Draft of the Final Completion Report of the Project on *"Secret Sharing Schemes Using DNA Cryptography*

Supported by

Ministry of Communications and Information Technology,
Department of Information Technology,
Govt. of India

Undertaken by

Department of Pure Mathematics,
Calcutta University
35 Ballygunge Circular Road,
Kolkata - 700019,
West Bengal,
India

Team Members of the Project

Calcutta University

Principal Investigator

Dr. Avishek Adhikari, Department of Pure Mathematics, Calcutta University.

Other Investigators (faculty)

Professor M R Adhikari, Department of Pure Mathematics, Calcutta University.
Dr. Rajat Bandyopadhyay, Department of Bio-Technology, Calcutta University.

Other Investigators (non-faculty Project Scientists)

Bappaditya Roy,

Partha Sarathi Roy.

DNA Secret Sharing

The broader objective of our project is to find a strong connection between the two emerging subjects, namely cryptography from computer science and DNA computing to develop a perfectly secured DNA secret sharing scheme for threshold as well as for general access structure by using mathematics and statistics.

Some natural questions come to our mind : what is a secret sharing scheme? What is the purpose of its use and how DNA may be used for secret sharing?

Let us first discuss the usefulness of secret sharing schemes. Due to the recent development of computers and computer networks, huge amount of digital data can easily be transmitted or stored. But the transmitted data in networks or stored data in computers may easily be destroyed or substituted by enemies if the data are not enciphered by some cryptographic tools.

However, we may have other threats such as troubles of storage devices or attacks of destruction. In order to prevent such attacks, we must make as many copies of the secret as possible. But, if we have many copies of the secret, the secret may be leaked out, and hence, the number of the copies should be as small as possible. In this situation, cryptography plays an important role. In a standard public-key cryptosystem [9], only the person who holds a secret key is able to perform the cryptographic task (decrypting or signing) corresponding to the related public key. It is desirable that actions or secrets to be protected by more than one key (jointly or separately), or that there be several keys and more than one way to recover the secret to initiate the action, using different combinations of keys. Secret sharing schemes are essential components of these distributed cryptosystems. Secret sharing schemes came into prominence in 1979 when two papers, one by Blakley [3] and one by Shamir [17], were published independently. A (t, n) *threshold secret sharing scheme* is a method whereby n pieces of information of the secret key K, called *shares* are distributed to n participants so that the secret key can be reconstructed from the knowledge of any t or more shares and the secret key can not be reconstructed from the knowledge of fewer than t shares.

But in reality, there are many situations in which it is desirable to have a more flexible arrangement for reconstructing the secret key. Given some n participants, the situation may demand to designate certain authorized groups of participants (called qualified sets of participants) who can use their shares to recover the key but certain sets of participants (called forbidden sets of participants) who can not get any information regarding the secret even though the use their shares collectively. This kind of scheme is called a general secret sharing scheme. Formally, we can define it as follows :
Let $\mathcal{P} = \{1, 2, \ldots, n\}$ be a set of n participants and $2^{\mathcal{P}}$ denote the power set of $\mathcal{P}$. Let Γ_{Qual} and Γ_{Forb} denote respectively the collections of all qualified and forbidden sets of participants. So, $\Gamma_{Qual}, \Gamma_{Forb} \subseteq 2^{\mathcal{P}}$ with $\Gamma_{Qual} \cap \Gamma_{Forb} = \phi$. Then a $(\Gamma_{Qual}, \Gamma_{Forb})$ *secret sharing scheme* is

a method of sharing a secret K among a finite set of participants $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ in such a way that

1. if the participants in $\mathcal{A} \subseteq \mathcal{P}$ are qualified to know the secret, they can reconstruct the secret K by pooling together their partial informations, known as shares.
2. any set $\mathcal{B} \subset \mathcal{P}$ which is not qualified to know K, cannot reconstruct the secret K.

The key is chosen by a special participant $\mathcal{D}$, called the *dealer*, and it is usually assumed that $\mathcal{D} \notin \mathcal{P}$. The dealer gives partial information, called *share*, to each participant to share the secret key K. Γ_0, called the basis of the scheme, is the family of all minimal qualified subsets. A secret sharing scheme is said to be perfect if the condition 2 of the above definition is strengthened as follows :
Any unauthorized group of shares cannot be used to gain any information about the secret key that is if an unauthorized subset of participants $\mathcal{B} \subset \mathcal{P}$ pool their shares, they can determine nothing more than any outsider about the value of the secret K. After the introduction of secret sharing, significant works have been done by many authors. Some of them are Blundo et al [5], Brickell and Davenport [6], Golic [11], Okada et al [14], Rabin and M. Ben-Or [15].

DNA secret sharing was first introduced by the author in [1]. In that paper the author dealt with only general access structure. But in this project we first develop a threshold $(2, n)$-DNA secret sharing scheme using identity matrices and then PBIBD. Next we construct a (n, n)-DNA secret sharing scheme. Finally, we develop a DNA secret sharing for general access structure using (n, n)-DNA threshold scheme.

Let us now explain why DNA may be used as a medium for secret sharing [1]. Three major reasons, namely very small size, huge storage capacity and massive parallel processing in DNA computing drive us to think about DNA as a medium for secret sharing. Moreover, high longevity of DNA and availability of synthesized DNA make the secret sharing scheme more useful. In our method, we will implement a perfectly secure DNA secret sharing scheme using two simple techniques of Biotechnology known as mixing and automated DNA sequencing. Due to the use of simple techniques, our scheme has low error rate and it is easy to implement.

One of the main advantages of our DNA secret sharing scheme over other secret sharing schemes is that a significant amount of secret information can be carried out in a limited amount of physical space due to the compact nature of DNA. Also due to the massive parallelism, one of the main operations in our scheme that is the "or" operation can be carried out very efficiently using DNA computing. Finally, the compact nature together with the high longevity of the DNA make our DNA secret sharing scheme applicable for secret agents and defense organizations. In the next section, we discuss about the DNA model of computation.

0.1 Preliminaries on DNA computation

DNA, the magic code of life, has been known for over 50 years as genetic material of living systems. In recent works for high performance, DNA computing has considerable attention as one of non-silicon based computing. The biggest achievement in this area was carried out by Adleman in 1994 [2]. Recently, cryptography has been shown to be one of the new applications of DNA computing [8, 10, 13].

DNA is found naturally as a double stranded molecule, with a form similar to a twisted ladder. The backbone of the DNA helix is an alternating chain of sugars and phosphates, while the association between the two strands are variant combinations of the four nitrogenous bases adenine (A), thymine (T), guanine (G) and cytosine (C). The two ends of the strand are distinct and are conventionally denoted as 3' end and 5' end. Two strands of DNA can form (under suitable conditions) a double strand if the respective bases are Watson-Crick [18] complements of each other - A matches with T and C matches with G, also 3' end matches with 5' end.

0.1.1 Biological operations

Our fundamental model of computation is to apply a sequence of operations to a set of DNA double strands in a test tube. The main operations for our scheme are mixing and reading the DNA strands. In mixing [12], the contents of two test tubes is poured into a third one to achieve union. Mixing can be done by rehydrating the tube contents (if not already in solution) and then combining the fluids together into a new tube, by pouring and pumping. A process called automated DNA sequencing may be used to read the DNA double stands.

0.1.2 DNA encoding of binary strings

For a given string X over a set of alphabet $\{A, T, G, C\}$, we denote by $\updownarrow X$, the double strand DNA. Every binary string can be represented as a set of integers that corresponds to the positions where the bits are 1 from right to left. For example, the binary string 1011, can be represented as a set $\{1, 3, 4\}$, as the 1st, 3rd and 4th bit positions of 1011 are 1. So, for any binary string we can always associate a unique non empty subset of natural numbers using the above technique. But, for any nonempty subset of natural numbers we do not get unique binary string. For an example, for the set $\{1, 3, 4\}$ we can always associate binary strings 1011 and 10110. So, to get the unique binary string from a subset of natural number, we need the length of the binary string. So, the set $\{1, 3, 4\}$ along with the length 4 is equivalent to the binary string 1011. Now each integer i can be represented in a DNA double strand

notation as follows [4]:

$$ds_i = \updownarrow S_0(GAATTGC^5)^i GAATTCS_1,$$

where $\updownarrow GATTC$ is the restriction site for EcoRI and S_0 and S_1 may be any suitable 20 to 30 base pair long DNA strand not containing $\updownarrow GAATTC$ as a sub-strand. So the binary string α can be represented as a test tube $T[\alpha] = \{ds_i : i\text{th bit of } \alpha \text{ is } 1\}$. For example, if $\alpha = 1011$, then the DNA double strand representation of α can be given by the test tube $T[\alpha] = \{ds_1,\ ds_3,\ ds_4\}$. So from now on, we will freely switch between a binary string and its DNA double strand representation along with the length of the string.

0.1.3 Mathematical Operations using DNA computing

Suppose we want to make the Boolean "*or*" operation between two binary strings α and β. For example, if $\alpha = 1011$ and $\beta = 1001$, then the binary "*or*" of two strings will be 1011, as 1 "*or*"1 is 1, 1 "*or*"0 is 1, 0 "*or*"1 is 1 and 0 "*or*"0 is 0.

Now we want to represent this operation in DNA terminology [4]. Let $T[\alpha]$ and $T[\beta]$ denote respectively, the test tubes corresponding to the binary strings α and β. If we simply use the mixing procedure with $T[\alpha]$ and $T[\beta]$, it yields $T[\alpha] \cup T[\beta] = \{ds_i : i\text{th bit of } \alpha \text{ is } 1 \text{ or } i\text{th bit of } \beta \text{ is } 1\}$.

For example, if $\alpha = 1011$ and $\beta = 1001$, then $T[\alpha] = \{ds_1, ds_3, ds_4\}$ and $T[\beta] = \{ds_1, ds_4\}$ with string length 4. So applying mixing procedure with $T[\alpha]$ and $T[\beta]$, it yields $T[\alpha] \cup T[\beta] = \{ds_1, ds_3, ds_4\}$. This corresponds to α"*or*"β, i.e., 1011.

0.2 DNA secret sharing scheme

Using Coding theoretic technique, any message written in any language can be encoded into a binary string. So we assume without loss of generality that our secret message is a binary string. Suppose we want to distribute a secret binary string to a set $\{P_1, P_2, \ldots, P_n\}$ of n participants in such a way that certain designated set of participants can reveal the secret by pulling their shares and certain set of participants have no information about the secret. This kind of access structure is known as general access structure. So the main point is how the dealer will distribute the shares to each participants. For that we need the following concept of generating matrices. Before that we first introduce some notations. Consider an $n \times m$ Boolean matrix M and let $X \subseteq \{1, 2, \ldots, n\}$. Let $|X|$ denote the cardinality of X, $M[X]$ denote the $|X| \times m$ sub matrix obtained from M by retaining only the rows indexed by the elements of X, M^X denote the Boolean "or" of the rows of $M[X]$ and $BW(V)$ be the number of 1's in a Boolean vector V.

Definition 0.2.1 *Let $(\Gamma_{Qual}, \Gamma_{Forb})$ be an access structure on a set $\mathcal{P} = \{1, 2, \cdots, n\}$ of n participants. Two $n \times m$ Boolean matrices G_0 and G_1 whose ith row is associated with the ith participant are said to be generating matrices for $(\Gamma_{Qual}, \Gamma_{Forb})$ if the following two conditions are satisfied :*

1. *For any $X \in \Gamma_{Qual}$, $BW(G_0^X) < BW(G_1^X)$.*
2. *For any $Y \in \Gamma_{Forb}$, $G_1[Y]$ and $G_0[Y]$ are identical up to column permutations.*

So to construct a $(\Gamma_{Qual}, \Gamma_{Forb})$ secret sharing scheme, it is sufficient to construct two generating matrices G_0 and G_1.

0.2.1 $(2, n)$-threshold DNA secret sharing scheme using Identity matrices

Suppose the access structure is such that the secret is to be distributed among a set of n participants in such a way that any two or more participants can reveal the secret by pulling their shares but it is not possible for a single participant to get any information about the secret. This is a special case of general access structure and it is known as $(2, n)$ threshold access structure. In the subsequent subsections we shall describe how the secret can be distributed among the participants and how the set of two or more participants can reveal the secret by pulling the corresponding shares. Moreover we shall prove that the scheme is perfectly secure and it is not possible for a single participant to get any information regarding the secret from the share that he/she holds.

0.2.2 Secret sharing algorithm

We assume that the algorithm of DNA encoding of binary string is public i.e., known to every body. Let the dealer want to share the secret binary string $x = x_1x_2 \ldots x_k$ among n participants P_1, P_2, ..., P_n where not all $x_i = 0$ and not all $x_i = 1$, for $i = 1, 2, \cdots, k$. The dealer follows the following algorithm :

1. If x_i is 0, the dealer considers the generating matrix G_0 and gives a random permutation to the columns of G_0 and constructs a new matrix M_i, $i = 1, 2, \ldots, k$. If x_i is 1, the dealer does the same with the generating matrix G_1.
2. Concatenate M_i's to get a new matrix $M = M_1||M_2|| \ldots ||M_k$.
3. Each row α_i of M, $i = 1, 2. \ldots, n$ represents a binary string of length km. Encode each row α_i of M by a test tube $T[\alpha_i]$ in DNA double strand notation, $i = 1, 2. \ldots, n$.

4. To each participant P_i, give the test tube $T[\alpha_i]$, $i = 1, 2, \ldots, n$.

5. To each participants, give also the values k and m.

0.2.3 Decryption algorithm

Let $X = \{P_{t_1}, P_{t_2}, \ldots, P_{t_q}\} \in \Gamma_{Qual}$. Then they have the test tubes $T[\alpha_{t_j}]$, $j = 1, 2, \ldots, q$. Also they know the algorithm for DNA encoding of binary strings, the values k and m. They will use the following algorithm to get the secret message.

1. Use mixing procedure as described earlier with the test tubes $T[\alpha_{t_j}]$, $j = 1, 2, \ldots, q$.

2. Execute automated DNA sequencing method to read the DNA double strands.

3. Since the algorithm for DNA encoding of binary strings and the values k and m are known to the participants, convert the DNA strand notation into binary string. Let the resulting binary string be $y = y_1 y_2 \ldots y_{mk}$. Note that this y represents the binary "or" of the rows α_{t_j}, $j = 1, 2, \ldots, q$.

4. Since the value of m is known to the participants, they break y into k substrings of length m as $y = (y_{1_1} y_{1_2} \ldots y_{1_m})(y_{2_1} y_{2_2} \ldots y_{2_m}) \ldots (y_{k_1} y_{k_2} \ldots y_{k_m})$.

5. The participants will compute $w_i = BW(y_{i_1} y_{i_2} \ldots y_{i_m})$. Since not all $x_i = 0$ and not all $x_i = 1$, not all w_i's are equal. Let $w_{min} = \min_{i \in \{1,2,\ldots,k\}} w_i$.

6. For $X \in \Gamma_{Qual}$, as $BW(G_X^0) < BW(G_X^1)$, the participants will compute $z = z_1 z_2 \ldots z_k$, where $z_i = 1$ if $BW(y_{i_1} y_{i_2} \ldots y_{i_m}) > w_{min}$; and $z_i = 0$, otherwise.

Theorem 0.2.1 *The binary string z and the secret binary string x are same.*

Proof : The result follows from the fact that $y_{i_1} y_{i_2} \ldots y_{i_m}$ is a permutation of G_0^X if $BW(y_{i_1} y_{i_2} \ldots y_{i_m}) = w_{min}$ or G_1^X if $BW(y_{i_1} y_{i_2} \ldots y_{i_m}) > w_{min}$.

Theorem 0.2.2 *The above scheme is perfectly secure.*

Proof : To proof this result it is sufficient to proof that for any $Y \in \Gamma_{Forb}$, the probability that these participants can predict the secret binary string $x = x_1 x_2 \ldots x_n$ correctly is $(1/2)^n$. The result follows from the fact that $G_0[Y]$ and $G_1[Y]$ are identical up to column permutations. So just looking at the $G_0[Y]$ ($G_1[Y]$), it is not possible to predict correctly whether it comes from G_0 or G_1.
Note : If all x_i's are equal, then our scheme can predict the secret binary string correctly with probability 1/2. But for all practical uses, not all x_i's are equal.

