Link to repository: https://github.com/simonskodt/secu1-virtual-dice

1 Roll virtual dice

1.1 Purpose of the project

In this hand-in, we are assigned to have Alice and Bob roll a virtual die. They do not trust that rolling a die locally on each one of their computers will produce a honest and fair outcome. Over an insecure network, and where Alice and Bob do not trust each other, we are to design a protocol that provides a way for them to throw a virtual dice without letting an adversary know which activity they are doing. Afterward, we must explain why the protocol is secure, and lastly, implement this protocol in a programming language.

I have chosen to implement this protocol in *Go*. I utilize the *gRPC*[1] framework to simply have two processes on localhost communicate. Besides, I have used frameworks to implement *Transport Layer Security* (*TLS*) as well as an *SHA-2* library for the hash-based commitment scheme. I have purposely avoided implementing security-related controls myself since these cryptographic implementations are extremely hard to perfect – a few bugs are an enormous vulnerability.

1.2 Design a secure protocol

When Alice and Bob are to communicate over an insecure network, the initial task is to establish a secure channel. This channel has to provide data security, and by using the *TLS* protocol together with hash-based commitments, confidentiality, integrity, and authenticity are achieved (see section 1.3).

To begin with, the *TLS* protocol ensures that data communication over a network is encrypted. Meaning that unauthorized individuals are not able to see the communication

[1] *gRPC* authors, *Introduction to gRPC*, accessed October 22, 2022, URL: https://grpc.io/docs/what-is-grpc/introduction/

between Alice and Bob (contributing to privacy). *TLS* uses both asymmetric and symmetric encryption where the asymmetric encryption is responsible for generating the secret shared key whilst the symmetric encryption takes care of encrypting the messages sent back and forth afterward. This is due to the symmetric encryption's faster performance.

The digital signature certificates are self-signing which is not signed by a *Certificate Authority* (*CA*); the certificate is instead signed by the client's private key. Normally, this signature would be requested from a *CA*. The advantages are that this method is faster and easier to implement in test environments, but the disadvantages are that if the secret key is compromised, the creates a major security risk.

Now, Bob and Alice have a secure channel, but in order to send messages back and forth, we will use coin-tossing protocol. By sending commitments, one party cannot change and value after having committed to it. Afterward, the party that committed, reveals the values in order for the receiving party (Bob) to authenticate the sending party (Alice).

1.3 Explain why this protocol is secure

All in all, we are to achieve confidentiality, integrity, and authenticity security goals.

The *TLS* symmetric encryption guarantees confidentiality which then prevents eavesdropping attacks. This means that the rolling of the dice should only be available to Alice and Bob. However, this gives us no integrity or authenticity guarantees. *TLS*'s included *Message Authentication Code* (*MAC*) and digital signature establish integrity and authenticity security goals. Integrity because the *MAC* algorithm detects that the message has not been altered, and authenticity since the digital signature certifies the original sender.

The hash-based commitments provide confidentiality because of the hiding and binding properties.

- **Hiding:** When Alice sends the commitment to Bob it is difficult for Bob to invert the hash function. This step must include randomness due to the fact that a predictable message allows Bob to brute force the message.
- **Binding:** It is difficult for Alice to lie about her initial original message.

1.4 Implementation in *Go*

The *TLS* implementation is found in the `./main.go` file from line `85-128` in the function `setupTLSProtocol()`. The function returns a configuration struct that consists of the certificates, the client authentication, the client *CA* (the clients own signature: self-signing), and the root of the *CA* (again own signature). The `./certificates` folder consists of a certificate and a private key for both Alice and Bob which have been generated with the `./certificates/cert.sh` bash file. By using the framework *gPRC credentials*, in `./main.go` at line `58`, the function is called to create transport credentials using *TLS*. In both the function `ServerSetup()` and `ConnectToPeer()` (both in `./node/node.go`), the transport credentials are passed as a parameter to instantiate a new server with the respective credentials as well as to create a client that is targeted towards this server with connection security credentials.

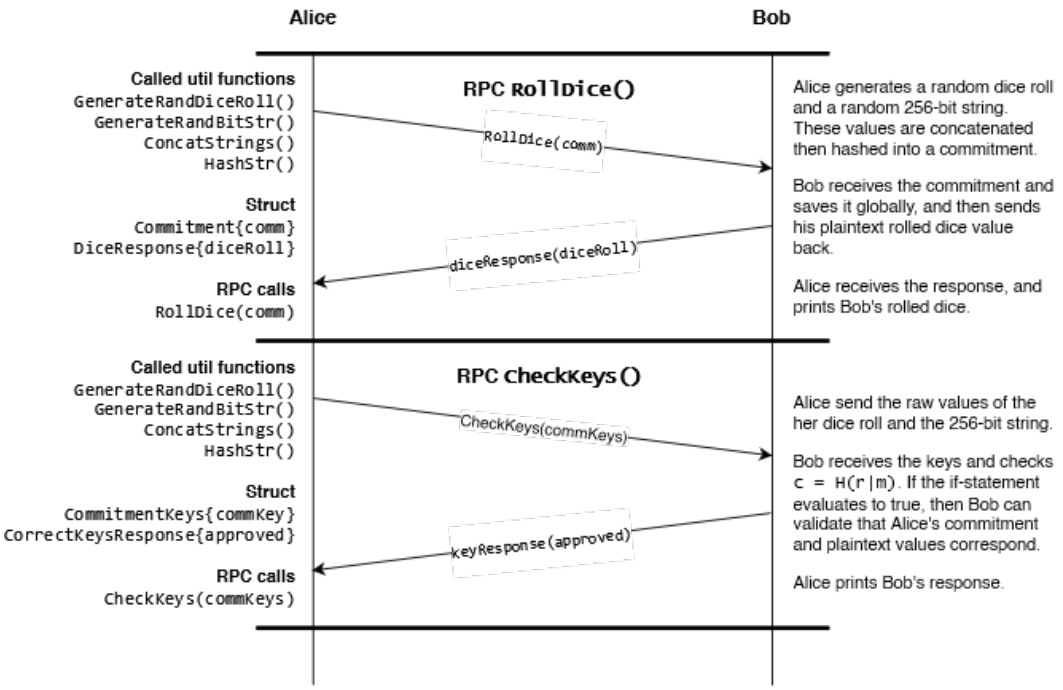

Figure 1: Sequence diagram depicting the two Remote Procedure Calls.

The `main()` function calls the `InitiateRequest()` function which is located in the `./node/node.go` line `73`. Before this, the dice roll of Alice and Bob have been randomly chosen. Alice initially generates a random k-bit string (256-bits), concatenates the rolled dice and the k-bit string, and lastly hashes this string with the imported library `crypto/sha256`.

The call to *RPC* `RollDice()` (line `95` in `./node/node.go`) sends the hashed concatenated

string in the `Commitment` struct, and then Bob stores the commitment for a later evaluation of Alice's raw values. Bob then returns his raw dice roll value.

The call to *RPC* `CheckKeys()` (line `111` in `./node/node.go`) sends Alices raw dice and bit string values to Bob. Bob evaluates concatenates and hashes the raw values and checks if the output string is equal to the previously stored commitment from Alice.