2009-01-01

Confidential Data Dispersion using Thresholding

Aravind Prakash
University of Miami, prakar177@gmail.com

UNIVERSITY OF MIAMI

CONFIDENTIAL DATA DISPERSION USING THRESHOLDING

By

Aravind Prakash

A THESIS

Submitted to the Faculty
of the University of Miami
in partial fulfillment of the requirements for
the degree of Master of Science

Coral Gables, Florida

May 2009

UNIVERSITY OF MIAMI

A thesis submitted in partial fulfillment of
the requirements for the degree of
Master of Science

CONFIDENTIAL DATA DISPERSION USING THRESHOLDING

Aravind Prakash

Approved:

_____	_____
Burton Rosenberg, Ph.D. Associate Professor of Computer Science	Terri A. Scandura, Ph.D. Dean of the Graduate School
_____	_____
Akmal Younis, Ph.D. Associate Professor of Electrical and Computer Engineering	Huseyin Kocak, Ph.D. Professor of Computer Science

PRAKASH, ARAVIND (M.S., Computer Science)

Confidential Data Dispersion using Thresholding (May 2009)

Abstract of a thesis at the University of Miami.

Thesis supervised by Dr. Burton Rosenberg.
No. of pages in text. (41)

With growing trend in "cloud computing" and increase in the data moving into the Internet, the need to store large amounts of data by service providers such as Google, Yahoo and Microsoft has increased over time. Now, more than ever, there is a need to efficiently and securely store large amounts of data. This thesis presents an implementation of a Ramp Scheme that confidentially splits a data file into a configurable number of parts or *shares* of equal size such that a subset of those shares can recover the data entirely. Furthermore, the implementation supports a threshold for data compromise and data checksum verification to verify that the data parts have not been tampered with. This thesis addresses two key problems faced in large-scale data storage, namely, data availability and confidentiality.

Table of Contents

List of Graphs

List of Figures

Chapter 1

Introduction

1.1 Background

Data storage, reliability and security are important aspects of data management. With the increase in the Internet traffic and growing number of services on the cloud, by different service providers such as Google, Yahoo, Microsoft and Facebook. There is a lot of data that these companies need to efficiently store, including sensitive data like user profile, banking information, personal files, etc. Also, there is company related data such as proprietary algorithms, Software source code, etc which need to be stored away securely and reliably as a contingency for both disasters and attackers (such as hackers). It is now more than ever that there is a need for these companies to efficiently, securely and reliable store data.

Current procedures used for data management, fault/disaster recovery and increasing availability, such as the ones discussed in [10] and [11] heavily rely on 1-2 or 1-N redundancy models, where a 1-N redundancy model implies a total of N copies including the 1 active copy. A 1-N redundancy model can be better understood using the following example. If the original data were in Miami, a copy of the "entire data" would be maintained in say Chicago in 1-2 redundancy model. In 1-N redundancy model, if original data were in Miami, a copy of the data would be maintained in multiple cities say, Chicago, Beijing and London. In either case, if the data in Miami were lost due to a natural calamity (say), a copy of the data from the backup in Chicago, in case of 1-2 redundancy model or a back up from Chicago, Beijing or London in case of 1-N

redundancy model would be restored as original. This method not only poses data security issues, but also for a given level of reliability, requires more data than actually needed.

1.2 Motivation

Conventional data security systems rely on Data encryption methods such as DES and RSA. These encryption methods convert a message called *plaintext* into an encrypted message called *ciphertext* using a *key,* which is usually much smaller than the *plaintext* message. The problem with this kind of a system is, one the key needs to be safeguarded, because if the key is compromised, the attacker will be able to decipher the *ciphertext* and two, the probability that an attacker will be able to reproduce the key without any knowledge of it should be incredibly small, meaning, the key will have to be big, random and should be from a really large sample space. In-spite of the above, it must be noted that these systems only provide Computational Security, meaning, the key recovery is possible, but the computing cost involved is too high. Their security heavily depends upon the attacker not being able to either get hold of the *ciphertext* or generate the *key*, which in turn depends on how well the *ciphertext* is guarded and also depends on the sophistication of the hardware used by the attacker. If the attacker is indeed able to generate the key, even a small amount of *ciphertext* could potentially leak a critical amount of secret information. Over time, thanks to Moore's laws, the cost of Hardware required to break these system will be so small that the systems will be rendered unsafe for use. In-fact experts already believe that RSA (1024 bits) is unsafe [8]. Scientists have constantly been forced to invent newer methods of encryption due to failure of older

methods due to Hardware Sophistication. This race between the hardware sophistication and the strength of encryption is a huge limitation of computational security.

In this thesis, we present an implementation of a (*c, t, w)* Ramp Scheme. A (*c, t, w*) Ramp Scheme is a protocol to distribute a secret s among a set*P* of *w* participants in such a way that sets of participants of cardinality greater than or equal to *t* have full information on *s,* whereas sets of participants of cardinality less than or equal to *c* have no information on *s*, whereas sets of participants of cardinality greater than *c* and less than *t* might have "some" information on *s*.

Since compromise of shares of up to *c* in number have NO INFORMATION on *s*, no matter how sophisticated the attacker's hardware is, he will not be able to recover any information. Hence, Ramp Schemes provide Information Theoretic security, which is clearly better and more preferable than Computational security. Since only a fraction of the total shares is required to reconstruct the actual data, Ramp Schemes provide tolerance to loss of shares (due to whatever means). Tolerance to loss of shares is nothing but increased reliability of data. Therefore, Ramp Schemes not only provide Information Theoretic security, they also increase the data reliability. In this thesis, we provide an implementation of a (*c, t, w*) Ramp Scheme.

Ramp Schemes rely on randomness as a source to obscure data and increase each share's entropy. Unfortunately, Ramp Schemes not only use randomness to split the data into parts, but also embed random data into each share, hence requiring the user to store redundant random information as a part of the shares. This redundancy of random information is a worthy tradeoff for the amount of reliability and confidentiality that the Ramp Schemes provide when compared to conventional 1-N redundancy systems. The

more random or purer the randomness is, higher is the entropy of a share and hence higher is the level of security. An implementation of a Ramp Scheme requires good quality randomness. In [6] Blundo and Santis discuss the role of randomness in Ramp Schemes and in [7], Blundo and Masucci provide bounds for the amount of randomness as a function of the size of data used in a (c, t, w) Ramp Scheme. Randomness plays a very important role in the implementation of a Ramp Scheme. The more "random" the randomness is, better the result, meaning, harder it is to guess the randomness, better secure the system is. More about randomness will be explained in a separate section to follow. In this thesis, we use randomness from /dev/urandom. We split the data and randomness into different shares using a Ramp Scheme implementation. Data (in the form of a file, say) is split into multiple individual shares (N in number) using a Lagrange interpolation polynomial such that a subset of them (M) can reconstruct the data entirely. A parameter called the threshold parameter (K) is used to indicate the tolerance the shares have to intrusion. This is equal to the amount of randomness added to the data. Compromise of up to K shares will reveal absolutely no information, while compromise of shares greater than K in number, but less than M might reveal some information. The security of the channels used for communicating the shares is of paramount importance since the compromise of those channels would defeat the system as a whole. In [12], Beimel and Chor discuss the implementation of Ramp Schemes in a public channel. In this thesis however, we assume that the channels used for communication are secure. Furthermore, detection of data manipulation, i.e., detecting if the data has been manipulated at one or more of the individual systems which hold data and identification of system(s) where the data is being manipulated is an important aspect of Ramp Scheme

implementation. These aspects are analyzed and a method to detect cheating and isolate cheaters is suggested by T. C. Wu and T. S. Wu in [15]. In this thesis, we present an implementation where we support detection of data manipulation in the form of a data integrity check, but implementation of the algorithm corresponding to the identification of the manipulating systems is left to the application which uses our library since.

