Security fall 2023

Nicolai Peter Seloy

Mandatory Hand-in #2

October 30, 2023

Contents

1 Adversarial model

I assume a static honest But curious adversarial model. Which means my users will only try to input in the terminal and not deviate from the intended behavior. There are however, some guards in the program to protect from wrongful input.

1.1 Building blocks

Technologies used used are Golang with gRPC and OpenSSL for this hand-in. My implementation is loosely based on my previous work in the distributed systems course using my own work. [1] [2]

Architecture

I chose a peer to peer architecture for all clients and then connected them all to a central server representing the hospital. This means that all clients secretly can communicate with each other and then all send information to the same central hospital server, which has no information about their connection. I have made a figure showing this with arrows going to and from every user but only arrows going to the hospital.

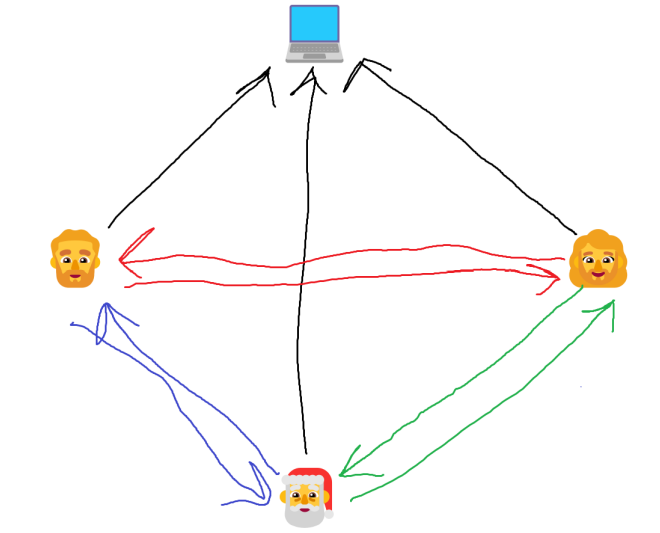

Figure 1: System architecture

Server

The server is responsible for setting up a server using TLS and opening connections for users that have a correct certificate using the generated cert.pem & key.pem that OpenSSL generates. The commands can be found in the

README. The server is also responsible for keeping track of who has sent their values and calculating the aggregate sum. This is all done in the SendPersonalInfo function.

Client

Several things are happening in the client. Firstly, it is both responsible for setting up a secure p2p connection with all other clients along with connecting to the hospital. I am using TLS connections with self-signed certificates for both of them. The important function for handling Multi-Party Computation is MPCScramble. The numbers returned by scramble are shared such that each user receives a share. shares are contained in a map where each name maps to a value, this makes it so that only the newest value from a user will be sent to the hospital. When a user sends their sum to the hospital, it will return a boolean showing whether everyone else has sent their value.

2 Why is it secure

Secure Multi-Part Computation

The requirements for multi-part computation is a that another user only receives a share of the value. i.e. *100* is split into *20,50,30* and that the hospital only learns the sum of three shares from a user.

The logic for splitting shares is done by the following code:

```
func MPCScramble (number int) (first,second,third int) {
   if(number == 0){
      return 0,0,0
   }
first = rand.Intn(number)
if(randBool() || first >= number-3){
   first = first *-1
}
for {
gen := rand.Intn(number)
if(gen + first < number){
   second = gen
   if(randBool()){
      second=second*-1
      }
   third = number - first - second
   return
   }
}}
```

To split a number into random parts I first take an integer between 0-input, randbool returns a random boolean which I use to flip the number into a negative to give me more randomization. After selecting the first number the second

number is selected randomly and then checked if the sum of first and second is lower than the total, this check runs until the condition is satisfied. the last number is selected by calculating the remainder for number - first - second. When sharing, a user will always set n1 for themselves and distribute n2 and n3 in ascending order for port numbers.

This makes it so that a user will only ever know a fraction of what another sent.

Transport layer security

To protect against adversaries on the network TLS is implemented using the following code

Setting up server

```
func loadCerts() credentials.TransportCredentials {
   cert, err := tls.LoadX509KeyPair("cert.pem", "key.pem")
   creds := credentials.NewTLS(&tls.Config{Certificates:
       []tls.Certificate{cert}})
   return creds
}

func startServer (server *Server) {
   grpcServer := grpc.NewServer(grpc.Creds(loadCerts()))
   listener, err := net.Listen("tcp", ":"+strconv.Itoa(server.port))
}
```

Client accessing server

```
func loadCertsClient() *x509.CertPool {
   certPool := x509.NewCertPool()
    caCert, err := os.ReadFile("cert.pem")
    certPool.AppendCertsFromPEM(caCert)
    return certPool
}

func SetupHospitalConnection(client *Client) {
   conn, err := grpc.Dial(fmt.Sprintf(":%v", *serverPort),
       grpc.WithTransportCredentials(
       credentials.NewClientTLSFromCert(loadCertsClient(), "")))
   client.hospitalConnection = proto.NewHospitalClient(conn)
}
```

This shows that my server is set up using a TLS certificate and only accepting connections on that. For client accessing, they load a certificate and dial the server using that. Thus securing the program against adversaries that should not be connected. It should be noted however that since I am the certificate authority and use self-signed certificates, this implementation only works on a

local system.

3 how to run the program

Please refer to the README in the zip file or on
https://github.com/nselpriv/securitystuff/tree/master/MedicalExperiment

3.1 The program running

In case the program won't run, I have included a screenshot of the process here. You see the server being started along with all the clients, their logs from different actions with timestamps and all users sending 100 as the value, what the value gets scrambled to and how the server calculated the sum using all results ending up with 300.

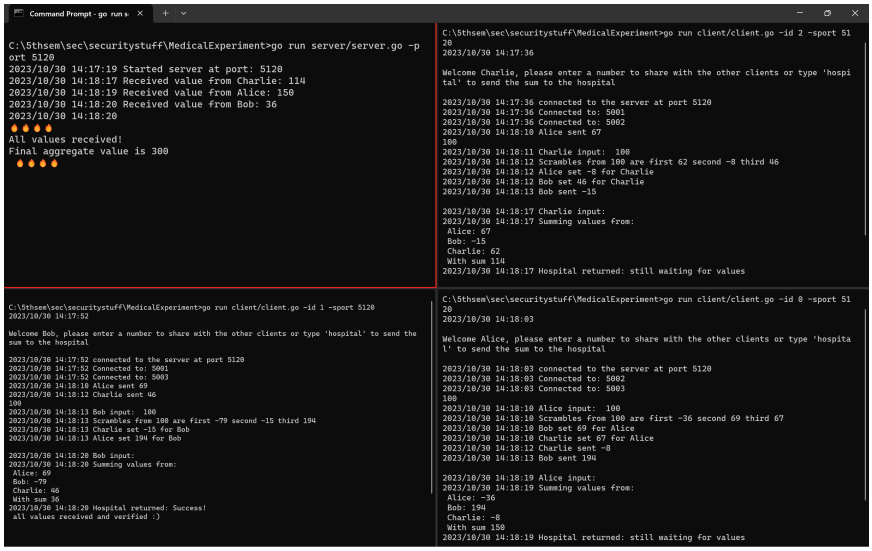

Figure 2: Printouts from running the program

References

[1] Nicolai Seloy Sigurd Holm Oscar Kankanranta. chittychat https://github.com/oskaitu/m.assignment$_0$3, 2023.

[2] Sigurd Holm Oscar Kankanranta, Nicolai Seloy. https://github.com/shhoitu/distributed-auction, 2023.