b
UNIVERSITÄT
BERN

Erasure Codes on Distributed Protocols

Implementing Monotone Erasure Codes with Access Structures

Bachelor Thesis

Felix Leichtle

from

Bern, Switzerland

Faculty of Science, University of Bern

22. April 2026

Prof. Christian Cachin
Annalisa Cimatti
Cryptology and Data Security Group
Institute of Computer Science
University of Bern, Switzerland

Abstract

This thesis concerns *monotone erasure codes*, which allow recovery according to an *access structure*, enabling more fine-grained control over which combinations of nodes can reconstruct the data, as compared to traditional threshold based erasure codes. A general solution based on linear programming is presented, but its reliance on floating-point arithmetic and potential exponential complexity make it impractical for many cases. To overcome this, we focus on *partitioned threshold access structures*. For this class of access structures a greedy optimization algorithm, that computes an optimal fragment assignment, while minimizing the *overhead*, is presented and implemented in Go. Validation on certain access structures extracted from the Stellar blockchain shows an average overhead of 1.06.

Contents

Chapter 1

Introduction

Erasure codes allow for efficient storage of data while providing integrity even if some data is lost. Distributed storage systems must protect against data loss. Storing entire copies of the data multiple times is excellent to protect against data loss, as only one such copy must survive for the data to remain accessible. It however also incurs a large storage *overhead*, meaning that a lot of storage is used by duplicated data. For practical solutions, a middle ground between having enough redundancy in the system to ensure data availability and reducing the storage requirements must be found. *Erasure codes* make this middle ground possible. They reduce the overhead, but introduce significant computational penalties.

With the amount of stored data in the world exploding in volume, erasure codes have become more prevalent. One of the most used erasure codes was invented by Reed and Solomon in their seminal work [9]. This code is called Reed-Solomon code and it found its foothold in many applications that require data to be stored or transmitted, and where it is important that the data is not lost. In 1977, Reed-Solomon was used to transmit images from the Voyager space mission back to earth, and later it was widely used in compact discs, among many other applications. Nowadays, almost all cloud storage providers employ some erasure code, and, more recently, erasure coding has become a crucial part of cryptographic networks. Ethereum data availability sampling, for example, makes use of erasure codes to guarantee that data stays available even when some nodes are offline [2].

Classical erasure codes are largely threshold based. For example, consider a company having three data centers, each storing some part of the data. The erasure code they are running ensures that if at least two of the three data centers are online, all the data is accessible. This kind of erasure code assumes that all data centers are equally reliable and equally accessible. If it were known that one of the three data centers is much more reliable than the other two, it would be helpful to have an erasure code that accounts for this. Indeed, we would like to ensure that the data stays accessible if either this dependable data center is online, or both of the other two data centers are online.

This thesis reviews and implements monotone erasure codes, a novel kind of erasure codes, that provides exactly this kind of control. Unlike traditional threshold-based codes, this code supports recovery according to an access structure. In the example above, this allows the company to specify precisely which data centers must be online to recover data. For instance, it could require either the dependable data center alone, or both of the less reliable data centers together. Thus, the code here generalizes threshold erasure codes while preserving the core advantages of low storage overhead and efficient recovery.

The remainder of this thesis is structured as follows. Chapter 2 provides the necessary background on erasure codes. Chapter 3 introduces a first implementation of a monotone erasure code, i.e., an erasure code that allows for recovery according to a given access structure. Chapter 4 outlines the design of a monotone erasure code that operates on a more specific access structure, which allows for more efficient computation, and Chapter 5 outlines its implementation. In Chapter 6, the code is tested, and Chapter 7 concludes the thesis.

The erasure codes implemented as part of this thesis, and the definitions they are reliant on, were

developed by Bammert et al. [1].

Chapter 2

Background

The problem of distributed storage concerns the preservation of information across multiple potentially unreliable nodes in a set $\mathcal{P}$.

Erasure codes provide redundancy by dividing a file into fragments and distributing them across n nodes, while ensuring that the original data can be recovered from a sufficiently large subset of nodes. In most cases it is sufficient to use a threshold erasure code where a threshold of k out of n nodes need to be available to reconstruct the data [7]. This works well under the assumption that all nodes are equally trusted, equally reliable and equally accessible. One could instead organize nodes into hierarchical clusters, geographical regions or trust zones among many other options. For this purpose we use a generalized access structure $\mathcal{A}$, a collection of subsets of $\mathcal{P}$. These subsets are called access sets, and each access set is explicitly capable of reconstructing the data.

In this work, we provide the implementation of an algorithm that allows us to store large amounts of information according to an access structure $\mathcal{A}$ expressed as a monotone boolean formula.

This chapter will familiarize the reader with the necessary definitions and concepts that are needed to understand our proposed solution.

2.1 Reed-Solomon Codes

Reed-Solomon erasure codes [9] play an important role in most distributed storage systems [12]. The algorithms treated in this thesis are no exception, they use Reed-Solomon as an important building block.

Reed and Solomon leveraged the fact that data can be represented as coefficients of a polynomial, and then from that polynomial an arbitrary number of points can be generated. Only k of these points are needed to reconstruct a polynomial of degree $k - 1$. This allows for the design of *threshold erasure codes*.

Definition 1. (Threshold Erasure Code) Given a file f and parameters k, m with $k \leq m$, a $[k, m]$-*threshold erasure code* provides the following two functions:

- Encode(f): given the file f as input, returns fragments $(g_1, \ldots, g_m)$;
- Decode(u): given k out of m fragments returns f or $\perp$ in the case that less than k fragments are given as input.

In Figure 2.1 we can see how this is done. Here, we are given some data and we encode it into symbols that are elements of the finite field we are working with. In practice, the finite field would be quite large, but in this example we use a finite field with 5 elements. The polynomial is of degree 2, so we must generate at least 3 such symbols to make reconstruction possible. Up to 2 more symbols can be generated in this small finite field. To reconstruct the data, at least 3 points on the polynomial need to be known. In the case that only 2 symbols are known, like in Figure 2.1, the data cannot be reconstructed.

It is important to note here that using erasure codes does not guarantee secrecy. Even if there are not enough nodes to reconstruct the entirety of the data, some of it can still leak. In the case that secrecy is

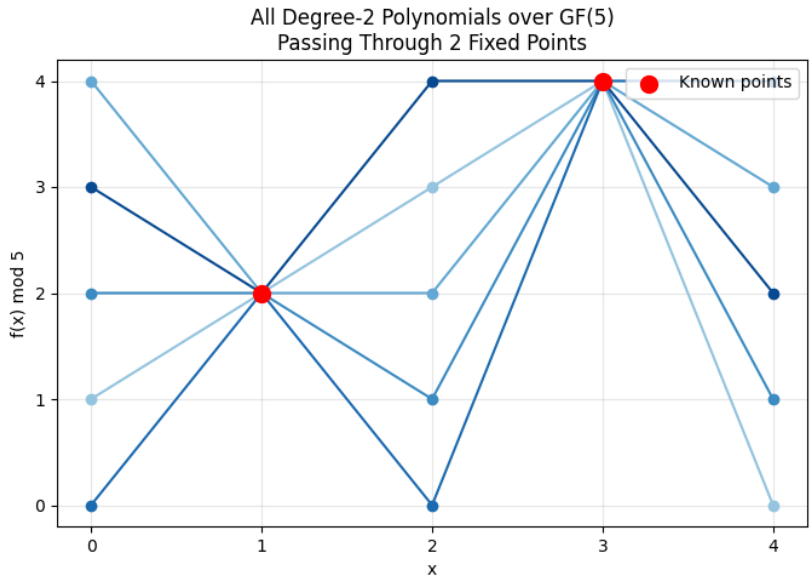


Figure 2.1

needed, one should consider using other methods, such as secret sharing [11]. This thesis however, is not concerned with secrecy.

2.2 Preliminary Definitions

We now introduce the concept of an *access structure*. An access structure defines which parties of a distributed storage environment can access or reconstruct the data that is being stored. This allows for much more control than traditional threshold erasure codes. Instead of just specifying a threshold of parties that are needed to reconstruct the data, an access structure defines exactly which parties can, by themselves reconstruct the data.

Definition 2. (Access Structure) An access structure $\mathcal{A}$ is a collection of subsets of $\mathcal{P}$, where $\mathcal{P}$ is a set of nodes. Each set $A \in \mathcal{A}$ is called an *access set*.

Access sets allow us to specify exactly which parties have *access* to the data we are encoding. In our setting each access set in the access structure can decode the data. If a set of nodes does not correspond to an access set then there is no guarantee that the data can be reconstructed.

In most applications, a *monotone access structure* is needed:

Definition 3. (Monotone Access Structure) An access structure $\mathcal{A}$ is *monotone* if for all $A \in \mathcal{A}$ and for all $A' \subseteq \mathcal{P}$ such that $A \subseteq A'$, it holds that $A' \in \mathcal{A}$.

In a monotone access structure, whenever some nodes form an access set and can therefore reconstruct the data, adding more nodes to the set means that they can always still reconstruct the data.

