Implementability of a Black-Box Secret Sharing Schemes

Mathieu David
UGA
Grenoble, France
mathieu.david1@etu.univ-grenoble-alpes

Supervised by: Pierre Karpman

I understand what plagiarism entails and I declare that this report is my own, original work.
Name, date and signature: DAVID Mathieu, 09/06/23

Abstract

A secret n-k threshold sharing scheme is composed of two algorithms. One will split a secret s, element of an Abelian group, into n shares, also element of the group. The other one will retrieve the secret if enough at least k shares are given as input. A black box secret sharing will works in the same way for any Abelian groups. This type of schemes are way difficult to build. R.Cramer C.Xing explicited a theoretical construction of such a scheme with interesting properties. The main idea of this report is to study the fluency to implement this scheme and possible hidden limitations. All of the implementation process has been done with the software SageMath which performs well for difficult mathematical computations. This construction seems to implies the manupilation of very large coefficients that could lead to slower the creation of the scheme.

1 Introduction

In modern cryptography, a Secret sharing scheme can be defined as a pair of algorithms. The first one is a non-deterministic algorithm which will be called A such that A takes as input a secret s and a parameter n. This algorithm will return n *shares* of s (let us call the vector of n shares $\boldsymbol{s}$). The second algorithm is a deterministic reconstruction algorithm which will be called R such that, if R has a sufficient amount of information (i.e. in other words enough shares as input), R can retrieve s (this is called the completeness property). Otherwise, R cannot retrieve the secret (either the algorithm R give partial information on s, or does not bring any information on s). This is called the privacy property and will be developed in the next paragraph.

To further detail the notion of a "sufficient amount of information", we define Γ an *access structure* as a set of subsets of $\{1, ..., n\}$ (i.e. $\Gamma \in \mathscr{P}(\{1, ..., n\})$) such that if we give an element of Γ to R, we provide enough informations to retrieve s. We will note Δ also an element of $\mathscr{P}(\{1, ..., n\})$, the set of all configuration which do not allow us to retrieve any information on s. In a general way, Γ and Δ do not necessarily form a partition of $\mathscr{P}(\{1, ..., n\})$. In that case, there exists an element in $\mathscr{P}(\{1, ..., n\})\backslash(\Gamma \cup \Delta)$ which can bring partial information on s. Our work will be focused on the case where Γ and Δ are complete (i.e. $\Delta \cup \Gamma = \mathscr{P}(\{1, ..., n\})$). Then, we talk about an (n, k) threshold secret sharing scheme if R can retrieve the secret s if and only if R has (at least) k shares of s as input. In other word the access structure for this secret-sharing scheme is $\Gamma = \{e \in \mathscr{P}(\{1, ..., n\})$ such that $card(e) > k - 1\}$ and $\Delta = \{e \in \mathscr{P}(\{1, ..., n\})$ such that $card(e) < k\}$.

Shamir explicits in [Shamir, 1979] a (now) well-known method to build such a scheme relying on linear algebra and polynomial interpolation. But the fact is that this scheme requires the shares being element of a finite field whose size is function of n. More generally, for any k and n, there exists a secret sharing scheme over a finite field of a specific size (with constraint on the size of the field). If we note q the size of the finite field, the constraint said that we need $q \geq n + 1$. With secret sharing schemes that exists today, it is much more difficult and expensive to deploy a scheme on any imposed finite Abelian group than building a scheme knowing in advance the group. That is why we would like to build a secret sharing scheme that is more convenient and works in the same way no matter the group. Then, we call black box secret sharing scheme (BBSSS) a secret-sharing scheme that works in the same way no matter the finite Abelian group. For the rest of the paper, when we will talk groups and field, we mean *finite Abelian groups* and *finite fields*. In addition, a finite field of size q will be noted $\mathbb{F}_q$.

Shamir's method is well suited for threshold secret sharing schemes but this method does not work for more general access structure (i.e. elements in Γ that not necessarily have the same size). In [Ito *et al.*, 1989], a method to build a secret sharing scheme is explicited. The main problem with this method is the size of each shares that may grow exponentially (i.e. the size of each piece of secret returned by the algorithm A). We call this size increase the *expansion factor*.