1.3 Organization of this Document

In Chapter 2 of this document, we discuss Ramp Schemes in detail and in chapter 3 we give details about an Algorithm to implement a (*c, t, w*) Ramp Scheme. In chapter 4, we present the architecture of our implementation followed by the library implementation of Ramp Scheme in chapter 5 using C programming language on Fedora Linux. The implementation is based on the algorithm described in chapter 3. We also discuss the performance of the Split and Merge functionalities of the implementation. Chapter 6 discusses the future work and chapter 7 concludes by summarizing our work.

Chapter 2

Ramp Schemes

Secret sharing schemes were introduced by Shamir[1] and Blakeley[2]. They analyzed sharing schemes for cases where a subset of shares (fixed) t out of w can reconstruct a secret. These schemes are called (t, w) threshold schemes.

2.1 Definition

In [4], Blundo, Santis and Vacarro define a (*c, t,* w) Ramp Schme as follows: A(*c, t, w*) Ramp Scheme is a protocol to distribute a secret *s* among a set *P* of *w* participants in such a way that set of participants of cardinality greater than or equal to t can reconstruct the secret s, sets of participants of cardinality less than or equal to c have no information on s, whereas sets of participants of cardinality greater than c and less than t might have "some" information on s. A (*t, w*) threshold scheme is a special case of a ramp scheme where $c = t - 1$. Formally defined, a (c, t, w) ramp scheme, where $1 \leq c < t \leq w$, is a sharing of a secretS among participants w, such that

1. Any set of at-least t participants can reconstruct the secret. That is, if A represents the collection of shares from number of participants ≥ t, it holds $H(S \mid A) = 0$.
2. Any set of at-most c participants has absolutely no information on the secret. That is, if A represents the collection of shares from number of participants ≤ c, it holds $H(S \mid A) = H(S)$

By definition of Ramp Schemes above, for any set of participants up to *c* in number, $H(S \mid A) = H(S)$, meaning, the entropy or the uncertainty of S doesn't change by knowledge of A (more details can be found in [9]), in other words, knowledge of up to *t*

shares will keep the secret as uncertain as it was without knowing any shares. Any set of more than *c* and less than *t*participants might have "some" information on the secret S. Formally from [4], for all A ⊆ P with $c < |A| < t$, it holds, $H(S \mid A) = (t - |A|) * H(S) / (t - c)$. This expression shows that the entropy,

$H(S \mid A) \leq H(S)$ for $c < |A| < t$, which implies that the Secret is less uncertain than it was without the knowledge of any shares.

From the above analysis, it can be concluded that Ramp Schemes provide information theoretic security rather than just computational security. Blakley and Meadows provide a detailed analysis of security of Ramp Schemes in [14]. We now present comparisons between Ramp Schemes and Conventional redundancies.

2.2 Ramp Scheme versus Conventional 1-N Redundancies

We compare the Ramp scheme against conventional 1-N redundancy in the following situations:

1. With respect to Data Storage Space: The amount of data storage space required is the same in both, Ramp Scheme and the 1-N redundancy.
2. With Respect to number of Individual Systems: We are provided with a set of highly vulnerable systems. That is, the probability of failure and or compromise of each system in both the Ramp Scheme and the 1-N redundancy is high. This analysis is practical in situations where there is less or no control over the individual systems. In particular when sensitive information is stored in nodes over WAN or Internet.

Scenario – 1: Amount of Data Storage is Constant

Relation between the size of shares of Ramp Scheme and size of share of 1-N Redundancy:

Though the user could store each share in an individual system or multiple shares on the same system, here, it has been assumed that the user stores individual shares in different systems, in which case, the total storage space required will be equal to the number of systems times the size of each share.

Let the size of Data to be stored be "*D'*.

Let the total number of generated shares under the Ramp Scheme be "*n'*.

Let the total number of shares required to reconstruct the data be "*m*'.

Let the total number of shares of randomness used be *"k'*

Hence, total number of shares of data used to split the data using the Ramp Scheme is (*m – k)*

The size of each of "*n*" generated shares is given by $s_i = D/(m - k)$

Hence, the total size of all the shares is given by, $S_1 = nD/(m - k)$

Share size under 1-N Redundancy is given by:

Size of Data is *D* and a copy is stored on each of the n^1.

Hence, total storage space required, $S_2 = n^1 D$.

We are concerned about a given amount of storage space, i.e., $S_1 = S_2$,

Therefore,

$$nD/(m - k) = n^1 D$$

or, $\boldsymbol{n/(m - k) = n^1}$

The above expression gives a relation between the Number of systems required in both the Setups, one using the Ramp Scheme and the other using the 1-N Redundancy.

Here it must be noted that, we do not consider the reparability of the individual systems. Meaning, if an individual system fails, the system administrator may be in a position to replace or repair the failed unit. Our assumption is that the systems will not be repaired or serviced once deployed. The repair and servicing aspect has been considered in detail and well analyzed in several reliability engineering books such as [13]. We now consider an example to demonstrate a setup where Ramp Scheme outperforms 1-N redundancy.

Example:

Consider a (c, t, w) Ramp Scheme with c = 1, t = 3 and w = 8. We will compare the reliability of this setup against a 1-3 redundancy model setup, which consists of 1 active and 3 failsafe redundant systems. In this example, it has been assumed that similar systems are used in both setups. Particularly, probability of failure of an individual system in Setup 1 using the (1, 3, 8) Ramp Schemes is the same as the probability of failure of the individual system in Setup 2 using the 1-3 Redundancy model.

Setup 1

We use 1 part of randomness and 2 parts of original data; this will make the tolerance of the system to 1 share. Meaning, compromise of 1 share will leak absolutely no information about the data while compromise of 2 shares will leak half the information about the data.

Let the size of the original data be D.

Hence, total storage space required to house this setup is the same as the sum total of sizes of all the shares. That is,

$$\frac{w}{t-c} \cdot D$$

In this particular case, the total storage space required is, 8D/(3 – 1) = 4D.

Setup – 2

We use a 1-4 redundancy model consisting of 1 active system and 3 failsafe redundant systems. Therefore, Total storage space required is (3+1)D = 4D, since we replicate original data into 3 redundant parts.

Now, for a given storage space (4D), we compare the reliabilities of Setup 1 and Setup 2.

Reliability of Setup – 1

Reliability of Setup 1 after time x is the probability that at-least t systems are up and running after time x. Hence, reliability is given by,

$$R_1(x) = \binom{w}{t} p_x^t (1-p_x)^{w-t} + \binom{w}{t+1} p_x^{t+1} (1-p_x)^{w-t-1} + \binom{w}{t+2} p_x^{t+2} (1-p_x)^{w-t-2} + \ldots + \binom{w}{w-1} p_x^{w-1} (1-p_x) + \binom{w}{w} p_x^w$$

Where, p_x is the probability that the system will be up after time x.