When specifying a monotone access structure, it can be cumbersome to specify all access sets. For this reason, we introduce the concept of a *minimal access structure*. Notice that if an access structure $\mathcal{A}$ is monotone, then it is uniquely determined by its minimal sets. In this case, minimal means that there is no set in $\mathcal{A}^-$ that is a proper subset of any element of $\mathcal{A}^-$.

Definition 4. (Minimal Access Structure) A *minimal access structure* $\mathcal{A}^-$ is the collection of all minimal access sets in $\mathcal{A}$.

Example 1. Let $\mathcal{P} = \{A, B, C, D\}$ and $\mathcal{A}$ ={{A,B,C,D},{A,B,C},{A,B}} then the corresponding minimal access structure is $\mathcal{A}^-$ ={{A,B}}.

Chapter 3

Monotone Erasure Codes

Building on the concepts introduced in Chapter 2 we present a solution to the problem of allocating data while respecting a given monotone access structure:

Definition 5. (Monotone Erasure Code) Let $\mathcal{A}$ be a monotone access structure on $\mathcal{P}$ and let f be a file from a set $\mathcal{F}$. A *monotone erasure code* C for $\mathcal{A}$ is a scheme that distributes f among the nodes in $\mathcal{P}$ and assigns to each $P_i \in \mathcal{P}$ a piece of information $g_i \in \mathcal{G}$ called fragment or shard of P_i. Such a code realizes two functions:

- Encode(f): given a file f, returns the fragment vector $g = (g_1, ..., g_n) \in \mathcal{G}$
- Decode(u): given a vector $u = (u_1, \ldots, u_n)$, where each $u_i \in \mathcal{G} \cup \{\perp\}$, returns either a file $f' \in \mathcal{F}$ or $\perp$ if the set of available fragments $\{u_i | i \in [n], u_i \neq \perp\}$, is not contained in any access set of $\mathcal{A}$.

For our purposes $[n]$ is defined as as the set of all natural numbers, excluding 0 up to and including n, i.e., $[n] = \{1, 2, \ldots, n\}$.

An important case of a monotone erasure code is the linear one. Linearity imposes an algebraic structure especially suited for efficient computation. The encoding operation for example becomes a simple and efficient matrix multiplication.

Definition 6. (Linear Monotone Erasure Code) Let $k, m \in \mathbb{N}$ be such that $k < m$. An $[m, k]$-*linear monotone erasure code* C for $\mathcal{A}$ is an $[m, k]$-monotone erasure code for $\mathcal{A}$, where the file is a vector $f \in \mathbb{F}_q^k$ and a fragment g_i consists of a vector in $\mathbb{F}_q^{m_i}$ or $g_i = \perp$, i.e., $\mathcal{G}_i = \mathbb{F}_q^{m_i} \cup \{\perp\}$. The encoding and decoding algorithms are based on a full-rank matrix $G \in \mathbb{F}_q^{k \times m}$ and a *labeling function*

$$\phi : \{1, \ldots, m\} \to \mathcal{P}$$

that assigns each column of G to a node in $\mathcal{P}$.

Let G_{P_i} be the matrix consisting of the columns of G assigned to P_i by ϕ, i.e.:

$$G_{P_i} = \left(G_{j_1} \mid \ldots \mid G_{j_{m_i}} \right)$$

where $\phi(j_1) = \ldots = \phi\left(j_{m_i}\right) = P_i$.

Then the algorithms are defined as follows:

- Encode(f): given $f \in \mathbb{F}_q^k$, returns the fragment vector $g = (g_1, ..., g_n) \in \mathcal{G}$, where $g_i \in \mathbb{F}_q^{m_i}$ such that

  $$g_i = f \cdot G_{P_i}$$

  is the fragment of $P_i \in \mathcal{P}$.
- Decode(u): given a vector $u = (u_i)_{i \in [n]} \in \mathcal{G}$, the algorithm returns either a file $f' \in \mathcal{F}_q^k$ or $\perp$.

The matrix G is also referred to as the *generator matrix* of C.

To measure the efficiency of the *monotone erasure code* the *overhead* is used.

Definition 7. (Overhead) Let C be a monotone erasure code that stores files of k bits and produces fragments of μ total bits. The *overhead* β of C is

$$\beta = \frac{\mu - k}{k}.$$

For example, an overhead of 0.5 means that the monotone erasure code produces fragments consisting of a total length of 1.5 times the length of the original file.

3.1 A first solution

We will now propose an implementation of a monotone erasure code. As mentioned previously, our monotone erasure code will make use of a Reed-Solomon erasure code, which will be referred to as the *base-code* that constructs base fragments. These base fragments are then assigned to the nodes in such a way that the specified access structure is respected. The choice of the threshold for the base-code and the number of fragments generated by it are set in a way that guarantees that the access structure is respected and minimizes the overhead.

The procedure is split up into 3 parts: setting the parameters, encoding and decoding.

3.1.1 Setting the Parameters

First we must find the optimal parameters m and k for our base-code. Where by optimal we mean, that they minimize the overhead. To do this we solve the following linear programming problem: Find $y = (y_1, ..., y_n) \in \mathbb{Q}^n$, such that

$$\min \sum_{i=1}^{n} y_i \tag{3.1}$$

$$\text{subject to } \Gamma \cdot y^\top \geq 1_r^\top \tag{3.2}$$

$$y_i \geq 0 \tag{3.3}$$

Where Γ is is a binary $r \times n$ matrix such that $\Gamma_{ij} = 1$ if and only if $P_j \in A_i$ and $\Gamma_{ij} = 0$ otherwise, and $1_r = (1, ..., 1) \in \mathbb{N}^r$. Note that Γ represents the access structure. Each row of the matrix corresponds to an access set in the access structure.
Given such a solution y, we take k to be the least common multiple of the denominators of all the y_i's. We define $m_i = y_i \cdot k$ for $i \in [n]$. We then compute m as the sum of all the m_i.
Minimizing $\sum_{i\in[n]} y_i$ corresponds to minimizing the overhead $\beta = \frac{\left(\sum_{i\in[n]} y_i \cdot k\right) - k}{k}$.

Additionally the solution to the linear programming problem guarantees that $\Gamma \cdot y^\top \geq 1_r^\top$. We have defined $m_i = y_i \cdot k$ for $i \in [n]$, so $y_i = \frac{m_i}{k}$ for $i \in [n]$ and $y = \frac{1}{k} \cdot (m_1, ..., m_n)$. This gives us $\Gamma \cdot (m_1, \ldots, m_n)^\top \geq (k, \ldots, k)$, meaning that for all access sets $A \in \mathcal{A}$ the nodes in A collectively hold at least k fragments, which suffice to reconstruct the data.

Example 2. Let $\mathcal{P} = \{A, B, C, D\}$ and $\mathcal{A}^- = \{\{A, B\}\}$. The corresponding monotone access structure is $\mathcal{A} = \{\{A, B\}, \{A, B, C\}, \{A, B, C, D\}\}$, which is represented by the following matrix:

$$\Gamma = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} \tag{3.4}$$

Now the linear programming problem yields some $y = (y_A, y_B, y_C, y_D)$. From this we calculate k as the least common multiple of the denominators of all the y_i's. And finally we calculate the fragment assignment $m = (m_A, m_B, m_C, m_D)$ with $m_i = y_i \cdot k$. The exact values aren't important here as we can show that they satisfy the core prerequisite, namely that any access set holds at least k fragments, without the numerical values:

$$\Gamma \cdot m = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} m_A \\ m_B \\ m_C \\ m_D \end{pmatrix} \tag{3.5}$$

$$= \begin{pmatrix} m_A + m_B \\ m_A + m_B + m_C \\ m_A + m_B + m_C + m_D \end{pmatrix} \tag{3.6}$$

$$= \begin{pmatrix} k \cdot (y_A + y_B) \\ k \cdot (y_A + y_B + y_C) \\ k \cdot (y_A + y_B + y_C + y_D) \end{pmatrix} \tag{3.7}$$

We also know that $\Gamma \cdot y^\top \geq 1_r^\top$, as it was a constraint of the linear programming problem. Meaning that:

$$\Gamma \cdot y^\top = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} y_A \\ y_B \\ y_C \\ y_D \end{pmatrix} \tag{3.8}$$

$$= \begin{pmatrix} y_A + y_B \\ y_A + y_B + y_C \\ y_A + y_B + y_C + y_D \end{pmatrix} \geq \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \tag{3.9}$$

With (3.7) and (3.9) we can see that each access set must indeed hold at lest k fragments.

Indeed in this example numeric evaluation yields the optimal result $y = (1, 0, 0, 0)$ and $k = 1$, since A is an element of each access set. $y = (0, 1, 0, 0)$ would be equivalent.

3.1.2 Encoding

We now explain how to encode a file f, given the set of nodes $\mathcal{P}$ along with their number of assigned fragments $m_1 \dots m_{|\mathcal{P}|}$. A $[k, m]$-base-code is used to generate the m fragments, where $m = \sum_{i=0}^{|\mathcal{P}|} m_i$. Then the base-fragments are distributed among the nodes, with each $P_i \in P$ receiving m_i fragments.