R. Cramer and C. Xing, explicit in [Cramer and Xing, 2019] a theoretical construction of such a threshold BBSSS with a lower bound on the expansion factor. The purpose of the internship is to study boundaries between the theoretical construction and the implementation. We will try to understand how this BBSSS is built and how it behaves when we encode or decode messages. The other goal is pointing possible limitations of this construction. One could ask the following question : Is there a trade off between the expansion factor and other possibly expensive operations in this construction ?

This work is difficult because this construction involves many new mathematical notions related to linear algebra and finite fields. In addition, a deep understanding of the theory behind the construction provided in [Cramer and Xing, 2019] is needed to do the implementation part. That is why the first part of the internship was dedicated to a literature study. The second one was more focused on all implementation related details and going deeper in the understanding of the BBSSS.

In this implementation attempt, we will focus ourselves on instantiating the BBSSS with a particular family of linear codes (that will be defined later in the report in the subsection 2.3). Other instantiations are possible with other family of linear codes but this will not be covered in this work. In the first part of the report, we are going to talk about some technical notions described in [Cramer and Xing, 2019] mandatory for the next part of the work. In the second one, we are going to talk about implementation details and possible limitations of this construction.

2 A detailed summary of the needed notions in [Cramer and Xing, 2019]

To provide a partial implementation of a BBSSS, a documentation step is needed. All the first part of my internship was to understand important notions related to secret sharing and BBSSS. In this section, I will talk about multiple notions and mathematical objects needed to build such a secret sharing scheme.

2.1 More details on Black-box secret sharing schemes

To make the link with monotone span program (described in the next sub-part), we need to provide clarifications. A more technical way to define BBSSS is described in [Cramer and Xing, 2019] : we consider a matrix $\boldsymbol{M}$ over a ring R and G an arbitrary group. In addition, a surjective function ψ from $\mathscr{P}(\{1,...,n\})$ (where n is the number of shares) to $\mathscr{P}(\{1,...,h\})$ (where h is the row dimension of the matrix) is defined in [Cramer and Xing, 2019]. Since we will only talk about threshold secret sharing schemes (i.e. $h = n$), ψ will not be involved in the rest of the paper.

A way to obtain shares from a secret $s \in G$ is to build a vector $\boldsymbol{g} = (s, g_2, ..., g_n)$, where $g_2, ..., g_n$ are uniformly sampled from G, and multiply this vector with the transpose of the matrix $\boldsymbol{M}$ (i.e. $\boldsymbol{S} = g\boldsymbol{M}^\intercal$).

In the rest of the report, since we considered ψ as the identity function, we will consider a matrix $\boldsymbol{M}_S$ the submatrix composed of the row of indices in $S \in \mathscr{P}(\{1,...,n\})$. For a vector, V_S , where $S \in \mathscr{P}(\{1,...,n\})$, can be seen as the vector V composed of the entries which have indices in S. We also define a *reconstruction vector* Λ for each element a of Γ such that the product $\Lambda \cdot \boldsymbol{s}_a$ allows us to retrieve s. The set $\mathscr{R}$ will define the collection of reconstruction vectors. The privacy condition said that for each element $b \in \Delta$, $\boldsymbol{s}_b$ is statistically independent from s.

2.2 Definition and purpose of a monotone span program

A monotone span program is defined in [Cramer and Xing, 2019] by a tuple of four elements : a ring R, a matrix $\boldsymbol{M}$ over the ring R, the same surjective function ψ as defined in the previous subsection on BBSSS, and a target vector called μ over the ring R such that $\mu = (1, 0, ..., 0)$.

There is a direct link between access structure and monotone span program since we say that a monotone span program can *compute an access structure* Γ if for any element S of Γ, $\mu \in im(\boldsymbol{M}_S)$. In other words, μ is generated by the submatrix of support S. In addition, for any element T of Δ, the first column of $\boldsymbol{M}_T$ must be generated by the other column of $\boldsymbol{M}_T$ (i.e. there exists a non-zero vector λ such that $\boldsymbol{M}_T\lambda = 0$).

