ProSS: an Educational tool for learning Secret Sharing

Chad Fallotin

Supervised by: Michael Huth

June 13, 2016

ABSTRACT

Secret Sharing is still a relatively new field of research. This project presents a Java implementation aimed for learning about several Secret Sharing Schemes (SSS): Shamir's (t,n)-threshold secret sharing, Asmuth-Bloom's (t,n)-threshold secret sharing, and a (n,n)-threshold XOR based secret sharing. We also introduce two image secret sharing schemes. Our application is made to assist students in learning about secret sharing by serving as a concrete example and as a code base for their own implementations. We first introduce the concept of multi-party computation as an example of a real-world application of secret sharing. Next, we focus on the SSS we have implemented. Finally, we discuss in detail how our program was built and how it can be modified for future extensions.

Key Words: secret sharing scheme, cryptography, threshold scheme, interpolation, exclusive-or, Chinese remainder theorem, Java implementation

ACKNOWLEDGEMENTS

I would like to express my gratitude to all those who provided me the opportunity to complete this project. A special thanks to my supervisor Michael Huth for being kind and motivating me from the start of this project. I'd like to thank my second marker Mahdi Cheraghchi for giving me good advice. My thanks to A.Rothan, a fellow student for helping me with the formatting of this report. Finally, i would like say to thank you to my friends a family for supporting me throughout this project, even during long afternoons of debugging.

CONTENTS

1 INTRODUCTION 11
2 BACKGROUND 13
2.1 Encryption 13
2.2 Multi-party computation (MPC) 13
2.3 Real World Application of MPC 14
2.4 Secret Sharing 16
2.4.1 Properties 16
2.4.2 Homomorphic encryption 16
2.4.3 Polynomial interpolation 17
2.4.4 Chinese remainder Theorem 19
2.4.5 Secret splitting using Exclusive-OR (XOR) 20
2.4.6 Image secret sharing 21
3 IMPLEMENTATION 25
3.1 Design 25
3.1.1 Command Line 25
3.1.2 Java vs Python 26
3.1.3 Graphic User Interface 27
3.2 Structure 28
3.3 User Interface 29
3.3.1 Main menu 29
3.3.2 SSS 30
3.3.3 Asmuth-Bloom 31
3.3.4 (n,n) XOR 31
3.3.5 Picture Sharing 32
3.4 Implemented Secret Sharing Schemes 34
3.4.1 Shamir's Secret Sharing 34
3.4.2 Asmuth-Bloom Secret Sharing 37
3.4.3 XOR secret sharing 40
3.5 Image Secret Sharing 41
3.5.1 XOR based Coloured Image Secret Sharing 41
3.5.2 SSS based Grayscale Image Secret Sharing 41
3.6 Extensions 44
3.6.1 The scheme class 44
3.6.2 The GUI 44
3.6.3 The Controller 44
3.7 Testing 45
4 EVALUATION AND FUTURE WORK 47
4.1 Strings vs Lists 47
4.2 Detailed output window 47
4.3 Multi-Threading 47
4.4 The Go To File Button 48

4.5 A Color image secret sharing based on SSS 48
4.6 More schemes and optimisations 48
5 CONCLUSION 49

LIST OF FIGURES

13figure.2.1
Figure 2 Double auction setup 15
Figure 3 Multi-party Computation 15
Figure 4 Polynomials of degree 1 (blue), 2 (red), 3 (green) and 4 (orange) with f(0) = 2 17
Figure 5 (3,3) visual secret sharing 22
Figure 6 Pixel overlaying 22
Figure 7 Command line concept 1 26
Figure 8 Command line concept 2 26
Figure 9 Concept 3 27
Figure 10 Scene Builder Interface 28
Figure 11 Main Menu 29
Figure 12 Example Share0.txt 30
Figure 13 ShareX.txt valid format 31
Figure 14 SSS splitting window 31
Figure 15 SSS reconstruction window 31
Figure 16 XOR splitting window 32
Figure 17 XOR reconstruction window 32
Figure 18 Picture Sharing splitting window 33
Figure 19 Picture Sharing reconstruction window 33
Figure 20 SSS Splitting example 34
Figure 21 SSS.split() 35
Figure 22 split_shares() 35
Figure 23 SSS Combining example 36
Figure 24 reconctruct_shares() 36
Figure 25 AB Splitting example 37
Figure 26 ABS.split() 38
Figure 27 AB Reconstruction example 39
Figure 28 ABS.reconstruct() 39
Figure 29 XOR.split() 40
Figure 30 computXOR() 40
Figure 31 XOR.combine() 40
41figure.3.32
Figure 33 (2,5)-threshold SSS Image splitting 42
Figure 34 (2,5)-threshold SSS Image Combining 43
Figure 35 Scheme selection in dothesplit 44
Figure 36 Performance of our implementation (X axis: number of shares, Y axis: time in milliseconds) 45
Figure 37 Performance of Image Secret Sharing (X axis: number of shares, Y axis: time in milliseconds) for a 192x108p .PNG 46

1

INTRODUCTION

Secure MultiParty Computation (SMPC) is probably one of the most important problem in cryptography. Since the late 80s, active research came up with a few protocols and even fewer real world implementation. Secret Sharing (SS) is the main building block of SMPC. The first (t,n)-threshold Secret Sharing Schemes (SSS) where independently discovered by Shamir [1] and Blackey [2] in 1979. Since then multiples schemes have been discovered, either based on existing work or completely new such as Jun Kurihara et al. scheme based on Exclusive-OR (XOR) [3]. While it is simple to find many papers describing different secret sharing schemes, it is nearly impossible to put a hand on their implementation to visualise their workings. The goal of this project is to offer a solution to this problem, that is to build a working implementation of several main SSS that will serve as a base application for a teaching tool to learn about Secret Sharing Schemes. Our program, ProSS, currently implements 4 main Secret Sharing Schemes: Shamir Secret Sharing Scheme, Asmuth-Bloom Secret Sharing Scheme, XOR Secret Sharing and Image Secret Sharing.

1 Shamir, Adi (1979). "How to share a secret". Communications of the ACM 22 (11): 612–613

2 Blakley, G. R. (1979). "Safeguarding cryptographic keys". Proceedings of the National Computer Conference 48: 313–317

3 Jun Kurihara et al."A New (k,n)-Threshold Secret Sharing Scheme and Its Extension", Proceedings of the 11th international conference on Information Security: 455-470

2

BACKGROUND

2.1 ENCRYPTION

Public key cryptography is an asymmetric scheme that uses two keys for encryption: a public key and a private key. The public key is visible to the world while your private key is kept secret. Anyone with a copy of your public key can encrypt information that only you can read. The security of this encryption holds to the fact that it is computationally infeasible to deduce the private key from the public key.

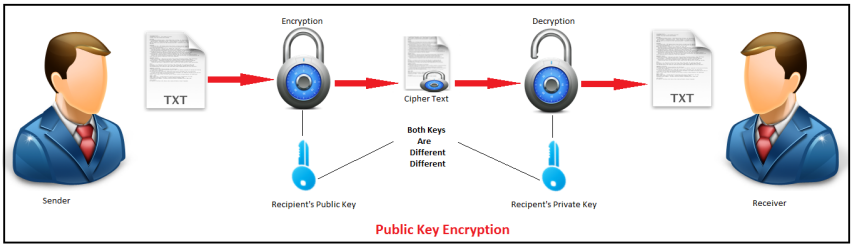

Figure 1: RSA public key encryption[1]

2.2 MULTI-PARTY COMPUTATION (MPC)

The idea of multi-party computation is simple: all necessary data is supplied and processed in encrypted form. The results made public will only be those that the involved parties agree to have decrypted. No information on the inputted data can be found from the result.

Computing without actually looking at the data may seem impossible; nevertheless this problem has been solved since the late 80s. As secure multiparty computation becomes more mature, we see more and more practical applications where computation moves from a theoretical scenario to an real world application where each computation node is hosted by a different party. In this chapter we will look at one of the first industrial implementation of MPC.

1 Rivest, R.; Shamir, A.; Adleman, L. (February 1978). "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems" (PDF). Communications of the ACM 21 (2): 120–126.

2.3 REAL WORLD APPLICATION OF MPC

Secure multiparty computation was first used in Denmark in 2008 to reallocate sugar beet production contracts between farmers and a sugar production company called Danisco. Over 5000 Danish farmers own contracts for production rights allowing them to grow a certain amount of beets per year. Farmers were allowed to trade contracts if required but had to take care of negociations themselves. A few years ago, the European Union drastically reduced the support for sugar beet producers. This and other factors meant that there was now an urgent need to reallocate contracts to farmers whose productions payed off the best. This resulted in an immediate need for a nation wide market for trading production rights: a double auction system.

A double auction is a process of buying and selling goods when potential buyers submit their bids and potential sellers simultaneously submit their ask prices to an auctioneer, who computes a market clearing price p where total demand equals total supply. In our scenario, a farmer willing to sell makes a bid for a selling price. Similarly, a buyers make a bid for a buying price. Once all bids have been submitted, a clearing price p is computed. Therefore, if a farmer submitted a bid, all transactions will be done at price p.

Double auction is fairly simple to implement and could easily be done using a trusted third party as an auctioneer. However in the case of Danish farmers, additional factors have to be taken into account. Danisco was first chosen to act as the third party but farmers were concerned about revealing their economic situation and productivity and therefore were reluctant to see Danisco as an auctioneer. The idea of using a consultancy company as a trusted party was put aside for financial reasons. The solution decided on was to implement an electronic double auction, where the role of the auctioneer would be played by a three-party multiparty computation done by representatives for Danisco, DKS (the farmer's organisation) and the SIMAP research project who implemented the auction.