Substituting values of w and t, we get,

$$R_1(x) = \binom{8}{3} p_x^3 (1-p_x)^5 + \binom{8}{4} p_x^4 (1-p_x)^4 + \binom{8}{5} p_x^5 (1-p_x)^3 + \binom{8}{6} p_x^6 (1-p_x)^2 + \binom{8}{7} p_x^7 (1-p_x) + \binom{8}{8} p_x^8$$

Reliability of Setup – 2

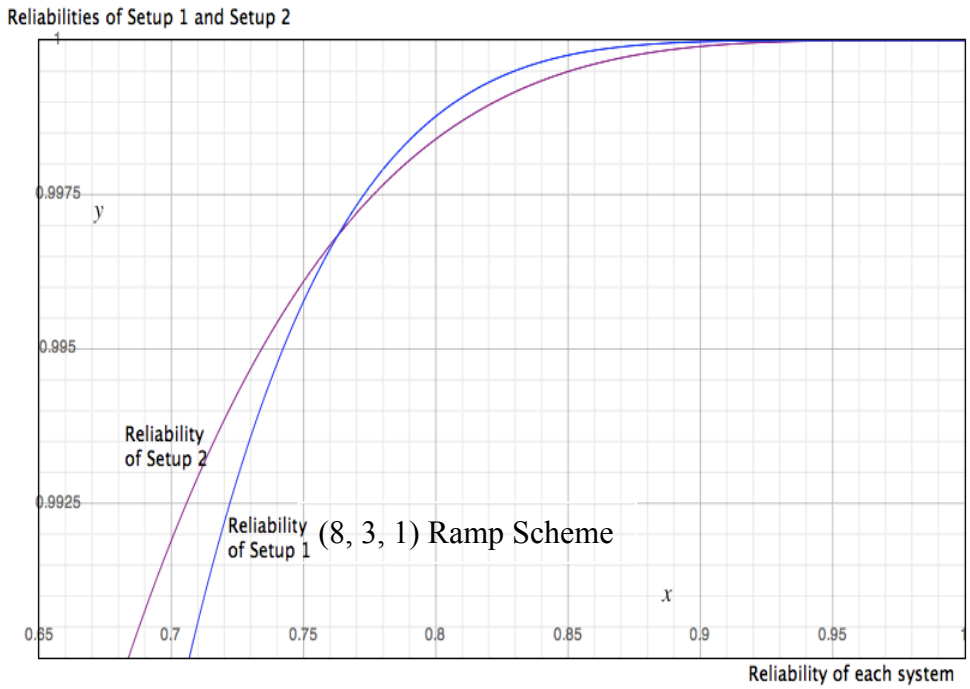
Reliability of Setup 2 after time x is the probability that at-least 1 system is up and running after time x. Reliability of such a setup with $n$systems is given by,

$$R_2(x) = 1 - (1 - p_x)^n$$

In this case, n = 4, since there are 4 systems in all. Substituting for n, we get,

$$R_2(x) = 1 - (1 - p_x)^4$$

Plotting $R_1(x)$ and$R_2(x)$ versus the p_x, the probability of a system being up after x units of time, we get the graph,

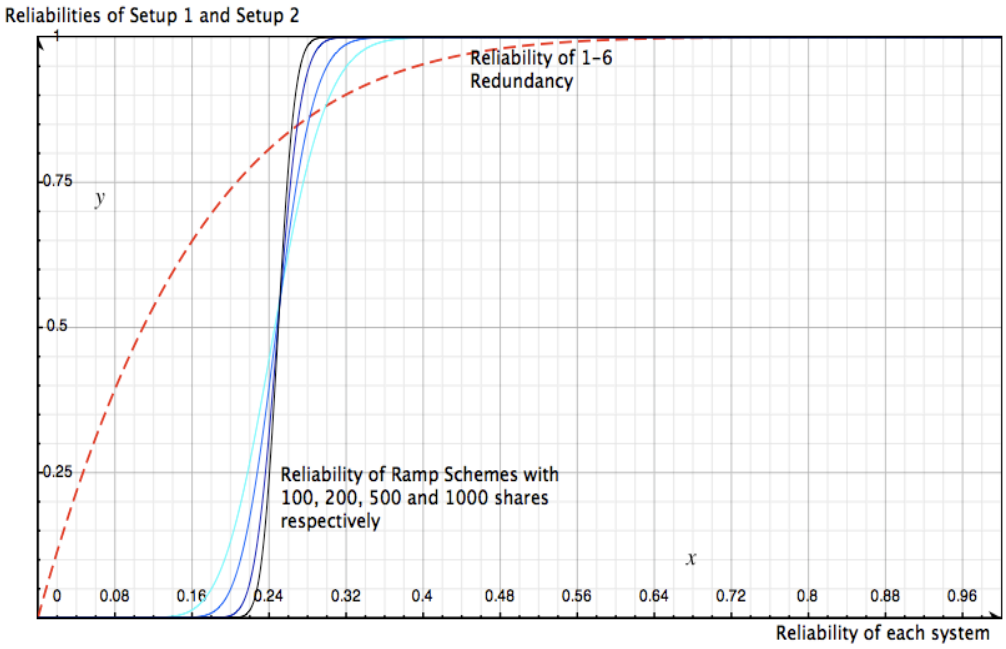


Graph 1 - Reliabilities of (8,3,1) Ramp Scheme and 1-2 Redundancy Setup

From the graph, it can be seen that as the reliability of individual systems increase over 0.76, the reliability of setup 1 becomes higher than reliability of setup 2. The increased reliability along with the threshold for tolerance (the confidentiality aspect, which allows the compromise of upto 1 share) makes Setup 1 a more suitable implementation. More such examples can be given to show that the ramp scheme out performs conventional 1-N redundancies. A (2, 10, 24) Ramp Scheme requires the same amount of storage space as a 1-3 Redundancy model, but gives better reliability than 1-3 redundancy at higher individual system reliabilities. A (5, 15, 40) Ramp Scheme requires the same amount of storage space as a 1-4 Redundancy model, but provides higher reliability at higher individual system reliabilities. In general, for a given amount of data storage space, Ramp Schemes provide higher reliability with higher individual system reliabilities.

Scenario – 2: Highly vulnerable individual systems

High vulnerability of a system implies that the system is prone to attack and or compromise. It will be looked upon as high probability of failure or compromise. In this scenario, we are interested in comparing the reliability of Ramp schemes against reliability of conventional 1-N redundant systems while the probabilities of failure of individual systems are very high. This analysis is specially useful in considering applications of Ramp Schemes in a Wide Area Network (or Internet) where there are a huge number of systems, where each of them can be pretty hostile or their down time can be pretty high. Ayari, Barbaron, Lefèvre and Primet [17] present different challenges and High Availability issues in Internet based services and Ayari, Barbaron and Lefèvre suggest improvements to Internet based High Availability systems in [10].



Graph 2 - Reliability of Ramp Schemes with 1000 nodes and 1-6 Redundancy System

The above figure shows that the Ramp Schemes out performs even the 1-6 Redundancy when the reliability of individual systems is small. The Reliability curve

gets steeper as the number of shares increase. This set up is particularly useful if Ramp Schemes were used as a model to store sensitive data over the Internet.

Chapter 3

Algorithm

In this chapter we discuss the algorithm used to implement a *(c, t, w)* Ramp Scheme. In a nutshell, we construct a Lagrange Interpolation Polynomial using the Original data to be split and the randomness as input points and generate the desired number of shares at the output points.

