Jared Faucher & Jonathan Penney

CS51 Final Project: Task 3 – Final Specification

TF: Ben Shryock

Signatures/Interfaces:

As we stated in the Draft Specification the most important thing our project must be able to do is encode a basic integer secret into some number of keys based on Shamir's Secret Sharing Algorithm and then reconstruct that secret when given some number of keys greater than or equal to the threshold value. Our key generation interface will be the following:

```
module type SHAMIR_ENCRYPT =
sig
  (* This is our abstract secret type. Initially, it will be an integer but eventually it will be string
or potentially some other type *)
  type secret
  (* This is our abstract polynomial type.  Because we are using the secret value to generate a
polynomial, we need some representation of that polynomial.  In our simple implementation this
will be a list of integers such that f(x) = 123 + 4x + 5x^2 ➔ [123; 4; 0; 5] *)
  type poly
  (* This is our abstract key type.  In our simple representation this will be an (int * int) for the x
and f(x) values *)
  type key
  (* This is our function that generates the keys from the given secret, the threshold number and
the total number of participants.  It outputs a list of keys to be given back to the user. *)
  val gen_keys: secret -> int -> int -> key list
  (* This is our print_keys function which is needed to print the keys back to the user.
Printf.printf wont work directly on our abstract type key so we need to give our interface the
ability to print our keys back to the user. *)
  val print_keys: key list -> unit
```

```
end
```

The signature for our interface is very simple, only containing a few abstract types and functions. However in reality we are going to need a lot of helper functions that are kept hidden in order to generate the keys. These functions will include additional helper functions to generate a random polynomial based on the secret and the threshold number, and another to evaluate a polynomial at some value x in order to produce our y-key values. By keeping most of the functionality of our program hidden ("under the hood"), this will allow us to reduce the complexity of the program to an outside user and ensure that our secret and the generated polynomial be exposed to outsiders.

Similarly to our key generation interface, out secret reconstruction interface will have a very similar layout:

```
module type SHAMIR_DECRYPT =
sig
  (* This is our abstract secret type as in the previous interface *)
  type secret
  (* This is our abstract key type as in the previous interface *)
  type key
  (* This is our abstract polynomial type as in the previous interface *)
  type poly
  (* This is our new abstract type to represent Lagrange Polynomials (here). In the integer
version of our program, the Lagrange Polynomial will be an (int * poly) tuple. In order to avoid
integer division rounding errors, we will keep the numerator and denominator separate. *)
  type lagrange_poly
  (* This will be our function that when given a list of keys returns the calculated secret. *)
  val get_secret: key list -> int -> secret
  (* Similarly to print_keys this will be a function to output our secret to the user. *)
  val print_secret : secret -> unit
end
```

In order to keep most of the functionality of the reconstruction phase hidden behind an interface, we are going to have a lot of hidden functions, similar to the key generation interface. These functions will include functions to sum together Lagrange polynomials, another to multiply a polynomial by an integer, to multiply a polynomial by an (x + a) (for Lagrange Polynomials) and possibly many others. This allows us the opportunity to add more features or use different types of secrets in the future without having to rewrite as much code.

Modules/Actual Code:

So far we have implemented a basic key generation module that generates keys based on the integer secret given, the threshold number given and the total number of participants given along with a basic secret reconstruction module. Initial function prototypes will require extensive testing. An immediate concern is adapting our project to allow for integer inputs that are beyond the range of Int.max_value and Int.min_value. Lessons learned from pset BigNums will come in handy for tackling this issue.   In our reconstruction phase, we are using integer division which could be problematic and we will need to be careful in dealing with "corner-cases".   These are some issues that we plan to overcome while implementing our core features.

Our code progress can be viewed at the repository provided below, but here is our basic integer version of the key generation module:

```
module Shamirint_encode : SHAMIR_ENCRYPT  =
struct
```

```
type poly = int list;;
type key = int * int;;

let gen_poly (s: int) (t: int) : poly =
  let rec helper (s: int) (t: int) : poly =
  match t with
  | 1 -> [s]
  | _ ->
    Random.self_init();
    Let r = (Random.int s) in
    r::(helper s (t – 1))
  in List.rev (helper s t);;

let eval_poly (x: int) (poly: poly) : int =
  let rec helper (x: int) (poly: poly) : int list =
    match poly with
    | [] -> []
    | hd::tl ->
      hd::(helper x (List.map ~f:(fun a -> x * a) tl))
  in
  List.fold_left (helper x poly) ~f:(+) ~init: 0;;

let gen_keys (s: int) (t: int) (n: int) : key list =
  let rec helper (n: int) (poly: poly) : key list =
    match n with
    | 0 -> []
    | _ ->
      (n, (eval_poly n poly))::(helper (n – 1) poly)
  in
  let poly = gen_poly s t in
  List.rev (helper n poly);;

let rec print_keys (keys: key list) : unit =
  match keys with
  | [] -> ()
  | h::t ->
    (match h with
     | (x, y) -> Printf.printf "(%i, %i)\n" x y; print_keys t);;
end
```

Timeline:

Week 4/20-4/25:

- Goal 1: Finish implementing secret type in module
- Goal 2: Finish Big_int implementation to avoid integer bounds errors
- Goal 3: Implement testing in both versions to determine errors and bounds for integers and Big_int values
- Goal 4: Create Makefile to compile integer version and Big_int versions separately

Week 4/27-5/1:

- Goal 1: Implement finite field arithmetic version for integers and Big_int using large generated prime numbers
- Goal 2: Implement secrets as strings with Big_int version

- Goal 3: Create some real-life scenarios to apply our program to, e.g. nuclear launch codes, treasure map, etc
- Goal 4: Create a "pretty" interface for our scenario via a local website or Graphics.ml. This will require creating another interface to generate HTML code based on the keys generated/given.
- Goal 5: Finish our project video and writing any documentation needed to use the program.

Progress Report:

In addition to the functions we provided in the Modules/Actual Code section, our progress can be viewed in our repository as well. The files shamirtrydec.ml and shamirtryenc.ml both compile and work correctly as long as the secret and number of participants do not cause any of the integer calculations to go out of bounds. The file Shamirint.ml has the encode and decode modules found in shamirtrydec.ml and shamirtryenc.ml. The file ShamirBigint.ml is our attempt to transfer our integer functionality into OCaml's built-in Big_int module, which we hope to avoid integer-bounding errors. We also hope to encode strings as secrets. The main.ml contains code to parse command-line arguments. Eventually we would like to have all of our modules/interfaces and command line functionality in separate files.

Version Control:

We have decided to create a new git repository on code.seas for our Version Control. The repository is located at:

https://code.seas.harvard.edu/faucher-penney-cs51-final-project/faucher-penney-cs51-final-proje
ct where staff are given access to read the repository.