3.1.3 Decoding

When decoding the data, it is important to know which fragments are available for reconstruction and which are not. In light of this, a labeling function is used. It maps the set of nodes to the set of fragments. With it we can infer which fragments are available from the available nodes. Additionally both when encoding and decoding, the length of the encoded file is stored. Since all fragments must be of the same size, but the number of fragments that must be generated may not cleanly divide the length of the file, the file at times must be padded. Storing the length of the file ensures that after decoding this padding can be removed again.

3.2 Shortcomings of this Approach

The implementation outlined above has a few issues, with the main ones originating in the linear programming problem.

Usual linear programming solvers generate floating point solutions. In contrast we need the solution to be rational. Rational numbers however cannot always be represented in floating point data types. For example, let us consider a linear programming problem where the true solution for one element of y is $y_i = \frac{1}{3}$. This is not truly representable as a floating point. Our solver might arrive at a solution of $y_i = 0.333$. Now we run into trouble trying to extract the denominator of this. A naive approach would assume $y_i = \frac{333}{1000}$, which in turn would make the least common multiple of all the y_i's much larger than needed, resulting in a gigantic overhead and the generation of a very large number of base fragments. This issue can be avoided by implementing a linear programming solver that operates on the rational numbers explicitly, making the extraction of the denominators trivial.

This issue was not further explored because the construction presented above is impractical in the general case. Indeed depending on the access structure and number of nodes, solving the linear programming problem can require exponential time, making the algorithm inefficient.

We will now introduce a new type of access structure for which a much more efficient and elegant solution exists.

Chapter 4

Design

4.1 Partitioned Threshold Access Structure

We now want to define a particular kind of access structure, called *partitioned threshold access structure*. To do this we must first introduce the notion of a monotone boolean formula.

Definition 8. (Monotone Boolean Formula) An MBF Γ is a formula composed of AND, OR, and threshold operators, along with literals. It describes a monotone function $\varphi : 2^{\mathcal{P}} \to \{0,1\}$ in the following way:

- when Γ consists of only a literal, then the value of Γ on input $S \subseteq P$ is 1 if and only if $\Gamma \in S$.
- when Γ is a threshold operator Θ_k^m, then $\Gamma(S) = 1$ if at least k of the $q_1, \ldots, q_m$ are recursively evaluated to 1 on input S.

In our case, each literal corresponds to a node P_i. The threshold operator $\Theta_k^m(q_1, \ldots, q_m)$ evaluates to 1 whenever at least k out of the m functions $q_1, \ldots, q_m$ evaluate to 1, where each q_i can be either a literal or an operator.

We can make use of the fact that a monotone access structure $\mathcal{A}$ can be described using a monotone function $\alpha : 2^{\mathcal{P}} \to \{0,1\}$ that returns 1 when evaluated on an access set and 0 otherwise. We can therefore represent the monotone access structure with a monotone boolean formula that describes α. We refer to the rooted tree representation of such an MBF as the *access tree* of $\mathcal{A}$.

Definition 9. Partitioned Threshold access structure A *(two-level) partitioned threshold access structure* is an access structure $\mathcal{A}$ on $\mathcal{P} = \bigcup_{i \in [r]} B_i$ with the following minimal access structure:

$$\mathcal{A}^- = \Theta_t^r \left(\Theta_{z_1}^{b_1}(B_1), \Theta_{z_2}^{b_2}(B_2), \ldots, \Theta_{z_r}^{b_r}(B_r) \right)$$

where $B_i \cap B_j = \emptyset$ for $i \neq j \in [r]$.

Example 3. Let us consider a minimal access structure

$$\mathcal{A}^- = \Theta_2^3 \left(\Theta_1^2(B_1), \Theta_1^1(B_2), \Theta_1^1(B_3) \right)$$

on $\mathcal{P} = \bigcup_{i \in [3]} B_i = \{A, B\} \cup \{C\}, \{D\}$. It is clear that this is a partitioned threshold access structure as all the B_i are disjoint. We can also easily derive the corresponding minimal access structure written out as a set containing all minimal access sets:

$$\mathcal{A}^- = \{\{A, C\}, \{A, D\}\{B, C\}\{B, D\}\{C, D\}\}$$

$\mathcal{A}^-$ can also be represented with a tree, as is shown in Figure 4.1, which makes it easy to see that it is indeed a two level partitioned access structure.

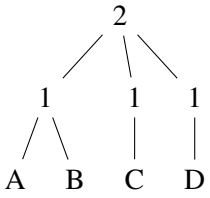


Figure 4.1. Partitioned threshold access structure for Example 3.

It is of course possible to generalize this definition to any number of levels. We generally use the term level, to represent the depth of the tree. When referring to a specific level of the tree we are addressing all nodes or threshold operators at the depth of that level.

For the purposes of this thesis we will always assume that any general partitioned access structure can be represented as a balanced tree. Balanced means that for all nodes, the length of the path from the root to the node is of the same length.

Figure 4.2. Example of how a tree can be balanced.

If this is not the case and there are nodes that are on the same level as threshold operators, we can insert Θ_1^1 threshold operators until the tree is balanced as can bee seen in Figure 4.2.

4.2 Setting the Parameters for Partitioned Threshold Access Structures

With the partitioned threshold access structure being a special case of an access structure the method proposed in 3.1 can be applied. The new access structure allows us to greatly increase the speed of the first phase, namely setting the parameters, while encoding and decoding stay the same.

To ensure that the children of the root collectively hold enough fragments, we assign each of them $\frac{k}{t}$ fragments. That way t of them together have k fragments to reconstruct the data. Their children in turn must each hold $\frac{1}{z} \cdot \frac{k}{t}$ fragments, where z is their number of children. In general, consider a path from the root to a node P_i. Let this path pass through internal thresholds $\Theta^{(l)}$ for $l \in [\omega_i]$, where ω_i is the depth of the parent threshold of node P_i. Each threshold $\Theta^{(l)}$ has its corresponding threshold specification $[b^{(l)}, z^{(l)}]$. Then P_i is assigned $\frac{k}{t \cdot \prod_{l=1}^{\omega_i} z^{(l)}}$ fragments.

This ensures that the access structure is respected but it is not yet optimal, as the overhead has not yet been minimized. To find the optimal k for this fragment assignment, the following approach is used:

Let us define the *cost* of a threshold. A Θ_2^3 threshold must hold 1.5 times the data it can restore. The cost of a node is at first one, while the cost of a threshold Θ_x^y is the average cost of its children multiplied by $\frac{y}{x}$.

Once the cost of all children are known, we check wether it is possible to improve efficiency by removing one of the children with the following criterion: If the cost of a child is large enough, it is more efficient to remove it. To evaluate this criterion the cost of the parent is evaluated with the child removed. The cost of a child is large enough to warrant removal, if it exceeds the previously evaluated cost of the parent.

More formally for a threshold Θ_x^y, let $\Theta^{(i)}$, $i \in [y]$ be the i-th child of that threshold, if

$$\text{cost}\left(\Theta^{(i)}\right) \geq \frac{1}{x-1} \cdot \sum_{j \in [y] \setminus i} \text{cost}\left(\Theta^{(j)}\right) \tag{4.1}$$

we remove that child. Specifically we start by removing the children with the highest cost, while reducing the parent's x and y values by 1 each time we remove a child. If a child cannot be removed then none of the children with a lower cost can be removed, so there is no need to check them.

With this method we can traverse up the tree of our threshold partitioned access structure until we have evaluated the cost of the root threshold. While traversing the tree we also update the cost of the nodes. In the end the cost of a node amounts to:

$$\text{cost (node)} = \prod_{\Theta \in B} \frac{1}{x_\Theta} \tag{4.2}$$

where B is the set of thresholds included in the path from the root to the node, and x_Θ is the x-value of the Θ_x^y threshold.

We then calculate the least common multiple of the denominators of the costs of each node to obtain our k. The number of fragments m_i assigned to the node is its cost multiplied with the obtained k, i.e., $m_i = \text{cost (node } i) \cdot k$.

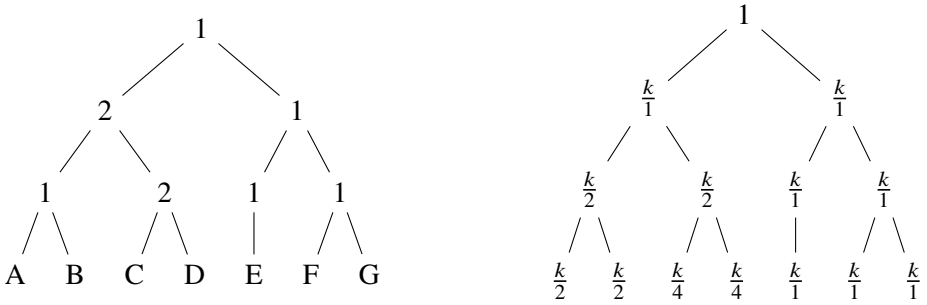


Figure 4.3. Access Tree on the left with the corresponding costs of the thresholds and nodes on the right.