The system is made of a Web server receiving bids, and three server performing the secure computation (see Figure 2). Each bidder installs a web applet on his/her computer which gives the option of submitting a bid for selling, buying or both for 4,000 different prices. The bids are first encrypted by the bidders machine, using one of the servers public key. The application encryption was based on Shamir's secret sharing scheme over a field $\mathbb{Z}_p$, with p a 64-bit prime number. We will see more on this in the following sections. The shares are then sent and stored in the central server running the ap-

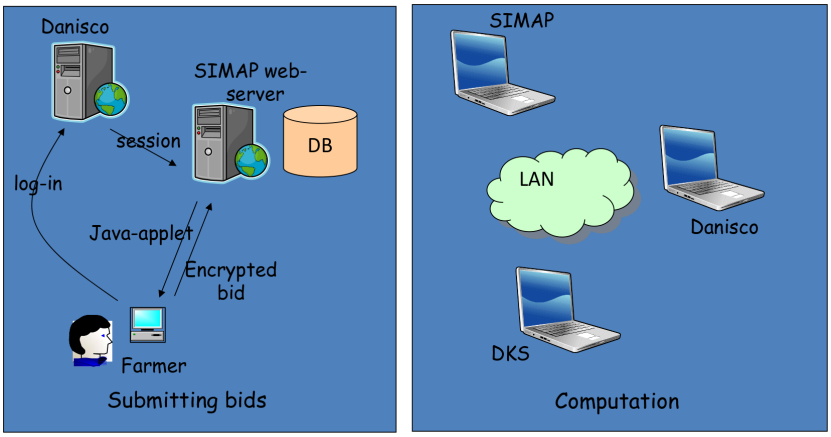

Figure 2: Double auction setup

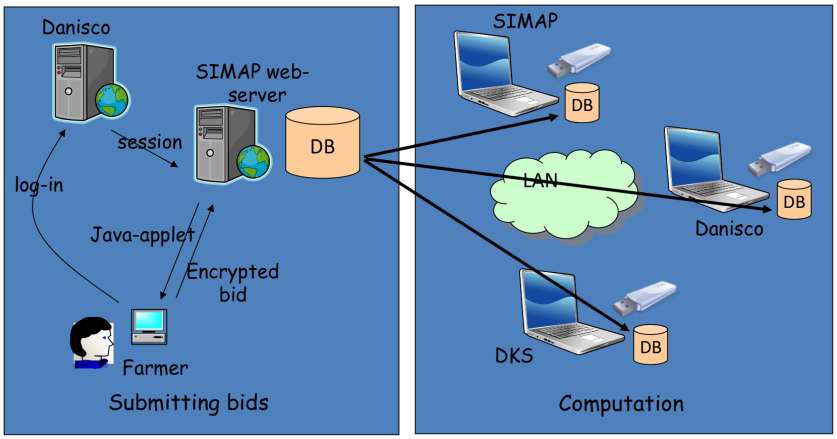

Figure 3: Multi-party Computation

plication until the bidding deadline. The secure multi-party computation phase is done in two steps. First, shares of each bid are loaded to their corresponding servers for decryption (see Figure 3). This is the most time consumming process since each server has to handle hundreds of encrypted shares. Finally, using comparison and binary search over all decrypted shares, the market clearing price is computed.

An on-line survey carried out in connection with the auctions showed that more than 85% of the farmers where satisfied with the new system. This application has been used every year since 2008 to reallocate sugar beet growing contracts between Danish farmers.

Multi-party computation would not have grown so much if it wasn't for secret sharing.

2.4 SECRET SHARING

Secret sharing is the core of secret multi-party computation. Before diving into details, here is an example of secret sharing:

Corporate Signing Keys are private keys used by a company to sign legal documents, sensitive personnel information, or press releases to authenticate their origin. In such a case, it is worthwhile for multiple members of the company to have access to the private key. However, this means that any single individual can act fully on behalf of the company. Secret sharing solves offers a convenient solution to this problem.
Consider n participants $P_1, P_2, \ldots, P_n$ who want to distribute a secret s amongst themselves such that each participant P_i has a share $[s]_i$ of the secret. It should be possible to reconstruct the secret when in possession of 'enough' shares and impossible otherwise.
Secret sharing is perfect for storing highly confidential information like bank account numbers, important passwords for example, nuclear missile launch codes. Since the secret is shared between several parties, it is difficult for a malicious agent to get all shares from all parties. Additionally, computation can be done with a certain number of missing parties. Therefore, loosing a share to a malicious party is equivalent to not loosing anything: the secret can still be computed and the malicious party doesn't learn anything from a single share.

2.4.1 *Properties*

A secure SS is meant to be:
Perfect: Any information about s cannot be obtained from $t-1$ or less shares.
Accessible: s can be fully recovered from t or more shares.
Ideal: The maximum size of a share always equals the size of s.

2.4.2 *Homomorphic encryption*

Threshold homomorphic encryption is a method that allows revealing a secret value to any coalition of at least t parties, while giving no information about that value to any coalition smaller than t. In addition, it enables operations over secret shares. That is, if two users P_1 and P_2 want to perform a secret addition, they first use a homomorphic secret sharing scheme to build two sets of shares P_{1_i} and P_{2_i}. Shares are then added together such as $P_{1_i} + P_{2_i} = P_{3_i}$. Enough shares P_{3_i} are then combine to recover the secret result.
Homomorphic encryption is a 4 step system:

- Process the secret using a homomorphism

- Split the secret into several parts and distribute a share to each recipient.
- Gather enough shares to recover the secret.
- Recover the secret by reversing the homomorphism from step 1.

An example of application of homomorphism is detailed in Craig Gentry's *Fully Homomorphic Encryption Using Ideal Lattices*.

2.4.3 *Polynomial interpolation*

"Threshold schemes are ideally suited to applications in which a group of mutually suspicious individuals with conflicting interests must cooperate", *Adi Shamir*

Shamir's secret sharing scheme (SSSS), invented in 1979 by Adi Shamir [2] one of the fathers of public key encryption, is based on the idea that two points are needed to define a line, three to define a parabola and so forth. Shamir's shares are evaluations of a randomly selected polynomial for different values.

Figure 4: Polynomials of degree 1 (blue), 2 (red), 3 (green) and 4 (orange) with f(0) = 2

Here is an overview of how SSSS works: A secret s is to be shared between n parties while t participants are needed to reconstruct s.

2 A. Shamir, "*How to share a secret*", Commun. ACM, vol.22, no.11, pp.612–613, 1979.

2.4.3.1 *Splitting*

A dealer picks a random polynomial $f \in \mathbb{F}_p[x]$ of degree $t-1$, where $t < n$, such that $f(0) = s \in \mathbb{F}_p$:

$$f(x) = s + a_1 x + a_2 x^2 + ... + a_{t-1} x^{t-1} \tag{1}$$

A prime $p > s$ is selected as a public key. The dealer then computes the values $[s]_i := f(i) \mod p$ for $1 \leq i \leq n$ and sends each share to the corresponding participant P_i where a share is a tuple $(x_i, [s]_i)$.

2.4.3.2 *Reconstruction*

Any t participants can reconstruct the polynomial f by applying Lagrange interpolation to the tuples $(i, [s]_i)$.

For every field $\mathbb{F}$, every t distinct values $x_1, ..., x_t$, and any t values $y_1, ..., y_t$, there exists a unique polynomial ℓ of degree at most $t-1$ over $\mathbb{F}$ such that $\ell(x_j) = y_j$ for $1 \leq j \leq t$. For $i \in \{0, \ldots, n\}$, the original polynomial is ℓ_i:

$$\ell_i(x) = \prod_{j=0,\ j \neq i}^{n} \frac{x - x_j}{x_i - x_j}. \tag{2}$$

The secret is recovered by taking the coefficient of degree 0 modulus the key p:

$$s = f(0) \ mod \ p \tag{3}$$

SSSS has been proven to be theoretically secure while the size of each share does not exceed that of the secret s. In addition, new shares can be generated dynamically without affecting existing shares or in the case where many shares have been stolen, a whole new set of shares can be generated by changing the polynomial's coefficients. However, this scheme requires heavy computation during splitting and recovery, especially for big secrets.

2.4.3.3 *Verifiable Secret Sharing (VSS)*

What if the dealer is malicious? How can players make sure they receive a valid share? VSS was first introduced by Chor et al. [3] with the intention to add a layer of robustness to secret sharing schemes like Shamir's. Fiedlamn's VSS scheme [4]is compatible with SSSS. The idea is to use a one-way function f such that $f(x+y) = f(x) * f(y)$ and broadcast $f(a_0), ..., f(a_{t-1})$, where $P(x)$ is th epolynomial used in Shamir's splitting phase. Correctness of each share $[S]_i$ can be verified by $f([S]_i) = \prod_{j=0}^{k-1} f(a_j)^{i^j}$. Publicly verifiable secret sharing schemes,

3 B. Chor, S. Goldwasser, S. Micali and B. Awerbuch, Verifiable Secret Sharing and Achieving Simultaneity in the Presence of Faults, FOCS85, pp. 383-395.

4 P.Feldman (1987). A practical scheme for non-interactive verifiable secret sharing, 28th Annual Aymposium on Foundations of Computer Sciences: p. 427-437

introduced by Stadler [5] makes anyone able to verify the correctness of the shares distributed by the dealer.

2.4.4 *Chinese remainder Theorem*

Asmuth-Bloom's threshold scheme and Mignotte's threshold scheme use a different sharing process: modular arithmetic and the Chinese remainder Theorem (CRT).

Chinese Remainder Theorem. Every system of linear congruences $x \equiv c_1 \ mod \ m_1, ..., x \equiv c_n \ mod \ m_n$, in which the moduli are relatively prime in pairs, is solvable, and the solution is unique modulo the product of the moduli.

2.4.4.1 *Mignotte's Scheme: Splitting*

We want to share a secret s between n participants such that t, with $t < n$, or more participants can uncover the secret. First, we choose a (t,n)-Mignotte sequence of positive integers such as $m_1 < ... < m_n$ where (m_i, m_j) are co-prime for all $1 \leq i < j \leq n$, and $m_{n-t+2}...m_n < m_1...m_t$. m_i numbers are chosen to satisfy $min < s < max$, where $min = m_{n-t+2}...m_n$ and $max = m_1 < ... < m_n$. The shares (S_i, m_i) are then chosen by $S_i = S \ mod \ m_i$, for all $1 \leq i \leq n$.