One important result of the paper [Cramer and Xing, 2019] is that, a Monotone Span Program $(R, \boldsymbol{M}, \psi, \mu)$ computing an access structure as defined in the introduction exists if and only if a BBSSS $(R, \boldsymbol{M}, \psi, \mathscr{R})$ computing the same access structure exists. Hence, it exists a direct link between monotone span programs

and Black Box secret sharing schemes. This is important for the rest of the paper [Cramer and Xing, 2019] since their construction relies on Monotone Span Program manipulation.

2.3 Construction of R.Cramer & C.Xing

Introduction

In this part, we will explicit the construction of the matrix of a monotone span program that computes an access structure over $\mathbf{Z}/\mathbf{pZ}$ for any p prime number. The way of thinking here is to know that a monotone span program follows the local-global principle (also called the Hasse principle). This property says that a phenomenon is true globally if and only if it is true locally everywhere. Hence, the idea from [Cramer and Xing, 2019] is to *glue* two monotone span programs that computes such an access structure modulo any prime number $p \leq n$ and another one computing the access structure modulo any prime number $p > n$. In the second sub section, we will shortly explained how [Cramer and Xing, 2019] create the matrix of the first monotone span program. In the third one, we will introduce the concept of linear code which is useful to create secret sharing schemes. Since we will use Shamir's secret sharing schemes, for each prime number p, we will need to create matrices over the field of size p^m (where m depends on n and $m \geq 1$) because of the constraint on the size of the field explicited in the introduction. In the fourth section, we will give details about this point. Finally, we will conclude.

Overview of the process of creation of the first monotone span program

To sum up the idea in [Cramer and Xing, 2019] to compute the first monotone span program, we need first to compute a matrix defined on $\mathbb{Z}^{n\times k}$ that generates a linear code (notion that we will detail further in the next paragraph) related to Shamir's construction [Shamir, 1979] over $\mathbb{F}_{p^m}$ for a single prime number p. Secondly, we will transform the matrix using a mapping to obtain another greater matrix defined on $\mathbb{Z}^{mn\times mk}$ over the field of size p. Thirdly, we will repeat this process for each prime number $p \leq n$ and finally lift all theses matrices to one matrix which is the desired matrix. Theses steps are more detailed in the next paragraphs.

Linear codes

In order to create the following secret sharing schemes and manipulate them, we will need to introduce the notion of linear codes. Linear codes are a common way to build linear secret-sharing schemes. Linear codes of size n, and dimension k over $\mathbb{F}_q$, noted $[n,k]q-ary$ linear code, are k-dimensional subspace of $\mathbb{F}_q$ generated by a matrix $\boldsymbol{M}$. The way of building a secret sharing scheme from a linear code is quite the same as the method for monotone span program. We will not define linear codes deeper since it is not the most important notion and linear codes are related to monotone span programs. Also, we will need to introduce the notion of *minimum distance*. For a code C, the minimum distance can be defined as the minimum number of non-zero entries in the difference of 2 distinct codewords (i.e. $\mathbf{min}_{x,y\neq x\in C} f(x,y)$ where f returns the number of non-zeros entries). This notion will help us to test our codes when we will implement the code creation.

Code in field extension of size p^m and Chinese remainder theorem (CRT)

