Jared Faucher & Jonathan Penney

CS51 Final Project: Task 2 – Annotated Draft Specification

TF: Ben Shryock

Brief Overview:

In today's age of computers, technology and the Internet, one of the biggest problems we face is security. Whether you are sending an email to a family member, or sending top secret encrypted files to a coworker, you expect those things to be accessed by authorized users. The problem with security is not determining what requirements a system should have to allow a particular user access to data, but how we implement these requirements in a way such that people who do not have these credentials are unable to gain access.

The goal of our project is to learn more about and implement Shamir's Secret Sharing Scheme. A secret sharing scheme is a way of splitting up some secret and sharing that secret among some group of participants. Each participant is given one part of the secret where some minimum number of participants must combine their parts to reconstruct the secret. If the minimum number of participants is not met when attempting to reconstruct the secret then the secret is not revealed and no information is gained about the secret.

We plan to create a program that when given some secret, some number of participants and a minimum number of participants needed to reconstruct that secret, the output is a list of passcodes/shares that can be distributed among participants. We also plan to have the program reconstruct the secret when given the correct number of shares. If the program is given less than the minimum number of shares needed to reconstruct the message, it should fail to reconstruct the message. Our final goal for this project is to apply our secret sharing scheme to some real life application, for example where a group of treasure hunters need 3 out of 5 keys to reconstruct the

secret location of the treasure or the White House needs 2 out of 3 nuclear launch codes to launch an attack. Although this last feature will not be necessary to the functionality of our secret sharing algorithm, it will provide a fun and informative way to apply our algorithm to the real world and expand our understanding of the core concepts of the algorithm and the language in general.

ANNOTATION: We successfully implemented SSSS in four different ways but we did not end up implementing the final goal to apply SSSS to some real life scenario

Feature List:

- Core feature: Implement some version of Shamir's Secret Sharing Scheme where when given a secret S, a number of participants N and a threshold T, distributes a list of N passcodes/shares back to each participant where T pieces are needed to reconstruct the secret. This will most likely will occur via the command line during the early phases. This is the encryption phase.

  ANNOTATION: We successfully implemented this

- Core feature: Implement the reconstruction phase of the secret sharing scheme such that when given T or more pieces of a secret S, we are able to reconstruct that secret and return it back to the participants. If less than T pieces of the passcodes are provided then the secret is not able to be reconstructed and no information is given about the secret. This reconstruction phase will require the creation of the Lagrange Basis polynomials from the pieces of the passcodes. As with the previous feature, this will most likely occur via the command line. This is the decryption phase.

  ANNOTATION: We successfully implemented this

- Cool extension: Applying this Secret Sharing Scheme to some real life application like we stated in the Brief Overview. This would be a feature from the core program that uses the secret sharing scheme to play out some scenario.

  ANNOTATION: We did not end up implementing this extension

- Cool extension: Creating some sort of "pretty" interface for our scenario either via a local website, similar to Moogle, or through some sort of graphics package. So far we plan on using OCaml because our program is very functional so Graphics.ml is an option but this could change through the course of the project.

  ANNOTATION: We did not end up implementing this extension and use used the command line as our interface.

- Cool extension: Adding some feature to our program where the use can select which algorithm is used to encode their secret into keys. This is the lowest priority for our project as the math needed can be very complicated but if we have time it would be a very interesting extension.

  ANNOTATION: We did not end up implementing this extension as it was not possible. We did not fully understand the mathematics involved in SSSS at this point.

- Cool extension: Adding extended functionality to encrypt a string value.
- ANNOTATION: We successfully implemented this extension by converting strings into bignums before performing the calculation and then converting it back to a string when reconstructed.
- Cool extension: Add another layer of encryption which will be applied to each participants' passcodes, instead of just integers. We could incorporate a known or custom encryption to the integer (passcode/share).

- ANNOTATION: We did not end up implementing this extension

Technical Specification:

Shamir's Secret Sharing Scheme is deeply rooted in mathematical formulas, thus it can be modularized into a list of clear and distinct pieces that we can work out separately and put back together in the end.

The first step is implement a program that takes in a secret S, which could be an integer or a string for example, the total number of participants N, and the number of participants needed to reconstruct the secret (threshold) T. The resulting initialization will return N number of keys. This phase of the algorithm will require the following types and functions:

- Type secret = int (This will be an int until we work out the math and then eventually a string)
- Type threshold = int, the minimum number of keys needed to reconstruct the message
- Type num_participants = int, the total number of keys to be generated
- Type key = (int * int), a pair consisting of an integer (1, 2, 3, etc.) plugged into the gen_keys function along with the result of plugging that integer into the function
- Val gen_keys : secret -> threshold -> num_participants -> key list, the function that actually generates the list of keys from the secret, threshold, num_participants and poly. The polynomial that is generated is not an argument of the gen_keys function, but an intermediary effect; generated, but never divulged.

  ANNOTATION: This interface stayed the same for the most part, except that threshold and num_participants are no longer abstract types in our final implementations.

We must be careful about designing our abstractions so that different types of secrets can be passed (first integers and then eventually strings), and that different types of encoding methods may be used eventually in addition to polynomials.

The next step in the algorithm is the reconstruction phase which will require taking in some X number of keys from a user to reconstruct the secret message. This phase of the algorithm will require implementing a function to compute the Lagrange basis polynomials from the keys and reconstruct the secret with them. This part of the algorithm will require the following types and functions:

- Type key = (int * int), the key pair as seen from the previous section
- Type secret = ‘a (int then string), the secret that we will reconstruct
- Type lagrange_poly : int * int list, similar to the type poly from the previous section, it is a representation of a Lagrange Polynomial as a list
- Val get_lagrange_poly : key list -> lagrange_poly list, this function would generate a list of a lagrange polynomials from a list of keys given.
- Val get_secret: lagrange_poly list -> secret, this function would take in the lagrange_poly list and attempt to generate the secret from it

  ANNOTATION: This also stayed the same for the most part except we changed lagrange_poly to a (int * poly) instead of (int * int list)

Besides these two stages of the algorithm the rest of the program would consist of functions printing to the user, whether it be through the command line or otherwise. In the case of a local web page we would need to create some function to take in the produced keys and output some built HTML code for our web page. However because the most important part of our program is

actually implementing the secret sharing algorithm we have not provided any function signatures for the additional features.

ANNOTATION: As we stated above, we did not end up implementing the web site interface

Next Steps:

Before we begin writing our final technical specification, we will ensure that we fully understand all key aspects of how SSSS components, especially when computing Lagrange basis polynomials. There is a naive method to implement SSSS that is not secure. If T-1 participants collude to determine the secret, much information can be gained for each additional colluder. We will implement the method of SSSS that addresses the naive approach, and is considered secure. The secure method ensures that when the threshold is not met, the colluders will not gain any information about the secret.

OCaml seems like a suitable language for this project. The tasks seem well suited to functional languages, but we will consider other languages if we hit a roadblock.

ANNOTATION: We did end up implementing both the naïve and secure versions of SSSS for both integer secrets and strings (converted to bignums) secrets.