Example 0.2.1 *Let us consider a (2,4)-DNA secret sharing scheme on a set* $\mathcal{P} = \{1, 2, 3, 4\}$ *of 4 participants, where* $\Gamma_0 = \{X \subseteq \mathcal{P} : |X| = 2\}$, $\Gamma_{Qual} = \{Y \subseteq \mathcal{P} : X \subseteq Y,$ *for some* $X \in \Gamma_0\}$ *and* $\Gamma_{Forb} = 2^{\mathcal{P}} \setminus \Gamma_{Qual}$. *So in a (2,4)-DNA secret sharing scheme the binary secret is to be distributed among a set of 6 participants in such a way that any set of two or more participants can reveal the secret but no information is gained about the secret by a single participant from his/her own share. To implement the scheme the dealer chooses two Boolean matrices* $G_0 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$ *and* $G_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$. *It can be shown that the two conditions of generating matrices are satisfied by* G_0 *and* G_1. *Now suppose that under this above secret sharing model the dealer wants to share the secret binary string* $x = x_1x_2x_3 = 101$ *to all the participants. Since* $x_1 = 1$, *the dealer considers the matrix* G_1 *and apply a random permutation to the columns of* G_1 *and produces a matrix* M_1. *Similarly, apply the same procedure for* x_2 *and* x_3 *on* G_0 *and* G_1 *respectively to produce two more matrices* M_2 *and* M_3. *Let* $M = M_1||M_2||M_3$, *where "*$||$*" denote the concatenation of matrices. For simplicity we assume that all the applied random permutations are identity permutation. So the first row of* M *is given by* $\alpha_1 = 100010001000$. *Similarly* $\alpha_2 = 010010000100$, $\alpha_3 = 0010100000010$, $\alpha_4 = 000110000001$. *Now the dealer converts the binary strings to DNA representations to get the test tubes* $T[\alpha_1] = \{ds_1, ds_5, ds_9\}$, $T[\alpha_2] = \{ds_2, ds_5, ds_{10}\}$, $T[\alpha_3] = \{ds_3, ds_5, ds_{11}\}$, $T[\alpha_4] = \{ds_4, ds_5, ds_{12},$ *where* ds_i *is defined earlier. The test tube* $T[\alpha_i]$ *is given to the participant* P_i, $i = 1, 2, \ldots, 4$. *Also the values* $m = 4$ *and* $k = 3$ *are given to the participants. For decryption, let the qualified set of participants* $\{P_1, P_2\}$ *come together. They use mixing procedure with test tubes* $T[\alpha_1]$ *and* $T[\alpha_2]$ *to get* $T[\alpha_1] \cup T[\alpha_2] = \{ds_1, ds_2, ds_5, ds_9, ds_{10}\}$. *With the knowledge of decoding the DNA representation to the binary string, the values of* $k = 3$ *and* $m = 4$, *the participants* P_1 *and* P_2 *can convert the DNA representation to the binary string* $y = 111011001110$. *Since, the value of* m *is known to the participants,* P_1 *and* P_2 *can break* y *as* $y = (1100)(1000)(1100)$. *Next they will find the value of* w_{min} *as 1 and then they will compute* $z = 101$, *as* $BW(1100) > 1$, $BW(1000) = 1$. *Thus* P_1 *and* P_2 *can recover the secret* 101. *Now suppose, any single participant say the* 1*st participant alone wants to get information about the secret. Then the two matrices* $G_0[\{1\}] = \begin{bmatrix} 1000 \end{bmatrix}$ *and* $G_1[\{1\}] = \begin{bmatrix} 1000 \end{bmatrix}$ *are identical up to column permutation. So* $BW(G_0^{\{1\}})$ *and* $BW(G_1^{\{1\}})$ *are equal. Thus just looking at these two matrices it is not possible to predict whether* $BW(G_0^{\{1\}})$ *or* $BW(G_1^{\{1\}})$ *correspond to 0 or 1. So no information is gained. Hence the scheme is totally secure.*

Now the question that comes to our mind is that how to construct generating matrices G_0 *and* G_1 *for a* $(2, n)$*-DNA secret sharing scheme. To construct the generating matrices we take the help of the Identity matrix.*

0.2.4 Construction of Generating Matrices for $(2, n)$ DNA secret sharing using identity marices

Theorem 0.2.3 *For any v, there exists a $(2, v)$-DNA secret sharing scheme can be constructed with v participants having $w_{min} = 1$ and v many columns in the generating matrices G_0 and G_1.*

Proof : Let N be the identity matrix of of order v. Take $G_1 = N$. Then $m = b$. Now in each row of N there are exactly one 1 and $w - 1$ 0's. Take the matrix G_0 to be a $w \times w$ boolean matrix with only first column to be all 1's and remaining all columns are zero columns. From the above discussion, it is clear that G_1 and G_0 satisfy the conditions of the Definition 0.2.1 with $w_{min} = 1$ and w many columns in the generating matrices G_0 or G_1.

0.2.5 $(2, n)$-threshold DNA secret sharing scheme using Partially Balanced Incomplete Block Designs

Suppose the access structure is such that the secret is to be distributed among a set of n participants in such a way that any two or more participants can reveal the secret by pulling their shares but it is not possible for a single participant to get any information about the secret. This is a special case of general access structure and it is known as $(2, n)$ threshold access structure. In the subsequent subsections we shall describe how the secret can be distributed among the participants and how the set of two or more participants can reveal the secret by pulling the corresponding shares. Moreover we shall prove that the scheme is perfectly secure and it is not possible for a single participant to get any information regarding the secret from the share that he/she holds.

0.2.6 Secret sharing algorithm

We assume that the algorithm of DNA encoding of binary string is public i.e., known to every body. Let the dealer want to share the secret binary string $x = x_1x_2 \dots x_k$ among n participants P_1, P_2, ..., P_n where not all $x_i = 0$ and not all $x_i = 1$, for $i = 1, 2, \cdots, k$. The dealer follows the following algorithm :

1. If x_i is 0, the dealer considers the generating matrix G_0 and gives a random permutation to the columns of G_0 and constructs a new matrix M_i, $i = 1, 2, \dots, k$. If x_i is 1, the dealer does the same with the generating matrix G_1.

2. Concatenate M_i's to get a new matrix $M = M_1||M_2|| \dots ||M_k$.

3. Each row α_i of M, $i = 1, 2. \ldots, n$ represents a binary string of length km. Encode each row α_i of M by a test tube $T[\alpha_i]$ in DNA double strand notation, $i = 1, 2. \ldots, n$.

4. To each participant P_i, give the test tube $T[\alpha_i]$, $i = 1, 2, \ldots, n$.

5. To each participants, give also the values k and m.

0.2.7 Decryption algorithm

Let $X = \{P_{t_1}, P_{t_2}, \ldots, P_{t_q}\} \in \Gamma_{Qual}$. Then they have the test tubes $T[\alpha_{t_j}]$, $j = 1, 2, \ldots, q$. Also they know the algorithm for DNA encoding of binary strings, the values k and m. They will use the following algorithm to get the secret message.

1. Use mixing procedure as described earlier with the test tubes $T[\alpha_{t_j}]$, $j = 1, 2, \ldots, q$.

2. Execute automated DNA sequencing method to read the DNA double strands.

3. Since the algorithm for DNA encoding of binary strings and the values k and m are known to the participants, convert the DNA strand notation into binary string. Let the resulting binary string be $y = y_1 y_2 \ldots y_{mk}$. Note that this y represents the binary "or" of the rows α_{t_j}, $j = 1, 2, \ldots, q$.

4. Since the value of m is known to the participants, they break y into k substrings of length m as $y = (y_{1_1} y_{1_2} \ldots y_{1_m})(y_{2_1} y_{2_2} \ldots y_{2_m}) \ldots (y_{k_1} y_{k_2} \ldots y_{k_m})$.

5. The participants will compute $w_i = BW(y_{i_1} y_{i_2} \ldots y_{i_m})$. Since not all $x_i = 0$ and not all $x_i = 1$, not all w_i's are equal. Let $w_{min} = \min_{i \in \{1,2,\ldots,k\}} w_i$.

6. For $X \in \Gamma_{Qual}$, as $BW(G_X^0) < BW(G_X^1)$, the participants will compute $z = z_1 z_2 \ldots z_k$, where $z_i = 1$ if $BW(y_{i_1} y_{i_2} \ldots y_{i_m}) > w_{min}$; and $z_i = 0$, otherwise.

Theorem 0.2.4 *The binary string z and the secret binary string x are same.*

Proof : The result follows from the fact that $y_{i_1} y_{i_2} \ldots y_{i_m}$ is a permutation of G_0^X if $BW(y_{i_1} y_{i_2} \ldots y_{i_m}) = w_{min}$ or G_1^X if $BW(y_{i_1} y_{i_2} \ldots y_{i_m}) > w_{min}$.

Theorem 0.2.5 *The above scheme is perfectly secure.*

Proof : To proof this result it is sufficient to proof that for any $Y \in \Gamma_{Forb}$, the probability that these participants can predict the secret binary string $x = x_1x_2 \dots x_n$ correctly is $(1/2)^n$. The result follows from the fact that $G_0[Y]$ and $G_1[Y]$ are identical up to column permutations. So just looking at the $G_0[Y]$ ($G_1[Y]$), it is not possible to predict correctly whether it comes from G_0 or G_1.
Note : If all x_i's are equal, then our scheme can predict the secret binary string correctly with probability 1/2. But for all practical uses, not all x_i's are equal.

Example 0.2.2 *Let us consider a (2,6)-DNA secret sharing scheme on a set* $\mathcal{P} = \{1, 2, 3, 4, 5, 6\}$ *of 6 participants, where* $\Gamma_0 = \{X \subseteq \mathcal{P} : |X| = 2\}$, $\Gamma_{Qual} = \{Y \subseteq \mathcal{P} : X \subseteq Y$, *for some* $X \in \Gamma_0\}$ *and* $\Gamma_{Forb} = 2^{\mathcal{P}} \setminus \Gamma_{Qual}$. *So in a (2,6)-DNA secret sharing scheme the binary secret is to be distributed among a set of 6 participants in such a way that any set of two or more participants can reveal the secret but no information is gained about the secret by a single participant from his/her own share. To implement the scheme the dealer chooses two Boolean matrices* $G_0 = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$ *and* $G_1 = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$. *It can be shown that the two conditions of generating matrices are satisfied by* G_0 *and* G_1. *Now suppose that under this above secret sharing model the dealer wants to share the secret binary string* $x = x_1x_2x_3 = 101$ *to all the participants. Since* $x_1 = 1$, *the dealer considers the matrix* G_1 *and apply a random permutation to the columns of* G_1 *and produces a matrix* M_1. *Similarly, apply the same procedure for* x_2 *and* x_3 *on* G_0 *and* G_1 *respectively to produce two more matrices* M_2 *and* M_3. *Let* $M = M_1||M_2||M_3$, *where "||" denote the concatenation of matrices. For simplicity we assume that all the applied random permutations are identity permutation. So the first row of* M *is given by* $\alpha_1 = 110011001100$. *Similarly* $\alpha_2 = 101011001010$, $\alpha_3 = 100111001001$, $\alpha_4 = 011011000110$, $\alpha_5 = 010111000101$ *and* $\alpha_6 = 001111000011$. *Now the dealer converts the binary strings to DNA representations to get the test tubes* $T[\alpha_1] = \{ds_1, ds_2, ds_5, ds_6, ds_9, ds_{10}\}$, $T[\alpha_2] = \{ds_1, ds_3, ds_5, ds_6, ds_9, ds_{11}\}$, $T[\alpha_3] = \{ds_1, ds_4, ds_5, ds_6, ds_9, ds_{12}\}$, $T[\alpha_4] = \{ds_2, ds_3, ds_5, ds_6, ds_{10}, ds_{11}\}$, $T[\alpha_5] = \{ds_2, ds_4, ds_5, ds_6, ds_{10}, ds_{12}\}$ *and* $T[\alpha_6] = \{ds_3, ds_4, ds_5, ds_6, ds_{11}, ds_{12}\}$, *where* ds_i *is defined earlier. The test tube* $T[\alpha_i]$ *is given to the participant* P_i, $i = 1, 2, \dots, 6$. *Also the values* $m = 4$ *and* $k = 3$ *are given to the participants. For decryption, let the qualified set of participants* $\{P_1, P_2\}$ *come together. They use mixing procedure with test tubes* $T[\alpha_1]$ *and* $T[\alpha_2]$ *to get* $T[\alpha_1] \cup T[\alpha_2] = \{ds_1, ds_2, ds_3, ds_5, ds_6, ds_9, ds_{10}, ds_{11}\}$. *With the knowledge of decoding the DNA representation to the binary string, the values of* $k = 3$ *and* $m = 4$, *the participants* P_1 *and* P_2 *can convert the DNA representation to the binary string* $y = 111011001110$. *Since, the value of* m *is known to the participants,* P_1 *and* P_2 *can break*

y as $y = (1110)(1100)(1110)$. Next they will find the value of w_{min} as 2 and then they will compute $z = 101$, as $BW(1110) > 2$, $BW(1100) = 2$. Thus P_1 and P_2 can recover the secret 101. Now suppose, any single participant say the 1st participant alone wants to get information about the secret. Then the two matrices $G_0[\{1\}] = \begin{bmatrix} 1100 \end{bmatrix}$ and $G_1[\{1\}] = \begin{bmatrix} 1100 \end{bmatrix}$ are identical up to column permutation. So $BW(G_0^{\{1\}})$ and $BW(G_1^{\{1\}})$ are equal. Thus just looking at these two matrices it is not possible to predict whether $BW(G_0^{\{1\}})$ or $BW(G_1^{\{1\}})$ correspond to 0 or 1. So no information is gained. Hence the scheme is totally secure.

Now the question that comes to our mind is that how to construct generating matrices G_0 and G_1 for a $(2, n)$-DNA secret sharing scheme. To construct the generating matrices we take the help of the statistical design, known as Balanced Incomplete Block Design (PBIBD).

0.2.8 Construction of Generating Matrices for $(2, n)$ DNA secret sharing using PBIBD

PBIB designs have been extensively studied in the literature in statistical design theory and for the sake of completeness, we give the following definitions following Raghavarao [16].

Definition 0.2.2 *Given v symbols $1, 2, \ldots, v$, suppose there is an association scheme with 2 classes such that*

(a) any two symbols are either 1st or 2nd associates, the relation being symmetrical;

(b) each symbol β has n_i ith associates;

(c) if any two symbols β and γ are ith associates, then the number of symbols that are jth associates of β, and kth associates of γ, is independent of the pair β and γ.

Then, a PBIB design $(v, b, r, k, \lambda_1, \lambda_2)$ is an arrangement of the v symbols into b blocks of size k ($k < v$) each, such that (i) every symbol occurs at most once in a set, (ii) every symbol occurs in exactly r blocks and (iii) if two symbols β and γ are ith associates, then they occur together in $\lambda_i (\geq 0)$ blocks, the number λ_i being independent of the particular pair of ith associates β and γ, $i = 1, 2$.

As an example, PBIBD(6,4,2,3,0,1) has blocks $\{1, 2, 3\}$, $\{1, 4, 5\}$, $\{2, 4, 6\}$ and $\{3, 5, 6\}$ and symbols within parenthesis below are first associates and otherwise second associates : $\{1, 6\}$, $\{2, 5\}$, $\{3, 4\}$.

Definition 0.2.3 *For a PBIB design, the incidence matrix is given by the $v \times b$ matrix $N = (n_{ij})$, with $n_{ij} = 1$ if the ith symbol is present in the jth block of the design, and 0 otherwise, $\forall\ i = 1, 2, \ldots, v$ and $\forall\ j = 1, 2, \ldots, b$.*

Now we have the theorem :

Theorem 0.2.6 *If there exists a PBIB$(v, b, r, k, \lambda_1, \lambda_2)$, then a $(2, v)$-DNA secret sharing scheme can be constructed with v participants having $w_{min} = r$ and b many columns in the generating matrices G_0 and G_1.*

Proof : Let the n participants of a $(2, n)$-DNA secret sharing scheme be identified with the v symbols of the design, resulting in $n = v$. Let N be the incidence matrix of the PBIB$(v, b, r, k, \lambda_1, \lambda_2)$. Take $G_1 = N$. Then $m = b$. By conditions (a) and (b) of Definition 0.2.2, in each row of N there are exactly r 1's and $b - r$ 0's. Again, from condition (c) of Definition 0.2.2, it follows that for any 2 participants i and j, $BW(G_1^{\{i,j\}}) = BW(G_1^{\{i\}}) + BW(G_1^{\{j\}}) - \lambda_q = 2r - \lambda_q$, if symbols i and j are qth associates in the PBIB design, $q = 1, 2$.

note that $BW(G_1^X) \geq 2r - \lambda_q$ and $BW(G_0^X) = r$ for any $X \subseteq \{1, 2, \cdots, n\}$ with $|X| \geq 2$. Since in PBIBD, $r - \lambda_q > 0$, $BW(G_1^X) > BW(G_0^X)$. Also, as $BW(G_1^{\{i\}}) = BW(G_0^{\{i\}})$, $G_1[\{i\}]$ and $G_0[\{i\}]$ are identical up to column permutations. From the above discussion, it is clear that G_1 and G_0 satisfy the conditions of the Definition 0.2.1 with $w_{min} = r$ and b many columns in the generating matrices G_0 or G_1.

