Secret Sharing: Mignotte's, Asmuth-Bloom's and Shamir's Schemes

Alex Bukhta, Buffalo Hird, Jefferson Lee and Adam Su

Video: http://www.youtube.com/watch?v=lnVDKwRi0RE

Overview

We implemented three different schemes for secret sharing: Mignotte's, Asmuth-Bloom's, and Shamir's. The three schemes have the same goal of taking some secret as input and generating a number of shares as output. A predetermined number of shares, called the critical number, is required to reconstruct the original secret.

Mignotte's and Asmuth-Bloom's are related: both fundamentally use the Chinese Remainder Theorem. First a list of pairwise relatively prime moduli $m_1, \dots, m_n$ are generated where n is the total number of shares, that is, the total number of people who will hold part of the secret. Both schemes require that the product of the k smallest moduli is greater than the product of the k-1 largest moduli. Then we then take the residues $r_1, \dots, r_n$ of the secret with respect to each modulus. Thus, each share consists of a pair (r_i, m_i). Due to the conditions on the moduli, the Chinese Remainder Theorem guarantees a unique solution to the system of linear congruence equations given by these shares, provided the solution is less than the product of the k smallest moduli. In addition Asmuth-Bloom's scheme requires the secret to be smaller than a modulus $m_0 < m_1$ and adds a random number $k * m_0$ to the secret S. The sum $S + k * m_0$ is encrypted and decrypted instead of S. Taking the residue modulo m_0 gives the original secret.

Shamir's fundamentally uses Lagrange interpolation. First a (k-1)-degree polynomial is generated with the secret as its constant term and random coefficients for its remaining terms. Lagrange interpolation provides a method to reconstruct this (k-1)-degree polynomial from any k unique points. For simplicity we let these points be (1, p(1)), … , (n, p(n)). However our Lagrange interpolation functions work for any set of points. Upon reconstructing the polynomial, the secret can be extracted by taking the constant term.

Planning

Here is our final specification:
https://docs.google.com/a/college.harvard.edu/document/d/1sIWN98i0o8alUSVpH3vhHVDcY_ktiHZ682KjQryggjo/edit?usp=sharing

We think our milestones and planning went very well and largely according to the weekly checkpoint requirements. Overall the team functioned very well together and we worked smoothly. Everyone was eager to work and contribute. We definitely played the strengths of every person and specialized according to skills. Nonetheless the division of labor in raw implementation was very equal: Jefferson and Adam implemented the two Chinese Remainder Theorem schemes while Alex and Buffalo implemented the Lagrange

Interpolation scheme. We all worked together in code reviews, writing the report, and producing the video.

Design and Implementation

Our experience with design was great. We always kept a high-level view of what our project was meant to do, and this guided us in how to implement the many functions that would carry out the final goal. We aimed for extreme clarity and carefully defined extra types for modular equations and polynomials to ensure readability. In important functions we declare all the input and output types. This proved invaluable in the debugging process. We found that interfaces were unnecessary for our project, and wrapping our functions in modules provided few benefits to our implementation. We had a lot of fun using OCaml, and its strengths seemed ideal for what we were doing.

It was very smart to implement the secret sharing algorithms using Big_ints. If we had used big nums from an earlier pset, we would have been bogged down tremendously by low level details such as implementing addition, multiplication, exponentiation, comparison, etc., to have excellent runtime.

The hardest part of the project was implementing the Chinese Remainder Theorem methods. It was difficult to get right and track down bugs because there are so many moving parts in each secret sharing scheme. To get around this we broke down functions into smaller pieces and thoroughly tested those functions before recombining them into larger functions. In addition we wrote tester functions for many of our functions to make it easier to perform larger amounts of tests. To make sure our encryption would work with truly large numbers, we incorporated Big_int exponentiation functions into our tests.

We got bogged down by what types decode_mignotte and decode_asmuth should take. For example, we originally had these decode functions take in the total and critical number of shares as arguments. However this doesn't make sense as that information would not necessarily be public. Instead these decode functions will take the base that splits the secret, which must be public, as this is the only way the secret can be reconstructed from its pieces. Moreover we mistook the requirement that the product of the smallest k moduli must be larger than the product of the largest k-1 moduli for the false requirement that product of the smallest k moduli must be larger than the product of the largest n-k moduli.