2.4.4.2 *Mignotte's Scheme: Reconstruction*

To reconstruct the secret, take t shares $S_{i_1}, ..., S_{i_t}$ and compute $s = x_0 \ mod \ r$ using CRT, where x_0 is the unique solution modulo M of the system:

$$\begin{cases} x \equiv S_{i_1} \ mod \ m_{i_1} \\ ... \\ x \equiv S_{i_t} \ mod \ m_{i_t} \end{cases}$$

2.4.4.3 *Asmuth-Bloom's Scheme: Splitting*

We want to share a secret s between n participants such that t, with $t < n$, or more participants can uncover the secret. First, we choose a sequence of pairwise co-prime positive integers $r < m_1 <...< m_n$ such that $r * m_{n-t+2}...m_n < m_1...m_t$.
The shares (S_i, m_i) are then chosen by $S_i = (S + \alpha * r) \ mod \ m_i$, for all $1 \leq i \leq n$, where α is an arbitrary integer such that $S + \alpha * r \in \mathbb{Z}m_1...m_t$.

5 M. Stadler (1996): Publicly verifiable secret sharing. Proceedings of the 15th annual international conference on Theory and application of cryptographic techniques, EUROCRYPT'96, Springer-Verlag, Berlin, Heidelberg, pp. 190–199,

2.4.4.4 *Asmuth-Blooms Scheme: Reconstruction*

Now, let M denote $\prod_{i=1}^{t} m_i$. To reconstruct the secret, take t shares $S_{i_1}, ..., S_{i_t}$ and compute $s = x_0 \ mod \ r$ using CRT, where x_0 is the unique solution modulo M of the system:

$$\begin{cases} x \equiv S_{i_1} \ mod \ m_{i_1} \\ ... \\ x \equiv S_{i_t} \ mod \ m_{i_t} \end{cases}$$

Since we know that for all i, $m_i, ..., m_t$ are pairwise coprime, according to Bezout's lemma stating that $(m_i, M/m_i)$, there exists positive integers p_i and q_i that satisfy $p_i * m_i + q_i * (M/m_i) = 1$. p_i and q_i are found using the extended Euclidean algorithm. x_0 is the solution to $\sum_{i=1}^{t} S_i e_i$ where $e_i = q_i * (M/m_i)$.

Although Asmuth-Bloom and Mignotte schemes are very similar, the latter is not perfect since a coalition smaller than t will get information about the secret. On the other hand, Asmuth-Bloom doesn't leak any information for 1 to t-1 stolen shares. Asmuth-Bloom is a perfect scheme but it very computation intensive.

2.4.5 *Secret splitting using Exclusive-OR (XOR)*

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

Table 1: XOR truth table

When $n = t$, that is when all participants are needed to recover the secret, XOR is by far the simplest solution. Let shares $r_1, ...r_{n-1}$ be random numbers of the same size as secret s. The last share r_n is the XOR of the secret and all those shares:

$$r_n = s \oplus r_1 \oplus r_2 \oplus ... \oplus r_{n-1} \tag{4}$$

Hence s can be found by XORing all shares $r_1, ..., r_n$:

$$s = r_1 \oplus r_2 \oplus ... \oplus r_n \tag{5}$$

XORing schemes security is based on the one-time pad system (OTP). In this method, a random secret key of the same length as the secret is generated. Then, each bit of the secret is combined with the corresponding bit from the secret key using a modular arithmetic.

The resulting cypertext is a sequence of bits the size of the original secret. Combining the cyphertext and the secret key uncovers the secret.

In the scheme described above, the random secret key K is the result of $r_1 \oplus ... \oplus r_{n-1}$ while the cyphertext is the last share r_n (since $r_n = s \oplus K$).

OTP was proven by Claude Shannon [6] to be information-theoretically secure assuming the secret key is truly random (as opposed to pseudorandom). If malicious parties somehow get hold of the cyphertext, they can only learn the length l of the secret. Indeed, since all keys of length l are possible and all resulting plaintexts have the same probability, there is now way for them to find out which plaintext is the actual secret. However, this is a double-edged property. Since this system is unbreakable, the loss of one share will always result in the definite loss of the secret information. A solution to this problem would be to keep multiple copies of each shares. Though this solution decreases the chance of completely losing the secret, it makes each shares more vulnerable to corruption.

Over the years, many have proposed (k,n)-threshold schemes. Jun Kurihara et al. [7] , describe a new (k,n) scheme based on XOR that keeps the security benefits of Shamir's scheme (a perfect scheme) while making the most of XOR computation speed. In their paper, they also show that their scheme is up to *"900-fold faster that Shamir's"* for the same level of security.

2.4.6 *Image secret sharing*

2.4.6.1 *Visual Cryptograpy*

Visual Secret Sharing was first introduced by Naor and Shamir [8] in 1995. They developed a (n,n) scheme that splits a greyscale visual support (text, image,...) into n pieces where n-1 pieces do not reveal any information about the secret. Each share consists of a transparency with a shares of the secret image. Superimposing all transparencies reveals the secret. This scheme requires no computation other than the splitting phase. As a matter of fact, reconstructing the secret relies on the human visual system capability to perceive an image from random pixels. However, this scheme isn't flawless. Indeed, there is a noticeable loss of contrast when adding 2 black and white shares to make a white pixel (see Figure 6). In addition, the size of each share

6 Shannon, Claude (1949). "Communication Theory of Secrecy Systems". Bell System Technical Journal 28 (4): 656–715.

7 *A New (k,n)-Threshold Secret Sharing Scheme and Its Extension*, Jun Kurihara et al.

8 Naor, M. and A. Shamir. *"Visual cryptography, Advances in cryptology"*. Eurocrypt '94 Proceeding LNCS, 950:1–12, 1995.

depends on the number of subpixels required to split each pixel from the secret image. If splitting is done by following the pattern in Figure 6, each share will be twice as wide as the secret.

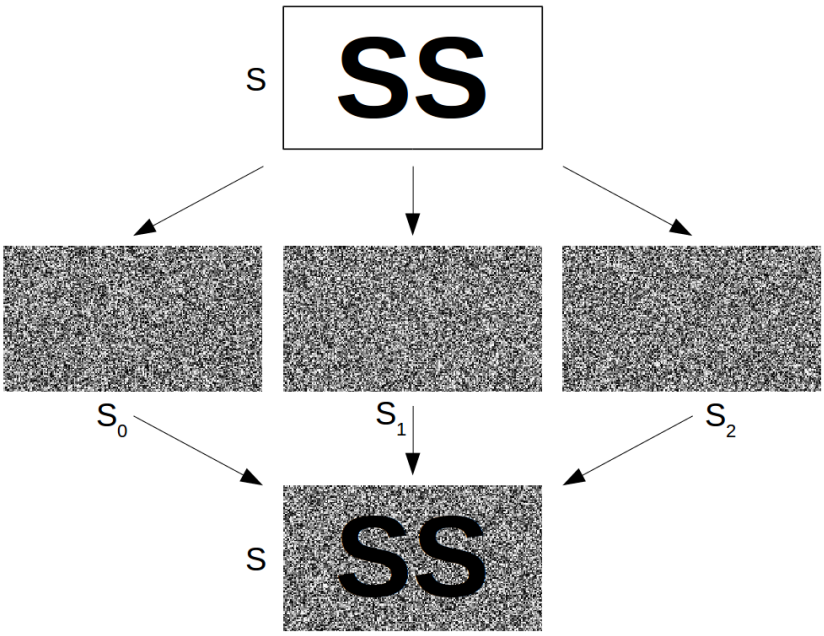

Figure 5: (3,3) visual secret sharing

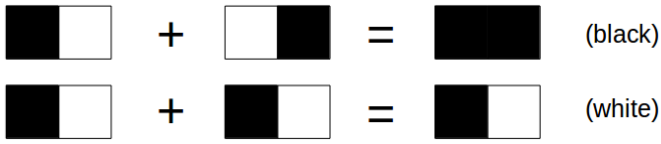

Figure 6: Pixel overlaying

2.4.6.2 *Image secret Sharing*

Thien and Lin [9] proposed a (k, n) threshold scheme base on Shamir's Secret Sharing Scheme to generate shares. Their scheme takes a grayscale image and uses a polynomial of degree t-1 to generate n shadow images. Each pixel gray value is shared between the shadow image as a new gray value. Thien and Lin describe how any t or more shadow images of size $1/t$ are able to reconstruct the secret while $t-1$ do not reveal enough information about the secret. To increase the randomness of the shadow images, they use a key to permute the order of pixels. Therefore an inverse permutation as to be applied to all shares in order to recover the secret image.

9 C.-C. Thien and J.-C. Lin, "Secret image sharing," Computers Graphics, vol. 26, no. 5, pp. 765–770, 2002.

In this project, we present two picture secret sharing schemes. One is based on XOR and the other on SSS.

3

IMPLEMENTATION

ProSS Secret Sharing is a Java implementation of several secret sharing schemes:

- Shamir's Secret Sharing
- Secret Sharing with CRT
- XOR Secret Sharing
- Image secret Sharing

We have built this application as a tool for visualising the workings of secret sharing schemes.

3.1 DESIGN

In this section we will detail the designing phase of our implementation.

3.1.1 *Command Line*

In the early stages of this project, ProSS was designed as a command line program. What we wanted was a base interface over which we could add extensions such as new Secret Sharing schemes. We came up with two main concepts for our prototype:

Concept 1 (Figure 7) is an interactive command line program. The user is first prompt to choose between several options which are selected by entering the corresponding number (ex: 1 for Reconstruction MODE in Figure 7). The user is then asked to select a secret sharing scheme and finally the number of shares, the threshold if required, and the secret message. The resulting shares and key are displayed in the terminal or can be saved to one or multiple files. Also, extensions can easily be added to the interface by updating the MODE and scheme list.

Concept 2 (Figure 8) is a program in which settings such as scheme, number of shares, threshold, file paths, and secret message are given as arguments. Shares are outputted in the same manner as with Con-

cept 1.