The following example illustrates a construction of generating matrices of a $(2, 6)$-DNA secret sharing scheme using PBIBD.

Example 0.2.3 *Consider the PBIB$(6, 4, 2, 3, 0, 1)$. By Theorem 0.2.6, we can construct a $(2, 6)$-DNA secret sharing scheme from it. The incidence matrix of this design is:*

$$N = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}. \text{ Take } G_0 = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix} \text{ and } G_1 = N.$$

Note : For a given value of n, if a PBIBD with $v = n$ does not exist, we choose a PBIBD with $v > n$. e.g., for $n = 2, 3$ we may use the PBIBD (4,4,2,2,0,1) from Clatworthy [7] with $v = 4$. Then from this PBIBD, we first construct the generating matrices G_1 and G_0 as in Theorem 0.2.6 for a $(2, v)$-DNA secret sharing scheme on v participants. Then, if we delete any $v - n$ rows from G_1 and G_0, the resulting matrices will be the generating matrices of the required $(2, n)$-DNA secret sharing scheme with n participants. Thus, using PBIB designs, we may construct a $(2, n)$-DNA secret sharing scheme for any number of participants and in practice, to construct a $(2, v)$-DNA secret sharing scheme, we use the PBIBD which exists with v as close to n as possible, $v \geq n$.

In the next section, we will develop (n, n)-DNA secret sharing scheme using linear algebra.

0.3 (n, n)-threshold DNA secret sharing scheme using Linear Algebra Technique

Suppose the access structure is such that the secret is to be distributed among a set of n participants in such a way that the secret will be revealed only when all the n participants pull their shares but it is not possible for a set of $n-1$ or less participants to get any information about the secret. This is again a special case of general access structure and is known as (n, n) threshold access structure. The encryption and the decryption algorithms are similar to that of $(2, n)$-DNA secret sharing scheme. As before, the same question that comes to our mind is that how to generate the generating matrices. For that purpose we take the help of basic linear algebra.

0.3.1 Construction of Generating Matrices for (n, n)-threshold DNA secret sharing scheme using Linear Algebra

In this section, we introduce a construction procedure for generating matrices for an (n, n)-threshold DNA secret sharing scheme. Let us consider the following associated system of linear equations on n variables $x_1, x_2, \cdots, x_n$ over the binary field $\mathbf{Z_2}$,

$$x_1 + x_2 + \cdots + x_n = 0 \tag{0.3.1}$$

and

$$x_1 + x_2 + \cdots + x_n = 1 \tag{0.3.2}$$

Now, both the equations (1) and (2) are consistent. In addition, the null space of (1) over $\mathbf{Z_2}$, i.e., the solutions of the equation (1), is a subspace of $\mathbf{Z_2}^n$ of dimension $(n-1)$. Hence there are 2^{n-1} solutions of (1). Let G_0 be an $n \times 2^{n-1}$ Boolean matrix whose columns are all possible solutions of the equation (1).

From the theory of linear equations, it is known that if $\mathbf{v}$ is a particular solution of the equation (2), then all the solutions of (2) can be obtained by adding $\mathbf{v}$ with all the solutions of (1). Consequently, there are 2^{n-1} solutions to (2). Let G_1 be the $n \times 2^{n-1}$ Boolean matrix whose columns are all possible solutions of (2) and is obtained from G_0 by adding a particular solution to individual columns over $\mathbf{Z_2}$. It is to be noted that by selecting different particular solutions one may obtain different matrices as G_1 but all of them will be identical up to column permutation. So, without loss of generality, we fix G_0 and G_1. Also note that G_0 and G_1 can be used as generating matrices provided conditions 1 and 2 of Definition 0.2.1 are satisfied. Consequently it is necessary to identify the qualified and forbidden sets, i.e., the access structure, if any. Since $(0, 0, \cdots, 0)$ is a solution of (1) and not a solution

of (2), such identification seems feasible. Let $\mathcal{Q} = \{X \subseteq \mathcal{P} : BW(G_1^X) \neq BW(G_0^X)\}$ and $\mathcal{F} = \{X \subseteq \mathcal{P} : BW(G_1^X) = BW(G_0^X)\}$. Then $\mathcal{Q} \cup \mathcal{F} = 2^{\mathcal{P}}$ with $\phi \in \mathcal{F}$.

Now we prove the following lemma.

Lemma 0.3.1 *Let $\mathcal{P} = \{1, 2, \ldots, n\}$ be a set of n participants and let $X = \{i_1, i_2, \ldots, i_k\} \subseteq \mathcal{P}$. Let the ith row of G_0 and G_1 corresponds to the ith participant of $\mathcal{P}$. Then the following statements are equivalent.*

(i) *There exists a particular solution $v = (v_1, v_2, \ldots, v_n)$ of (2) such that $v_{i_j} = 0$, $\forall\ j = 1, 2, \ldots, k$.*

(ii) *$G_0[X]$ and $G_1[X]$ are identical up to column permutation.*

(iii) *$X \in \mathcal{F}$.*

Proof : $((i) \Rightarrow (ii))$ Let $\mathbf{v} = (v_1, v_2, \ldots, v_n)$ be a particular solution of (2) such that $v_{i_1} = v_{i_2} = \cdots = v_{i_k} = 0$, $0 < k < n$ and let $X = \{i_1, i_2, \ldots, i_k\}$. It follows that if $\mathbf{v}$ is used to generate G_1 from G_0 then $G_0[X]$ and $G_1[X]$ are identical. If this particular $\mathbf{v}$ is not used to generate G_1 from G_0 then $G_0[X]$ and $G_1[X]$ will be identical up to column permutation.
$((ii) \Rightarrow (iii))$ If $G_0[X]$ and $G_1[X]$ are identical up to column permutation then $BW(G_0^X) = BW(G_1^X)$. Thus, $X \in \mathcal{F}$.
$((iii) \Rightarrow (i))$ Let $X \in \mathcal{F}$. Then $BW(G_0^X) = BW(G_1^X)$. As $\mathbf{x} = \mathbf{0}$ is a solution of (1), $G_0[X]$ contains a column of all zeros which in turn implies $G_1[X]$ contains a column of all zeros, a necessary condition for $BW(G_0^X)$ to be equal to $BW(G_1^X)$. It follows that there exists a $\mathbf{v} = (v_1, v_2, \ldots, v_n)$ such that $v_{i_j} = 0$, $\forall j = 1, 2, \ldots, k$ as a solution to (2).

Lemma 0.3.2 *If $X \in \mathcal{Q}$, $BW(G_0^X) < BW(G_1^X)$.*

Proof : If $BW(G_1^X) = 2^{n-1}$ then clearly, $BW(G_0^X) < BW(G_1^X)$. Let $BW(G_1^X) = 2^{n-1} - t, t > 0 \Rightarrow$ $G_1[X]$ has t all-zero columns. Let these columns represent t solutions of (2), namely $\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_t$. Define $\mathbf{x}_i = \mathbf{y}_1 + \mathbf{y}_i,\ i = 1, 2, \ldots, t$. Note $\mathbf{x}_i$ is a solution of (1) and $\mathbf{x}_i[X]$ is all-zero $\forall i = 1, 2, \ldots, t$. $\Rightarrow$ $G_0[X]$ has at least t all-zero columns $\Rightarrow$ $BW(G_0^X) \leq BW(G_1^X)$. But if $BW(G_0^X) = BW(G_1^X)$ then $X \in \mathcal{F}$, a contradiction. Therefore, $BW(G_0^X) < BW(G_1^X)$.

Note: Of the 2^{n-1} solutions of (2) at most one solution may have all 1's. So the remaining solutions have one or more coordinate(s) as zero(s). Thus $\mathcal{F}$ is non-null. Also let z_i be the number of zeros in the ith solution of (2), $i = 1, 2, \ldots, 2^{n-1}$ and let $z = \max_i z_i$. Clearly $z < n$. It is easy to note that for any $X \subseteq \mathcal{P}$ such that $|X| > z$, $BW(G_1^X) > BW(G_0^X)$. [In fact, for any $X \in \mathcal{Q}$, $BW(G_1^X) = 2^{n-1}$ and $BW(G_0^X) \leq 2^{n-1} - 1$.] Hence $\mathcal{F}$ is also not

empty. From Lemma 0.3.1, it is clear that $\mathcal{F}$ represents the forbidden set of participants i.e., Γ_{Forb}. As $\Gamma_{Qual} \cup \Gamma_{Forb} = 2^{\mathcal{P}}$ and $\mathcal{Q} \cup \mathcal{F} = 2^{\mathcal{P}}$, $\Gamma_{Qual} = \mathcal{Q}$. Henceforth $\mathcal{Q}$ and $\mathcal{F}$ will be represented as Γ_{Qual} and Γ_{Forb} respectively. Therefore, conditions 1 and 2 of Definition 0.2.1 hold on G_0 and G_1 and they can be generating matrices.

Thus we have obtain a method of getting the generating matrices and the access structure based on associated system of linear equations, one homogeneous and the other non-homogeneous.

Theorem 0.3.1 *Let $(\Gamma_{Qual}, \Gamma_{Forb})$ be an access structure of an (n, n)-DNA secret sharing scheme on a set $\mathcal{P} = \{1, 2, \ldots, n\}$ of n participants with $\Gamma_0 = \{X \subseteq \mathcal{P} : |X| = n\}$. Then there exists generating matrices G_0 and G_1 for $(\Gamma_{Qual}, \Gamma_{Forb})$.*

Proof : Let us consider the following system of linear equations over the binary field $\mathbf{Z_2}$ as given below :

$$x_1 + x_2 \cdots + x_n = 0 \tag{0.3.3}$$

$$x_1 + x_2 \cdots + x_n = 1 \tag{0.3.4}$$

Let G_0 and G_1 denote the matrices corresponding to the solutions of (3) and (4) respectively as defined before. Then G_0 and G_1 are $n \times m$ Boolean matrices where $m = 2^{p-1}$. Now the Theorem follows from Lemmas 0.3.1 and 0.3.2.

Example 0.3.1 *Let us consider an example where the binary secret message is to be distributed among a set of 3 participants in such a way that the secret will be revealed only when all the three participants are willing to pull their shares to get the secret but no set of two or less participants have any information about the secret. This secret sharing scheme is known as (3,3)-DNA secret sharing scheme. The encryption and the decryption schemes are similar as described in Example 0.2.2. The only difference is the construction of the generating matrices. Here we take help of the subject linear algebra. For each participant i we associate a variable x_i, $i = 1, 2, 3$ and consider the two linear equations :*

$$x_1 + x_2 + x_3 = 0 \tag{0.3.5}$$

$$x_1 + x_2 + x_3 = 1 \tag{0.3.6}$$

Let G_0 and G_1 denote respectively the generating matrices for corresponding to the set of participants $\{1, 2, 3\}$. The columns of G_0 and G_1 are all possible solutions of the equations (5) and (6) respectively. Consequently, $G_0 = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$ and $G_1 = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix}$. The two matrices satisfy all conditions for generating matrices.

In the next section we will deal with more general scenario known as general access structure constructed from (n, n)-threshold access structure.

0.4 DNA secret sharing schemes for general access structure

Until now we discussed about threshold secret schemes. But, in reality, sometimes general access structure is more useful from practical view point. For an example, suppose there is an important locker in the office of defense ministry and the Government wants that the key of the locker will be distributed among a set of 4 participants, namely {Defense Minister, General 1, General 2, General 3} in such a way that the locker will be opened only when Defense Minister and any two out of three Generals are present. In this scenario, threshold scheme for secret sharing won't work. To overcome this situation we need to think about general access structure. For that we need the generating matrices with respect to the access structure with $\Gamma_0 = \{\{\text{Defense Minister, General 1, General 2}\}, \{\text{Defense Minister, General 1, General 3}\}, \{\text{Defense Minister, General 2, General 3}\}\}$,
$\Gamma_{Qual} = \{Y \subseteq \mathcal{P} : X \subseteq Y, \text{ for some } X \in \Gamma_0\}$ and $\Gamma_{Forb} = 2^{\mathcal{P}} \setminus \Gamma_{Qual}$. To construct the required generating matrices, we take help of the (n, n)-DNA secret sharing scheme and the following lemma that provide us a method to construct the generating matrices of the combined access structures from the two given access structures.

Lemma 0.4.1 *Let G_0^1 and G_1^1 (G_0^2 and G_1^2) denote the generating matrices of a given access structure $(\Gamma_{Qual}^1, \Gamma_{Forb}^1)$ $((\Gamma_{Qual}^2, \Gamma_{Forb}^2))$ on the set of participants $X_1 = \{i_{1_1}, i_{1_2}, \cdots, i_{1_k}\}$ $(X_2 = \{i_{2_1}, i_{2_2}, \cdots, i_{2_s}\})$. Then there exist generating matrices G_0 and G_1 for the access structure $(\Gamma_{Qual}^1 \cup \Gamma_{Qual}^2, \Gamma_{Forb}^1 \cap \Gamma_{Forb}^2)$ on the set of participants $X = X_1 \cup X_2$.*

Proof : From G_0^1 we construct a matrix $\hat{G}_0^1$ having $|X|$ rows. For $i = 1, 2, \cdots, |X|$, if i is a participant of $(\Gamma_{Qual}^1, \Gamma_{Forb}^1)$, i.e., if $i \in X_1$, the ith row of $\hat{G}_0^1$ is the row corresponding to the row of G_0^1 for the participant i in $(\Gamma_{Qual}^1, \Gamma_{Forb}^1)$; else it is a row having all zero entries. Similar construction is done for $\hat{G}_1^1$, $\hat{G}_0^2$ and $\hat{G}_1^2$. Finally the generating matrix G_0 (G_1) is constructed by concatenation of the two matrices $\hat{G}_0^1$ and $\hat{G}_0^2$ ($\hat{G}_1^1$ and $\hat{G}_1^2$). From the construction it is clear that G_0 and G_1 are the generating matrices of the given access structure.

Next we will prove the theorem on the existence of generating matrices for a given access structure using the preceding extensions of G_0 and G_1.

Theorem 0.4.1 *Let $(\Gamma_{Qual}, \Gamma_{Forb})$ be a strong access structure on a set $\mathcal{P} = \{1, 2, \ldots, n\}$ of n participants with $\Gamma_0 = \{B_1, B_2, \ldots, B_k\}$ where $B_i \subseteq \mathcal{P}, \forall\ i = 1, 2, \ldots, k$. Then there exists generating matrices G_0 and G_1 for the access structure $(\Gamma_{Qual}, \Gamma_{Forb})$ on $\mathcal{P}$.*

Proof : Let us define $\Gamma_{0i} = \{B_i\}, \forall\ i = 1, 2, \ldots, k$. Then we can think of Γ_{0i} as an (n_i, n_i)-DNA threshold access structure with $n_i = |B_i|$, $\forall i = 1, 2, \cdots, k$. Then by Theorem 0.3.1 we

can construct generating matrices G_0^i and G_1^i, $\forall i = 1, 2, \cdots, k$. Thus by using Lemma 0.4.1, we can construct generating matrices for the access structure $(\Gamma_{Qual}, \Gamma_{Forb})$.