Example 4. See Figure 4.3 for an example of evaluating the cost of nodes and thresholds on an access tree.

Example 5. Consider the following 2-level partitioned threshold access structure

$$\mathcal{A} = \Theta_2^3\left(\Theta_1^1(A), \Theta_1^1(B), \Theta_1^3(C, D, E)\right).$$

Its corresponding access tree can be seen in Figure 4.4.

In the setup phase of the algorithm, the costs of nodes and threshold operators are evaluated and stored in their corresponding maps. The costs of the nodes and threshold operators are displayed In Figure 4.5.

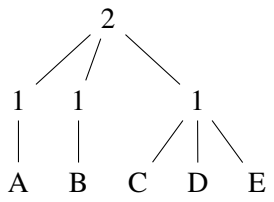


Figure 4.4. Access tree corresponding to access structure in Example 5.

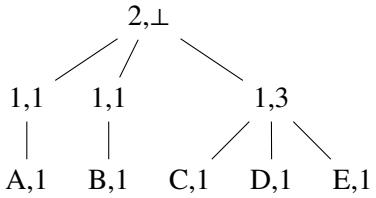


Figure 4.5. Access tree corresponding to access structure in example 5, with the cost of nodes and thresholds specified (node/threshold,cost).

Notice that of the thresholds below the root $(\Theta_1, \Theta_2, \Theta_3)$ the cost of the Θ_3 threshold (corresponding to $\Theta_1^3(C, D, E)$), is much higher than the others on the same level. So, in the second phase, the algorithm checks it first to see if it warrants removal. Indeed, it checks that the following holds:

$$\text{cost}(\Theta_3) \geq \frac{1}{x-1} \cdot \sum_{j \in [3] \setminus 3} \text{cost}\left(\Theta_j\right) \tag{4.3}$$

$$3 \geq \frac{1}{2-1}(1+1) \tag{4.4}$$

when considering the elimination rule (4.1).

After the removal of a threshold operator, the values of the root threshold operator change from $(2, 3)$ to $(1, 2)$. With this new value $x = 1$, the cost of the nodes is updated according to (4.2), yielding $\text{cost}(A) = \text{cost}(B) = 1$. From this, k is computed as the lcm of the denominators, resulting in $k = 1$. Both nodes A and B are assigned $m = k \cdot \text{cost}(A) = 1$ fragment each, while C, D and E receive no fragments.

Example 6. Consider the following general partitioned threshold access structure

$$\begin{aligned}\mathcal{A} = \Theta_2^3\Big[& \\ & \Theta_3^4\Big[\Theta_2^3(B_1),\ \Theta_1^3(B_2),\ \Theta_1^2(B_3),\ \Theta_2^4(B_4)\Big], \\ & \Theta_2^3\Big[\Theta_1^2(B_5),\ \Theta_2^3(B_6),\ \Theta_1^4(B_7)\Big], \\ & \Theta_1^2\Big[\Theta_3^3(B_8),\ \Theta_1^2(B_9)\Big]\Big].\end{aligned}$$

Its corresponding access tree can be seen in Figure 4.6

Setup is similar to Example 5. Instead of iterating over the roots children we must iterate over all thresholds at the lowest level, level 3. The evaluated costs of these thresholds and their children as evaluated during the setup are displayed in Figure 4.6.

In the second phase, the algorithm goes through the remaining levels of thresholds. First level 2 then level 1 which only contains the root.

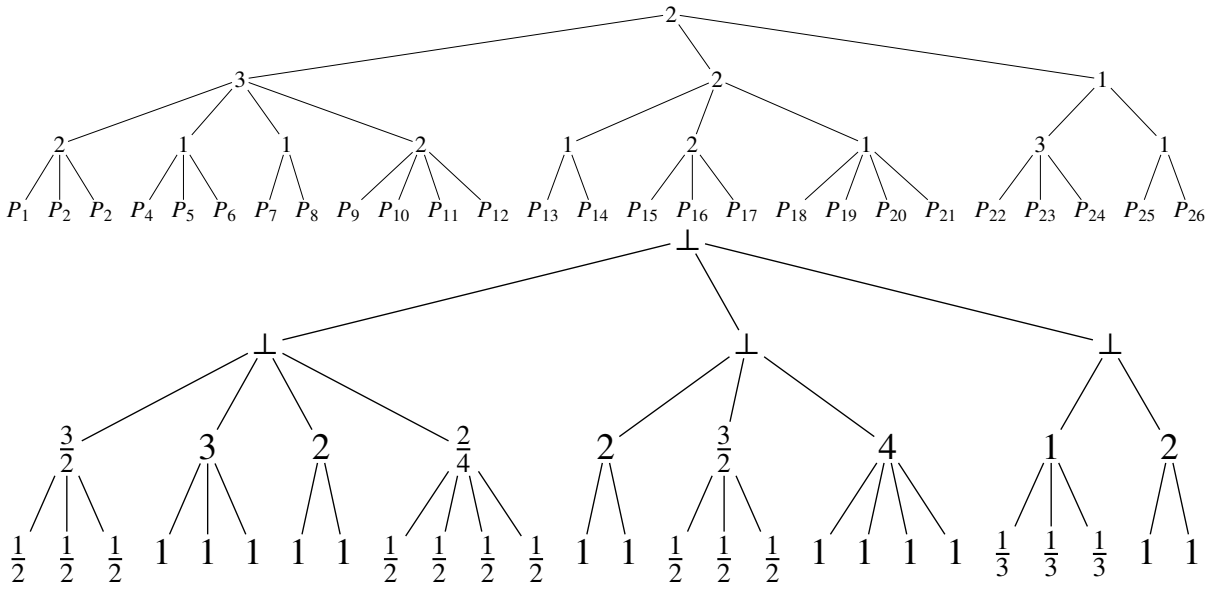


Figure 4.6. Accesstree (top) and evaluation of cost (bottom) during setup corresponding to the access structure in example 6

Chapter 5

Implementation

5.1 Development Environment and Dependancies

The choice of Go as the implementation language for this erasure code is motivated by a combination of security and ecosystem factors that align with the demands of both cryptographic operations and erasure coding.

5.1.1 Security

The official Go blog [4] reports that Trail of Bits conducted a comprehensive audit of Go's cryptography libraries, producing only a single low-severity finding in a legacy component. This is a remarkable outcome that supports the claim that Go's crypto ecosystem security is strong. The audit covered key exchange, digital signatures, encryption, hashing, key derivation, and authentication, as well as the cryptographic random number generator.

5.1.2 Cryptographic Ecosystem

In this thesis we will make heavy use of the *reedsolomon* erasure coding Go package published on github by Klaus Post. It is a pure Go port of the Java-Reed-Solomon library released by Blackblaze with additional optimizations, resulting in speeds exceeding 1GB/s/cpu core [6].

5.2 Data Architecture

A partitioned threshold access structure is represented, using three Go-structs, by a recursive structure that uses three Go-structs and distinguishes between threshold operators and nodes.

```
type Node struct {
  ID       string
  Name     string
  HeldData [][]byte
}

type Threshold struct {
  X       int64
  Y       int64
  Children []any
}

type AccessTree struct {
  Root  *Threshold
  Nodes []*Node
}
```

Each threshold operator contains a pair of integers (x, y) a slice `Children []any`. A slice is the Go implementation of a mutable array. The use of any is a deliberate trade-off: it allows a single slice to hold both `*Threshold` and `*Node` values. This allows a threshold operator to either have threshold operators or nodes as children. This does sacrifices some type safety and requires type assertions while traversing the tree. As a tradeoff it avoids the complexity of defining a separate interface or using recursive generic types. Note that y is redundant in the integer pair, since it is the length of the `Children` slice. It was used to make the code more readable.

`Root` and `Children` always store pointers to `Threshold` or `Node` structs, never the structs themselves. The algorithm relies on several maps to track per-node or per-threshold states. Using pointers as map keys provides O(1) average-case lookups and a low memory footprint. If the map keys were value types the map would need to hash the entire struct or a derived identifier. Additionally, the `AccessTree` struct maintains a separate slice `Nodes []*Node` that aggregates all leaf nodes. This flat list simplifies batch operations without requiring recursive descent each time.

5.3 Algorithmic Building Blocks

5.3.1 Basecode

Both the encoding and decoding algorithms make heavy use of the *reedsolomon* erasure coding Go package by Klaus Post [6]. Here, new terminology is used. What we have been calling fragments, here are called shards and they are subdivided into two groups. Data shards hold the data, and concatinating them yields our encoded data. Parity shards are the additional shards generated, allowing the reconstruction of the data in the case that shards are lost. For our purposes, a shard is always either available or not available, but never corrupted. With these preliminaries the library allows us to create an encoder, acting as our base code. The resulting algorithm can encode our data, generating parity shards, and reconstruct the data.