3.1 Background on Lagrange Interpolation

The Lagrange interpolating polynomial is a polynomial P(x) of degree less than or equal to (n-1) that passes through the n points (x_1, y_1), (x_2, y_2), ..., (x_n, y_n), where $y_1 = f(x_1)$, $y_2 = f(x_2)$, …, $y_n = f(x_n)$. It is given by,

$$P(x) = \sum_{j=1}^{n} P_j(x)$$

,

where,

$$P_j(x) = y_j \prod_{k=1,\, k \neq j}^{n} \frac{x - x_k}{x_j - x_k}$$

,

Explicitly written,

$$P(x) = \frac{(x - x_2)(x - x_3)\ldots(x - x_n)}{(x_1 - x_2)(x_1 - x_3)\ldots(x_1 - x_n)} \cdot y_1$$

$$+ \frac{(x - x_1)(x - x_3)\ldots(x - x_n)}{(x_2 - x_1)(x_2 - x_3)\ldots(x_2 - x_n)} \cdot y_2$$

$$+ \ldots + \frac{(x - x_1)(x - x_2)\ldots(x - x_{n-1})}{(x_n - x_1)(x_n - x_2)\ldots(x_n - x_{n-1})} \cdot y_n$$

Note that the Lagrange Interpolation Polynomial P(x) can not be reconstructed without at-least "n" points. This property is the basis of the algorithm presented here. We construct a Lagrange Interpolation Polynomial of degree m – 1, where "m" is the minimum number of shares required to recover the original data, use "m – k" inputs of original data parts and "k" inputs of random data at "m" points beginning from point 0 and evaluate the output at "n" output points which are different from the input points. To reconstruct the original data, we take any "m" out of "n" input points and evaluate the output at "m – k" points beginning from 0.

3.2 Assumptions

We make the following assumptions to facilitate the description of the algorithm:

1. We assume that we are provided with a field with size of at-least 256 with an implementation of addition, subtraction, multiplication and inverse functions (The algorithm has been tested for GF (256). However, the Algorithm will work with any GF's with greater than 256 elements. The number of shares that can be generated depends on the number of distinct points on the field. If a field has "N" distinct points, "*N – m*" shares can be generated in the field). A method SUM(*a, b*), which adds two field elements *a* and *b* and returns the resulting field element. A method DIFF(*a, b*), which subtracts field element *b* from field element *a* and returns the resulting field element (Note that if we are using a GF with characteristic 2, then SUM and DIFF will both be the same, which is an XOR operation of the two elements). A method PROD(*a, b*), which returns a field

element which is a product of two field elements *a* and *b*. We also assume the presence of a function called INV (*x*) which returns the inverse of the element *x* in the field.

2. We assume an infinite source of Randomness RAND (), which on each call returns a single byte of randomness.
3. We assume the existence of a function HASH (data, hash), which generates the hash for the data and returns the size of the hash. We also assume that the character "-" is not a part of the hash. This is important because "-" is used as padding at the end of data and hash.

3.3 Useful Sub-Routines

We define the following methods that will be used in the algorithms for SPLIT and the MERGE.

We define method GETBYTE, a method used to retrieve a byte from a part starting from the first byte. On successive calls, successive bytes are obtained. This works similar to *getc(FILE *fp)* in stdio C library.

GETBYTE(*in_part*)

1. **return**one byte from *in_part*.

We define PUTBYTE method, used to put a byte into a part. On successive calls to PUTBYTE, bytes are appended to the original content. PUTBYTE works similar to the *putc(int data, FILE *fp)* in stdio C library.

PUTBYTE(*byte, out_part*)

1. *out_part[last_byte]* ← *data*

The following two procedures CALC_INCROSS and CALC_OUTCROSS contribute towards forming the Lagrange Interpolation Polynomial. CALC_INCROSS is a method, which calculates the product of $(x_i - x_j)$ with j not equal to i, where x_i are input points. That is, it evaluates,

$$incross[i] = \prod_{i=0,\, i \neq j}^{m-1} (x_i - x_j)$$

CALC_INCROSS(*in_points*)

1. **for***i* ← *0* **upto** *length*[*in_points*]
2. **do***val*←1
3. **for***j* ← *0* **upto** *length[in_points]*
4. **do if** (*j not equal to i)*
5. **then***val* ← PROD (*val,* DIFF(*in_points[i], in_points[j]*))
6. *in_cross[i]* ← *val*
7. **return***in_cross*

CALC_OUTCROSS is a method, which calculates the product of $(z_i - x_j)$ for all *i*, where z_i are output pointsand $x_j$are input points. That is, it evaluates,

$$outcross[i] = (z_i - x_1)(z_i - x_2) \ldots (z_i - x_n)$$

CALC_OUTCROSS(*in_points, out_points*)

1. **for***i* ← *0* **upto** *length*[*out_points*]
2. **do***val*←1
3. **for***j* ← *0* **upto** *length[in_points]*

4. **do***val*← PROD (*val,*DIFF(*out_points[i], in_points[j]*))
5. *out_cross[i]* ← *val*
6. **return***out_cross*

EVAL_POINT is a method, which plugs in the *m* input data values into the Lagrange Interpolation Polynomial generated from CALC_INCROSS and CALC_OUTCROSS and interpolates the *n* output data values.

$$outdata[i] = \sum_{j=0}^{m-1} \left(\frac{(z_i - x_1)(z_i - x_2) \ldots (z_i - x_n)}{(z_i - x_j) \cdot \prod_{k=0, k \neq j}^{m-1} (x_j - x_k)} \right) . indata[j]$$

This is nothing but,

$$outdata[i] = \sum_{j=0}^{m-1} \left(\frac{outcross[i]}{(z_i - x_j) \cdot incross[j]} \right) . indata[j]$$

EVAL_POINT(*in_points, out_points, in_cross, out_cross, in_data*)

1. **for***i* ←0 **upto** *n* – 1
2. **do***val* ←0
3. **if***out_cross[i] is equal to* 0 //Means output point is an input point retain data
4. **then for** *j* ←0 **upto** *m – 1*
5. **do if** *out_points[i] equals in_points[j]*
6. **then***val* ← *in_data[j]*
7. **else**
8. **for***j* ←0 **upto** *m – 1*

9. **do***term1 ← DIFF(out_points[i] – in_points[j])*
10. *term1← PROD (term1, in_cross[j])*
11. *term1← INV(term1)*
12. *term1← PROD(term1, out_cross[i])*
13. *term1← PROD(term1, in_data[j])*
14. *val← SUM(val, term1)*
15. *out_data[i] ← val*

We now move on to the Algorithms for splitting and merging.

3.4 SPLIT

We define the method SPLIT used to split data (*D*) of size (*D_size)* into "*n"*parts of equal size such that at-least "*m*" of the "*n*" parts are required to reconstruct the data successfully. Compromise of up-to "*k*" parts will reveal absolutely zero information, while compromise of less than m, but greater than k number of parts might reveal some information.

SPLIT (*D, D_size, m, n, k*)

1. **if***n< 0* or *m < 0* or *n < m* or *m < k*
2. **thenreturn** config_error
3. *hash_len*←HASH (*D, cs*)
4. *pad_size*←*m – k –* ((*D_size + hash_len) mod (m – k)*)
5. *D*← Concatenate *D* and *cs*
6. **for***i* ← 1 **upto** *pad_size*
7. **do***D* ←Concatenate *D* and "-"

8. **for***i*← 0 **upto** *m* – 1
9. **do***in_points[i]* ← *i*
10. **for***i* ← *m* **upto** *m* + *n* – 1
11. **do***out_points[i]* ← *i*
12. *part_size* ← (*D_size* + *hash_len* + *pad_size*)/ (*m – k)*
13. **for***i* ← *0* **upto** *m – k – 1*
14. **do***in_parts[i] = D[i * part_size]* //Split data into *m - k* parts
15. *in_cross* ← CALC_INCRORSS(*in_points*)
16. *out_cross* ←CALC_OUTCROSS(*in_points, out_points*)
17. **for***i* ←1 **upto** *part_size*
18. **do for** *j* ← *0* **upto** *m - 1*
19. **do if** *j* < *m – k – 1*
20. **then***in_data[j]* ←GETBYTE(*in_part[j]*)
21. **else***in_data[j]* ←RAND()
22. *out_data* ←EVAL_POINT(*in_points, out_points, in_cross, out_cross, in_data*)
23. **for***j* ← *0* **upto** *n – 1*
24. **do**PUTBYTE(*out_parts[j], out_data[j]*)
25. **return***out_parts, out_points*

3.5 MERGE

We define method MERGE, used to merge the data parts into the original data. Merging works similar to splitting. We plug the *m* parts at their respective points in the Lagrange Interpolation Polynomial and evaluate the outputs at points 0 to *m – k – 1*,

which will be the original data parts. That is, *m* out of *nout_points* of SPLIT become the *in_points* of MERGE and the *in_points* of SPLIT become the *out_points* of merge. *Parts* is an array of parts with data and the point of evaluation (in_point), *part_size* is the size of each part, *m, n* and *k* form the configuration parameters which must be identical to what they were when the data was split using SPLIT.