Assuming that we have the matrix of the linear code of size $n\times k$ over $\mathbb{F}_{p^m}$ (because of the constraint on the size of the field where m is the same as in the introduction). Next, we will need to resize this matrix in the same time we reduce the size of the field to p. Indeed, if we reduce the size of the field, some coefficient of the matrix should not be represented anymore and we will lose informations about the code. The idea is to link linear code over extension field of size p^m to a field of size p. Here are some idea of the properties written in [Cramer and Xing, 2019] that will show the equivalence between a code over $\mathbb{F}_{p^m}$ where, and a code over $\mathbb{F}_p$. The paper shows that for each prime number p there exists a mapping from $\mathbb{F}_{p^m}$ to $\mathbb{F}_p^m$ ϕ such that ϕ map a p^m-ary $[n,k]$ linear code to p-ary $[nm,km]$ linear code. The matrix generating the p-ary $[nm,km]$ ($\boldsymbol{M_1}$) linear code is simply the matrix generating the p^m-ary $[n,k]$ ($\boldsymbol{M_2}$) linear code with ϕ applied to each of the coefficient of the first matrix (i.e. $\boldsymbol{M_2} = \phi(\boldsymbol{M_1})$). After repeating this process for each $p \leq n$, we use the CRT on each coefficients of the matrices, in order to find a matrix that satisfying all constraints on the remainders modulo every prime p (see [Wikipedia contributors, 2023a] for more details on CRT). At the end, we obtain a matrix with values over $\mathbb{Z}/\kappa\mathbb{Z}$, where κ is the product of all prime number less or equal to n. All of this steps will be detailed in the third part of the report with corresponding instructions in SageMath.

Gluing method of [Cramer and Xing, 2019]

It is shown in [Cramer and Xing, 2019] that the second monotone span program can be built from a Vandermonde matrix. Then, after obtaining the matrix that computes the access structure for any prime number $p \leq n$ and the Vandermonde matrix that compute this access structure for each prime $p \geq n$, we can glue these matrices following a specific construction in a way that the new matrix computes a desired monotone span program. As we obtained a matrix of a monotone span program computing a given access structure modulo all prime p, we can build secret sharing scheme from this

matrix

3 Outline of the implementation

In this section, we will explain which steps are needed to build the glued matrix of the monotone span program in [Cramer and Xing, 2019]. We will also detail some of theses steps, and bring explanations about my work. The code is available Here. First thing, we have chosen SageMath [The Sage Developers, 2023] to implement the matrix creation. SageMath is a python-like language that provide specific functions for mathematical computations, numbers manipulation and more. As SageMath is a high-level language and easy to use, it allows us to avoid implementations problems like bugs, memory management and so on. Hence, we can focus on the construction itself and not wasting time on details. SageMath is well suited regarding the remaining time and the difficulties of managing numbers in finite fields.

To perform a complete implementation, we will need to build the glued matrix and to understand how to write encoding and decoding functions. As we said in the previous section, the encoding function only consists in multiplying a message vector $\boldsymbol{g} = (s, g_2, ..., g_n)$ (the same as in the subsection 2.1) by the matrix, where $g_2, ..., g_n$ are uniformly sampled from the desired group. However, the decoding part is not straightforward and we can ask some questions about the process. Theoretically, given $S \in \Gamma$ we need to perform a matrix inversion on the submatrix $\boldsymbol{M_S}$ of $\boldsymbol{M}$ the glued matrix of the monotone span program of support S (i.e. entries of S determine which part of $\boldsymbol{M}$ I should take in my submatrix). However, there is still non-trivial points regarding the decoding function like how to form the submatrix of support S from the glued matrix knowing that the gluing method is not simple. Then, we will try to figure out theses points in the next parts of the report.

3.1 Tasks needed to compute the matrix of the MSP

First, we recall the construction of the matrix in [Cramer and Xing, 2019], then, we will list the important tasks to achieve. The matrix of the linear code described in section 2.3 after lifting with CRT will be noted $\boldsymbol{G}$. We recall that the matrix $\boldsymbol{G}$ has size of $mn \times km$. Then, $\boldsymbol{G}$ can be decomposed as follows:

$$\boldsymbol{G} = \begin{pmatrix} {}^t c_1 & N_1 \\ {}^t c_2 & N_2 \\ \vdots & \vdots \\ {}^t c_n & N_n \end{pmatrix}$$

where the c_i are row vectors of size m and N_i are block matrices of size $(km-1) \times mn$ for $i \in \{1, \dots, n\}$