The final selection was done by surveying students in the Computer lab. Concept 1 was selected by a large majority.

```
+---------------------+
| Welcome to ProSS!   |
+---------------------+
Select a MODE:
0 - Split secret
1 - Reconstruct secret
>0

Select a scheme:
0 - Shamir's Secret Sharing
1 - Asmuth-Bloom SS
2 - XOR
3 - Image SS
>0

===== SSS MODE =======+
Enter number of shares:
>3
Enter threshold:
>2
Enter secret:
>This is my secret.
----------------------+
Shares:
(1,XXXXXXXXXXXXXXXXX)
(2,XXXXXXXXXXXXXXXXX)
(3,XXXXXXXXXXXXXXXXX)

Key:
XXXXXXXXXXXXXXXXXX
```

Figure 7: Command line concept 1

```
Process -m split -sch SSS -n 3 -t 2 -S "This is my secret."
Process -m reconstruct -sch SSS -s "Shares.txt" -k "Shares.txt"
```

Figure 8: Command line concept 2

3.1.2 *Java vs Python*

We first began our implementation in Python. Python offers dynamic typing while handling very large number as normal integer (ex: long*long = long). Our program in meant to comput very big numbers. Additionally, we chose Python for its many available secret sharing libraries. However, we hit a wall in the early stage of our implementation, while implementing SSS, that made us switch to Java (section 3.1.3). From the start, Java was our second choice. Java's BigInteger type is perfectly suited for our application. However, as

opposed to Python's long type, BigIntegers require a specific syntax to write basic operation (BigInteger newBig = new BigInteger("251") or newBig.multiply(new BigInteger("251"))). On the other hand, it provides built-in operations for modular arithmetic, primality testing, prime generation, bit manipulation, and more. Here is a non exhaustive list of helpful BigInteger methods:

- **int bitLength()**: gets the size of secret and generate random numbers of same size.
- **BigInteger mod(BigInteger b)** and **modInverse(BigInteger b)**: those were used to compute finite field arithmetic over $\mathbb{F}_p$.
- **BigInteger probablePrime(int l, Random rnd)**: returns a positive BigInteger that is probably prime of length l.
- **BigInteger xor(BigInteger b**: convenient way of XORing two BigIntegers.

Java was also selected for its cross-platform portability meaning ProSS will run flawlessly on Linux, windows and OSX without having to modify the source code.

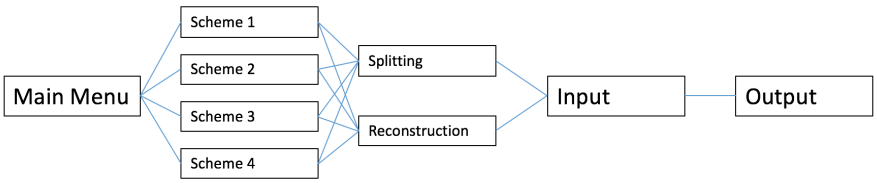

Figure 9: Concept 3

3.1.3 *Graphic User Interface*

While implementing the first prototype of Concept 1 (Figure 7), we noticed how unpractical the interactive command line system was. Every time we needed to run a test for Shamir's Scheme, we had to run the program, press 0 (to select Split Mode) and Enter, press 0 (to select SSS) and Enter again, then enter the number of share, threshold and finally the secret. In order to try the same scheme with 1 more share or a different secret, we had to go through everything again. Why not switch to Java and translate Concept 1 into a GUI?

JavaFX is a graphical library integrated into Java SDK for creating desktop applications. **Scene Builder** [1] is a drag-and-drop interface that enables fast and clean GUI creation. Using JavaFX along with Scene Builder we were able to build Concept 3 (Figure 9).

1 http://gluonhq.com/labs/scene-builder/

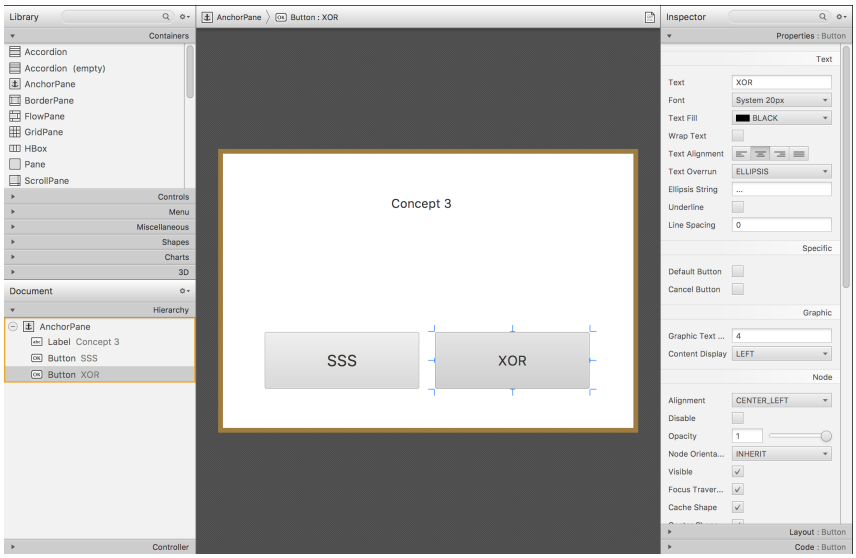

Figure 10: Scene Builder Interface

We used IntelliJ IDEA CE [2], a powerful IDE focused on java development because it integrates JavaFX and Scene Builder. Netbeans and Eclipse were good choices as well. Furthermore, IDEA has a built-in version control extension which helped a lot. Git was used to save our work throughout the entire project.

3.2 STRUCTURE

Our application's structure is simple; ProSS is currently made of one Controller class, one Main class and four SS classes corresponding to each secret sharing scheme currently implemented. Each SS class is paired to an .fxml file generating a window showing up in the GUI. The Controller takes care of the interaction between the GUI and the classes.

Main.java: Loads the main menu window `mainmenu.fxml` and launches the GUI.

mainmenu.fxml: Contains the layout settings for the Main Menu window.

SSS.java: Contains all methods for Shamir's Secret Sharing Scheme, `split()`, `reconstruct()` as well as utility methods such as `newPrime()` and `getRandomBigInteger()`.

sss.fxml: Contains the layout settings for the SSS window.

ABS.java: Contains all methods for the Asmuth-Bloom Secret Sharing Scheme, `split()`, `reconstruct()` and utility methods.

ab.fxml: Contains the layout settings for the Asmuth-Bloom window.

2 https://www.jetbrains.com/idea/

XOR.java: Contains all methods for the XOR Secret Sharing Scheme, `split()`, `reconstruct()` and utility methods.

xor.fxml: Contains the layout settings for the Asmuth-Bloom window.

PSS.java: Contains all methods for the Picture Secret Sharing, all `split()` and `reconstruct()` methods for each scheme (currently XOR and SS).

Process.java: Contains all utility methods for `PSS.java`.

pss.fxml: Contains the layout settings for the PSS window.

ControllerSS.java: This class is linked to all `.fxml` files. It handle events such as button clicks and calls the right methods for each secret sharing scheme. For instance, when the "split" button is clicked on the XOR window, `Controller.java` calls `split()` in `XOR.java` and outputs the result in the ouputbox of the right window.

Pair.java: Implementation for Pair(K,V) handling

picture (file): Contains classes for picture processing (`Color.java`, `Picture.java`,...) Used in Picture secret sharing.

3.3 USER INTERFACE

We tried our best to keep our interface simple yet intuitive. This section shows the different windows available in our program.

3.3.1 *Main menu*

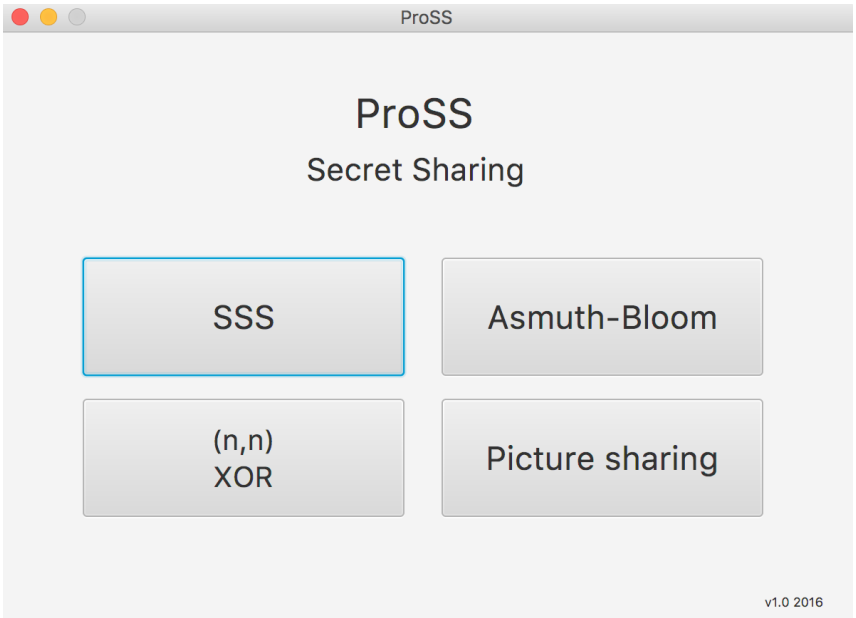

Figure 11: Main Menu

The main menu is a simple window with 4 buttons corresponding to an implemented Secret Sharing Scheme.

1. SSS - section 3.3.2
2. Asmuth-Bloom - section 3.3.3
3. (n,n) XOR - section 3.3.4
4. Picture sharing - section 3.3.5

3.3.2 *SSS*

Each Secret sharing scheme window is made of two tabs to conveniently separate the Split and Reconstruct interfaces.

The SSS Split tab (Figure 14) is divided into two panels: input and output. The left side contains the input boxes, `Secret`, `Number of Shares` and `Threshold`. Clicking the "Split" button calls `SSS.split()` which computes the shares[3]. The TextArea on the right outputs the result of the splitting process while the "Copy" button copies the content of the Output box into the input boxes on the SSS reconstruction window (Figure 15) to facilitate testing. Shares can be saved in one file using the "Save" button. Clicking the "Save" button opens a `FileChooser` dialog. The default directory is the one from which ProSS was launched and the default file name is `Shares.txt`. If the "One share per file" checkbox is selected, the "Save" button opens the same window but saves n files `Share`n`.txt` (Figure 12) in the same directory.

```
Share 0:
(1,136510872213167)
Key;
146814818097499
```

Figure 12: Example Share0.txt

On the reconstruction tab (Figure 15), users can input the number of shares to combine, along with the actual shares and the key to perform modular arithmetic. Shares can be entered manually or loaded from one file or multiple files. The "Load File" button opens a `FileChooser` windows where the user can select a share to load. Similarly, the "Load > 1 Files" button opens a `FileChooser` from which multiple files can be loaded. The key is loaded with the shares. Notice that loading only works if the file(s) containing the shares have not been altered, that is as long as they respect the format in Figure 13.