MERGE (*parts, part_size, n, m, k*)

1. **if***n*< *0* or *m* < *0* or *n* < *m* or *m* < *k*
2. **thenreturn** config_error
3. **for***i*← 0 **upto** *m* – 1
4. **do***in_points[i]* ←*parts[i].id*
5. **for***i* ← *0* **upto** *m* - *k* – 1
6. **do***out_points[i]* ← *i*
7. *in_cross* ← CALC_INCRORSS(*in_points*)
8. *out_cross* ←CALC_OUTCROSS(*in_points, out_points*)
9. **for***i* ←1 **upto** *part_size*
10. **do for** *j* ← *0* **upto** *m* - *1*
11. **do***in_data[j]* ←GETBYTE(*parts[j]*)
12. *out_data* ←EVAL_POINT(*in_points, out_points, in_cross, out_cross, in_data*)
13. **for***j* ← *0* **upto** *m – k – 1*
14. **do**PUTBYTE(*out_parts[j], out_data[j]*)
15. **for***i* ←1 **upto** *m – k – 1*
16. **do***D* ← *out_parts[i]* concatenated to *D*
17. **while**the last character of *D* is "-"

18. **do** remove "-" //Remove padding

19. *cs* ←removehash from end of D

20. HASH (*D, cs_new*)

21. **if***cs* not equal to *cs_new*

22. **return** checksum error

23. **return***D*

Chapter 4

Architecture

In this chapter we discuss the architecture of the library.

4.1 Hashing

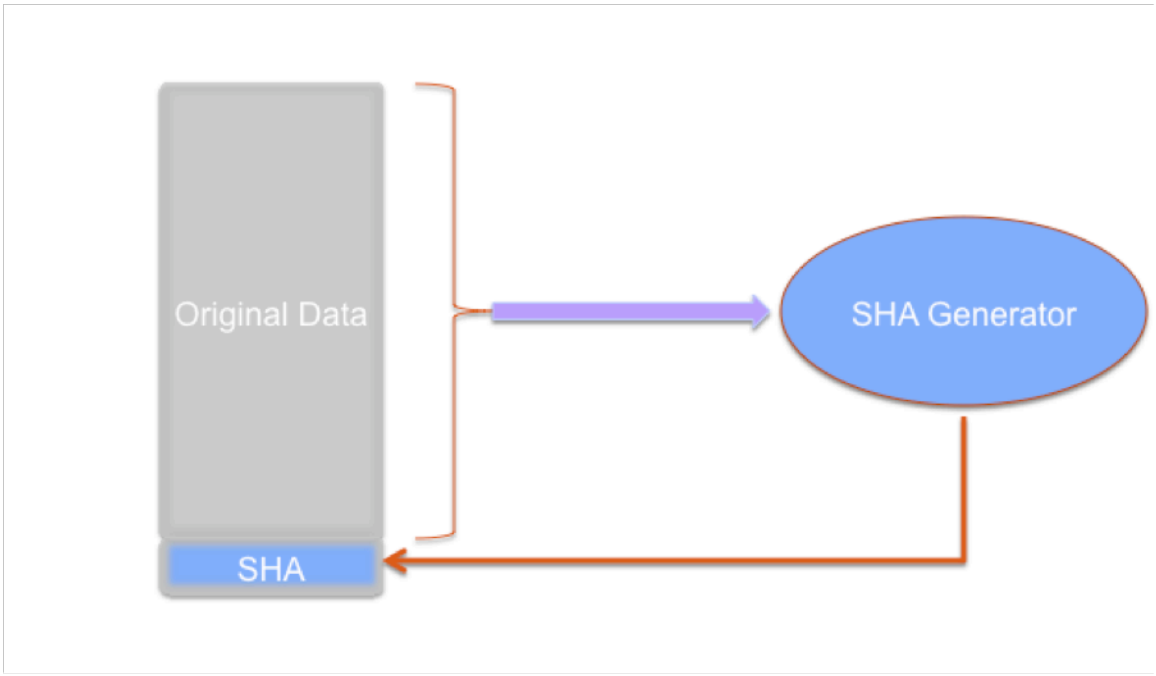


Figure 1 - Updating Data with the SHA hash string

The library depends on an external SHA library [19] to generate the SHA hash. A reference to the data, which needs to be split, is passed to the SHA library. The SHA library generates a SHA hash string corresponding to the data, which is then appended to the data itself.

4.2 Splitting

The following diagram shows data is split into shares. The middle portion of the diagram shows the Lagrange Interpolation polynomial constructed out of the *m* input points, which include *m* – *k* data points and *k* random points.

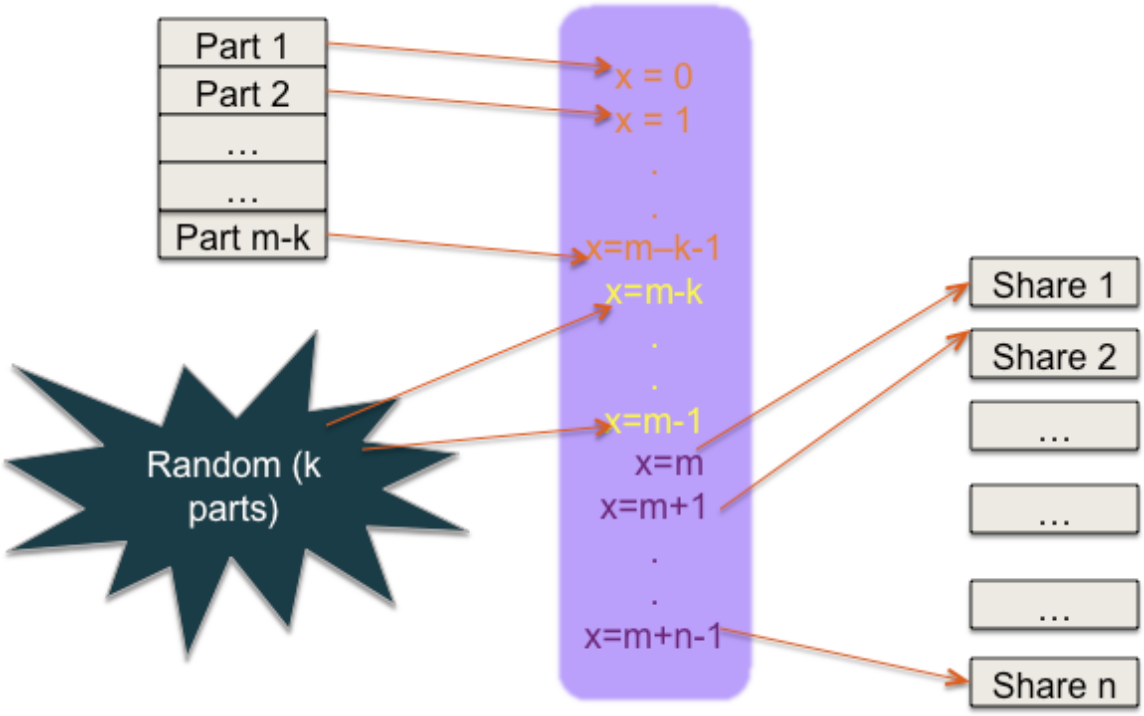


Figure 2– Splitting data using Lagrange Interpolation Polynomial

It is a matter of choice to choose the points where the actual data will correspond . The *m* – *k* parts of Original data (appended with SHA) are used at input points 0, 1, 2, …,*m* – *k* – *1,* along with *k* parts of random data are used as input points at *m* – *k, m* – *k* + *1, ..., m* and a Lagrange interpolation polynomial is constructed. The data from the input points is plugged into the Lagrange Interpolation Polynomial and the data at the *n* output points (which are not the same as the input points) are evaluated to obtain the *n* shares. The

point that a share corresponds to is stored as a part of the share. This information is essential during the reconstruction of the original data from the shares.