Algorithm 1 Klauspost Reed-Solomon Encoder

1: enc, err ← `reedsolomon.New`$(k, m-k)$
2: err ← `enc.Encode`(data)
3: err ← `enc.Reconstruct`(data)

The data here must be a two dimensional byte-array of size m where the first k elements, when concatenated, are the data we are storing and the last $m-k$ elements are the parity shards. When it is not possible to divide the data into k shards of equal size, some padding is added, which can be removed after reconstruction.

To simplify things, from now on we will go back to calling our data packets fragments regardless of wether they are data or parity shards.

5.3.2 Helper Functions for Access Tree Traversal

The algorithms described in Section 5.5 require additional tree-traversal utilities. These functions operate on the AccessTree structure and its maps, which are used to track nodes and threshold operators states during the greedy elimination process. When eliminating threshold operators we do not delete them during runtime, instead they are masked using a map, which maps each threshold operator to a boolean value. This allows us to easily and efficiently check if the threshold operator must be skipped.

- ComputeMinChildCost and ComputeMaxChildCost: These functions iterate over the children of a given threshold operator, skipping those that are masked. They return the child with the smallest or largest accumulated cost respectively.

- GetNodesUnder: Performs a depth-first search from a given *Threshold*, collecting all leaf Node pointers reachable without traversing through masked thresholds. This function is essential for resetting costs or collecting fragment sets after masking decisions.

- GetMaxDepth and GetThresholdsAtLevel: These methods on AccessTree compute the maximum nesting depth of thresholds and retrieve all thresholds at a specific depth using breadth-first search (BFS). They are used for debugging and for validating that the access tree conforms to expected structural constraints.

- ComputeK: Takes a slice of rational weights y and computes the least common multiple of the denominators, as outlined in Section 4.

5.4 Implementation for any Access Structure

In the general case, which can be applied to any given access structure, a linear programming problem is solved to find the optimal parameters. To do this the linear programming Go library GoLP [8] was used. GoLP lets us generate a linear programming solver, where we set the objective function, add the constraints and solve it, finding our vector y, as is outlined in Section 3.1.1.

Algorithm 2 Computation of y for the general access structure

Input: Matrix $\Gamma \in \mathbb{Z}^{r \times n}$ representing the access structure
Output: Vector $y \in \mathbb{Q}^n$ (rational approximations)

lp $\leftarrow$ golp.NewLP$(0, n)$
obj $\leftarrow [1, 1, \ldots, 1] \in \mathbb{R}^n$
lp.SetObjFn(obj) // Set objective to minimize the sum of all y_i (i.e., minimize $\sum y_i$)

for $i = 0$ **to** $r - 1$ **do**
    row $\leftarrow [\Gamma_{i,0}, \Gamma_{i,1}, \ldots, \Gamma_{i,n-1}]$
    lp.AddConstraint(row ≥ 1) // Add constraint: $\Gamma_i \cdot y \geq 1$, where Γ_i is the i-th row of Γ.

lp.Solve() // solves for a minimal y while respecting the constraint

yFloat $\leftarrow$ lp.Variables()
for $i = 0$ **to** $n - 1$ **do**
    $y[i] \leftarrow$ yFloat, rounded to two decimal points
return y

With the resulting y-vector we can compute k as the least common multiple of the denominators of the components of y. Each node P_i is then assigned $m_i = y_i \cdot k$ fragments.

The problem of this approach is that the linear programming solver operates exclusively on floating-point numbers. Therefore the resulting y vector is a slice of float64 values. Many optimal rational solutions (e.g. $y_i = \frac{1}{3}$) cannot be represented exactly in floating-point arithmetic.

As a compromise, the result is rounded to two decimal points and then converted into a rational number. This rounding can introduce a large error in the exactness of the final fragment counts. A proper solution would require a linear programming solver that operates directly on rational numbers. However implementing such a solver is outside the scope of this thesis.

In the following section, we describe the implementation for a partitioned threshold access structure. Here the problem is circumvented by working with rational numbers directly. Specifically, we use Go's standard *math/big* package, which provides a rational number datatype. While computation is less efficient with *math/big* rational numbers they, allow for easy and crucially exact extraction of the denominator.

5.5 Implementation for a Partitioned Threshold Access Structure

Both following algorithms work conceptually in the same way. They are split into 3 phases. First the setup is done: Maps to track the state of the nodes and threshold operators are generated, and their values are initiated. This is identical for both the 2-level and general partitioned access structure, with the exception that with the 2-level case the level of the access structure doesn't have to be computed.

Second comes the main stage of the algorithm where the optimization is done. Inefficient thresholds are masked and the costs of all nodes below them is set to 0. Lastly, the optimal number of fragments is calculated and a number of fragments is assigned to each node.

Type assertions are made regularly, the reason for this is discussed in Section 5.2 and performance implications are discussed in Sections 6.2.1.

Algorithm 3 ($\mathcal{A}$)

Input: Access tree $\mathcal{A}$ with root R (a threshold) and set of nodes $\mathcal{N}$
Output: Integer list m of length $|\mathcal{N}|$, and integer k

thresholdMask $\leftarrow$ empty map (Threshold $\rightarrow$ Boolean)
thetacost $\leftarrow$ empty map (Threshold $\rightarrow$ rational)
$h \leftarrow$ empty map (Node $\rightarrow$ rational)

Level = $\mathcal{A}$.GetMaxDepth()
ThresholdsAtLowestLevel = $\mathcal{A}$.GetThresholdsAtLevel(Level)

for each Node u in $\mathcal{N}$ **do**
    $h[u] \leftarrow 1$

costSum $\leftarrow 0$
for each threshold Θ in ThresholdsAtLowestLevel **do**
    costSum $\leftarrow 0$
    for each child u of Θ **do**
        $h[u] \leftarrow \frac{h[u]}{\Theta.X}$
        costSum $\leftarrow$ costSum $+h[u]$
    thetacost[Θ] $\leftarrow$ costSum

Algorithm 4 Fragments Assignment for 2-Level partitioned threshold access structures

Input: Access tree $\mathcal{A}$ with root R (a threshold) and set of nodes $\mathcal{N}$
Output: Integer list m of length $|\mathcal{N}|$, and integer k

```
Setup(A)

if R.X = R.Y then
    minΘ ← ComputeMinChildCost(R, thetacost, thresholdMask)
    for each child v of R.Children except minΘ do
        if v is a Threshold Θ then
            for each u in Θ.Children do
                if u is a Node then
                    h[u] ← 0
            thresholdMask[Θ] ← true
else
    x ← R.X,    y ← R.Y
    (maxΘ, maxCost) ← ComputeMaxChildCost(R, thetacost, thresholdMask)
    while true do
        costSum ← 0
        for each child v of R.Children except maxΘ do
            if v is a Threshold Θ then
                if thresholdMask[Θ] then continue
                costSum ← costSum + thetacost[Θ]
        if maxCost < 1/(x−1) · costSum or maxΘ = null or x ≤ 1 then break
        for each u in maxΘ.Children do
            if u is a Node then
                h[u] ← 0
        thresholdMask[maxΘ] ← true
        x ← x − 1,    y ← y − 1
        (maxΘ, maxCost) ← ComputeMaxChildCost(R, thetacost, thresholdMask)
    for each child v of R.Children do
        if v is a Threshold Θ then
            if thresholdMask[Θ] then continue
            for each u in Θ.Children do
                if u is a Node then
                    h[u] ← h[u] · x
            if Θ.Children[0] is a Node first then
                thetacost[Θ] ← h[first] · Θ.Y

k ← ComputeK(h)
m ← empty map    (Node → integer)
for each node u in AT.Nodes do
    m[u] ← k · h[u]
return (m, k)
```

Algorithm 5 Fragment Assignments for general partitioned threshold access structures

Input: Access tree $\mathcal{A}$ with root R (a threshold) and set of nodes $\mathcal{N}$
Output: Integer list m of length $|\mathcal{N}|$, and integer k