Now, we can describe what the glued matrix should look like (using notation similar to [Cramer and Xing, 2019]):

$$\boldsymbol{M} = \begin{pmatrix} \delta & \mathbf{0} & e_1 \\ \zeta_N c_1 & N_1 & \mathbf{0} \\ \delta & \mathbf{0} & e_2 \\ \zeta_N c_2 & N_2 & \mathbf{0} \\ \vdots & \vdots & \vdots \\ \delta & \mathbf{0} & e_n \\ \zeta_N c_n & N_n & \mathbf{0} \end{pmatrix}$$

where δ stands for the determinant of the Vandermonde matrix, $e_i = (i^1, i^2, ..., i^{k-1})$, and ζ_N is the following product:

$$\prod_{S \subset [n], |S| = k-1} \Big(\prod_{A \in M_{nt}(N_S), Det(A) \neq 0} Det(A) \Big)$$

where we suppressed all prime factors $\leq n$. Here, $M_{nt}(N_S)$ is the set of the submatrix of size $(k-1)m \times (k-1)m$ of N_S. **In this formula of the report (and only this one), N_S stands for the submatrices N_i grouped where $i \in S$ and N_i are the block matrices defined at the begining of the subsection 3.1.**

To build the glued matrix, we compute all the elements described above. The Vandermonde matrix, decomposed as the e_i vectors, is quite straightforward to implement. We will focus a little bit more on the process needed to compute $\boldsymbol{G}$ and especially the step where we obtain the matrix in the extension field. We need also to compute the product ζ_N but we will not spend a lot of time since it does not involve specific mathematical notions or process.

To instantiate the code, we need to choose a family of linear code that are easy to implement. We will use Reed-Solomon codes to generate theses matrices. Reed Solomon Codes have the properties to be *Maximum distance separable* codes (i.e. the minimum distance is equal to $n - k + 1$). This property will be an interesting test for the implementation part.

3.2 Partial implementation in SageMath

In this subpart, we will explain how to instantiate a Reed Solomon code over a given extension field. To do so, we will take an simple example over $\mathbb{F}_{2^4}$ (i.e. $p = 2$ and $m = 4$). We will detail all the steps needed to obtain a matrix before the lifting with the CRT. Let us also define the dimension and the length of the linear code taking $n = 4$ and $k = 2$.

We express a Reed-Solomon linear code as follow : $RS[n, k] := \{f(\alpha_1), f(\alpha_2), ..., f(\alpha_n)\}$ where

$f \in \mathbb{F}_{p^m}[X]_{\leq k}$ and $\alpha_1, ..., \alpha_n$ are pairwise distinct (see [van Lint, 1999] for more informations on Reed Solomon codes). In our implementation of the code generation, the α_i are chosen at random from $\mathbb{F}_{p^m}^n$ and publicly known, but this is not a mandatory point.

Now, to build the generator matrix of the linear code, we need to choose a basis of $\mathbb{F}_{p^m}[x]_{<k}$ (i.e. the space of polynomials over $\mathbb{F}_{p^m}$). We will keep the process simple, and take the canonical basis : $\{1, x, x^2, ..., x^{k-1}\}$.

In our example, the message $\boldsymbol{g}$ that we need to encode will be the coefficients $(s, g_2, ..., g_k)$ such that the shares will be the n evaluations of the polynomial $\boldsymbol{Q}$ where $\boldsymbol{g}$ is the coordinate vector of $\boldsymbol{Q}$ in the canonical basis (i.e. $\boldsymbol{Q} = s + g_1 x + g_2 x + ... + g_{k-1} x^{k-1}$ and the shares are $\{\boldsymbol{Q}(\alpha_1), \boldsymbol{Q}(\alpha_2), ..., \boldsymbol{Q}(\alpha_n\})$.

To do so, we will need to build a matrix $\boldsymbol{G'}$ such that $g\boldsymbol{G'}$ return the n shares.

Then, the matrix should look like :

$$\boldsymbol{G'} = \begin{pmatrix} \alpha_1^0 & ... & \alpha_n^0 \\ \alpha_1^1 & ... & \alpha_n^1 \\ \vdots & \vdots & \vdots \\ \alpha_1^{k-1} & ... & \alpha_n^{k-1} \end{pmatrix}$$

Recall : $\mathbb{F}_{p^m} \cong \mathbb{F}_p[y]/\langle P \rangle$ where P is an irreducible polynomial, then α_i are polynomials over $\mathbb{F}_{p^m}$

Then, let us take arbitrarily 4 elements $\{\alpha_1 = y^3 + y^2 + y + 1, \alpha_2 = y^3 + y^2 + 1, \alpha_3 = y^3 + y + 1, \alpha_4 = y^3 + 1\}$.