4.3 Merging

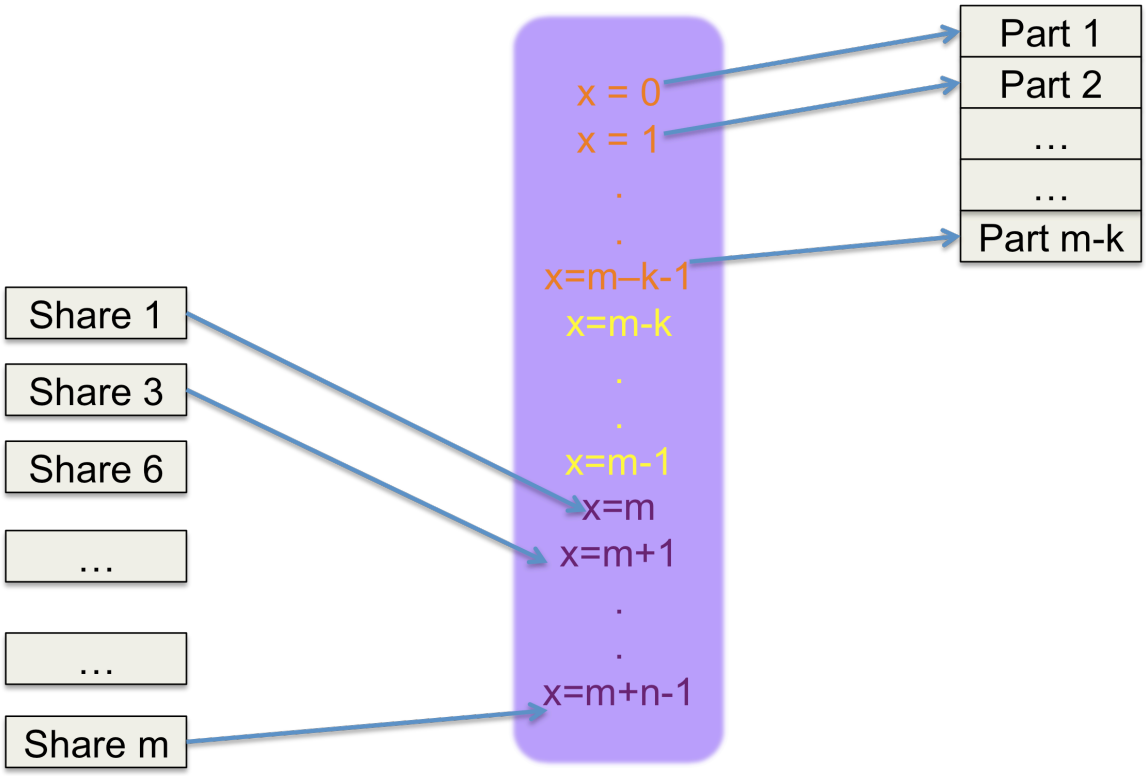


Figure 3 – Merging data using the Lagrange Interpolation Polynomial

Merging happens very similar to splitting. Any *m*out of *n*shares are used as inputs to generate the data parts at the output. Lagrange interpolation is used to create a polynomial passing through *m* shares. This polynomial is then evaluated at 0, 1, 2, … ,*m* – *k* – *1* points to recover the original data parts. These recovered parts are appended together in the increasing order of point value to obtain the merged data. It can be noted that the data at points,*m* – *k* – *1* to *m* – *1* correspond to the randomness that was added.

4.4 Integrity Check

The last part of the reconstruction procedure is the integrity check. This part validates that the shares haven't been altered intentionally or unintentionally.

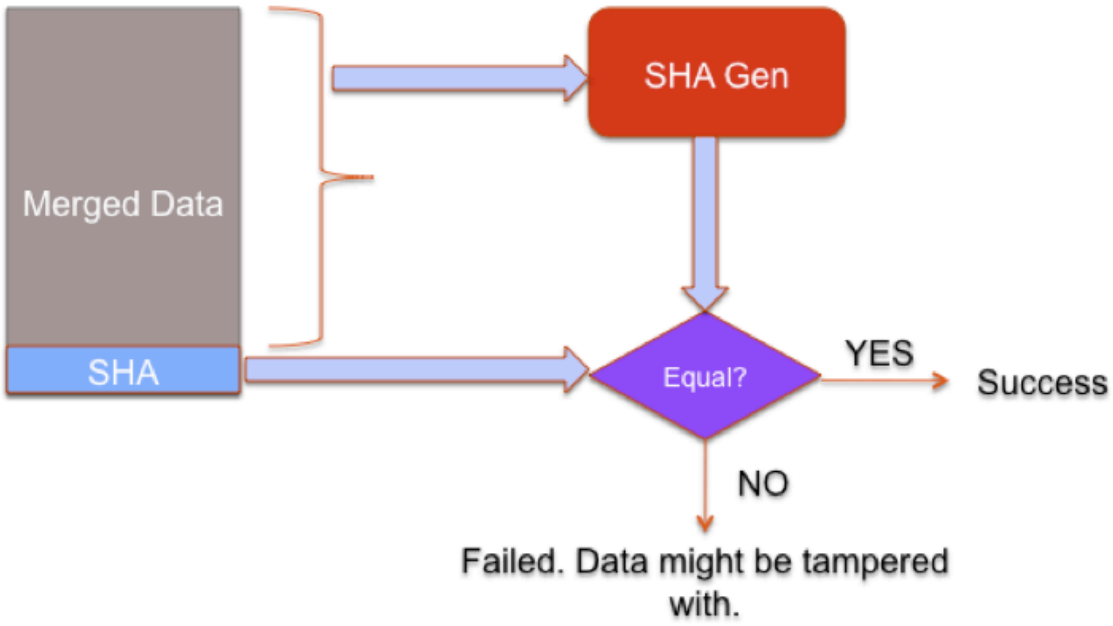


Figure 4 - Checking the integrity of the merged data

In this part, the merged data, without the SHA at the end of it is passed through the SHA generator to generate the new SHA. This new SHA is compared against the previously generated SHA to verify integrity. If the two SHAs do not match, there is a potential threat of intrusion.

Chapter 5

Implementation

We implemented the above algorithm into a Unix library using C programming language. Functionalities such as synchronous Splitting and Merging are supported. We worked on a GF(256) using Conway Multiplication for the field multiplication. The field implementation by David Madore was used for the field. We used an Open Source implementation of Secure Hash Algorithm (SHA) [3] library to provide support for data integrity. We obtain the random data from /dev/urandom.