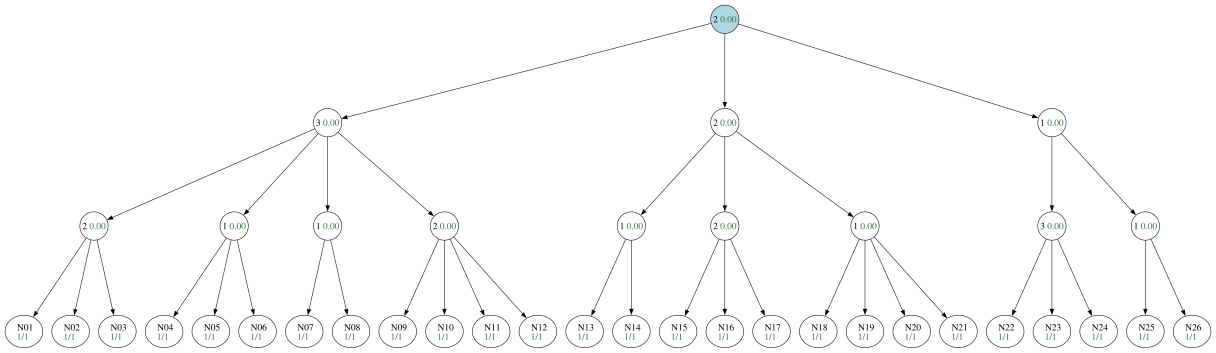
```
1: SETUP(A)

2: for i = 0 to L − 1 do
3:     level ← L − i − 1
4:     thresholds ← A.GetThresholdsAtLevel(level)
5:     for each threshold Θ ∈ thresholds do
6:         costSum ← 0
7:         if Θ.X = Θ.Y then
8:             (minΘ, minCost) ← COMPUTEMINCHILDSUM(Θ, thetacost, thresholdMask)
9:             thresholdsAtHigherLevel ← A.GetThresholdsAtLevel(level + 1)
10:            for each Θ in thresholdsAtHigherLevel do
11:                if Θ = minΘ then continue
12:                connectedNodes ← GETNODESUNDER(Θ, thresholdMask)
13:                for each node u in connectedNodes do
14:                    h[u] ← 0
15:                thresholdMask[Θ] ← true
16:                Θ.X ← Θ.X − 1
17:                Θ.Y ← Θ.Y − 1
18:            thetacost[Θ] ← minCost
19:        else
20:            x ← Θ.X, y ← Θ.Y
21:            (maxΘ, maxCost) ← COMPUTEMAXCHILDSUM(Θ, thetacost, thresholdMask)
22:            costSum ← 0
23:            while true do
24:                costSum ← 0
25:                for each child v of Θ.Children do
26:                    if v = maxΘ then continue
27:                    if v is a Threshold Θ then
28:                        costSum ← costSum + thetacost[Θ]
29:                if maxCost < 1/(x−1) · costSum or maxΘ = null or x ≤ 1 then break
30:                connectedNodes ← GETNODESUNDER(maxΘ, thresholdMask)
31:                for each node u in connectedNodes do
32:                    h[nd] ← 0
33:                thresholdMask[maxΘ] ← true
34:                x ← x − 1
35:                y ← y − 1
36:                (maxΘ, maxCost) ← COMPUTEMAXCHILDSUM(Θ, thetacost, thresholdMask)
37:            Θ.X ← x, Θ.Y ← y
38:            connectedNodes ← GETNODESUNDER(Θ, thresholdMask)
39:            for each node u in connectedNodes do
40:                h[u] ← h[u]/Θ.X
41:                costSum ← costSum + h[u]
42:            thetacost[Θ] ← costSum

43: k ← COMPUTEK(h)
44: m ← empty map   (Node → integer)
45: for each node u in AT.Nodes do
46:     m[u] ← k · h[u]
47: return (m, k)
```

5.6 Visualization

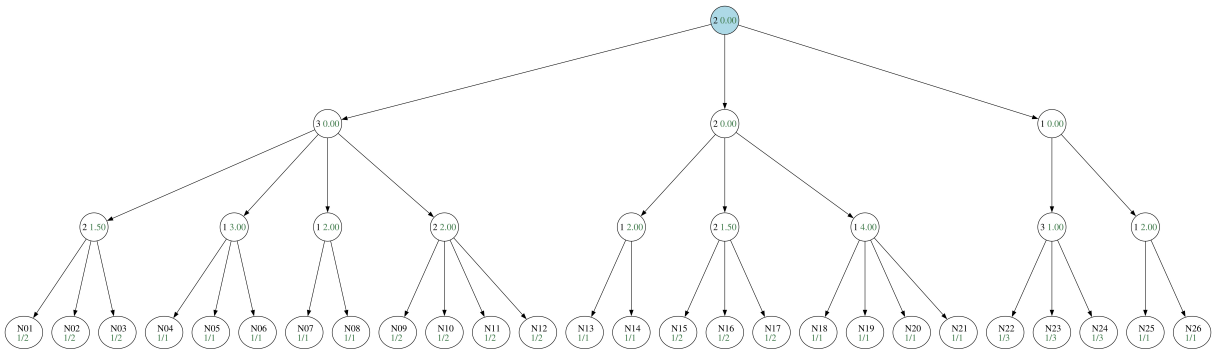
Additionally a tool was created to visualize the output of the fragment distribution algorithm. The VisualFragmentDistribution algorithm, uses a Go package [10] for generating graphviz trees to make snapshots of the access tree during the execution of the algorithm.

Graphviz is an open source graph visualization software that allows for the generation and rendering of graphs. In our case snapshots of the access tree are rendered both as a vector graphic and a pdf file. VisualFragmentDistribution outputs a single pdf file per snapshot, which can later be merged into one pdf file, using a simple shell script.

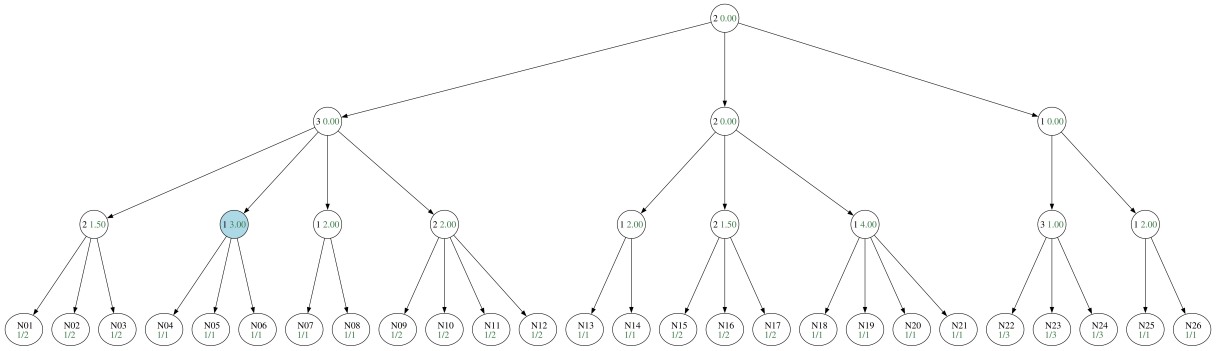
Example 7. Below is an example output of the visualization tool on the access structure from Example 6.



The access tree is loaded.

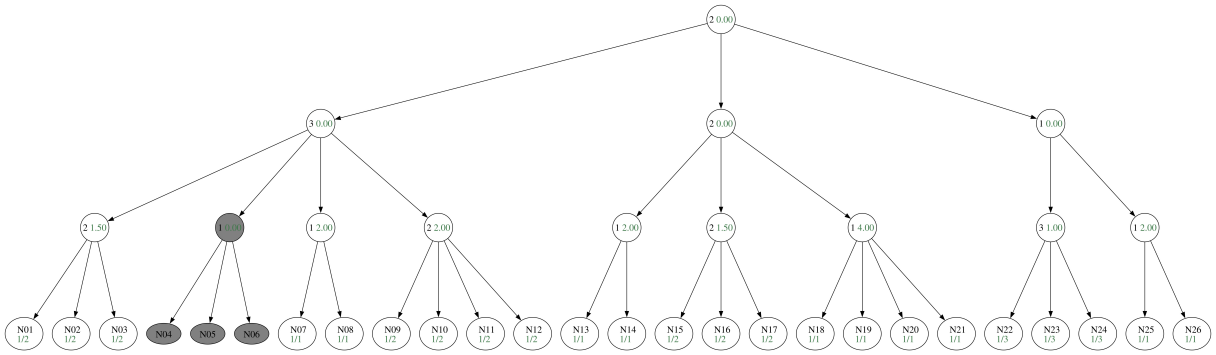


The thresholds on level 3 have their cost initiated.

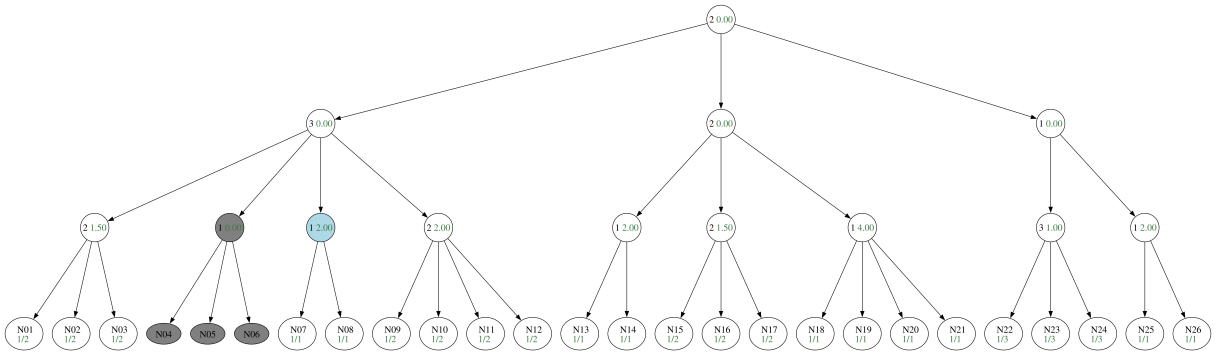


The algorithm checks for children of thresholds on level 2, that warrant removal.