Then our matrix should look like :

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ y^3 + y^2 + y + 1 & y^3 + y^2 + 1 & y^3 + y + 1 & y^3 + 1 \end{pmatrix}$$

A way that can be used to verify correctness of the generator matrix of the linear code is to compute the minimal distance. Reed-Solomon codes are Maximum distance separable code. That means the minimal distance is $= n - k + 1$. Here, we just need to process the Linear Code creation with our matrix as input and compute the minimal distance. See appendix number 3 for the corresponding commands in SageMath.

The program finally returns the desired output 3. Now, since the elements of the matrix are polynomials over $\mathbb{F}_p$ and not only integers, we need to transform the matrix into another greater matrix with values over $\mathbb{F}_p$. The current matrix has dimensions $n \times k$ and the extended matrix has a size of $nm \times km$. We need to map each element α_{ij} of the matrix $\boldsymbol{G'}$ to a new block matrix $\boldsymbol{G}_{i,j}$ and concatenate them over a greater matrix $\boldsymbol{G}$ (where i, j are respectively in $1, ..., n, 1, ..., k$ and each $\boldsymbol{G}_{i,j}$ have size of $m \times m$). In [Cramer and Xing, 2019], the way to obtain this result is to apply the following transformation to each element of the matrix.

First, we need to create a mapping ϕ from $\mathbb{F}_{p^m}$ to $\mathbb{F}_p^m$ which returns the coordinate of a given element in a polynomial basis over the variable y (i.e. ϕ return the coordinates of an element according to a chosen basis of $\mathbb{F}_p[y]_{\leq m}$). To keep things simple, let us take the canonical basis $\{1, y, y^2, ..., y^{m-1}\}$. Thus, $\phi(y^3 + y^2 + y + 1)$ will return the vector $(1, 1, 1, 1)$. In SageMath, a *vector()* function is available and returns the coordinates of an element in an extension field according to the canonical basis. See the appendix number two for the corresponding function in SageMath. For each element $a \in \mathbb{F}_{p^m}$ of the current matrix, we will build the following 4×4 block matrix :

$$\begin{pmatrix} \phi(1 \times a) \\ \phi(y \times a) \\ \phi(y^2 \times a) \\ \phi(y^3 \times a) \end{pmatrix}$$

This matrix is in fact nothing more than *a companion matrix* or *a multiplication matrix* of the element a. This notion of multiplication matrix steps in because the multiplication of one element and a constant over is a linear operation. Since an element from a extension field can be written as a vector over the sub-field, a multiplication to a constant can be expressed as a matrix multiplication. Hence, such a multiplication is a vector-matrix product. Then, if we want to multiply one element a to a constant b, we need to get a multiplication matrix of the element b and the coordinate vector of a (here we will use the following notation : $\boldsymbol{a}$ is the coordinate vector of a and $\boldsymbol{B}$ the multiplication matrix of the constant b):

$$\boldsymbol{a} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{pmatrix} \boldsymbol{B} = \begin{pmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,m} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m,1} & b_{m,2} & \cdots & b_{m,m} \end{pmatrix}$$

such that $\boldsymbol{a} \times \boldsymbol{B} = \boldsymbol{c}$ where $\boldsymbol{c}$ is the vector of the element obtained by the simple multiplication of $a \times b$ in $\mathbb{F}_{p^m}$. If we can write the element $b = \gamma_0 + \gamma_1 y + \gamma_2 y^2 + ... + \gamma_{m-1} y^{m-1}$, then the desired matrix can be obtained by the sum $\sum_{i=0}^{m-1} \gamma_i \times M_{y^i}$ where M_{y^i} is the multiplication matrix of the monomial y^i. In sageMath, a *matrix()* operation is available to return the block multiplication matrix of an element in a finite field. Then, we only need to build the desired matrix concatenating corresponding block matrices. See Appendix number 1 for the matrix given by the trace code.