3 see section 3.4.1.1

```
Share X:
(X,XXXXX…XXXXXX)
(X,XXXXX…XXXXXX)
…
(X,XXXXX…XXXXXX)
Key;
XXXXXXXX…XXXXXXX
```

Figure 13: ShareX.txt valid format

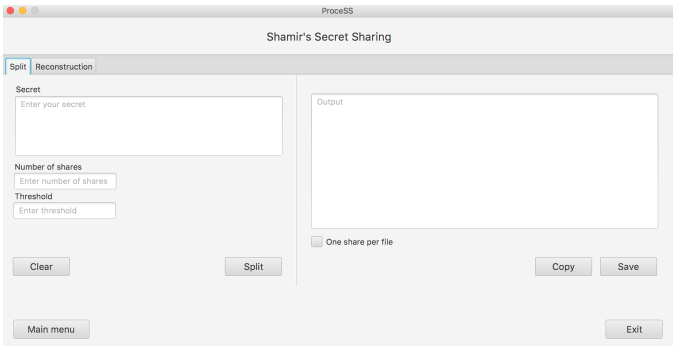

Figure 14: SSS splitting window

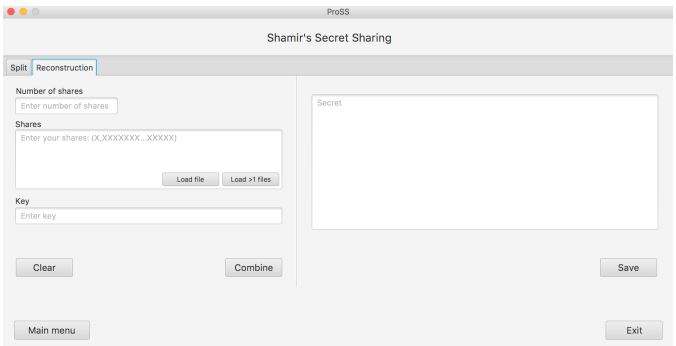

Figure 15: SSS reconstruction window

3.3.3 *Asmuth-Bloom*

The Asmuth-Bloom interface is exactly the same as SSS's. The only difference here, is that the "Split" button (in Figure 26) and "Combine" button (in Figure 28) call `ABS.split()` and `ABS.recontruct()` respectively.

3.3.4 *(n,n) XOR*

The XOR window (Figure 16 and 17) is simpler than the two previous ones. Since there is no threshold, the user can only input the secret message and the required number of shares. Also, as no key is needed to reconstruct the secret, the reconstruction phase only

takes shares and the number of shares, which can be inputted in the "Shares" and "Number of Shares" TextAreas respectively.

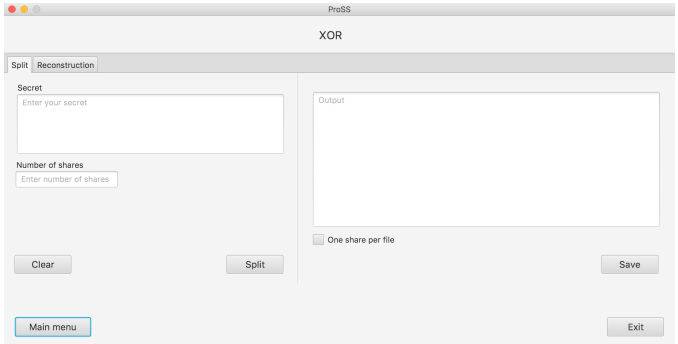

Figure 16: XOR splitting window

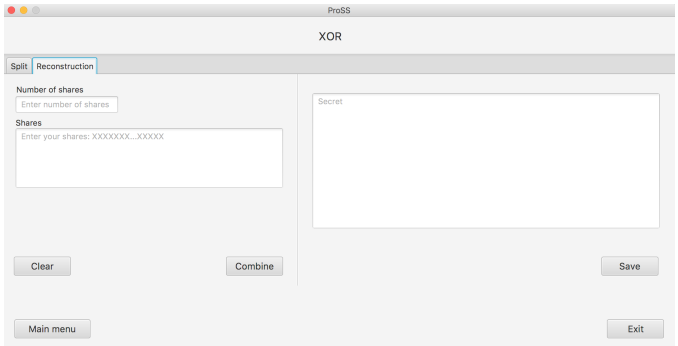

Figure 17: XOR reconstruction window

3.3.5 *Picture Sharing*

In Figure 18, `pss.fxml` creates a form with 3 TextFields that will accept the path of a secret picture, the number of shares to plit the image in, and the threshold. The path can be written manually or loaded via the `FileChooser` window opened by the "Load Image" button. If a valid path is entered or loaded, that is if the image exists, an ImageView below the "Secret" Textfield displays a preview of the selected image. Scheme selection is done by selecting the right Checkbox (currently XOR or SS). For convenience, when XOR is ticked, the threshold Textfield is disabled. The output TextArea, on the right, displays a validation message ("Success! # shares have been saved") once splitting is done. A "Go to File" button opens the directory where the shadow images have been saved.

On the reconstruction tab (Figure 19, the user can input the number of shares to be combined together and load them by clicking the "Load Images" button. Alternatively, paths to shadows images can be entered manually. The "Combine" button starts the reconstruc-

tion and the image is displayed on the right hand side if the process was successful. A "Go to File" button opens the directory where the reconstructed secret image has been saved.

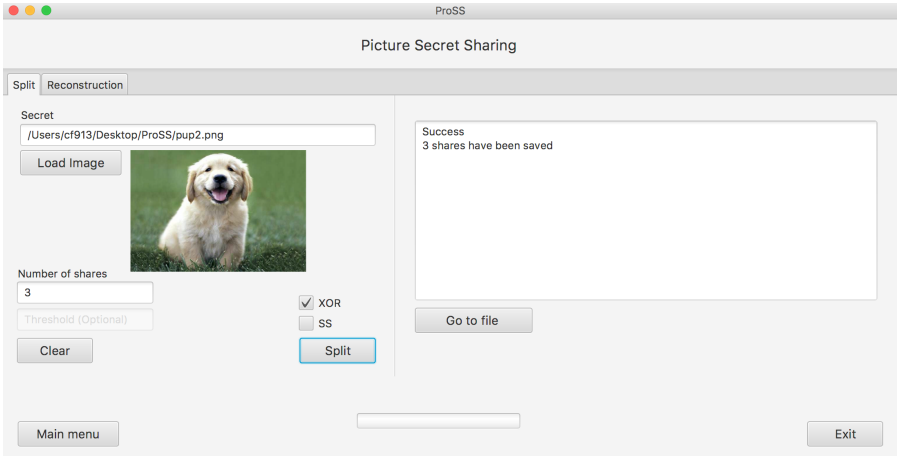

Figure 18: Picture Sharing splitting window

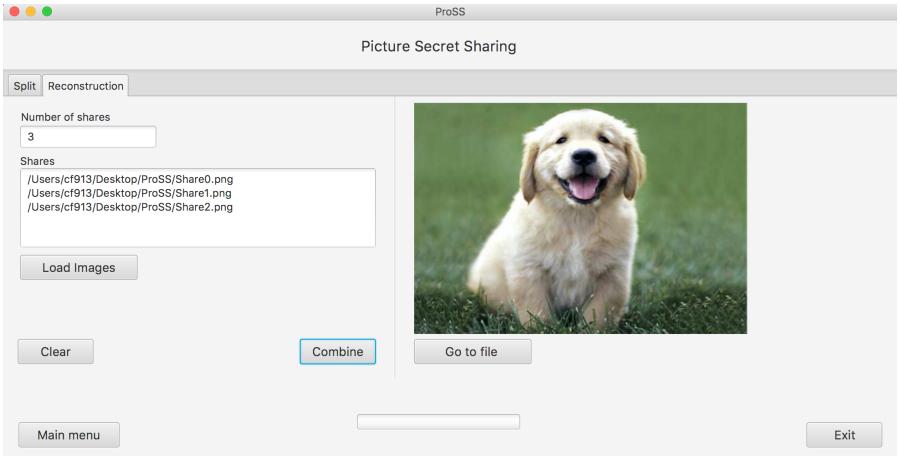

Figure 19: Picture Sharing reconstruction window

3.4 IMPLEMENTED SECRET SHARING SCHEMES

In this section, we are looking at how each scheme works internally. We explain how we implemented our splitting and reconstruction methods. For this, we will go through each scheme with the same input:

Secret s = "SEC" (for SECret)
Number of shares $n = 3$
Threshold $t = 2$

3.4.1 *Shamir's Secret Sharing*

3.4.1.1 *Split*

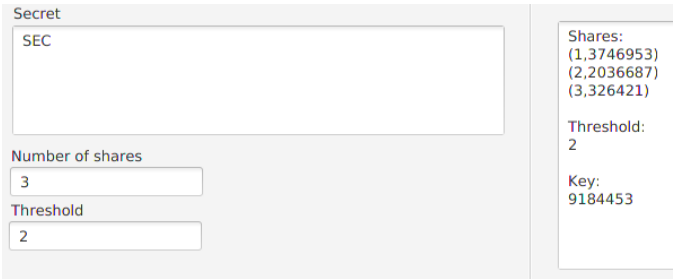

Figure 20: SSS Splitting example

Once the "Split" button is pressed, the Controller checks if all inputs are valid. If not, an error message pops up (Figure **??**). Otherwise, the Controller calls `SSS.split("SEC", 4, 3)`.

`SSS.split(...)` (Figure 21) first converts "SEC" into a BigInteger *secret*. Then a prime number *prime* the same bitsize as *secret* is generated with `newPrime(...)`. The following loop makes sure *prime* > *secret*. Next, a random polynomial of degree $t-1$ is generated and coefficients of degree 1 to $t-1$ are added to a list `coefs[]` of length t while `coef[0]` = *secret*.