5.1 Interfaces

The following structure is used as a configuration structure to set the values of *m*, *n* and *k*.

```
struct config {
     unsigned int m;
     unsigned int n;
     unsigned int k;
};
```

Where, *n* is total number of parts into which you want the data split, *m* is the total number of partsneededto recover the original data and *k* is number of random parts to use (this indicates the tolerance. Compromise of up to *k*parts will reveal no information about the original data whatsoever).

The following conditions apply:

1. It is required that $m \geq n$ and $n > k$.
2. *k* can be 0, but *m* and *n* can't.

If there is an error in configuration, INVALID_CONF error will be returned.

Error codes are enumerated as follows:

```
typedef enum {
     SUCCESS = 0,
     INVALID_CONF = 1,
     OUT_OF_MEM = 2,
     MERGE_CHECKSUM_ERROR = 3,
     DATA_ERROR = 4,
     IO_ERROR = 5,
     GOD_KNOWS_WHAT_WENT_WRONG_ERROR = 6
} SplitError;
```

The following structure is used to provide original data as input to split

```
struct req_orig_data{
     void *data;
     unsigned int data_size;
};
```

The following structure contains the split parts. "*parts*" is an array of *"m*" pointers, each pointer pointing to a single part of size *"part_size"* bytes. The number of pointers will be the same as the number of parts requested in the "*struct config* structure". If memory allocation fails for the parts, error code will be set to OUT_OF_MEM.

```
struct res_split_parts {
     struct part **parts;
     int part_size;
     SplitError err;
};
```

The following structure holds a part.Data for the parts will be allocated by the library.

```
struct part {
     unsigned char *part_data;
     int id;
};
```

*id*corresponds to the point at which the part was evaluated. This information is used in the construction of Lagrange Interpolation polynomial during reconstruction.

The following structure should be used as request to merge the parts into the original data. "*parts*" should have an array of "*num_parts*" pointers, each pointing to an individual part. "*part_size*" bytes will be read from each part.

```
struct req_orig_parts {
     struct part **parts;
     int num_parts;
     int part_size;
};
```

The following structure is used to return the merged data with size "*size*" bytes. Appropriate error code will be set.

```
struct res_merged_data {
     void *data;
     int data_size;
     SplitError err;
};
```

The following method is used to split data into parts. *data* will contain the Original data and its size, while *conf* will contain the requested configuration.

The *struct res_split_parts* will hold the pointers to the requested parts. If an error was encountered, appropriate error code will be set. If the error doesn't indicate SUCCESS, the contents of the parts will be unreliable (may contain some garbage value).

```
struct res_split_parts split(struct req_orig_data data,
struct config conf);
```

The following method is used to merge the shares into the original data and check for integrity. The *struct req_orig_parts* will contain the individual parts.

struct res_merged_data will contain the merged data along with an appropriate error code. If the checksum fails (which would indicate that one or more parts might be compromised), MERGE_CHECKSUM_ERROR error will be returned.

```
struct res_merged_data merge(struct req_orig_parts
parts, struct config conf);
```

5.2 Randomness

Randomness is a crucial resource for cryptography, and random number generators are therefore critical building blocks of almost all cryptographic systems. Be it to generate session and message keys for symmetric ciphers such as triple-DES or Blowfish, be it to generate seeds for routines that generate large primes (for RSA, say), be it for salts to combine with passwords during password storage, and so on, randomness is used everywhere. Similarly, randomness is crucial in the implementation of a threshold scheme or a (c, t, w) Ramp Scheme in general. The security of a Ramp Scheme heavily depends on the randomness that is used during the generation of the shares.

True random numbers are defined as follows: a number generated in the range 0, 1, …, $2^n - 1$ is “truly random” if an observer, with any amount of computational resourcecannot predict the number with probability better than $1/2^n$. True random numbers are, unfortunately, very difficult to generate, especially on computers, which are designed to be deterministic. Nature is the best source of true randomness. Temperature variations in the CPU, keystrokeson a keyboard, mouse movements and hard-drive activity are activities which are non-deterministic in nature and hence are a source of true randomness. Hardware based true random number generators use some physical phenomenon that is expected to be random and then compensate for possible biases in the measurement process. Software based random number generators on the other hand use computational algorithms that produce long sequences of apparently random results, which are in fact completely determined by a shorter initial value, known as a seed or

key. Such a number is called a pseudorandom number if the generator's output is not distinguishable from a truly random number by any polynomial time algorithm. The quality of a pseudorandom number generator depends on the algorithm used to generate random numbers and the source of the key. In a cryptographic sense (including that in Ramp Schemes), pseudorandom numbers (PRNGs) are acceptable only if they are indistinguishable in polynomial computational time from true random numbers. Such cryptographically acceptable pseudorandom numbers are called "cryptographically secure pseudo random numbers" (CSPRNG).

PRNG's such as Peter Gutmann's PRNG in Cryptlib [20], Colin Plumb's PRNG in PGP [21], RSAREF 2.0 PRNG [22], Yarrow [23] are established PRNGs. Design challenges of this thesis included obtaining good randomness but at the same time keeping the library small. Linux provides two sources of cryptographically secure randomness through two special files, /dev/random (blocking) and /dev/urandom (non-blocking). Since /dev/urandom is cryptographically secure and comes built into most flavors of Linux, randomness required to generate the shares has been obtained from /dev/urandom (stands for "unlocked" random) on Fedora Linux (/dev/random and /dev/urandom generally come built into most Linux flavors). Ituses environmental noise collected from device drivers and other hardware sources, primarily from keyboard, mouse, hard-disk activity and interrupts to generate randomness. The kernel maintains a pool of entropy (which is fed using the hardware activity), which the /dev/random uses to generate random data. Since the hardware activity during the boot sequence of a system is almost identical every time the system boots, the hardware activity during the initial phase is predictable and thus poses a security threat. To mitigate this threat, the Linux

kernel reads 512 bytes of data from /dev/urandom and stores it on the hard disk during shutdown and writes it back to the entropy pool at boot up. At any point, if the pool is empty, /dev/random will block until additional environmental noise is gathered and entropy is added into the pool. /dev/urandom on the other hand reuses the internal pool to produce more pseudo-random bits. Since /dev/urandom re-cycles the entropy without blocking, it is considerably faster than /dev/random. However, the quality of randomness generated by /dev/urandom degrades if the entropy available in the entropy pool falls below a certain level. More data about /dev/random and /dev/urandom can be found in [24].

Performance of the Ramp Scheme, as will be shown below, depends heavily on the rate of generation of random data. One way of improving performance is to use the randomness from /dev/random to seed a faster PSRNG and gather randomness from the PSRNG. This has been left as future work.

5.3 Experimental Setup

The experiment was run on Fedora Linux. Identical system states were maintained with respect to the available free memory and CPU before running the split and merge. Data Files of sizes 100KB, 500KB, 1MB, 5MB, 10MB, 20MB, 50MB, 100MB, 200MB and 500MB were used to test the performance of the split and merge functions of the library. 3 trials of the experiments corresponding to split and merge were run for each file and the times taken to split/merge in each trial were noted. The average of the 3 trials was used to plot the graph. The following code was inserted to time the execution of the split function.

```
clock_t start = clock();
```

```
res = split(data, conf);
printf("Done.\nTime elapsed: %f\n", ((double)clock() -
start)/CLOCKS_PER_SEC);
```

Similarly, the following code was inserted to time the execution of the merge function.

```
clock_t start = clock();
res = split(data, conf);
printf("Done.\nTime elapsed: %f\n", ((double)clock() -
start)/CLOCKS_PER_SEC);
```