If we reprocess the same command as above to compute the minimal distance, we get a minimal distance of 4. This result is consistent with the properties of [Cramer

and Xing, 2019]) which says that the minimum distance of a extended matrix over $\mathbb{F}_p$ has a minimum distance greater or equal to the minimum distance of the linear code over $\mathbb{F}_p^m$.

3.3 Structure of the shared vector

In this subsection, we will detail a little bit more the encoding algorithm and the structure of the vector of shares. Let us assume the matrix M described in the subsection 3.1 and $\boldsymbol{g} = (s, g_2, g_3, ..., g_{km+k-1})$, the vector composed of the secret and uniformly sampled elements. The vector **s** obtained after the multiplication is composed of $(m+1) \times n$ entries.

$$\boldsymbol{s} = \left(\begin{array}{c} s \times \delta + \sum_{i=1}^{k-1}(1^i) \times g_{l-k+1+i} \\ s \times \zeta_N \times c_{11} + \sum_{i=1}^{m\times k-1} N_{1_{1,i}} \times g_{i+1} \\ s \times \zeta_N \times c_{12} + \sum_{i=1}^{m\times k-1} N_{1_{2,i}} \times g_{i+1} \\ \vdots \\ s \times \zeta_N \times c_{1m} + \sum_{i=1}^{m\times k-1} N_{1_{m,i}} \times g_{i+1} \\ \hline s \times \delta + \sum_{i=1}^{k-1}(2^i) \times g_{l-k+1+i} \\ s \times \zeta_N \times c_{21} + \sum_{i=1}^{m\times k-1} N_{2_{1,i}} \times g_{i+1} \\ s \times \zeta_N \times c_{22} + \sum_{i=1}^{m\times k-1} N_{2_{2,i}} \times g_{i+1} \\ \vdots \\ s \times \zeta_N \times c_{2m} + \sum_{i=1}^{m\times k-1} N_{2_{m,i}} \times g_{i+1} \\ \hline \\ \vdots \\ \hline \\ s \times \delta + \sum_{i=1}^{k-1}(n^i) \times g_{l-k+1+i} \\ s \times \zeta_N \times c_{n1} + \sum_{i=1}^{m\times k-1} N_{n_{1,i}} \times g_{i+1} \\ s \times \zeta_N \times c_{n2} + \sum_{i=1}^{m\times k-1} N_{n_{2,i}} \times g_{i+1} \\ \vdots \\ s \times \zeta_N \times c_{nm} + \sum_{i=1}^{m\times k-1} N_{n_{m,i}} \times g_{i+1} \end{array}\right)$$

Since the secret sharing scheme should return n shares, each tuple of $m+1$ elements will stand for a share. In each share, the first element is the component that use the glued Vandermonde matrix. All other elements involve the lifted matrix G by blocks. Now, we know the skeleton of the vector of shares. The remaining work is to implement the manipulation of shares in the encoding and decoding functions.

3.4 Limitations

One of the limitation in the construction of the matrix is the factor ζ_N which is much large. Hence processing computations on ζ_N is a costly operation. When we instantiate the glued Matrix with the following parameters : $n = 6, k = 3, m = 3$, we obtained $\zeta_N \geq 10^{1800}$. In this subsection, we will try to give an upper bound of this factor to characterize it a little bit more.

The idea is to upper bound the value of the determinant of a matrix of size $m(k-1) \times m(k-1)$ using the *Hadamard's inequality* in [Wikipedia contributors, 2023b]. In our case, each entries of the matrix are bounded by κ (where κ is the product of all prime numbers less or equal to n). Hence, we can write : $|Det(\boldsymbol{A})| \leq (\sqrt{m(k-1)}\kappa)^{m(k-1)}$

If we retake the expression of ζ_N, the first product has $\binom{n}{k-1}$ elements. The second product is a little bit more tricky, we need to compute an upper bound of the number of submatrix of size $m(k-1) \times m(k-1)$ in a matrix of size $m(k-1) \times mk-1$. The number of elements in the second product is $\binom{mk-1}{m(k-1)}$.

