Security Hand-in 2

atro
Andreas Severin Hauch Trøstrup

October 31 2023

1 Introduction

In this assignment, we were tasked with implementing a solution allowing three parties, and a central server to communicate secret values securely, using Secure Multiparty (Three party) Computation. I am basing my solution on the related slides, and the provided lecture notes on Secure Three Party Computation.

2 The problem and the adversarial model

Alice, Bob and Charlie all hold some secret data, that a central server, in this case the Hospital needs to know. However, since the users don't want to share their secret data, with each other or the hospital, we need some other method. It turns out the hospital only needs an aggregate value of all the clients secret values. Thus, we can use secure multiparty computation and additive secret sharing to securely share the data with the hospital.

In this scenario, the clients, Alice, Bob and Charlie, all follow the protocols we establish. However, we can't be certain that they are honest, and we can't be certain that they won't try to monitor or manipulate the data transferred through the network, as is possible in the Dolev-Yao model. This means the adversarial model is the "Passive" model. Because of this, we need some method to ensure the confidentiality, integrity and authenticity of the messages sent through the network.

3 The building blocks of my solution

3.1 Additive secret sharing

To ensure that no other clients learn the secret value a client holds, we use additive secret sharing. In additive secret sharing, a secret value (in this case, an integer value), is split up into shares, and the shares are distributed such that no other party holds enough shares to calculate the secret value.

In our case, the secret value is some integer value $s \in \mathbb{F}_q$, where $\mathbb{F}_q$ is a field of order q with operations $+$. Now, if we want to split our secret value into n shares, (in our case 3), we do the following:

- Pick, at random, $s_1, \ldots, s_{n-1} \xleftarrow{\$} \mathbb{F}_q$. That is, we pick random values from the field.
- Calculate the final share $s_n = s - \sum_{i=1}^{n-1} s_i$

The last share s_n is kept as the client's secret share. The other shares are sent to the other clients. This way, no other clients hold all shares of the secret value, and as such can't calculate the actual secret value.

If client 1 holds shares x_1, x_2, x_3, from a secret x, the client then sends share x_2 to client 2 and x_3 to client 3. In return, it receives shares y_1 and z_1, one from each client. Then, that client's aggregate value is calculated as $out_1 = x_1 + y_1 + z_1$. This value is then sent to the central server, the hospital, who in the end can calculate the final aggregate value as $out_1 + out_2 + out_3$. This value can be trivially proven to be the same value as the sum of the original secret values x, y, z, but we have now shared no information about the individual parts of the sum.

3.2 TLS - Transport Layer Security

Because of the adversarial model described, we need to ensure confidentiality, integrity and authenticity of the messages sent through the network. This can be done by using TLS, which uses certificates to ensure these three factors. TLS ensures confidentialty by encrypting messages with symmetric-key encryption, integrity with the use of MAC (message authentication codes) and authentication using public-key cryptography. A TLS certificate is confirmed to be authentic by a trusted Certificate Authority.

4 Implementation

I have written my implementation in Ruby, using the `sockets` and `openssl` libraries.

As stated in the previous section, we need to use TLS to ensure the security of communication. For TLS certificates to be trusted, they need to be verified by a trusted Certificate Authority. Since I am not, in fact, a trusted Certificate Authority, my certificates are self-signed. This means they are invalid in the real world, but they will suffice for this demonstration. Because all clients will be acting as servers (to accept connections from the other clients), all clients need their own certificate. Thus, I have generated four certificates, one for each client and one for the hospital. These are found in the `certs` folder, along with their private keys. In reality, these should of course be kept secret, and I shouldn't be sharing them – but to make all our lives easier I have included them in the folder.

For the value q, which is the order of the field $\mathbb{F}_q$, I have picked the value 47. The secret values of the clients are random integers in this field.

My implementation then follows the following procedure:

- A client is generated with id 0, 1 or 2. At initialization, the clients randomly generates its secret value. It then generates the shares as described in section 3.1.
- Using the `sockets` library, the client creates TCP websockets to communicate with the other clients. The websockets are wrapped in SSL sockets, to ensure security. This is done using the `openssl` library.
- Using the secure sockets, the clients communicate their shares to each other.
- The clients calculate their intermediate aggregate value and send this to the Hospital, again using the SSL-secured websockets.
- The hospital calculates the final aggregate value.

By following this procedure, the hospital ends up with the sum of all the clients secret values, but has no way of knowing what parts of the sum came from which client. The clients themselves have also not shared enough information between them to be able to calculate the actual sums – they each only have a single share from the other clients – and as such, we have succeeded in communicating without sharing any confidential information. Using TLS-protected websockets, we have ensured secure communication while doing this.

5 Specifics and running the implementation

To run the implementation, you need to have Ruby installed. I have only tested with Ruby 3.2.2, but it should work for any >3 version.

In the project folder, three Ruby files are present:

- `client.rb` – contains the code for a secret-holding Client. Run with:
  `ruby client.rb [options] id`
  where `id` is either 0, 1 or 2. You can supply the `-v` option to increase logging.
- `hospital.rb` – contains the code for the hospital, the central server that calculates the final aggregate value. Run with:
  `ruby hospital.rb [options]`
  Also accepts `-v` option.
- `ssl_helpers.rb` – contains helper functions for making SSL connections.

To run the implementation, you need to start the hospital and the clients 0, 1 and 2 in separate terminals. The clients will automatically establish connection to each other when run. They use ports 4747, 4748, 4749 and 5000 for the hospital.

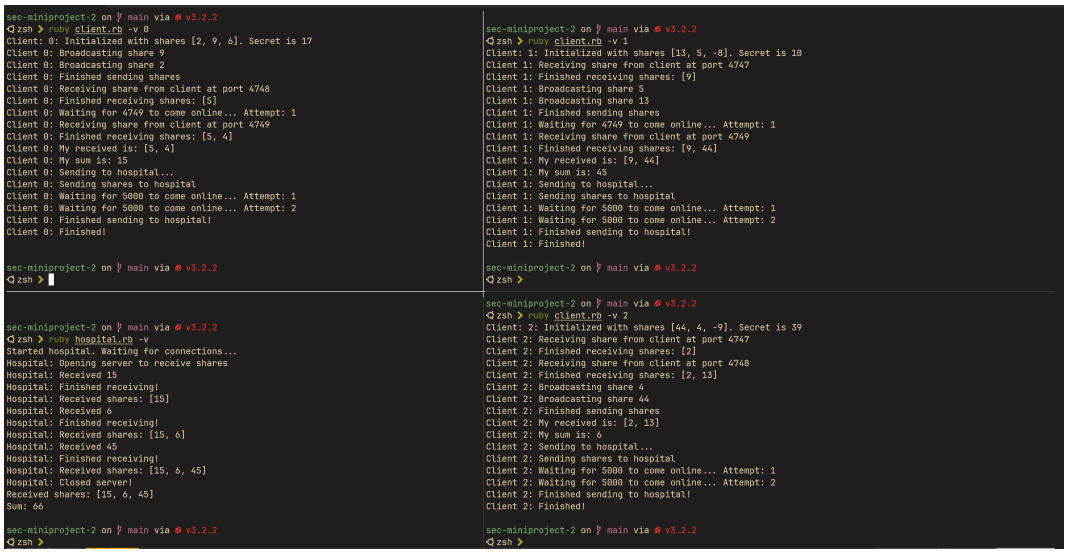

Figure 1: An example of running the implementation in four different terminals.