Example 0.4.1 *Let us continue with the example as posed at the beginning of this section. Here,* $\Gamma_{01} = \{$*Defense Minister, General 1, General 2*$\}$*. So, we can think of* Γ_{01} *as an access structure for (3,3)-DNA secret sharing scheme and by using Theorem 0.3.1 we can construct* $G_0^1 = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$ *and* $G_1^1 = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix}$*. Similarly we can construct* G_0^2, G_1^2, G_0^3 *and* G_1^3*. Now using Lemma 0.4.1, we can construct* $\hat{G}_0^1 = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$ *and* $\hat{G}_1^1 = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$*. Similar construction holds for* $\hat{G}_0^i$, $\hat{G}_1^i$, *for* $i = 2, 3$*. Now using Theorem 0.4.1 we can construct the generating matrices* $G_0 = \hat{G}_0^1 || \hat{G}_0^2 || \hat{G}_0^3$ *and* $G_1 = \hat{G}_1^1 || \hat{G}_1^2 || \hat{G}_1^3$.

Bibliography

[1] Avishek Adhikari, "DNA Secret Sharing", IEEE World Congress on Evolutionary Computation 2006, CEC 2006, July 16-21, 1407-1411, 2006.

[2] L. Adleman, "Molecular Computation of Solutions to Combinatorial Problems," *Science*, vol. 266, pp. 1021-1024, 1994.

[3] G. R. Blakley, "Safeguarding cryptographic keys," *AFIPS 1979,* National Computer Conference, Vol. 48, pp. 313-317, 1979.

[4] R. Barua, J. Misra, "Binary Arithmetic for DNA Computers," *Proc. 8th Int. Conf. on DNA-Based Computers (DNA 8)*, Eds M. Hagiya and A. Ohuchi, LNCS 2568, Springer-Verlag, 2003.

[5] C. Blundo, A. De Santis and U. Vaccaro, "Efficient sharing of many secrets," *STACS'93,* Lecture Notes in Computer Science, Vol. 665, Springer-Verlag, pp. 692-703, 1993.

[6] E. F. Brickell and D. M. Davenport, "On the classification of ideal secret sharing schemes," *Journal of Cryptology,* Vol. 4, pp. 123-134, 1991.

[7] Clatworthy, W.H., (1973), "Tables of Two-associate Partially Balanced Design", National Bureau of Standards, Applied Maths, series No. 63, Washington D.C.

[8] C. T. Clelland, V. Risca, C. Bancroft, "Hiding messages in DNA microdots," *Nature*, vol. 399, pp. 533-534, 1999.

[9] W. Diffie and M. Hellman, "New directions of cryptography," *IEEE Transactions on Information Theory,* Vol. IT-22, No. 6, pp. 644-656, 1976.

[10] A. Gehani, T. H. LeBean, J. H. Reif, "DNA-based Cryptography," *5th DIMACS Workshop on DNA Based Computers*, MIT, 2000.

[11] J. D. Golic, "On matroid characterization of ideal secret sharing schemes," *Journal of Cryptology,* Vol. 11, pp. 75-86, 1998.

[12] L. Kari "DNA computing: arrival of biological mathematics," *The Mathematical Intelligencer*, vol. 19, pp. 9-22, 1997.

[13] A. Leier, C. Richter, W. Banzhaf, H. Rauhe, "Cryptography with DNA binary strands", *BioSystem*, vol. 57, pp. 13-22, 2000.

[14] K. Okada, W. Ogata, K. Sakano and K. Kurosawa, "Analysis on secret sharing schemes with non-graphical access structures," *IEICE Transactions Fundamentals,* Vol. E80-A, No. 1, pp. 85-89, 1997.

[15] T. Rabin and M. Ben-Or, "Verifiable Secret Sharing and Multiparty Protocols with Honest Majority," *In Proc. 21st STOC*, ACM, pp. 73-85, 1989.

[16] Raghavarao, D., "Constructions and Combinatorial Problems in Design of Experiment," *John Wiley and Sons, New York*, 1971.

[17] Adi Shamir, "How to share a secret," *Communications of the ACM*, vol. 22(1), pp. 612613, 1979.

[18] J. D. Watson, F. H. C. Crick, "A structure for deoxyribose nucleic acid", *Nature*, vol. 25, pp. 737-738, 1953.

Technical Details of the DNA-Lab Experiments

Of

Secret Sharing Schemes Using DNA Cryptography

Secret Sharing Schemes Using DNA Cryptography

Final Technical Details and Road Map of the DNA-Lab Experiments:

DNA–lab experiment model for (2,2)-DNA secret sharing

Our fundamental model of DNA computation for (2,2)-DNA secret sharing scheme is to apply some molecular biology operations i.e., DNA colning, polymerase chain reaction (PCR) and automated DNA sequencing. By using the share generation algorithm the dealer can encode any message into a binary string and finally in to a DNA base notation. After DNA encoding of binary strings for each participants, the dealer generates the shares using DNA cloning procedure. These recombinant DNAs as a form of DNA share are then distributed among the participants on a small paper form. To achieve this aim, the following Lab-experiment may be carried out.

Work Done By The Dealer:

DNA encoding of binary strings:

i. Suppose the secret binary string is 1101.
ii. By using the share generation algorithm we get the binary strings in the form 10011010 and 01101001.
iii. A binary string can be represented as a set of integers that corresponds the positions where the bits are 1 from left to right.
iv. 10011010 and 01101001 can be represented as a set {1, 4, 5, 7} and {2, 3, 5, 8}.
v. Suppose we represent 1 by 5' ATG 3' and 5' CAT 3' for template and complement strand respectably.
vi. Two binary strings can be represented by DNA bases as:

Template strands:

For share-1

5'GAATTCccATGcccATGATGATGATGcccATGATGATGATGATGcc
cATGATGATGATGATGATGATGccGGATCC3'

For share-2

5'GAATTCccATGATGcccATGATGATGcccATGATGATGATGATGcc
cATGATGATGATGATGATGATGATGccGGATCC3'

Complement strands:

For share-1

5’GGATCCggCATCATCATCATCATCATCATgggCATCATCATCATCATgg
gCATCATCATCATgggCATggGAATTC3’

For share-2

5’GGATCCggCATCATCATCATCATCATCATCATgggCATCATCATCATC
ATgggCATCATCATgggCATCATggGAATTC3’

vii. Annealing of synthetic oligonucleotide by heating and slowly cooling procedure and produce two individual double stranded DNA for further cloning experiment in pGEM-T vector.

viii. Gel purification of annealing dsDNA by Wizard SV Gel and PCR Clean-Up System.

ix. Kinasing of this dsDNA with PNK1 enzyme by our lab standardized protocol.

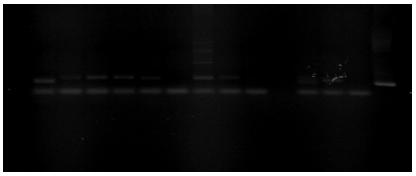
x. Ligation of one adenine nucleotide base at the 3’-ends of the each DNA strands using 10μM dATP, 2.5U taq Polymerase, and 1X PCR buffer at 37^0C for four hours.

xi. The cloning experiments were carried out using a pGEMT vector and adenine nucleotide base overhang dsDNA. Approximately 50ng of DNA vector and insert was ligated into the pGEMT vector (5:1 of insert to vector molar ratio) at 4^0C for overnight ligation.

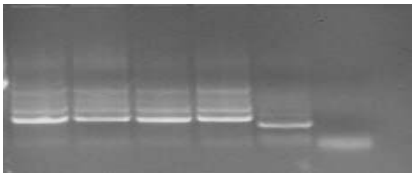
xii. Transformed the overnight ligated product into competent cells (E.coli DH5α).

xiii. The cells were spread/plated on Luria agar plates containing ampicillin (100mg/ml) as selection marker were incubated overnight (16hr) at 37^0C.

xiv. The transformed Colonies were screened by PCR using M13 universal primer set (M13-40, M13-48) and was run on 2 % agarose gel.



Colony PCR for Share-1



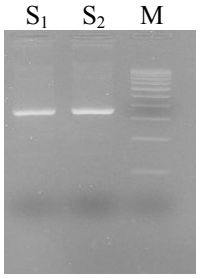
Colony PCR for Share-2

xv. The size and concentration of the amplicon were estimated by comparison with 100bp DNA marker (Fermentas) on 2 % agarose gel.

xvi. The positive clones were selected and plasmid DNA was isolated using miniprep plasmid isolation kit (Qiagen).

xvii. Presence of inserts was further checked by restriction digestion using EcoR1 and BamH1 using gel electrophoresis method.

xviii. The plasmid DNA of positive colonies was sequenced using Big-Dye sequencing kit.



PCR of positive clones comparison with 100bp DNA marker

xix. The DNA sequences obtained were matching alignment done with the desired sequence, using Bio-Edit software.

xx. After getting two recombinant DNA corresponding to each share, the dealer will place these two DNA on two different nitrocellulose filter paper and dry them into centrifugal evaporator at sterilized condition.

xxi. Then these two papers containing the required DNA share will be distributed among the two participants as there shares. This ends the work of dealer.

Work Done By The Share Holder :

i. 1st the share holder will bring their shares.

ii. Then extract the DNA from nitrocellulose filter papers into the water soluble form by random vortex and centrifugal process.

iii. Then quantify the DNA concentration through agarose gel electrophoresis and finally spectrophotometrically for DNA sequencing.

iv. After sequencing this inserted DNA strand using M13 primer they will get their secret in DNA encoding form.

DNA–lab experiment model for "DNA General Access Structure" for a given small access structure with a given secret

Our fundamental model of DNA General Access Structure is almost same as our (2,2)-DNA secret sharing model. But it is more generalized than the (2,2)-scheme. To show that correctness of the proof, we carried out the experiment for a fixed access structure with fixed binary secret. To achieve this aim, the following methodology may be carried.

Work Done By The Dealer:

DNA encoding of binary strings:

1. Suppose the secret binary string is 1001 and there are three participants $\{P_1, P_2, P_3\}$. Here the qualified set of participants are $\{P_1, P_2\}$ and $\{P_1, P_3\}$ and the forbidden set of participants are $\{P_2, P_3\}$ and any single ton set.

2. By using the share generation algorithm we get the fixed length, i.e., 8 bits binary strings in the form 10101010, 01101001 and 01101001 for participants P_1, P_2 and P_3 respectively.
3. A binary string can be represented as a set of integers that corresponds the positions where the bits are 1 from left to right.
4. 10101010, 01101001 and 01101001 can be represented as a set {1, 3, 5, 7}, {2, 3, 5, 8} and {2, 3, 5, 8}.
5. Suppose we represent 1 by 5' ATG 3' and 5' CAT 3' for template and complement strand respectably.
6. Three binary strings can be represented by DNA bases as:

Template strands:

For P_1:

5' TCTAGAccATGcccATGATGATGcccATGATGATGATGATGcccATG
ATGATGATGATGATGATGccGGATCC 3'

For P_2:

5' GAATTCccATGATGcccATGATGATGcccATGATGATGATGATGcc
cATGATGATGATGATGATGATGATGccGGATCC 3'

For P_3:

5' GAATTCccATGATGcccATGATGATGcccATGATGATGATGATGcc
cATGATGATGATGATGATGATGATGccGGATCC 3'

Complement strands:

For P_1:

5' GGATCCggCATCATCATCATCATCATCATgggCATCATCATCATC
ATgggCATCATCATgggCATggTCTAGA 3'

For P_2:

5' GGATCCggCATCATCATCATCATCATCATgggCATCATCATCATC
ATgggCATCATCATgggCATCATggGAATTC 3'

For P_3:

5' GGATCCggCATCATCATCATCATCATCATgggCATCATCATCATC
ATgggCATCATCATgggCATCATggGAATTC 3'

7. Annealing of synthetic oligonucleotide with the compatible pair of DNA by heating and slowly cooling procedure and produce three individual double stranded DNA for further cloning experiment in pGEM-T vector.
8. Gel purification of annealing dsDNA by Wizard SV Gel and PCR Clean-Up System.
9. Kinasing of this dsDNA with PNK1 enzyme by our lab standardized protocol.

10. Ligation of one adenine nucleotide base at the 3'-ends of the each DNA strands using 10μM dATP, 2.5U taq Polymerase, and 1X PCR buffer at 37^0C for four hours.

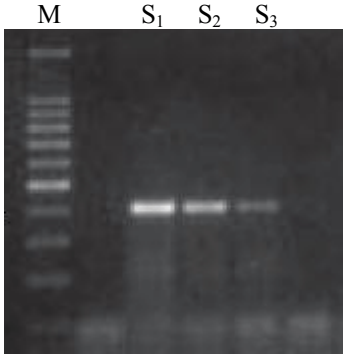
11. The cloning experiments were carried out using a pGEMT vector and adenine nucleotide base overhang dsDNA. Approximately 50ng of DNA vector and insert was ligated into the pGEMT vector (5:1 of insert to vector molar ratio) at 4^0C for overnight ligation.

12. Transformed the overnight ligated product into competent cells (E.coli DH5α).

13. The cells were spread/plated on Luria agar plates containing ampicillin (100mg/ml) as selection marker were incubated overnight (16hr) at 37^0C.

14. The transformed Colonies were screened by PCR using M13 universal primer set (M13-40, M13-48) and was run on 2 % agarose gel.

15. The size and concentration of the amplicon were estimated by comparison with 100bp DNA marker (Fermentas) on 2 % agarose gel.



PCR of positive clones comparison with 100bp DNA marker

16. The positive clones were selected and plasmid DNA was isolated using miniprep plasmid isolation kit (Qiagen).

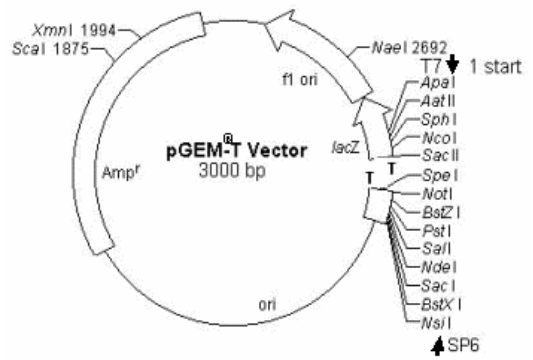
17. Presence of inserts was further checked by restriction digestion using EcoR1 and BamH1 using gel electrophoresis method.

18. The plasmid DNA of positive colonies was sequenced using Big-Dye sequencing kit.

19. The DNA sequences obtained were matching done with desired sequence, using Bio-Edit software.

20. After getting three recombinant DNA corresponding to each share for each participants, the dealer will place these three DNA on three different nitrocellulose filter paper and dry them into centrifugal evaporator at sterilized condition.

21. Then these three papers containing the required DNA share will be distributed among the three participants as there shares. This ends the work of dealer.



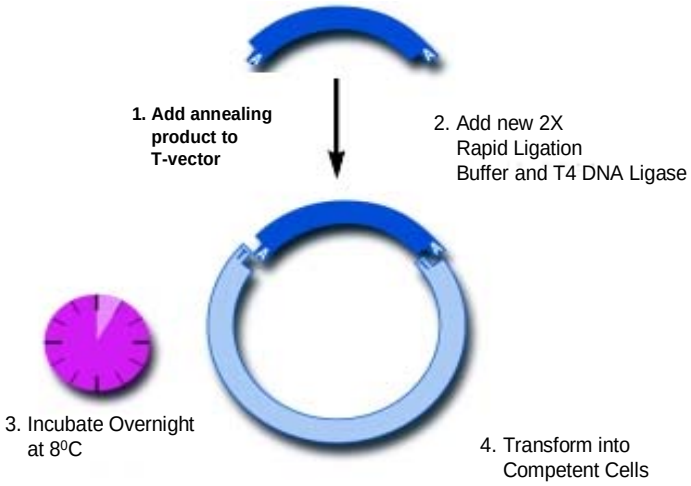


Fig. Clone into pGEM-T Vector

Work Done By The Share Holders:

1. 1st the share holder will bring their shares.
2. Then extract the DNA from nitrocellulose filter papers into the water soluble form by random vortex and centrifugal process.
3. Then quantify the DNA concentration through agarose gel electrophoresis and finally spectrophotometrically for DNA sequencing.
4. After sequencing this inserted DNA strand using M13 primer they will get their secret in DNA encoding form.

The Software Implementation Of The DNA Secret Sharing Scheme for General Access Structures

OpenSesame: AN OVERVIEW

The software model of the algorithm described in this project has been creatively christened: “OpenSesame”. It is an effective tool for Secret Sharing which can be used to both generate shares and verify the results of the DNA experiment. The source code has been written in Java. Thus the program is platform independent, secure and robust. The application is a small .jar file which can be launched either from the console or as a window based program. The NetBeans IDE 6.0.1 has been used to write the code.