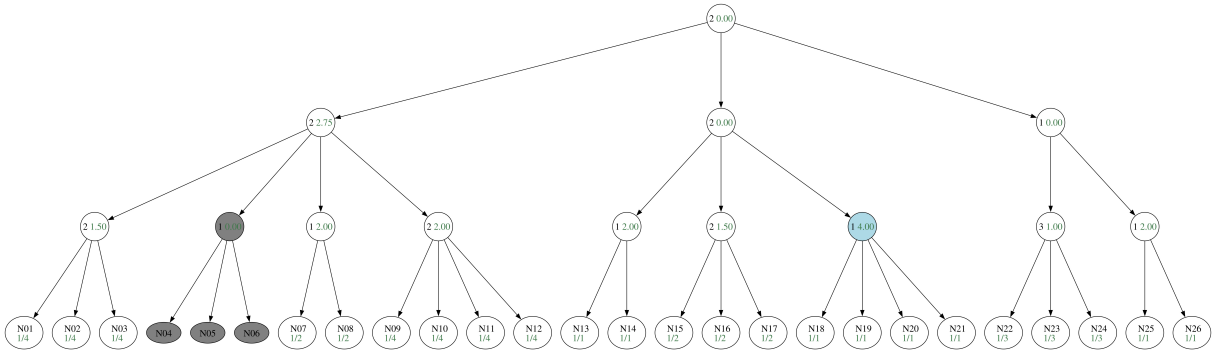
A threshold is evaluated for removal. Notice, that it has the highest cost of all of its parents children.



The threshold is removed since its cost is high enough.

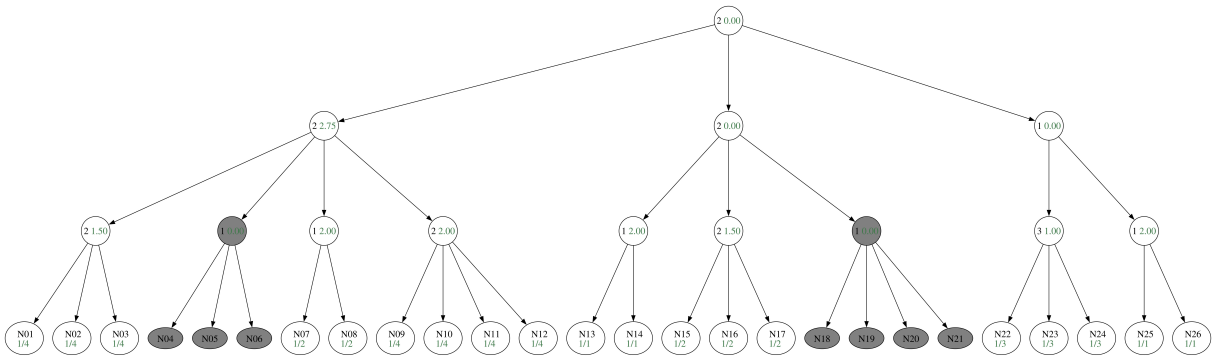


Another threshold is evaluated for removal.

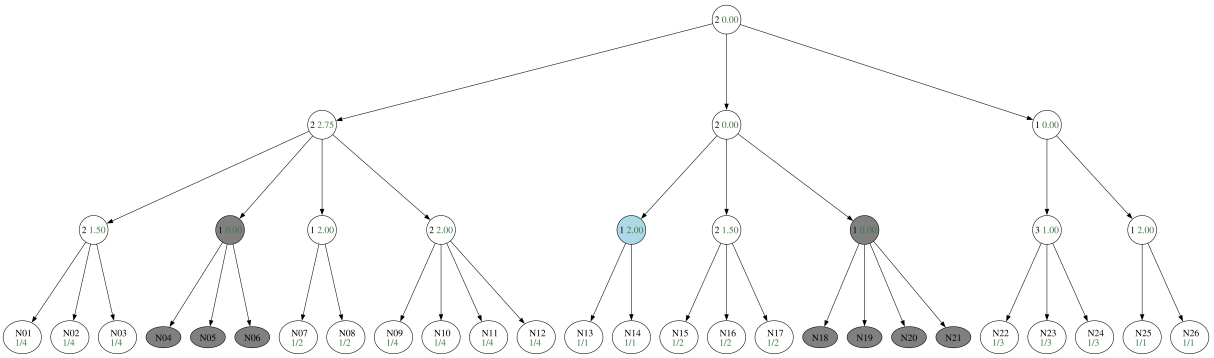


The previous threshold was not removed, since it's cost was not high enough.

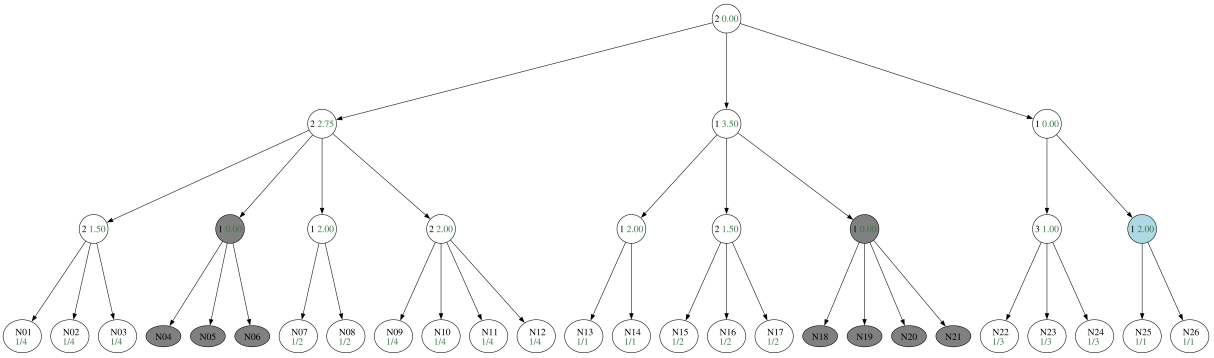
The parent threshold has been updated and the process repeats.



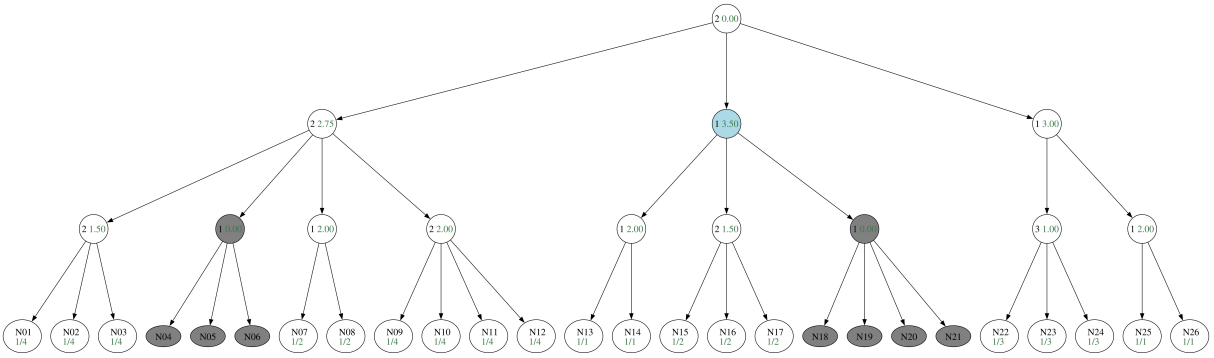
A threshold is removed.



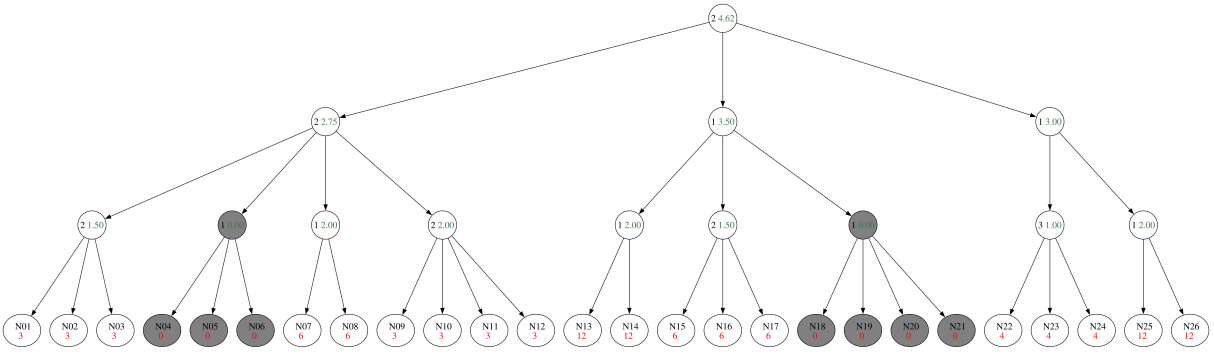
A threshold is evaluated but not removed.



