

```
member = Member()
member.set_parameters(t, n,
    a_coeff, x_arr, g_matrix, points)
```

You can change the threshold and calculate the C_v using

```
member.calculate_cv()
```

Create a Validator object using the following code:

```
validator = Validator()
validator.set_parameters(t, a_coeff, hash)
```

and reconstruct the secret with enough C_v's.

VIII. Appendix II - The Code

All the code is hosted and publicly shared on Github.

IX. Appendix III - Mathematics in Finite Fields

All of the calculations in the code are preformed inside a finite field. We implemented the basic math operations and Lagrange interpolate optimized and modified for finite field arithmetic inside $utils\_finite\_field\_helper.py$.