The documentation of the software has been organized into four parts:

1. **Requirements Documentation:** A brief overview of the functioning of OpenSesame along with the computational resources required by the software.
2. **Architecture Documentation:** Schematic diagrams explaining the design and development of this project.
3. **Technical Documentation:** The source code of the software along with its operating parameters and limitations.
4. **User Documentation:** A hands on tutorial for users of the software.

Sufficient time and energy has been spent in testing the software and trying to ensure that it is as flexible and resource efficient as possible. The user interfaces have been designed with special care in order to appeal to the aesthetic senses of the users. The following chapter covers the documentation of the software in detail.

OpenSesame: UNDER THE HOOD

1. Requirements Documentation:

This document presents a brief overview of the purpose and scope of OpenSesame. The emphasis is on the functional requirements and usability requirements of the software. While preparing this document, the goal was to create a software model which would effectively run parallel to the DNA experiments.

Functional Requirements:

- The software MUST provide two different views for two mutually exclusive categories of users: the Dealer and the Participants.
- The Dealer MUST be require password authentication to use the software to generate shares.
- The Dealer MUST be able to enter a “secret key” file (.txt) by browsing the file system of the native computer. He/ She MUST also specify the access structure.
- The software MUST generate the required number of share files (.txt) in the raw binary format within an acceptable time span.
- The software MUST also generate a .txt file which mentions the length of the secret key in bits as well as the number of columns in the generating matrices.
- The software SHOULD run smoothly for a secret length of 100 kb and for up to 5 participants.
- Participants MUST specify the length of the “Secret Key” in bits and the number of columns in the Generating Matrices.
- Participants MUST be required to browse the file system of the computer and select the share file (.txt).
- For a qualified set of participants, the software MUST reconstruct the secret key and output a text file with the same. For an unqualified set of participants the output file MUST contain a continuous string of ‘1’ bits.

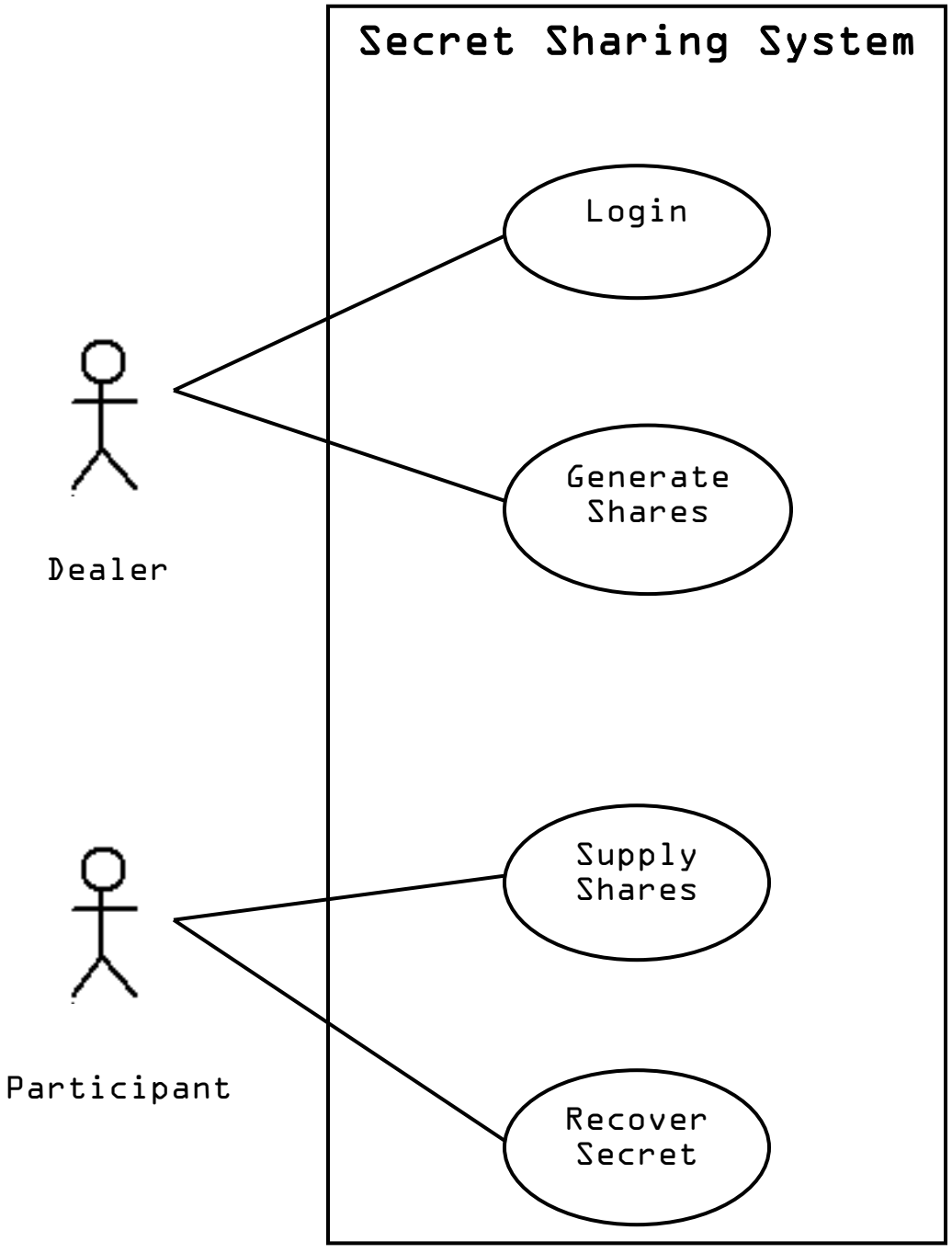
Usability Requirements:

- The interfaces must be user friendly and visually attractive.
- It must be easy to learn how to operate the software.
- Error reports must be lucid and clear and MUST not alarm the user.

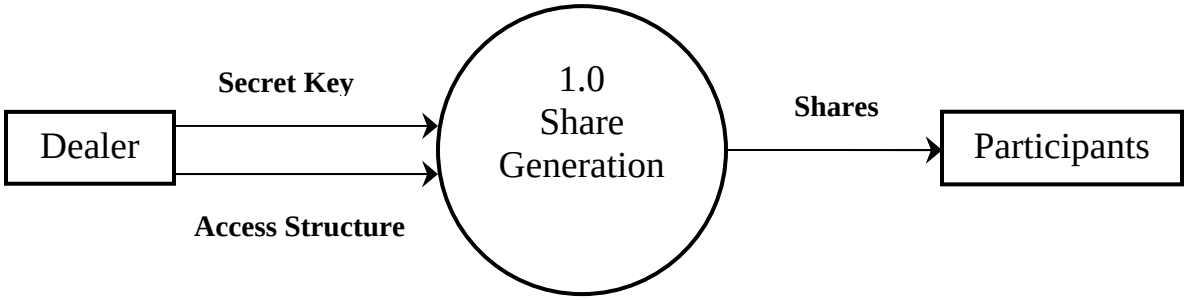
2. Architecture Documentation:

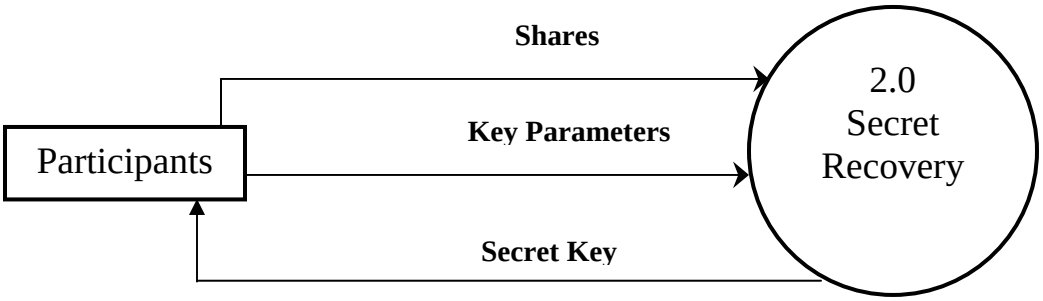
This section explains in explicit details the design and architecture of the software. It does so with the help of various schematic diagramming methods including Use Case models, DFD and an E-R Diagram. This document is meant more for a future designer of a similar system than the end user.