A threshold is evaluated but not removed.



Of the roots children even the threshold with the highest cost does not warrant removal.



The algorithm has concluded. In red is shown how many base fragments are assigned to each node. The root node shows a cost of 4.62. This means we incur an overhead of 3.62.

Figure 5.1. Example output of visualization tool on the access structure from Example 6

Chapter 6

Validation

All code was thoroughly tested for bugs and issues using the standard go *testing* package [14].

To provide additional data on runtime and viability we wanted to run our fragment distribution algorithm on a large set of real world partitioned threshold access structures, which the Stellar [13] blockchain provided.

6.1 Stellar

Stellar is an open source blockchain used for a variety of payments and remittance applications [13]. The Stellar network is composed of nodes operated by a diverse set of organizations, including companies like MoneyGram, non-profits like the Stellar Development Foundation, financial institutions and individual validators. Each organization typically runs one or more nodes, and together they maintain the distributed ledger. To achieve consensus without a central authority, Stellar uses the Stellar Consensus Protocol (SCP). Unlike Bitcoin's proof of work or Ethereum's proof of stake, SCP is based on federated Byzantine agreement. At the core of Stellar's consensus protocol is the notion of local trust, i.e., each node decides for itself whom they trust. Specifically, each node defines some number of quorum slices, namely sets of nodes believed to be sufficient to reach agreement, without external arbitration. For example, a node operated by Bank A might define a slice consisting of Bank B, Bank C, Non-profit D, trusting that if at laest two of those three agree, the outcome is valid. From these local slices, global quorums emerge. A quorum is a set of nodes in which every node also has at least one of its slices contained within the quorum. In other words, when some nodes all satisfy each others slice requirements, they form a quorum.

To make progress on stellar, a quorum must reach agreement on a set of transactions to be committed to the ledger.

Compared to other blockchains, where everyone with computation or stake can vote, on Stellar to be able to vote, one must convince those already voting, of one's trustworthiness. Therefore, to influence consensus a node must be trusted by other nodes.

The Stellar Consensus Protocol is faster and cheaper than many other blockchains. Most crucially for us here, Stellar provides us with a sufficiently large set of example quorum slices. These quorum slices take the form of partitioned threshold access structures.

6.2 Evaluation on Stellar

A crucial tool that made this evaluation possible was the Stellar network explorer [5], which allowed for API access to the Stellar network. Through this tool, we could easily extract the slices of all $\sim$ 100 nodes.

The API provides structured snapshots of the stellar network in JSON format, which was imported and converted to the access tree data structure that was needed.

All testing was done on a surface pro 4 Intel core i5-6300U cpu @ 2.4GHz Go's standard testing library.

On average, we achieved an overhead of 1.06 with an average runtime of $368\mu s$.

6.2.1 Performance Implications of Access Tree Data Structure

In Section 5.2 we mentioned that letting threshold operators store both threshold operators and nodes as their children makes it necessary to do a lot of type assertions during runtime. To ease worries about the performance implication of this, a quick test was done.

Go type assertions seem to be incredibly fast. Computing 10^9 assertions takes approximately $690ms$, which comes to less than $0.7ns$ per assertion. As of Apr. 13. 2026, computing a fragment assignment on all nodes in the Stellar network encounters a total of 4986 type assertions, with about 50 type assertions per node on average. Considering the average runtime of a fragment assignment, which is $368\mu s$, and an average of $50 \cdot 0.7ns = 35ns$ per node the performance implication of these type assertions are negligible.

Chapter 7

Conclusion

This thesis described and implemented monotone erasure codes. The proposed scheme allows data recovery according to a monotone access structure $\mathcal{A}$. A general solution was presented, but its reliance on the extraction of rational numbers from floating points and potential exponential runtime make it impractical.

To overcome these limitations, we focused on partitioned threshold access structures, for which a greedy optimization algorithm was presented. The algorithm computes an optimal fragment assignment that minimizes storage overhead while respecting the given access structure.

The algorithm was implemented in Go and validated on real-world access structures extracted from the Stellar blockchain.

7.1 Critical Evaluation

Two main limitations of our scheme should be acknowledged. Firstly, Algorithm 2 relies on rounding, to circumvent the issue of extracting denominators from floating point values. This is inaccurate and can lead to a blow up in the number of base fragments that are generated.

Secondly, Stellar quorum slices generally generate 2-level partitioned threshold access structures that rarely contain threshold operators with costs large enough to warrant removal. The threshold operators having similar costs and similar number of children, also leads to nodes receiving similar amounts of data. When all nodes receive similar amounts of data, in the context of distributed storage, this means that all nodes are similarly trusted, reliable and accessible. Recall that this is the scenario in which regular threshold based erasure codes perform well. Our scheme aimed to provide utility beyond this rigid assumption of equal reliability. It would have been interesting to validate the scheme on a set of access structures that leverage this fine-grained control more.

7.2 Future Work

Implementing an LP solver that operates directly on rational numbers would guarantee exact solutions in the general case, since we would no longer have to extract denominators from floating point values.

More importantly, a complete distributed storage protocol based on this monotone erasure code remains to be built. Evaluating such a protocol under realistic network conditions would present a natural next step.

Additional work has been done by Raphael Fehr, who extends our subject of allocating storage according to a given access structure, but incorporates secrecy in his master thesis on linear codes for secret sharing [3].

For additional material and rigorous proofs of the algorithms implemented as part of thesis, we refer to the research paper [1].

Bibliography

[1] V. Bammert, A. Cimatti, O. Alpos, G. Losa, and C. Cachin. "Monotone Erasure Codes". Manuscript in preparation. 2026.

[2] *Ethereum Improvement Proposals*. [Online; accessed 6. May 2026]. May 2026. URL: https://eips.ethereum.org/EIPS/eip-7594.

[3] R. M. J.-M. Fehr. "Linear Codes for Secret Sharing: Exploring Two Approaches to Generate a Secret Sharing Scheme for a Given Access Structure". Master's thesis. University of Bern, 2025. URL: https://crypto.unibe.ch/archive/theses/2025.msc.raphael.fehr.pdf.

[4] *Go Cryptography Security Audit*. Go Blog. 2025. URL: https://golang.google.cn/blog/tob-crypto-audit#.

[5] Obsrvr. *Obsrvr Radar: Stellar Network Explorer*. Website. URL: https://radar.withobsrvr.com/.

[6] K. Post. *reedsolomon: Reed-Solomon Erasure Coding in Go*. GitHub repository. 2026. URL: https://github.com/klauspost/reedsolomon.

[7] M. O. Rabin. "Efficient dispersal of information for security, load balancing, and fault tolerance". In: *J. ACM* 36.2 (Apr. 1989), 335–348. ISSN: 0004-5411. DOI: 10.1145/62044.62050.

[8] D. Raffensperger. *Golp: Golang wrapper for the LPSolve linear programming library.* GitHub repository. 2026. URL: https://github.com/draffensperger/golp.

[9] I. S. Reed and G. Solomon. "Polynomial Codes over Certain Finite Fields". In: *Journal of the Society for Industrial and Applied Mathematics* 8.2 (1960), pp. 300–304.

[10] W. Schulze. *gographviz: Parses the Graphviz DOT language in Golang*. GitHub repository. 2026. URL: https://github.com/awalterschulze/gographviz.

[11] A. Shamir. "How to share a secret". In: *Commun. ACM* 22.11 (Nov. 1979), 612–613. DOI: 10.1145/359168.359176.

[12] Z. Shen, Y. Cai, K. Cheng, P. P. C. Lee, X. Li, Y. Hu, and J. Shu. "A Survey of the Past, Present, and Future of Erasure Coding for Storage Systems". In: *ACM Trans. Storage* 21.1 (2025), 4:1–4:39. DOI: 10.1145/3708994.

[13] *Stellar | Blockchain Network for DeFi, Payments & Asset Tokenization*. [Online; accessed 6. May 2026]. May 2026. URL: https://stellar.org.

[14] *testing: Support for automated testing of Go packages*. Go standard library. 2026. URL: https://pkg.go.dev/testing.

Statement on the use of Artificial Intelligence

I used a large language model for practice and learning about the GO programming language, which I learned in preparation for this thesis. Beyond practice no AI tools were used to write any of the code or any other content of this thesis.

Erklärung

Erklärung gemäss Art. 30 RSL Phil.-nat. 18

Ich erkläre hiermit, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.

Für die Zwecke der Begutachtung und der Überprüfung der Einhaltung der Selbständigkeitserklärung bzw. der Reglemente betreffend Plagiate erteile ich der Universität Bern das Recht, die dazu erforderlichen Personendaten zu bearbeiten und Nutzungshandlungen vorzunehmen, insbesondere die schriftliche Arbeit zu vervielfältigen und dauerhaft in einer Datenbank zu speichern sowie diese zur Überprüfung von Arbeiten Dritter zu verwenden oder hierzu zur Verfügung zu stellen.

Bern 9.5.2026

Ort/Datum

Unterschrift