Variable values in `SSS.split()` (Figure 21) for our model input:

```
secret: 5457219
prime: 9184453
degree: 1
coefs: [5457219, 7474187]
newShares: (1,3746953)\n(2,2036687)\n(3,326421)
Runtime:12630838
```

Now, `split_shares(...)` (Figure 22)takes the list of coefficients, n and *prime* to compute the polynomial value for all n over field *prime*. Shares are returned as a String of the form "share1\nshare2\n...\nshare$n$". Finally, `SSS.split(...)` returns `String[3]` where `String[0] = shares`,

```
    public static String[] split(String message, int shares, int
                                 threshold) {

          String msg         = message;
          BigInteger secret  = new BigInteger(msg.getBytes());

          BigInteger prime = newPrime(secret.bitLength());
          if ((secret.compareTo(prime)>= 0)) {
                prime = newPrime((secret.bitLength() + 1);
          }

          int degree = threshold - 1;

          BigInteger[] coefs;
          coefs = newPolynomial(degree, secret);

          String newShares;
          newShares = split_shares(coefs, shares, prime);

          String[] result = new String[3];
          result[0] = newShares;
          result[1] = prime.toString();
          result[2] = String.valueOf(threshold);

          return result;
    }
```

Figure 21: SSS.split()

```
private static String split_shares(BigInteger[] coef, int shares,
                                   BigInteger prime) {

   String newShares = "";
   for (int j = 1; j <= shares; j++) {
      BigInteger k = BigInteger.valueOf(j);
      BigInteger v = coef[0];

      for (int i = 1; i < coef.length; i ++) {
         v = (v.add(k.pow(i).mod(prime).multiply(coef[i])).mod(prime)).mod(prime);
      }

      newShares = newShares.concat("(" + k + "," + v + ")\n");

   }
   return newShares;
}
```

Figure 22: split_shares()

`String[1]` = t and `String[2]` = *prime*. Then the Controller updates the Output TextArea (Figure 20). Using Strings makes output formatting easier.

3.4.1.2 *Reconstruction*

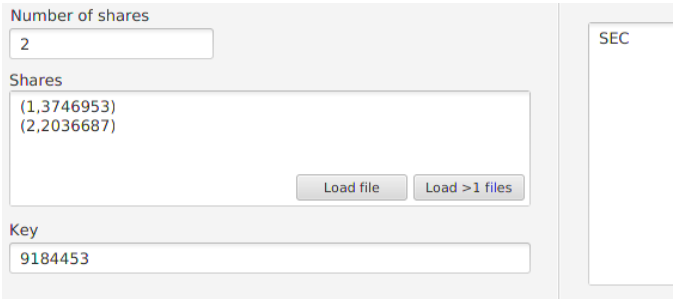

Figure 23: SSS Combining example

The reconstruction function `SSS.reconstruct(...)` first translate inputted shares into a list `newShares` of Pair$< k,v >$. For instance, in Figure 23, for share 1, $k = 1$ and $v = 507716$. The resulting list and *prime* are given to `recontruct_shares(...)` (Figure 24) which takes care of Lagrange Interpolation [4] over *prime*.

```
private static BigInteger reconstruct_shares(List<Pair<Integer,
                                BigInteger>> newShares, BigInteger prime) {
  BigInteger s = BigInteger.ZERO;

  //Lagrange interpolation to find coef[0]
  for (int i = 0; i < newShares.size(); i++) {
    BigInteger num = BigInteger.ONE;
    BigInteger denum = BigInteger.ONE;
    for (int j = 0; j < newShares.size(); j++) {
      if (i != j) {
        num   = num.multiply(BigInteger.valueOf(-
                newShares.get(j).getL())).mod(prime);

        denum = denum.multiply(BigInteger.valueOf(newShares.get(i).getL()-
                newShares.get(j).getL())).mod(prime);
      }
    }
    BigInteger value = newShares.get(i).getR();
    s = s.add(value.multiply(num).multiply(denum.modInverse(prime))).mod(prime);
  }
  return s;
}
```

Figure 24: reconctruct_shares()

$$\ell_i(x) = \prod_{j=0,\ j\neq i}^{n} \frac{x - x_j}{x_i - x_j} mod\ prime \tag{6}$$

Here are variable values in `SSS.reconstruct_shares()` (Figure 24) for our model input: (in this example, `<var_name>[i][j]` or `<var_name>[i]` are not Arrays. `i` and `j` act as loop counters)

4 see Section 24

```
num[0][1]: 9184451
denum[0][1]: 9184452
value[0]: 3746953
s[0]: 7493906
num[1][0]: 9184452
denum[1][0]: 1
value[1]: 2036687
s[1]: 5457219
```

This function returns the coefficient of degree 0: the secret. BigInteger secret is then converted back to a String (using String's `to.ByteArray()`) and displayed in the Output TextArea.

3.4.2 *Asmuth-Bloom Secret Sharing*

3.4.2.1 *Split*

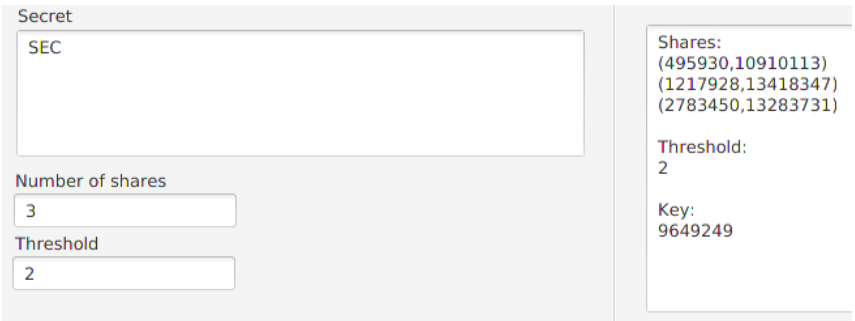

Figure 25: AB Splitting example

The Asmuth-Bloom Scheme differs from Shamir's in that the first part of the process consists in generating a list of pairwise co-prime numbers [5] instead of a list of random coefficients. In Figure 26 (lines 15-32), the do-while loop verifies the validity of the list, that is if $r, m_1 <...< m_n$ and $r * m_{n-t+2}...m_n < m_1...m_t$. Next, α is generated such that $S + \alpha * r \in \mathbb{Z}m_1...m_t$. Our function `getAlpha(...)` randoms a prime α of size $m_1 * ... * m_t$ and decreases its size by one bit until is reaches a valid value. It takes up to three iterations to find a valid α. Finally, `computShares(...)` concatenates all shares into the output String. As for Shamir's, the output format String[3] is suited for displaying the results on the Output TextArea (Figure 25).

Here are variable values in `ABS.split()` (Figure 26) for our model input: (in this example, `<var_name>[i][j]` or `<var_name>[i]` are not Arrays. `i` and `j` act as loop counters)

5 see section 2.4.4.3

```
secret: 5457219
m: [9649249, 10910113, 13418347, 13283731]
alpha: 818143
bloom: 7894470981826
newshares: (495930,10910113)\n(1217928,13418347)\n(2783450,13283731)
```

```
public static String[] split(String message, int shares, int threshold) {

  String[] results  = new String[3];
  String msg        = message;
  BigInteger secret = new BigInteger(msg.getBytes());

  BigInteger[] m = new BigInteger[shares + 1];

  // list of ms
  BigInteger left;
  BigInteger right;
  int p = 0;
  m[0] = SSS.newPrime(secret.bitLength()+1);

  do {
    for (int i = 1; i < m.length; i++) {
      do {
        m[i] = SSS.newPrime(secret.bitLength() + 1);
      } while (inList(m[i],m, i));
    }
    left = m[0];
    right = BigInteger.ONE;

    for (int i = shares-threshold+2; i <= shares; i++) {
      left = left.multiply(m[i]);
    }

    for (int i = 1; i <= threshold; i++) {
      right = right.multiply(m[i]);
    }
    p++;
  } while (left.compareTo(right) >= 0);

  key = m[0];

  BigInteger[] temp = m;
  BigInteger alpha = getAlpha(m, secret, threshold);
  BigInteger bloom = (secret.add(m[0].multiply(alpha)));

  String newshares = computShares(temp, bloom, shares);
  results[0] = newshares;
  results[1] = String.valueOf(threshold);
  results[2] = String.valueOf(key);

  return results;
}
```

Figure 26: ABS.split()

3.4.2.2 *Reconstruction*

The first step of reconstruction is the computation of M [6] (Figure 28). The next loop calculates values for Bezout's lemma using the extended Euclidian Algorithm implemented in the `euclid(...)` function. The BigInteger resulting from the loop is then moded by M and the key revealing the secret number. The secret is converted back to a String and displayed on the ABS output form.

6 see section 2.4.4.3

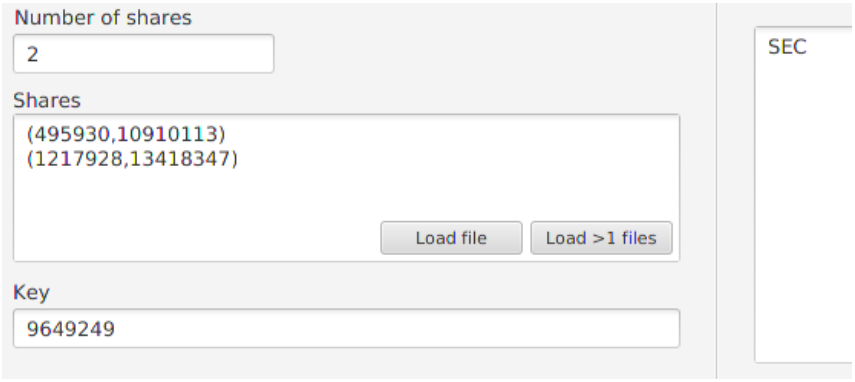

Figure 27: AB Reconstruction example

```
public static String reconstruct(int num, String shares, BigInteger key) {

        String result;
        String[] list = shares.split("\\n");

        String nobracket;
        String[] s;
        BigInteger bigm = BigInteger.ONE;
        for (int i = 0; i < list.length; i++) {
            nobracket = list[i].replaceAll("[\\[\\](){}]", "");
            s = nobracket.split(",");
            bigm = bigm.multiply(new BigInteger(s[1]));
        }

        BigInteger res = BigInteger.ZERO;
        for (int i = 0; i < list.length; i++) {
            nobracket = list[i].replaceAll("[\\[\\](){}]", "");
            s = nobracket.split(",");
            BigInteger a = new BigInteger(s[1]);
            BigInteger b = bigm.divide(new BigInteger(s[1]));
            BigInteger[] eu = euclid(a,b);
            res = res.add((new BigInteger(s[0])).multiply(eu[1].multiply(b)));
        }
        result = new String(res.mod(bigm).mod(key).toByteArray());

        return result;
}
```