I. OpenSesame: Use Case Diagram

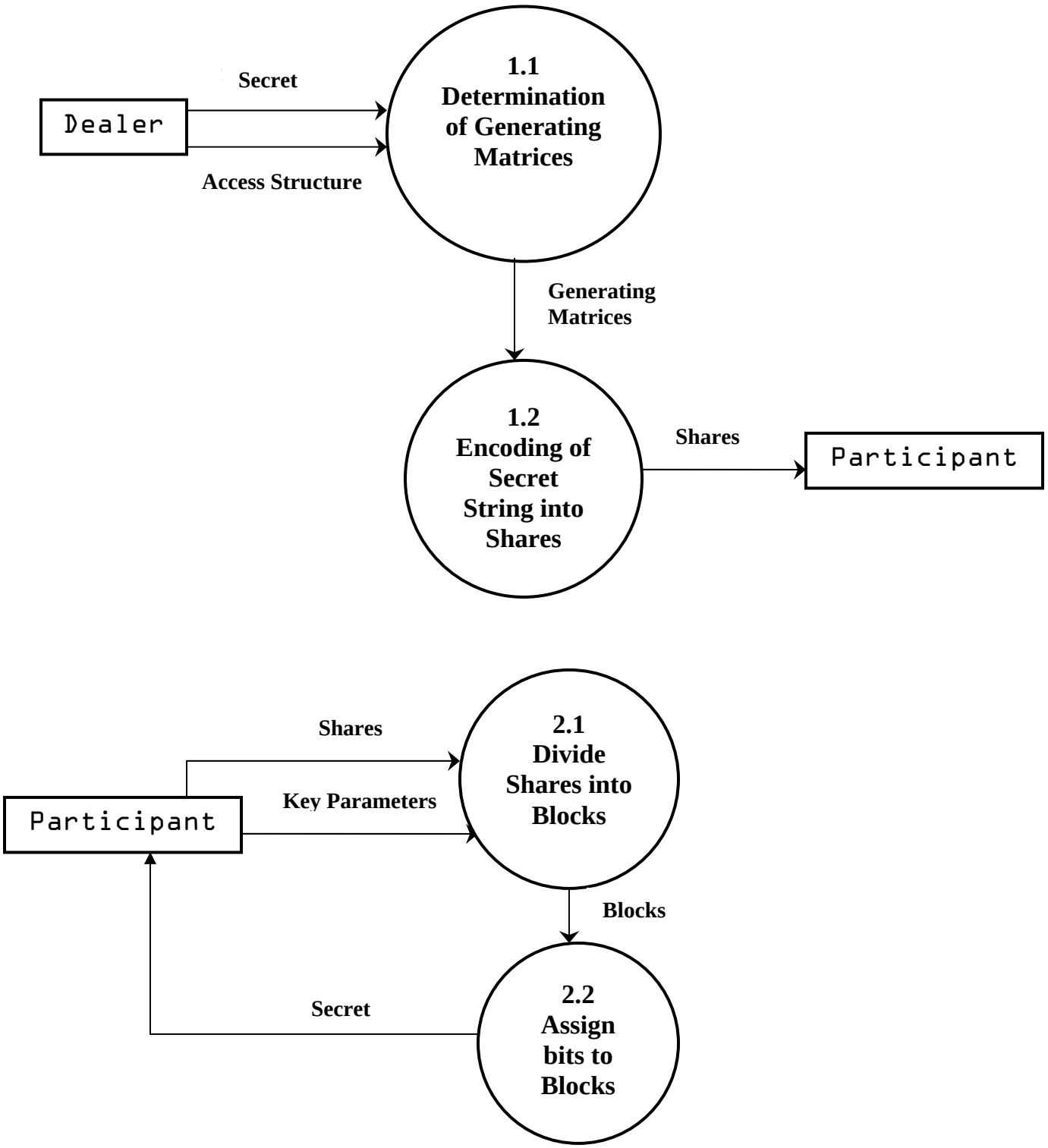


II. OpenSesame: Context Level Data Flow Diagram

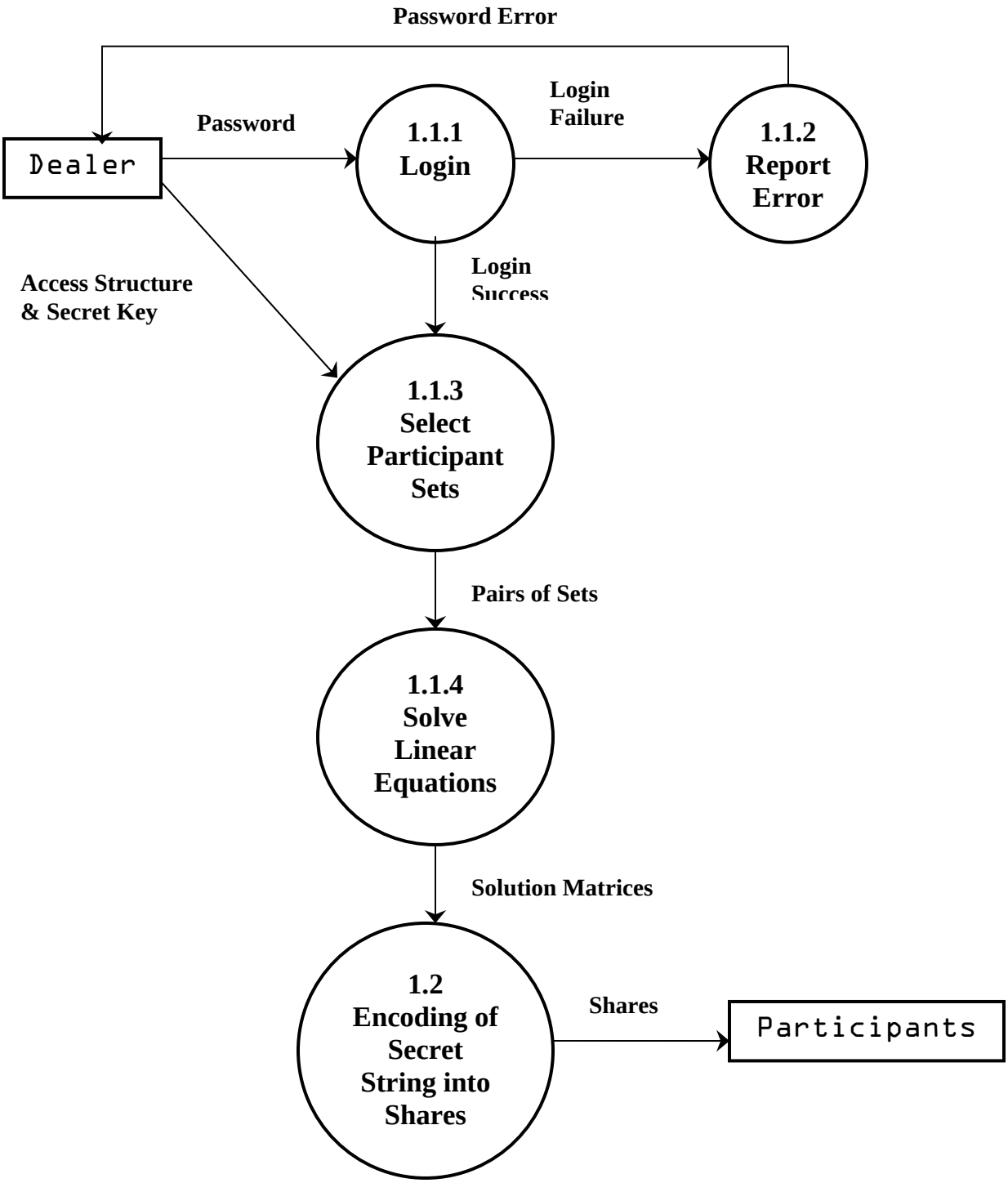




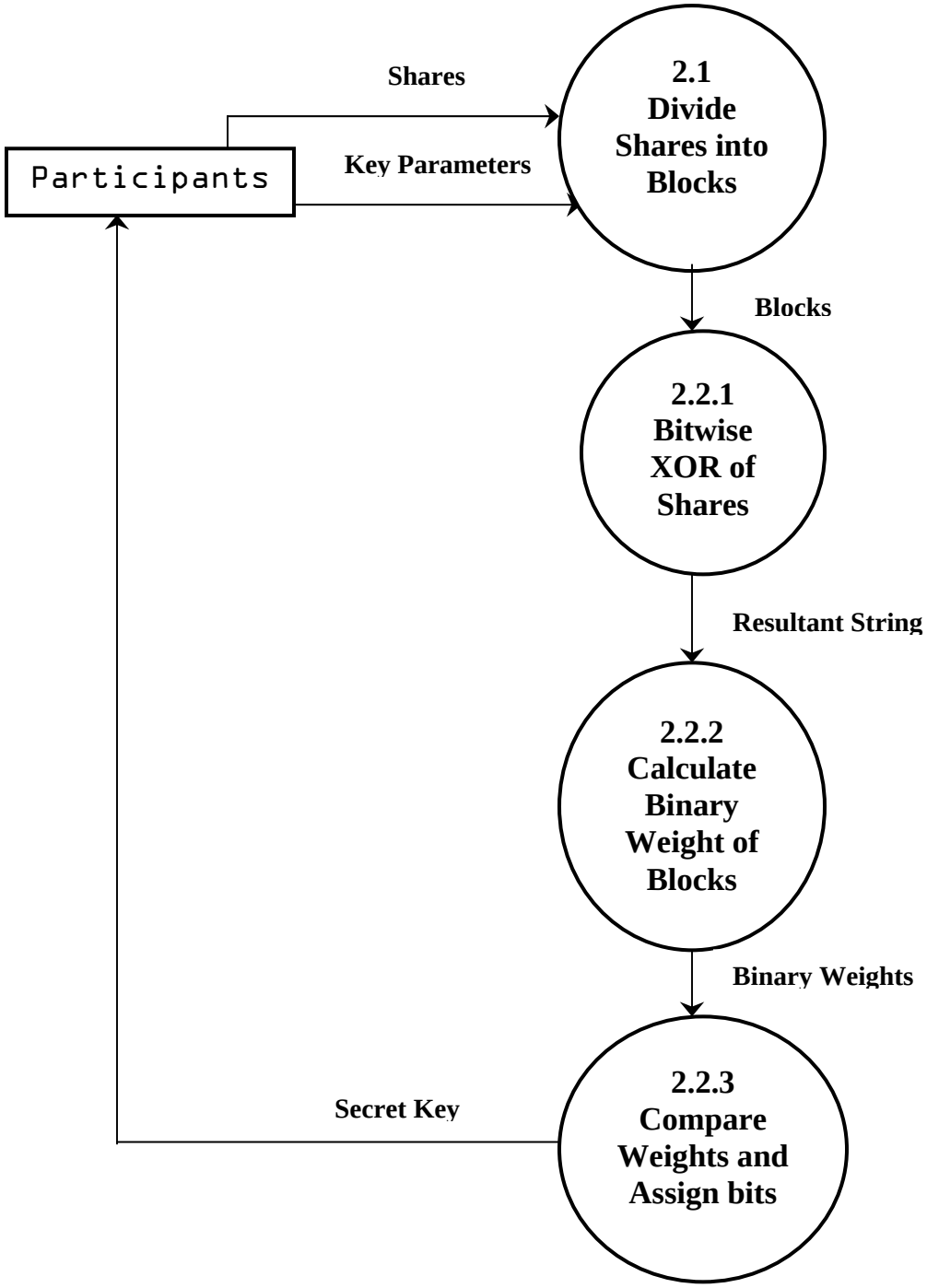
III. OpenSesame: Level - 1 Data Flow Diagram



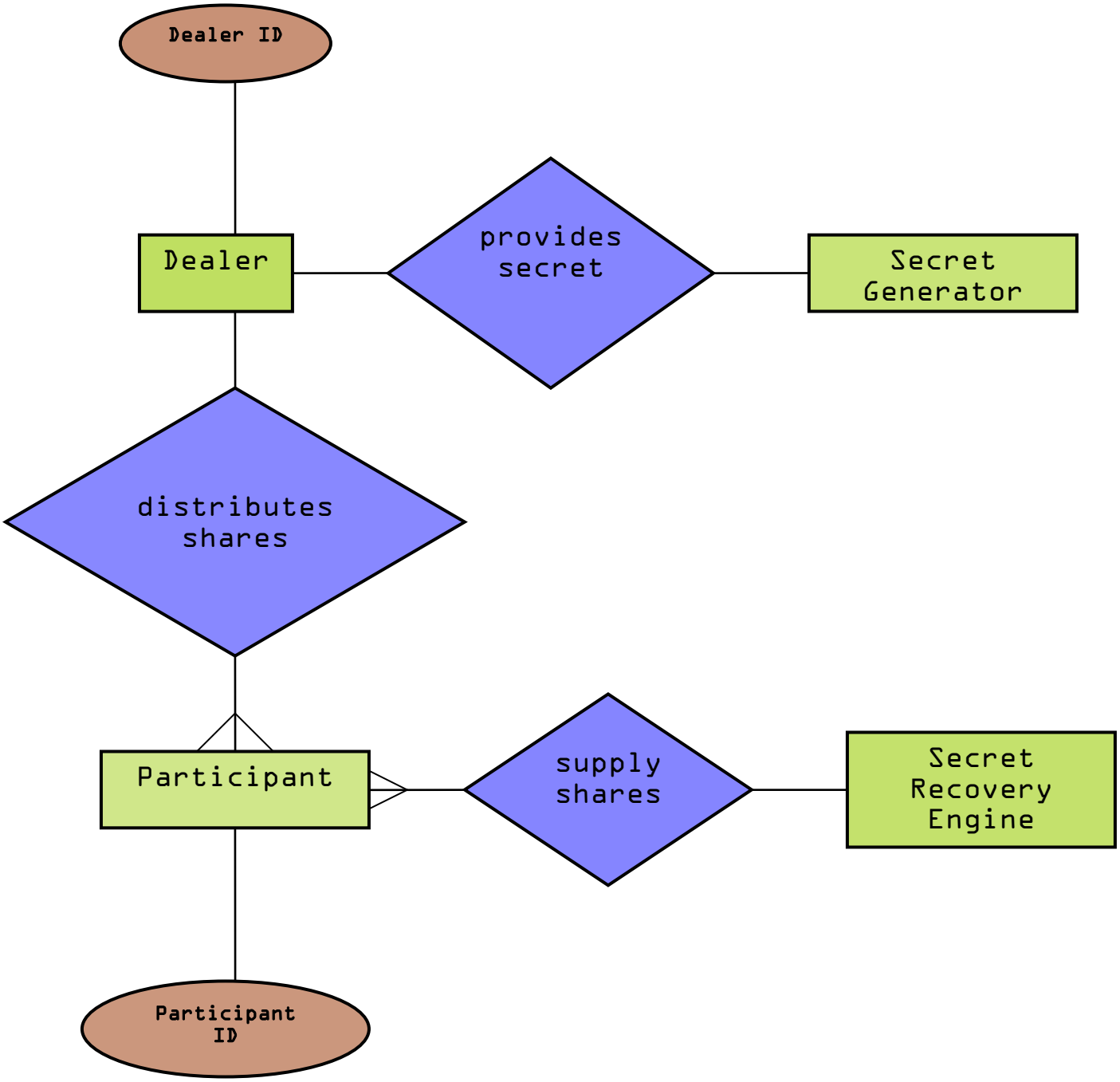
IV. OpenSesame: Level - 2 Data Flow Diagram



OpenSesame: Level - 2 Data Flow Diagram (contd…)



V. OpenSesame: Entity Relationship Diagram



3. Technical Documentation:

This section presents important parts of the source code of OpenSesame and an analysis of the system requirements for using this software. The source code has been commented carefully so that any future modifications to the original code are easy to implement.

Source Code:

Login Window:

```
import java.io.BufferedWriter;
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.FileWriter;
import java.util.Arrays;

/*
 * LogIn.java
 *
 * Created on Feb 6, 2009, 6:14:18 PM
 */
/** author: INDRANIL BANERJEE **/
public class LogIn extends javax.swing.JFrame {

    /** Creates new form LogIn */
    public LogIn() {
        initComponents();
    }

    /** This method is called from within the constructor to
     * initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is
     * always regenerated by the Form Editor.
     */
    @SuppressWarnings("unchecked")
    // <editor-fold defaultstate="collapsed" desc="Generated Code">//GEN-
BEGIN:initComponents
    private void initComponents() {

        userMessage = new javax.swing.JOptionPane();
        changePass = new javax.swing.JDialog();
        jPanel1 = new javax.swing.JPanel();
        jLabel5 = new javax.swing.JLabel();
        jLabel6 = new javax.swing.JLabel();
        oldPass = new javax.swing.JPasswordField();
        newPass = new javax.swing.JPasswordField();
        okPass = new javax.swing.JButton();
        cancel = new javax.swing.JButton();
        jPanel2 = new javax.swing.JPanel();
        jLabel1 = new javax.swing.JLabel();
        jLabel2 = new javax.swing.JLabel();
```

```
        jLabel3 = new javax.swing.JLabel();
        passwd = new javax.swing.JPasswordField();
        exit = new javax.swing.JButton();
        d_logIn = new javax.swing.JButton();
        jSeparator1 = new javax.swing.JSeparator();
        jLabel4 = new javax.swing.JLabel();
        ext = new javax.swing.JButton();
        p_logIn = new javax.swing.JButton();
        changePasButton = new javax.swing.JButton();.
```

Event Handlers:

```
    private void d_logInActionPerformed(java.awt.event.ActionEvent evt) {//GEN-
FIRST:event_d_logInActionPerformed

        // Fetch the entered password.
        password = passwd.getPassword();

        // Fetch the actual password.
        String Password = "";
        try {
            // Open the file
            FileInputStream fstream = new FileInputStream("Password.txt");

            // Convert our input stream to a
            // DataInputStream
            DataInputStream in = new DataInputStream(fstream);

            // Continue to read lines while
            // there are still some left to read
            while (in.available() != 0) {
                // Print file line to screen
                Password = in.readLine();
            }
            in.close();

        } catch (Exception e) {
            System.err.println("File Input Error " + e);
        }

        dealerPass = new char[Password.length()];
        for (int i = 0; i < dealerPass.length; ++i) {
            dealerPass[i] = Password.charAt(i);
        }
        decrypt(dealerPass);

        // Compare password.
        if (Arrays.equals(dealerPass, passwd.getPassword())) {
            new ShareGenerate().setVisible(true);
            this.dispose();
        } else {
            userMessage.showMessageDialog(this, "Incorrect Password !!");
            passwd.setText("");
        }

    }//GEN-LAST:event_d_logInActionPerformed

    private void p_logInActionPerformed(java.awt.event.ActionEvent evt) {//GEN-
FIRST:event_p_logInActionPerformed

        new SecretRecovery().setVisible(true);
```

```
        this.setVisible(false);

    }//GEN-LAST:event_p_logInActionPerformed

    private void messageKeyPressed(java.awt.event.KeyEvent evt) {//GEN-
FIRST:event_messageKeyPressed
    }//GEN-LAST:event_messageKeyPressed

    private void exitActionPerformed(java.awt.event.ActionEvent evt) {//GEN-
FIRST:event_exitActionPerformed
        this.dispose();
    }//GEN-LAST:event_exitActionPerformed

    private void extActionPerformed(java.awt.event.ActionEvent evt) {//GEN-
FIRST:event_extActionPerformed
        this.dispose();
    }//GEN-LAST:event_extActionPerformed

    private void changePasButtonActionPerformed(java.awt.event.ActionEvent evt)
{//GEN-FIRST:event_changePasButtonActionPerformed

        changePass.setSize(410, 215);
        changePass.setVisible(true);
}//GEN-LAST:event_changePasButtonActionPerformed

    private void okPassActionPerformed(java.awt.event.ActionEvent evt) {//GEN-
FIRST:event_okPassActionPerformed
        // Fetch and compare old password
        String Password = "";
        try {
            // Open the file
            FileInputStream fstream = new FileInputStream("Password.txt");

            // Convert our input stream to a
            // DataInputStream
            DataInputStream in = new DataInputStream(fstream);

            // Continue to read lines while
            // there are still some left to read
            while (in.available() != 0) {
                // Print file line to screen
                Password = in.readLine();
            }

            in.close();
        } catch (Exception e) {
            System.err.println("File Input Error " + e);
        }

        dealerPass = new char[Password.length()];
        for (int i = 0; i < dealerPass.length; ++i) {
            dealerPass[i] = Password.charAt(i);
        }
        decrypt(dealerPass);

        // If password matches, write new password in file
        if (Arrays.equals(dealerPass, oldPass.getPassword())) {
            try {
                // Create file
                FileWriter fstream = new FileWriter("Password.txt");
                BufferedWriter out = new BufferedWriter(fstream);
```

```
                char[] pass = newPass.getPassword();
                encrypt(pass);
                out.write(pass);

                out.close();
            } catch (Exception e) {//Catch exception if any
                userMessage.showMessageDialog(this, "Error in Changing Password: "
+ e);
            }

        } else {
            userMessage.showMessageDialog(this, "Incorrect Password !!");
        }

        changePass.dispose();
    }//GEN-LAST:event_okPassActionPerformed

    private void cancelActionPerformed(java.awt.event.ActionEvent evt) {//GEN-
FIRST:event_cancelActionPerformed
        changePass.dispose();
    }//GEN-LAST:event_cancelActionPerformed

    /**
     * @param args the command line arguments
     */
    public static void main(String args[]) {
        java.awt.EventQueue.invokeLater(new Runnable() {

            public void run() {
                Runtime.getRuntime().gc();
                new LogIn().setVisible(true);
            }
        });
    }

    // Variables declaration - do not modify//GEN-BEGIN:variables
    private javax.swing.JButton cancel;
    private javax.swing.JButton changePasButton;
    private javax.swing.JDialog changePass;
    private javax.swing.JButton d_logIn;
    private javax.swing.JButton exit;
    private javax.swing.JButton ext;
    private javax.swing.JLabel jLabel1;
    private javax.swing.JLabel jLabel2;
    private javax.swing.JLabel jLabel3;
    private javax.swing.JLabel jLabel4;
    private javax.swing.JLabel jLabel5;
    private javax.swing.JLabel jLabel6;
    private javax.swing.JPanel jPanel1;
    private javax.swing.JPanel jPanel2;
    private javax.swing.JSeparator jSeparator1;
    private javax.swing.JPasswordField newPass;
    private javax.swing.JButton okPass;
    private javax.swing.JPasswordField oldPass;
    private javax.swing.JButton p_logIn;
    private javax.swing.JPasswordField passwd;
    private javax.swing.JOptionPane userMessage;
    // End of variables declaration//GEN-END:variables

    // Global Variables used throughout the program.
    int count = 0;
    public static char[] dealerPass;
```

```
    public char[] password;

    // Method to encrypt password
    public void encrypt(char[] a) {
        for (int i = 0; i < a.length; ++i) {
            a[i] += 4;
        }
    }

    // Method to decrypt password
    public void decrypt(char[] a) {
        for (int i = 0; i < a.length; ++i) {
            a[i] -= 4;
        }
    }
}
```

Share Generator:

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

/*
 * ShareGenerate.java
 *
 * Created on Feb 6, 2009, 6:46:01 PM
 */
public class ShareGenerate extends javax.swing.JFrame {

    /** Creates new form ShareGenerate */
    public ShareGenerate() {
        Runtime.getRuntime().gc();
        initComponents();
    }

    /** This method is called from within the constructor to
     * initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is
     * always regenerated by the Form Editor.
     */
    @SuppressWarnings("unchecked")
    // <editor-fold defaultstate="collapsed" desc="Generated Code">//GEN-
BEGIN:initComponents
    private void initComponents() {

        basisDialog = new javax.swing.JDialog();
        basisPanel = new javax.swing.JPanel();
        jLabel6 = new javax.swing.JLabel();
        basisStr = new javax.swing.JTextField();
        cancel = new javax.swing.JButton();
        ok = new javax.swing.JButton();
        userMessage = new javax.swing.JOptionPane();
        browse = new javax.swing.JFileChooser();
        jPanel1 = new javax.swing.JPanel();
        jLabel1 = new javax.swing.JLabel();
        jLabel2 = new javax.swing.JLabel();
```

```
        jLabel4 = new javax.swing.JLabel();
        jLabel5 = new javax.swing.JLabel();
        inputBasis = new javax.swing.JButton();
        reset = new javax.swing.JButton();
        genShare = new javax.swing.JButton();
        partc = new javax.swing.JTextField();
        mqs = new javax.swing.JTextField();
        browse1 = new javax.swing.JButton();
        secretFile1 = new javax.swing.JTextField();
        time = new javax.swing.JLabel();
```

Event Handlers:

```
private void inputBasisActionPerformed(java.awt.event.ActionEvent evt) {//GEN-
FIRST:event_inputBasisActionPerformed

    basisDialog.setSize(525, 275);
    basisDialog.setVisible(true);

}//GEN-LAST:event_inputBasisActionPerformed

private void okActionPerformed(java.awt.event.ActionEvent evt) {//GEN-
FIRST:event_okActionPerformed

    String str = basisStr.getText();
    String[] arr = str.split(",");
    basis[count++] = new int[arr.length];

    for (int i = 0; i < arr.length; ++i) {
        basis[count - 1][i] = Integer.parseInt(arr[i]);
    }



    basisStr.setText("");
    temp1--;

    if (temp1 < 1) {
        basisDialog.setVisible(false);
        inputBasis.setEnabled(false);

    }


}//GEN-LAST:event_okActionPerformed

private void partcFocusLost(java.awt.event.FocusEvent evt) {//GEN-
FIRST:event_partcFocusLost
    p = Integer.parseInt(partc.getText());
}//GEN-LAST:event_partcFocusLost

private void mqsFocusLost(java.awt.event.FocusEvent evt) {//GEN-
FIRST:event_mqsFocusLost
    n = Integer.parseInt(mqs.getText());
    temp1 = n;

    basis = new int[n][];
    common = new int[n][n];
}//GEN-LAST:event_mqsFocusLost
```

```
private void genShareActionPerformed(java.awt.event.ActionEvent evt) {//GEN-
FIRST:event_genShareActionPerformed

    long start = System.currentTimeMillis();

    //To identify pairs of minimal qualified sets with maximum intersection.
    init_flag(n);
    init_common(n);
    basis_comp();
    pair();

    //End of this block
    /*This block solves the system of linear equations and generates the shares*/

    create_g();
    Runtime.getRuntime().gc();
    //share = new short[p][columns * secretKey.length()];

    //Code to determine which minimal qualified elements should be paired.
    for (int i = 0; i < n; ++i) {
        if (flag1[i] == 0) {
            solve_eq1(i);   //the code to solve a single equation.
        } else {
            for (int j = i + 1; j < n; ++j) {
                if (flag1[i] == flag1[j]) {
                    int x = common[i][j];
                    int y = (basis[i].length - x);
                    int z = (basis[j].length - x);
                    solve_eq(x, y, z, i, j);
                }
            }
        }
    }

    out_share();

    /*for (int j = 0; j < p; ++j) {

    int k = j + 1;
    String name = "Share" + k + ".txt";

    try {
    // Create file
    FileWriter fstream = new FileWriter(name);
    BufferedWriter out = new BufferedWriter(fstream);

    for (int i = 0; i < secretKey.length() * columns; ++i) {
    out.write("" + share[j][i]);
    }
    //Close the output stream
    out.close();
    } catch (Exception e) {//Catch exception if any
    System.err.println("Error: " + e.getMessage());
    }
    //printShare(str, name);
    }*/

    String col = "Number of columns is: " + columns + "\n\n" + "Length of the
Secret is: " + secretKey.length() + " bits";
    printShare(col, "keyParam.txt");
```

```
    long end = System.currentTimeMillis();

    float duration = (end - start) / 1000;
    time.setText("Time Elapsed: " + duration + " s");

}//GEN-LAST:event_genShareActionPerformed

private void resetActionPerformed(java.awt.event.ActionEvent evt) {//GEN-
FIRST:event_resetActionPerformed
    new LogIn().setVisible(true);
    this.setVisible(false);
}//GEN-LAST:event_resetActionPerformed

private void frameWindowClosed(java.awt.event.WindowEvent evt) {//GEN-
FIRST:event_frameWindowClosed
    System.out.println("Bye !!");
}//GEN-LAST:event_frameWindowClosed

private void browseActionPerformed(java.awt.event.ActionEvent evt) {//GEN-
FIRST:event_browseActionPerformed

    // To measure the runtime
    long start = System.currentTimeMillis();

    try {
        // Open the file
        FileInputStream fstream = new FileInputStream(browse.getSelectedFile());

        // Convert our input stream to a
        // DataInputStream
        DataInputStream in = new DataInputStream(fstream);

        // Continue to read lines while
        // there are still some left to read
        while (in.available() != 0) {
            // Print file line to screen
            input += in.readLine() + "\n";
        }
        in.close();

    } catch (java.lang.StringIndexOutOfBoundsException e) {
        userMessage.showMessageDialog(this, "Values of some fields may be incorrect
!!");

    } catch (Exception e) {
        userMessage.showMessageDialog(this, "Error in Reading File: " + e);
    }

    System.out.println("\nFile reading complete " + input.length());

    try {
        // Create file
        FileWriter fstream = new FileWriter("binaryKey.txt");
        BufferedWriter out = new BufferedWriter(fstream);

        //this part takes time
        for (int i = 0; i < input.length(); ++i) {
            char a = input.charAt(i);

            String temp = Integer.toBinaryString((int) a);
            while (temp.length() < 8) {
                temp = "0" + temp;
```

```
            }
            //System.out.println("\n"+temp+"\n");
            //secretKey += temp;
            out.write(temp);
        }

        out.close();
    } catch (Exception e) {//Catch exception if any
        userMessage.showMessageDialog(this, "Error in Writing File: " + e);
    }

    try {
        BufferedReader in = new BufferedReader(new FileReader("binaryKey.txt"));
        String str;
        while ((str = in.readLine()) != null) {
            secretKey += str;
        }
        in.close();
    } catch (IOException e) {
        userMessage.showMessageDialog(this, "Error in Reading File: " + e);
    }
    input = "";
    long end = System.currentTimeMillis();

    float duration = (end - start) / 1000;
    time.setText("Time Elapsed: " + duration + " s");

}//GEN-LAST:event_browseActionPerformed

private void browse1ActionPerformed(java.awt.event.ActionEvent evt) {//GEN-
FIRST:event_browse1ActionPerformed
    browse.showDialog(this, "Select Key");
    secretFile1.setText(browse.getSelectedFile().getPath());

}//GEN-LAST:event_browse1ActionPerformed

private void cancelActionPerformed(java.awt.event.ActionEvent evt) {//GEN-
FIRST:event_cancelActionPerformed
    basisDialog.dispose();
}//GEN-LAST:event_cancelActionPerformed

    // Variables declaration - do not modify//GEN-BEGIN:variables
    private javax.swing.JDialog basisDialog;
    private javax.swing.JPanel basisPanel;
    private javax.swing.JTextField basisStr;
    private javax.swing.JFileChooser browse;
    private javax.swing.JButton browse1;
    private javax.swing.JButton cancel;
    private javax.swing.JButton genShare;
    private javax.swing.JButton inputBasis;
    private javax.swing.JLabel jLabel1;
    private javax.swing.JLabel jLabel2;
    private javax.swing.JLabel jLabel4;
    private javax.swing.JLabel jLabel5;
    private javax.swing.JLabel jLabel6;
    private javax.swing.JPanel jPanel1;
    private javax.swing.JTextField mqs;
    private javax.swing.JButton ok;
    private javax.swing.JTextField partc;
    private javax.swing.JButton reset;
    private javax.swing.JTextField secretFile1;
    private javax.swing.JLabel time;
```

```
    private javax.swing.JOptionPane userMessage;
    // End of variables declaration//GEN-END:variables

    // Global Variables Used
    public int p,  n,  temp1,  count = 0,  count1 = 0,  columns = 0,  count2 = -1,
count3 = -1;
    //p: no. of participants, n: no. of minimal qualified sets
    public int[][] basis,  common;
    int[][] G0, G1, share;
    public String secretKey = "",  output,  input = "";
    public int[] flag1,  flag2;


    // Global Methods Used

    //Method to initialize the flag arrays
    public void init_flag(int len) {
        flag1 = new int[n];
        flag2 = new int[n];

        for (int i = 0; i < len; ++i) {
            flag1[i] = 0;
            flag2[i] = 0;
        }
    }

    //Method to initialize the common array
    public void init_common(int len) {
        for (int i = 0; i < len; ++i) {
            for (int j = 0; j < len; ++j) {
                common[i][j] = -1;
            }
        }
    }



     //Method to return the number of elements common between two specified rows of
     basis array
    public int compare(int row1, int row2) {
        int com = 0;

        for (int i = 0; i < basis[row1].length; ++i) {
            int x = basis[row1][i];

            for (int j = 0; j < basis[row2].length; ++j) {
                if (x == basis[row2][j]) {
                    com++;
                }
            }
        }
        return (com);
    }

    //Method to compare each pair of minimal qualified sets for common participants
    public void basis_comp() {

        for (int i = 0; i < n; ++i) {
            for (int j = (i + 1); j < n; ++j) {
                common[i][j] = compare(i, j);
            }
        }
```

```
}

//Method to return the biggest element of the common[][] array.
public void max_common() {
    int max = 0;
    int row1 = -1, col1 = -1;

    for (int i = 0; i < n; ++i) {
        for (int j = (i + 1); j < n; ++j) {
            if (flag1[i] != 0 || flag1[j] != 0) {
                continue;
            }

            if (common[i][j] > max) {
                max = common[i][j];
                row1 = i;
                col1 = j;

            }
        }
    }
    if (row1 != -1 && col1 != -1) {
        flag1[row1] = flag1[col1] = ++count1;
    }
}

//Method to pair the minimal qualified elements.
public void pair() {
    int i, j;

    if ((n % 2) > 0) {
        j = (n - 1) / 2;
    } else {
        j = n / 2;
    }

    for (i = 0; i < j; ++i) {
        max_common();
    }

}

//Method to create and initialize the generating matrices.
public void create_g() {
    int sum = 0, com, x, y;

    //To calculate the number of columns in the Generting matrices.
    for (int i = 0; i < n; ++i) {

        if (flag1[i] == 0) {
            x = (basis[i].length) - 1;
            columns += Math.pow(2, x);
        }

        for (int j = (i + 1); j < n; ++j) {

            if (flag1[j] == 0) {
                continue;
            }

            if (flag1[i] == flag1[j]) {
                com = compare(i, j);
```

```
                x = basis[i].length;
                y = basis[j].length;
                sum = (x + y) - (com + 2);
                columns += Math.pow(2, sum);
            }
        }
    }

    //To initialize the generating matrices
    G0 = new int[p][columns];
    G1 = new int[p][columns];

    for (int i = 0; i < p; ++i) {
        for (int j = 0; j < columns; ++j) {
            G0[i][j] = 0;
            G1[i][j] = 0;

        }
    }
}

//Method to return the XOR of the digits of an integer.
public int num_XOR(int num) {
    int mask = 1, acc = 0;

    for (int i = 0; i < 8; ++i) {
        int M = mask & num;
        acc = acc ^ M;
        num = num >>> 1;
    }
    return (acc);
}

//Methods to update the elements of the generating matrices.
public void update_G0(int v1, int v2, int v3, int row1, int row2, int c2) {
    char mask = 1, flag = 0;

    for (int i = 0; i < basis[row1].length; ++i) {
        int X = basis[row1][i];
        flag = 0;
        for (int j = 0; j < basis[row2].length; ++j) {
            if (X == basis[row2][j]) {
                flag = 1;
            }
        }

        if (flag == 1) {    //it is a common element between row1 and row2
            short M = (short) (mask & v1);
            G0[X - 1][c2] = M;
            v1 = v1 >>> 1;

        } else {
            short M = (short) (mask & v2);
            G0[X - 1][c2] = M;
            v2 = v2 >>> 1;
        }
    }

    for (int i = 0; i < basis[row2].length; ++i) {
        int X = basis[row2][i];
        flag = 0;
        for (int j = 0; j < basis[row1].length; ++j) {
```

```
                if (X == basis[row1][j]) {
                    flag = 1;
                }
            }

            if (flag == 0) {    //its an extra variable of the second equation
                short M = (short) (mask & v3);
                G0[X - 1][c2] = M;
                v3 = v3 >>> 1;
            }
        }

    }


    // Method to update the Generating matrix
    public void update_G1(int v1, int v2, int v3, int row1, int row2, int c3) {
        int mask = 1, flag = 0;

        for (int i = 0; i < basis[row1].length; ++i) {
            int X = basis[row1][i];
            flag = 0;
            for (int j = 0; j < basis[row2].length; ++j) {
                if (X == basis[row2][j]) {
                    flag = 1;
                }
            }

            if (flag == 1) {    //it is a common element between row1 and row2
                short M = (short) (mask & v1);
                G1[X - 1][c3] = M;
                v1 = v1 >>> 1;

            } else {
                short M = (short) (mask & v2);
                G1[X - 1][c3] = M;
                v2 = v2 >>> 1;
            }
        }

        for (int i = 0; i < basis[row2].length; ++i) {
            int X = basis[row2][i];
            flag = 0;
            for (int j = 0; j < basis[row1].length; ++j) {
                if (X == basis[row1][j]) {
                    flag = 1;
                }
            }

            if (flag == 0) {    //its an extra variable of the second equation
                short M = (short) (mask & v3);
                G1[X - 1][c3] = M;
                v3 = v3 >>> 1;
            }
        }

    }

    //Method to solve the systems of linear equations.
    public void solve_eq(int com_var, int ex_var1, int ex_var2, int row1, int row2)
{
```

```
        //int mask=1;

        for (int i = 0; i < Math.pow(2, com_var); ++i) {
            for (int j = 0; j < Math.pow(2, ex_var1); ++j) {
                for (int k = 0; k < Math.pow(2, ex_var2); ++k) {

                    if (num_X0R(i) == num_X0R(j) && num_X0R(i) == num_X0R(k)) {
/*Condition for solution G0*/
                        count2++;
                        update_G0(i, j, k, row1, row2, count2);
                    }

                    if (num_X0R(i) != num_X0R(j) && num_X0R(i) != num_X0R(k)) {
/*Condition for solution G1*/
                        count3++;
                        update_G1(i, j, k, row1, row2, count3);
                    }

                }
            }
        }
    }

    // Method to solve single linear equation
    public void solve_eq1(int row) {
        System.out.println("solve eq has been called !!  " + row);
        for (int i = 0; i < Math.pow(2, basis[row].length); ++i) {
            if (num_X0R(i) == 0) {
                count2++;
                update1_G0(i, row, count2);

            } else {
                count3++;
                update1_G1(i, row, count3);
            }
        }

    }

    // Method to update generating matrices when no participants are in common
    public void update1_G0(int var1, int row, int c2) {
        int mask = 1;

        for (int i = 0; i < basis[row].length; ++i) {
            short M = (short) (mask & var1);
            int a = basis[row][i];
            //System.out.println(a+"   "+M);
            //System.out.println(columns+"");
            G0[a - 1][c2] = M;
            var1 = var1 >>> 1;
        }
    }

    // Method to update generating matrices when no participants are in common
    public void update1_G1(int var1, int row, int c3) {
        int mask = 1;

        for (int i = 0; i < basis[row].length; ++i) {
            short M = (short) (mask & var1);
            G1[basis[row][i] - 1][c3] = M;
            var1 = var1 >>> 1;
        }
```

```
    }

    //Method to calculate the shares.
    public void out_share() {

        for (int i = 0; i < p; ++i) {
            try {
                // Create file
                int num=i+1;
                String name="Share"+num+".txt";
                FileWriter fstream = new FileWriter(name);
                BufferedWriter out = new BufferedWriter(fstream);

                for(int j=0;j<secretKey.length();++j){
                    if(secretKey.charAt(j)=='0'){
                        for(int k=0;k<columns;++k){
                            out.write(""+G0[i][k]);
                        }
                    }

                    if(secretKey.charAt(j)=='1'){
                        for(int k=0;k<columns;++k){
                            out.write(""+G1[i][k]);
                        }
                    }

                }

                out.close();
            } catch (Exception e) {//Catch exception if any
                userMessage.showMessageDialog(this, "Error in Writing File: " + e);
            }

        }

    }

    public static void printShare(String share, String name) {
        try {
            // Create file
            FileWriter fstream = new FileWriter(name);
            BufferedWriter out = new BufferedWriter(fstream);
            out.write(share);
            //Close the output stream
            out.close();
        } catch (Exception e) {//Catch exception if any
            System.err.println("Error: " + e.getMessage());
        }
    }
}
```

Secret Recovery Engine:

```
import java.io.BufferedWriter;
```

```
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.FileWriter;


/*
 * SecretRecovery.java
 *
 * Created on Feb 6, 2009, 8:45:31 PM
 */
public class SecretRecovery extends javax.swing.JFrame {

    /** Creates new form SecretRecovery */
    public SecretRecovery() {
        initComponents();
    }

    /** This method is called from within the constructor to
     * initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is
     * always regenerated by the Form Editor.
     */
    @SuppressWarnings("unchecked")
    // <editor-fold defaultstate="collapsed" desc="Generated Code">//GEN-
BEGIN:initComponents
    private void initComponents() {

        fselect = new javax.swing.JFileChooser();
        userMessage = new javax.swing.JOptionPane();
        jPanel1 = new javax.swing.JPanel();
        jLabel1 = new javax.swing.JLabel();
        jLabel2 = new javax.swing.JLabel();
        jLabel3 = new javax.swing.JLabel();
        jLabel4 = new javax.swing.JLabel();
        enterShares = new javax.swing.JButton();
        reset = new javax.swing.JButton();
        recovKey = new javax.swing.JButton();
        len = new javax.swing.JTextField();
        cols = new javax.swing.JTextField();
        time = new javax.swing.JLabel();
```

Event Handlers:

```
    private void lenActionPerformed(java.awt.event.ActionEvent evt) {//GEN-
FIRST:event_lenActionPerformed
        // TODO add your handling code here:
}//GEN-LAST:event_lenActionPerformed

    private void enterSharesActionPerformed(java.awt.event.ActionEvent evt) {//GEN-
FIRST:event_enterSharesActionPerformed

        fselect.showDialog(this, "Select Share");
}//GEN-LAST:event_enterSharesActionPerformed

    private void recovKeyActionPerformed(java.awt.event.ActionEvent evt) {//GEN-
FIRST:event_recovKeyActionPerformed

        long start = System.currentTimeMillis();

        for (int i = 0, block = 0; i < lenShare; i += columns, block++) {
```

```
            int temp = 0;
            for (int j = 0; j < columns; ++j) {
                if (store[i + j] == '1') {
                    temp++;
                }
            }
            binWt[block] = temp;
        }

        int max = 0;
        for (int i = 0; i < lenSec; ++i) {
            if (binWt[i] > max) {
                max = binWt[i];
            }
        }

        for (int j = 0; j < lenSec; ++j) {
            if (binWt[j] == max) {
                secretKey[j] = '1';
            } else {
                secretKey[j] = '0';
            }
        }

        convChar();
        //ShareGenerate.printShare(secretKey1,"secretKey.txt");
        try {
            // Create file
            FileWriter fstream = new FileWriter("secretKey.txt");
            BufferedWriter out = new BufferedWriter(fstream);
            //for(int k=0;k<secretPrint.length;++k){
            out.write(secretPrint);
            out.close();
        } catch (Exception e) {//Catch exception if any
            System.err.println("Error: " + e.getMessage());
        }
        //}
        long end = System.currentTimeMillis();

        float duration = (end - start) / 1000;
        time.setText("Time Elapsed: " + duration + " s");


    }//GEN-LAST:event_recovKeyActionPerformed

    private void fselectActionPerformed(java.awt.event.ActionEvent evt) {//GEN-
FIRST:event_fselectActionPerformed

        long start = System.currentTimeMillis();
        try {
            // Open the file
            FileInputStream fstream = new
FileInputStream(fselect.getSelectedFile());

            // Convert our input stream to a
            // DataInputStream
            DataInputStream in = new DataInputStream(fstream);

            // Continue to read lines while
            // there are still some left to read
            while (in.available() != 0) {
```

```
                in.readFully(share);

            }

            accum(share);
            //System.out.println("till here!");

            in.close();
        } catch (java.lang.StringIndexOutOfBoundsException e) {
            userMessage.showMessageDialog(this, "Values of some fields may be
incorrect !!");

        } catch (Exception e) {
            System.err.println("File Input Error " + e);
        }

        long end = System.currentTimeMillis();

        float duration = (end - start) / 1000;
        time.setText("Time Elapsed: " + duration + " s");

    }//GEN-LAST:event_fselectActionPerformed

    private void lenFocusLost(java.awt.event.FocusEvent evt) {//GEN-
FIRST:event_lenFocusLost
        lenSec = Integer.parseInt(len.getText());
        binWt = new int[lenSec];
        secretKey = new char[lenSec];
        secretPrint = new char[lenSec / 8];
    }//GEN-LAST:event_lenFocusLost

    private void colsFocusLost(java.awt.event.FocusEvent evt) {//GEN-
FIRST:event_colsFocusLost
        columns = Integer.parseInt(cols.getText());
        lenShare = lenSec * columns;
        store = new char[lenShare];
        share = new byte[lenShare];

        for (int i = 0; i < lenShare; ++i) {
            store[i] = '0';
        }
    }//GEN-LAST:event_colsFocusLost

    private void resetActionPerformed(java.awt.event.ActionEvent evt) {//GEN-
FIRST:event_resetActionPerformed
        new LogIn().setVisible(true);
        this.setVisible(false);
    }//GEN-LAST:event_resetActionPerformed
    // Variables declaration - do not modify//GEN-BEGIN:variables
    private javax.swing.JTextField cols;
    private javax.swing.JButton enterShares;
    private javax.swing.JFileChooser fselect;
    private javax.swing.JLabel jLabel1;
    private javax.swing.JLabel jLabel2;
    private javax.swing.JLabel jLabel3;
    private javax.swing.JLabel jLabel4;
    private javax.swing.JPanel jPanel1;
    private javax.swing.JTextField len;
    private javax.swing.JButton recovKey;
    private javax.swing.JButton reset;
    private javax.swing.JLabel time;
    private javax.swing.JOptionPane userMessage;
```

```
    // End of variables declaration//GEN-END:variables

    //Global Variables
    int lenSec, columns, lenShare;
    int[] binWt;
    char[] store, secretKey, secretPrint;
    byte[] share;

    public void accum(byte[] s) {

        for (int i = 0; i < s.length; i++) {
            if (s[i] == '1' || store[i] == '1') {
                store[i] = '1';
            } else {
                store[i] = '0';
            }
        }
    }

    public void convChar() {


        for (int i = 0; i < secretKey.length; i = i + 8) {

            int temp = 0;
            for (int j = 0; j < 8; ++j) {
                if (secretKey[i + j] == '1') {
                    temp += Math.pow(2, (7 - j));
                }
            }
            char c = (char) temp;
            secretPrint[i / 8] = c;
        }

    }
}
```

System Requirements:

- Windows 98 or higher or any other equivalent platform.
- P-III or higher or any other equivalent processor.
- At least 256 MB of primary memory.
- Java Runtime Environment.

4. User Documentation: An interactive tutorial

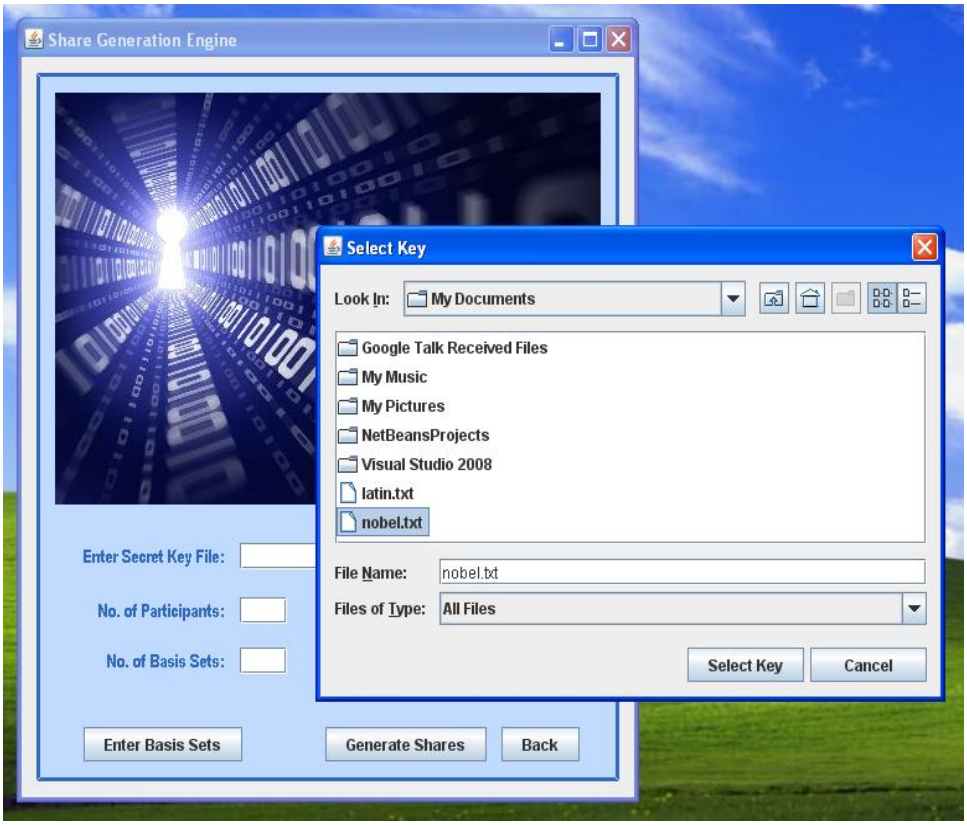
This section presents a simplified interactive tutorial on the use of OpenSesame. This document is meant for the use of the end user. No special background is required on the use of computers to operate this software. Following is a step by step rendering of the process:

The Dealer:

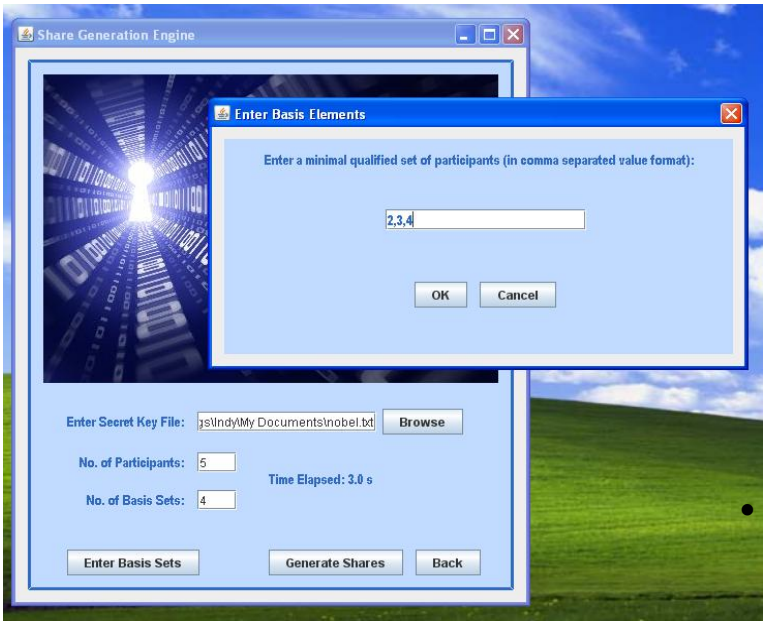
- Login to the Share Generator using the password provided. The default password is “dealer”.

- Upon logging in the Dealer is presented with the Share Generator window.

- Click on the “Browse” button to select the “secret key” file. A file browser window opens.
- Select the .txt secret key file from the file browser window.



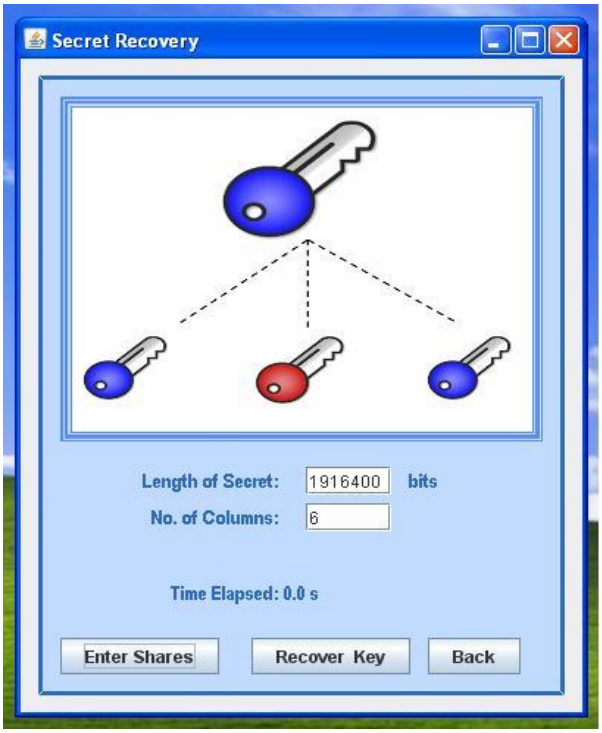
- Enter the number of Participants and the number of Minimal Qualified Sets into the text boxes labeled the same.

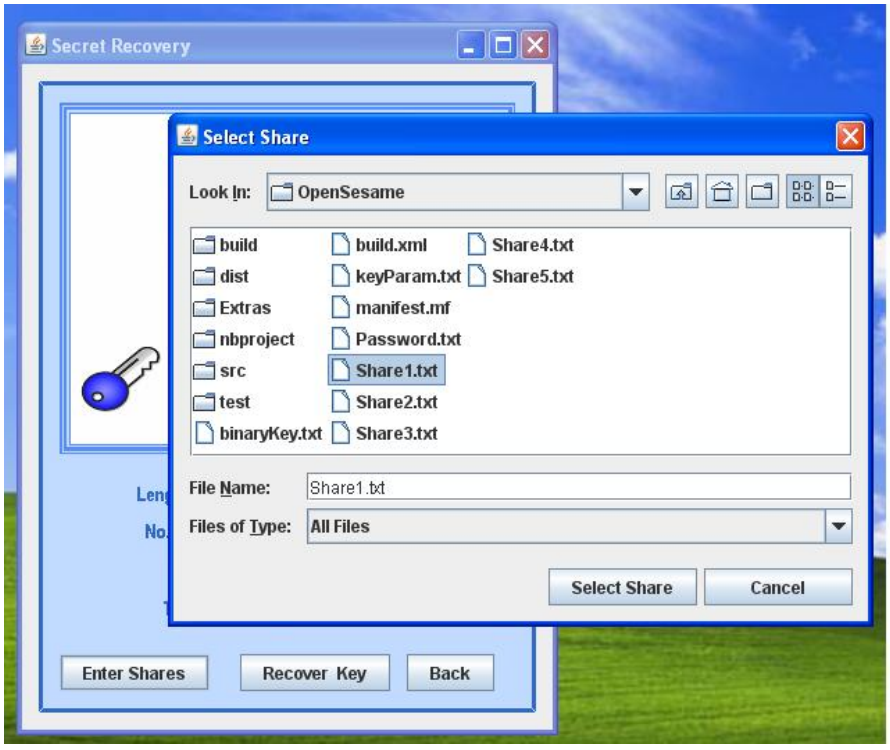


- Click “Enter Basis” to display the dialog box as shown.
- Enter the basis sets in comma separated value format as shown.
- Click “OK” to confirm the sets.
- Click “Cancel” to cancel an entry.
- After finishing, click “Generate Shares”
- Click the “Back” button to go back to the login window.

The Participants:

- Click “Login” to enter the Secret Recovery Engine.
- Enter the key parameters.





- Click “Enter Shares” to open a file browser.
- Select the Share files.
- Click “Select Share” to confirm the entry.
- Click “Recover Key” to begin the key reconstruction process.
- The secret key is output as a .txt file in the working directory.

9. DISCUSSION OF RESULTS

Input:

- Secret Key: The lecture of renowned physicist Richard Feynman at the dinner of the Nobel Prize ceremony was selected as the Secret File: “nobel.txt”
- Access Structure: 5 participants in the form {{1,2}, {1,3},{2,3,4},{2,4,5}}

Output:

- 5 Share files were created in the .txt format labeled: “Share1.txt”, “Share2.txt” and so on.
- One .txt file named “keyParameters” containing the length of the secret and the number of columns in the generating matrices.

Operating Parameters:

- It took 6 seconds time to generate the shares.
- It took 4 seconds time to reconstruct the secret from the shares of participant 2, 3 & 4.
- If the secret key file exceeds 150 kb, the JVM runs out of heap memory.

The software runs both on the Microsoft Windows platform and the Linux platform. It is slightly sluggish on any machine which contains less than 512 MB of primary memory.