At the end, our bound is greater of equals to

$$(((\sqrt{m(k-1)}\kappa)^{m(k-1)})^{\binom{mk-1}{m(k-1)}})^{\binom{n}{k-1}}$$

We finally need to decompose this expression in prime factors and only keep prime factors greater to n to obtain ζ_N.

Back to our instantiation attempt, our upper bound is greater than 10^{4703} taking the same parameters as in our example. In our implementation, the generation of the glued matrix is pseudo-randomly generated (i.e. α_i where $i \in 1, ..., n$ are randomly sampled with the *random_element* function from SageMath). We can test the program multiple times to compare ζ_N to our upper bound. After tens of test, ζ_N does not change much and stay close to 10^{1800}. As we can see, our upper bound is not tight.

4 Conclusion and future work

The scheme provided in [Cramer and Xing, 2019] has an interesting limit on the expansion factor. However, we need to pay the price of a small lower bound by big coefficients in the matrix of the MSP. One can ask the following question, can we do better ? It is important to note that the big coefficients are probably not optimized and make the creation of the MSP slower. However, it should not make the encoding and decoding functions slower because all the entries of the matrix should be reduced modulo the order of the group where the scheme works. Hence, the coefficients of the shares obtained in the subsection 3.3 should also be reduced (especially the coefficients with ζ_N). As the result, the high coefficients are only involved in the creation of the MSP matrix.

In this paper, only partials results are described and should be studied deeper. We will continue the internship for two more weeks until the end of June. There are still multiple tasks that are not finished as the implementation of the encoding and decoding functions. In addition, we will continue to look for limitations and try to measure their impacts on the computations.

References

[Cramer and Xing, 2019] Ronald Cramer and Chaoping Xing. Blackbox secret sharing revisited: A coding-theoretic approach with application to expansionless near-threshold schemes. Cryptology ePrint Archive, Paper 2019/1134, 2019. https://eprint.iacr.org/2019/1134.

[Ito *et al.*, 1989] Mitsuru Ito, Akira Saito, and Takao Nishizeki. Secret sharing scheme realizing general access structure. *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)*, 72(9):56–64, 1989.

[Shamir, 1979] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, nov 1979.

[The Sage Developers, 2023] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 9.5)*, 2023. https://www.sagemath.org.

[van Lint, 1999] J. H. van Lint. *Linear Codes*, pages 33–46. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[Wikipedia contributors, 2023a] Wikipedia contributors. Chinese remainder theorem — Wikipedia, the free encyclopedia, 2023. [Online; accessed 7-June-2023].

[Wikipedia contributors, 2023b] Wikipedia contributors. Hadamard's inequality — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Hadamard%27s_inequality&oldid=1156416002, 2023. [Online; accessed 7-June-2023].

5 Appendix

Appendix number 1, Complement of the section 3.2
The final matrix displayed by the program is :

```
Matrix after extension in Fp of size
km * nm :
[1 0 0 0 1 1 1 1]
[0 1 0 0 1 0 0 0]
[0 0 1 0 1 1 0 0]
[0 0 0 1 1 1 1 0]
[1 0 0 0 1 1 1 0]
[0 1 0 0 0 0 0 1]
[0 0 1 0 1 0 0 0]
[0 0 0 1 1 1 0 0]
[1 0 0 0 1 1 0 1]
[0 1 0 0 1 0 1 1]
[0 0 1 0 0 1 0 1]
[0 0 0 1 1 0 1 0]
[1 0 0 0 1 1 0 0]
[0 1 0 0 0 0 1 0]
[0 0 1 0 0 0 0 1]
[0 0 0 1 1 0 0 0]
```

Appendix number 2, complement of the section 3.2

```
sage: a = GF(16).random_element()

(taking a random element
from the finite field of size 16)

sage: a

>>> z4^2 + 1

sage: vector(a)

(printing the coordinate
of a according to the canonical
basis over the variable z4)

>>> (1, 0, 1, 0)
```

Appendix number 3, complement of the section 3.2

```
    C = LinearCode(our_matrix)
    d = C.minimal_distance()
    print(d)
```