In the end we were able to compare and contrast the three different schemes, as promised in the Final Specifications. As we stated before, Mignotte's and Asmuth-Bloom's schemes are both derivatives of the Chinese Remainder Theorem. Asmuth-Bloom's scheme is severely limited by the fact that the secret must be less than m_0, the smallest modulus. In contrast, Mignotte's only requires the secret be smaller than the product of the smallest k moduli. As a result, we must split the secret many times more in the Asmuth-Bloom scheme, and the runtime is consequently increased. However, in Asmuth-Bloom's scheme no information is leaked when fewer than the critical number of shares is leaked. The secret is protected by the $k * m_0$ added to the secret, where k is a random integer. In our implementation k ranges from 0 to max_int, so it is very hard to crack an Asmuth-Bloom

secret without the critical number of shares. Using Mignottes's it is clear that every share helps a hacker converge on the secret. Our hacker function was sometimes able to hack Mignotte-encoded secrets in reasonable amounts of time when given close to the critical number of shares.

Because Asmuth-Bloom's method is slow and Mignotte's method leaks information, we believe Shamir's method is the best scheme overall. Having any less than the critical number of shares gives you no information about the secret as you can fit an infinite number of degree k polynomials to less than k points. In addition we found that OCaml's Big_int library was very fast---so fast that it was unnecessary to divide secrets as big as 10^1000 into smaller pieces. Thousands of tests of this magnitude encrypted and decrypted in seconds.

Reflection

We were pleasantly surprised by how naturally OCaml could implement the Chinese Remainder Theorem and Lagrange Interpolation. To take advantage of Ocaml's powerful list processing abilities, we implemented polynomials as lists of coefficients and shares of Mignotte's and Asmuth-Bloom's Schemes as lists of modular equations. In addition we were delighted to find the List.map2 function, which we used on many occasions.

We were not so pleasantly surprised by how hard it was to integrate OCaml into a website. Eventually we found a way to link OCaml to PHP, thus producing the website you will in see in our video. However the code was very inelegant, so we chose not to submit it as part of our final project. Hopefully this example highlights the power of secret sharing in examples such as sharing missile launch codes.

Some good ideas we had along the way were:

We found a clever way to implement Lagrange interpolation without using fractions. Granted we have denominators in polynomials in the process of Lagrange interpolation, but we realized that these denominators must evenly divide into the coefficients in the numerator after adding the Lagrange basis polynomials together. This is because our random polynomial generator only produces integral coefficients and the random polynomial p(x) is integral for any integer x. This allowed us to simply use integer division on our fractional polynomials, which in turn allowed us to represent polynomials solely as lists of big_ints.

Our mod_generator function kept on failing even for trivial values of n (total shares) and k (critical shares). Thus we incorporated a shift parameter into this function that puts a limit on the smallest prime index mod_generator can return. We decided to do this upon realizing that prime numbers get logarithmically denser as they get larger, at least up to a certain point. For we values we have, the function works fine.

Finally we decided to split secrets in both Chinese Remainder Schemes into smaller pieces. By doing this mod_generator does not depend on the size of the secret, and instead the secret is split according to the magnitudes of the moduli mod_generator returns.

It was annoying to test whether two big_ints were equal because they are abstract values. Likewise, we could not directly see the value of big_ints without first calling int_of_big_int. To get around this we wrote test functions that took care of this overhead, so we could test our functions on more manageable int types. In addition we used Big_int compare functions and wrote our own eq_poly function to compare lists of big_ints. Finally, in a Helpers.ml file we have the functions list_to_big and list_to_int that convert a list of big_ints to a list of ints and vice versa. This is called many, many times in testing.

If we were to start over, we would still implement the project in OCaml. Ocaml highlights tools that proved very powerful in our implementation. More specifically, functional programming, recursion, list processing, and the Big_int library helped us a lot. One thing we might change would be to stop worrying about runtime until after we have a working implementation. We need to remember Donald Knuth's advice to implement before optimizing.

If we had more time we would probably work on verifiable secret sharing. However this is a very hard problem. The first step might be learning more number theoretical math just to get the background in how Feldman's and Benaloh's Scheme works.

Advice for Future Students

In this project we learned about the power of list processing and recursion. To give a specific example, we implement polynomials as a list of coefficients. We originally thought it was very hard to calculate coefficients of arbitrarily long binomial products, but with some more careful thinking we realized it was relatively simple to implement multiplication of a polynomial by a binomial. Recursion allowed us to extend this to arbitrarily long binomial products. A take home lesson might be: exploit the strengths of the language you are using.