Figure 28: ABS.reconstruct()

Here are variable values in `ABS.reconstruct()` (Figure 28) for our model input: (in this example, `<var_name>[i][j]` or `<var_name>[i]` are not Arrays. `i` and `j` act as loop counters)

```
list: [(495930,10910113), (1217928,13418347)]
bigm: 146395682043211
a[0]: 10910113
b[0]: 13418347
eu[0]: [2384236,-1938561]
res[0]: -12900272092726325310
a[1]: 13418347
b[1]: 10910113
eu[1]: [-1938561,2384236]
res[1]: 18780817152430434594
result: SEC
```

`Runtime:2608843`

3.4.3 *XOR secret sharing*

This implementation is self-explanatory. The splitting phase starts with `newshares(...)` which generates a list of random numbers of the same bitlength as the secret message: shares 1 to $n-1$. Next, `computXor(...)` (Figure 30) computes the XORing of the secret and all elements of the previous list into share n. The reconstruction phase is the splitting phase reversed.

```
public static String split(String message, int shares) {

    String result = "";
    BigInteger bitsize = new BigInteger(message.getBytes());


    String[] list = newshares(bitsize, shares);
    BigInteger px = computeXor(bitsize, list);
    list[shares-1] = String.valueOf(px);

    for (String s: list) {
        result = result.concat(s+"\n");
    }

    return result;
}
```

Figure 29: XOR.split()

```
private static BigInteger computeXor(BigInteger bitsize, String[] list) {

  BigInteger result = bitsize;
  for (int i = 0; i < list.length-1; i++) {
    result = result.xor(new BigInteger(list[i]));
  }

return result;
}
```

Figure 30: computXOR()

```
public static String combine(int num, String shares) {

  String[] list = shares.split("\\n");
  BigInteger result = computeXor(list);

  return new String(result.toByteArray());
}
```

Figure 31: XOR.combine()

3.5 IMAGE SECRET SHARING

3.5.1 *XOR based Coloured Image Secret Sharing*

In this scheme, colors are represented in RBG (R,G,B) with values ranging from 0 to 255, where 0 is black and 255 is bright red, green or blue. We first take a $w * h$ pixel image I. For each pixel $I_{i,j}$, we take values $R_{i,j}$, $G_{i,j}$ and $B_{i,j}$. We then generate 3 lists Rs, Gs and Bs of n random numbers between 0-255. We will name the three lists $R_{i,j}$, $G_{i,j}$ and $B_{i,j}$. In each list, the first element is set to the values of $I_{i,j}$. For each shadow image S_k where $k \in 1...n-1$, the pixel $S_{ki,j}$ is set as (Rs[k], Gs[k], Bs[k]). Finally, for the last share S_n, pixel $S_{ni,j}$ takes the value (Rs.XOR, Gs.XOR, Bs.XOR) where $list$.XOR is the value resulting from XORing all elements from list $list$.

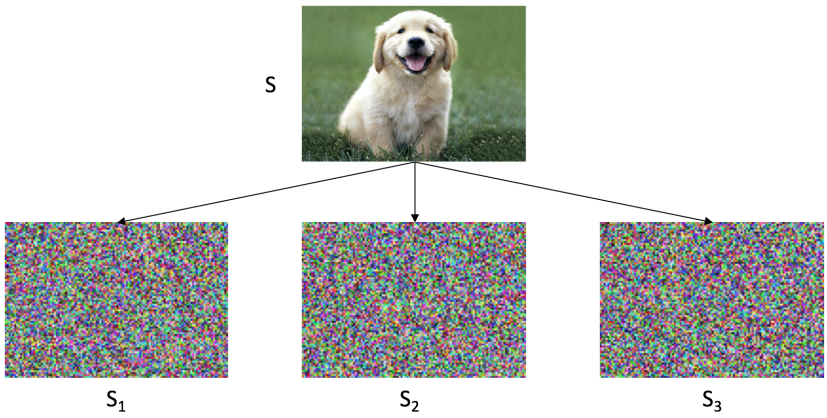

Figure 32: (3,3)-XOR Image Sharing [7]

The recovery of the secret is done by taking each pixel from each share and XORing all red, blue, and green values into the original values R, G, and B respectively. Since this scheme is based on XOR, it inherits its properties. Therefore, for an image S, all shared are of size S and less than n shadow images do not reveal any information about S. This scheme works fine if all participants have to be present in order to reconstruct the secret.

3.5.2 *SSS based Grayscale Image Secret Sharing*

In this scheme, we want to share a $w * h$ pixel image I between n parties such as t parties are need to recover i. As for the previous scheme, colors are represented in RBG (R,G,B) with values ranging from 0 to 255, where 0 is black and 255 is bright red, green or blue. This scheme takes any image and returns a grayscale version of it.

7 We used a lower definition version of image S to make the shadow images clearer

We first take a pixel $I_{i,j}$. We then calculate the average value of $R_{i,j}$, $G_{i,j}$ and $B_{i,j}$. Lets call it *avg*. This value is then given to our modified version of `SSS.split()`. Since 251 is the biggest prime smaller than 255, and for SSS to work, *prime* has to be bigger than S, we have to slightly alter the image. If $avg >= 251$, we set it down to 250 to allow computation. Next, the pixel is split into n shares $(n+z, U)$ using SSS. n is the share number, z is a random integer to increase the shadow image randomness and U is the value $f(n+z)$ of the polynomial. The corresponding pixel of the first share will look like this: $(1+z, U, prime)$. The order of R,G,B depends on a $imod3$. For instance, if the first pixel of the first share is $(1+z, U, prime)$, the second would be $(prime, 1+z, avg)$, the third $(U, prime, 1+z)$ and so forth. Repeat those steps for all $I_{i,j}$.

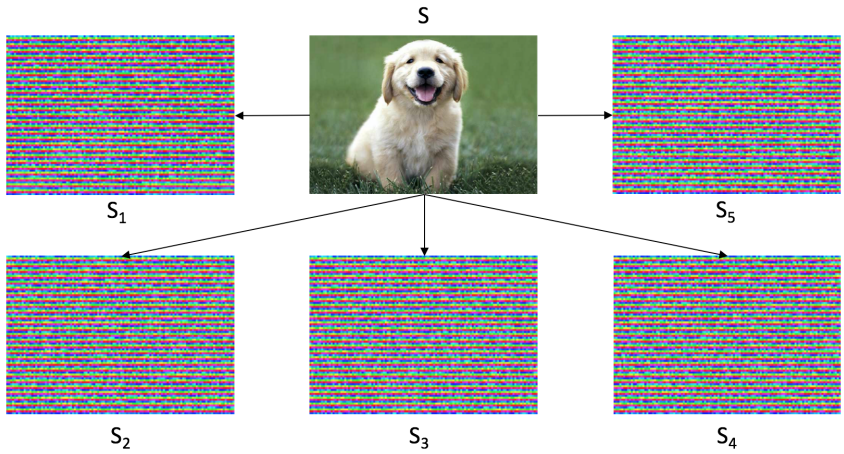

Figure 33: (2,5)-threshold SSS Image splitting

In order to successfully recover the secret image I, we need at least t shares. We take the first pixel $P_{1,1}$ from each shadow image and extract the reconstruction parameters from the color values. The original *avg* color is calculated using Lagrange interpolation. Finally the resulting value is set to $I_{1,1}$ for R,G and B. Once this is done for all pixels in each shares, the secret image appears in black and white (Figure 34).

Our scheme generates is not perfect since it generates lossy shares of size I. Compared to a scheme like Thien and Lin [8] our scheme is less efficient in storing space. Additionally, our shadow images look less random. However, we only use shadow images to rebuild the secret image. This makes our scheme more robust than Thien and Lin because reconstruction does not depend on a separate key.

8 C.-C. Thien and J.-C. Lin, "Secret image sharing," Computers Graphics, vol. 26, no. 5, pp. 765–770, 2002.

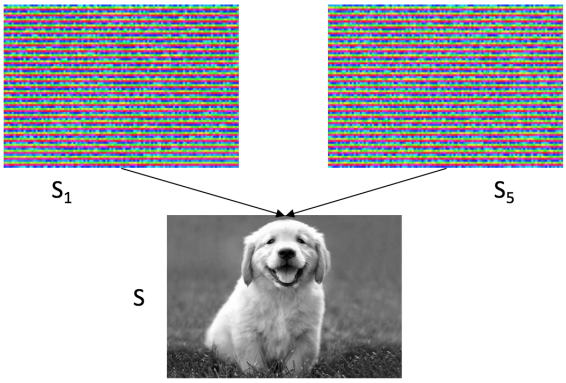

Figure 34: (2,5)-threshold SSS Image Combining

3.6 EXTENSIONS

We have built our application in an extension-friendly manner so that it only takes a few step to add a new scheme. The code is well commented[9] and of course open to adjustments. Here are a few example to follow in order to add a new scheme to our program:

3.6.1 *The scheme class*

The new scheme class contains all important methods to perform the secret sharing. It is recommended to use the same format as our existing classes. Thus, it has to contain two main functions, `split()` and `combine`, and auxiliary methods. It is possible to use auxiliary methods from other classes by setting them visibility to public or protected.

3.6.2 *The GUI*

Users can create new windows for their scheme by taking the existing `.fxml` files as template. For threshold scheme, `sss.fxml` or `abs.fxml` are a great start. For secret splitting, `xor.fxml` would be more adequate. `pss.fxml` can be used for any kind of image secret sharing.

3.6.3 *The Controller*

The controller class `ControllerSS.java` handles the interactions between the UI and the main classes. It is here that the user can link the new class with its corresponding window. Two main methods `dothesplit` and `dothecombine` control the action following a click on "Split" and "Combine" respectively, for all schemes. As seen on Figure 35, add a call to the new scheme in the main `if` statement of `dothesplit` or `dothecombine`. Most of the functions in `ControllerSS` can be reused unless the user adds new buttons with new functions.

```
if (e.getSource()==splitbutton) {
        s = SSS.split(secret, shares, thresh);
} else if (e.getSource()==splitbuttonxor) {
        s = ABS.split(secret, shares, thresh);
} else if (e.getSource()==NEWBUTTON{
        // Some action
}
```

Figure 35: Scheme selection in dothesplit

9 comments have been removed from figures to save space

3.7 TESTING

Our building process was often interrupted by a testing phase. We tested every scheme before and after adding them to the GUI. Our testing process was first done manually, by inputting different setting one by one. We then automatised our testing using a function that would run a given scheme over many input settings. Our main issue, that took a while to debug was bad uses of `Integers` where we assumed they would not exceed the maximum value. For instance in the early stage of our SSS implementation, for secrets of different sizes, there was a different maximum working threshold value. Past this value, the shares would be corrupted.

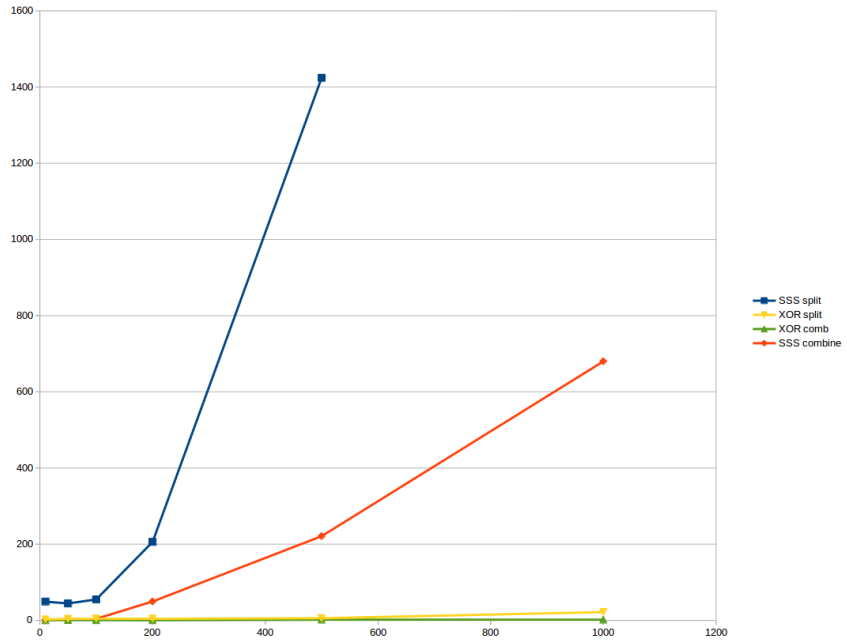

Figure 36: Performance of our implementation (X axis: number of shares, Y axis: time in milliseconds)

Figure 36 shows the performance of our implemented secret sharing schemes for the secret message **"Hello World, this is a secret"**. As our schemes convert the secret from string to BigInteger, the secret is processed as the 70-digits number:

`1951797304363278098593211625885632000047329579345786637569862164178292`

In this test, number of shares = threshold. XOR is clearly faster than SSS because it does not require as much processing and mathematical calculations. However, SSS is still fast enough for convenient use compared to Asmuth-Bloom. The latter is not shown on the graph because it took 45 seconds to split the message into 10 shares (see 2). For the same secret, processing 20 shares takes more than one hour. The problem not being the number of shares, but the threshold. For

instance, AB is able to generate 100 shares with threshold 5 in about 5 seconds.

Shares	Split	Combine
2	411.9	9.9
3	709	5.5
4	1841	4.6
5	2791	3.4
6	9338	4.9
7	11966	9.3
8	20239	6.1
9	39314	6.3
10	45501	8.2
15	257054	20.3

Table 2: ABS performance in milliseconds for secret "Hello world, this is a secret" (n = t)

The performances of our image sharing schemes follow the trend of the previous schemes; the SSSS one is noticeably slower than the XOR (see Figure 37). Also, since the secret varies from 0 to 255, computing Shamir's scheme takes roughly the same time for each pixel. This means that the running time of SS depends mainly on the size of the image.

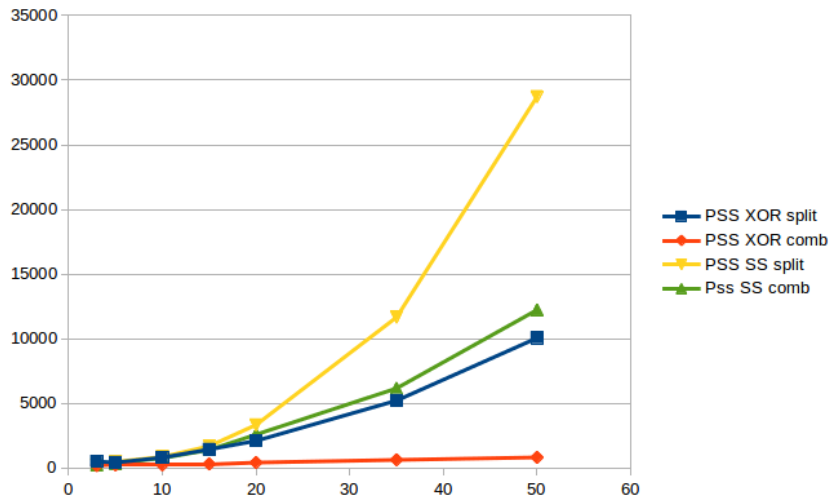

Figure 37: Performance of Image Secret Sharing (X axis: number of shares, Y axis: time in milliseconds) for a 192x108p .PNG

4

EVALUATION AND FUTURE WORK

The goal of this project was to implement a working secret sharing learning tool based on existing secret sharing protocols. ProSS is close to what we were expected to achieve. We have a user friendly interface that includes 4 working implementations of secret sharing schemes. However, here are some things we could have done/added to make our program better:

4.1 STRINGS VS LISTS

Our program uses Strings, lots of them, everywhere. ProSS would probably be more efficient speed-wise if we had used Arrays instead of concatenating Strings. However, String handling is easy and our main goal was to make a simple application.

4.2 DETAILED OUTPUT WINDOW

Since our application is destined to be used in a teaching environment, we assumed it would be ran from source in an IDE, and therefore all `System.out.println()` statements would be visible. Those outputs are important to understand each scheme. To solve this issue, we could have added a verbose mode, which when activated would display in a window the details of secret sharing (secret size, coefficients, prime numbers, pixel color, and more...) Such outputs were extremely useful especially during the testing phase and would probably be helpful for a user new to secret sharing.

4.3 MULTI-THREADING

One thing that we have not implement yet, but that would have been helpful for testing and convenience of use, is multithreading handling. The problem is that when the size of the secret is significantly big (ex: a very high definition picture), the program freezes completely while a splitting or combining process is running. For example, using XOR image secret sharing, splitting a 1920x1080 image in 3 shares will lock the program for about 14 seconds. Since there is

no indication of completion such as a progress bar, we can't know for sure if the program crashed of if it is still working on splitting the image. A solution to this problem is to open a new thread for splitting and combining so that the application stays usable instead of not responding and sometimes shutting down. Also, since the program doesn't freeze, we could add a progress bar that displays a correct estimation of remaining processing time.

4.4 THE GO TO FILE BUTTON

During our testing phase, we thoroughly tested our GUI on Windows, Ubuntu and OS X. The only issue we discovered is one we were not able to fix. The "Go to file" button on both "split" and "reconstruction" windows of picture secret sharing does not work on Linux. It is supposed to open a file browser in the directory where the shadow images were saved. When a user clicks the button, the program crashes. Surprisingly, it works fine on other operating systems.

4.5 A COLOR IMAGE SECRET SHARING BASED ON SSS

Our SSS-based scheme presented in section 3.5.1 does not enable colour image sharing. The output image is always grayscale. Our system is limited by the use of (R,G,B) as a color format. If we keep the structure of our scheme, we are going to need a pixel format that can store 3 colors, a share number and a prime.

4.6 MORE SCHEMES AND OPTIMISATIONS

Adding new schemes is one of the reason why we made this application. We look to keep it updated to the latest secret sharing standard. Eventually, we could try to optimise the efficiency of our schemes.

5

CONCLUSION

In this project, we have presented an educational Java implementation of secret sharing: **ProSS**. This program currently implements 4 main Secret Sharing Schemes: Shamir Secret Sharing Scheme, Asmuth-Bloom Secret Sharing Scheme, XOR Secret Sharing and Image Secret Sharing. Those schemes are put together in a user-friendly GUI compatible with most operating systems. Easily extensible, ProSS offers a clear, well commented code base for students new to secret sharing.

BIBLIOGRAPHY

[1] Shamir, Adi (1979). *How to share a secret*, Communications of the ACM22(11): p. 612–613

[2] Blakley, G. R. (1979). *Safeguarding cryptographic keys*, Proceedings of the National Computer Conference 48: p. 313–317

[3] Jun Kurihara et al. (2008). *A New (k,n)-Threshold Secret Sharing Scheme and Its Extension*, Proceedings of the 11th international conference on Information Security: p. 455-470

[4] Rivest, R.; Shamir, A.; Adleman, L. (1978). *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*, Communications of the ACM 21 (2): p. 120–126.

[5] B. Chor, S. Goldwasser, S. Micali and B. Awerbuch. *Verifiable Secret Sharing and Achieving Simultaneity in the Presence of Faults*, FOCS85: p. 383-395.

[6] P.Feldman (1987). *A practical scheme for non-interactive verifiable secret sharing*, 28th Annual Aymposium on Foundations of Computer Sciences: p. 427-437

[7] M. Stadler (1996). *Publicly verifiable secret sharing*, Proceedings of the 15th annual international conference on Theory and application of cryptographic techniques, EUROCRYPT'96, Springer-Verlag, Berlin, Heidelberg, p. 190–199.

[8] Shannon, Claude (1949). *Communication Theory of Secrecy Systems*, Bell System Technical Journal 28: p. 656–715.

[9] Naor, M. and A. Shamir. (1995). *Visual cryptography, Advances in cryptology*, Eurocrypt '94 Proceeding LNCS, 950:1–12.

[10] C.-C. Thien and J.-C. Lin. (2002). *Secret image sharing* Computers Graphics, vol. 26, no. 5, pp. 765–770.