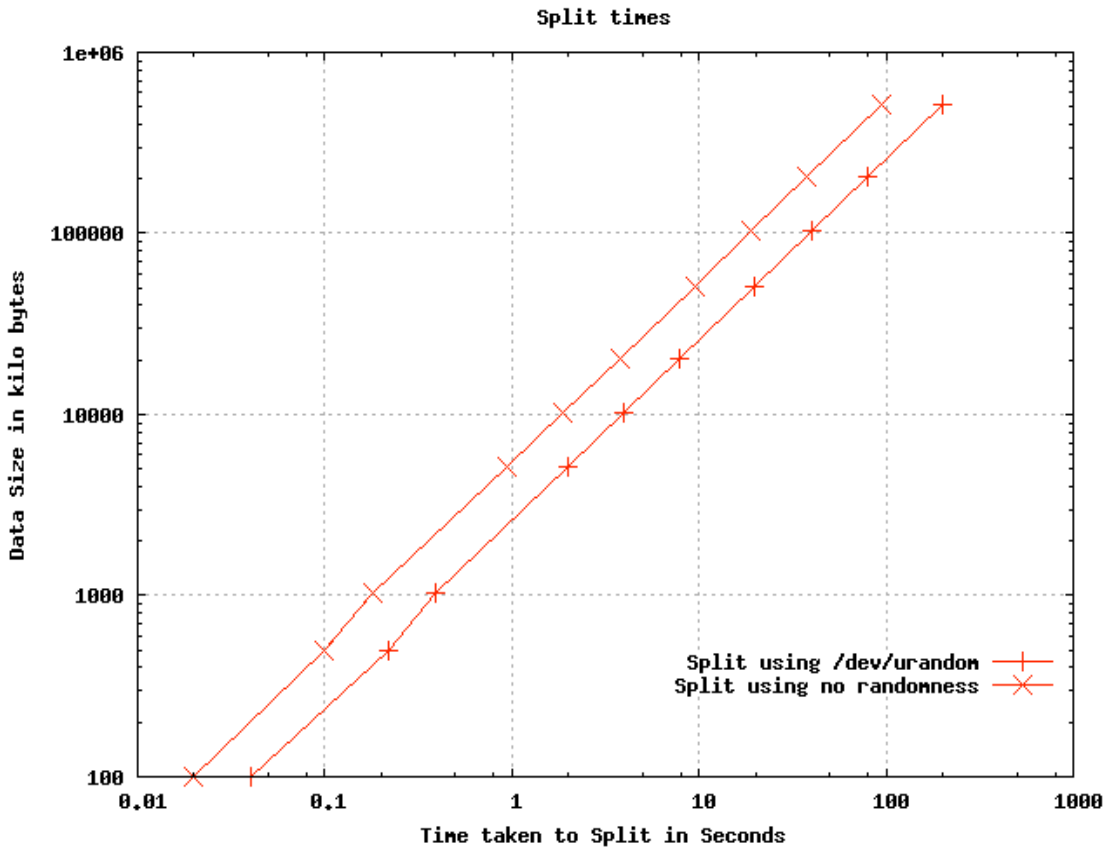
Different input files as listed above were used as inputs and the respective output times were recorded. A graph of Data File Size v/s Time Taken was plotted. The experiment was run under two setups:

1. Randomness from the /dev/urandom was used as the main source of randomness during split.
2. No randomness was used during splitting.

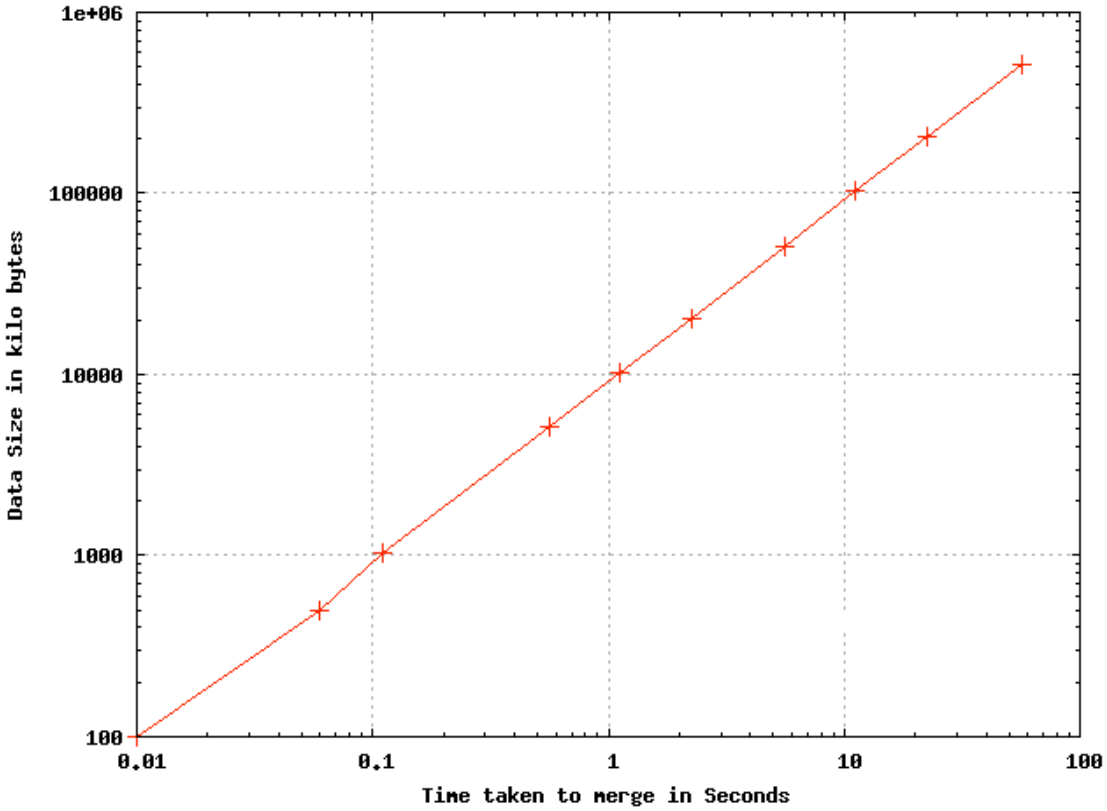
While obtaining the randomness from /dev/urandom, sufficient time was provided between trials to ensure that the entropy available in the system was above 3000. The entropy available was calculated using,

```
cat /proc/sys/kernel/random/entropy_avail
```

5.4 Results



Graph 3 - Split times using /dev/urandom against using no randomness



Graph 4-Time taken to merge

A graph of Size of the Data File versus Time taken was plotted. The times taken while using no randomness was significantly less than the time taken while using /dev/urandom as a source of randomness to split a data file of given size. In each trial, the entropy available in the system after running the experiment reduced to a value below 200, which is considered dangerously low.

It can be seen from the graph that Split function performs better when no randomness is used. Also, it can be seen that the graph is a straight line indicating that the time taken is directly proportional to the data size. This agrees with our runtime analysis of *O(n)* as mentioned in Chapter 3. The rate of splitting data which is the same as the slope of the line in the graph is found to be approximately 5.4 MB per second in the case of split using no randomness as compared to 2.6MB per second while using /dev/urandom as the main source of randomness.

On the other hand, the merge function has nothing do to with acquiring randomness. The rate of merging data equal to the slope of the line is approximately equal to 9.1 MB per second. As it can be seen this is very much higher than the data rate in the split functionality. This is clearly due to the fact of reading randomness during splitting. The faster performance can also be attributed to the fact that the number of points where the outputs are evaluated during merge is less than the number of points where output is evaluated during split.

Chapter 6

Conclusion and Future work

6.1 Conclusion

In this thesis, we highlighted the problems with conventional 1-N redundancy copy based systems. We then defined Ramp Schemes and showed how Ramp Schemes are suitable for data storage. In particular, we highlighted the following differences.

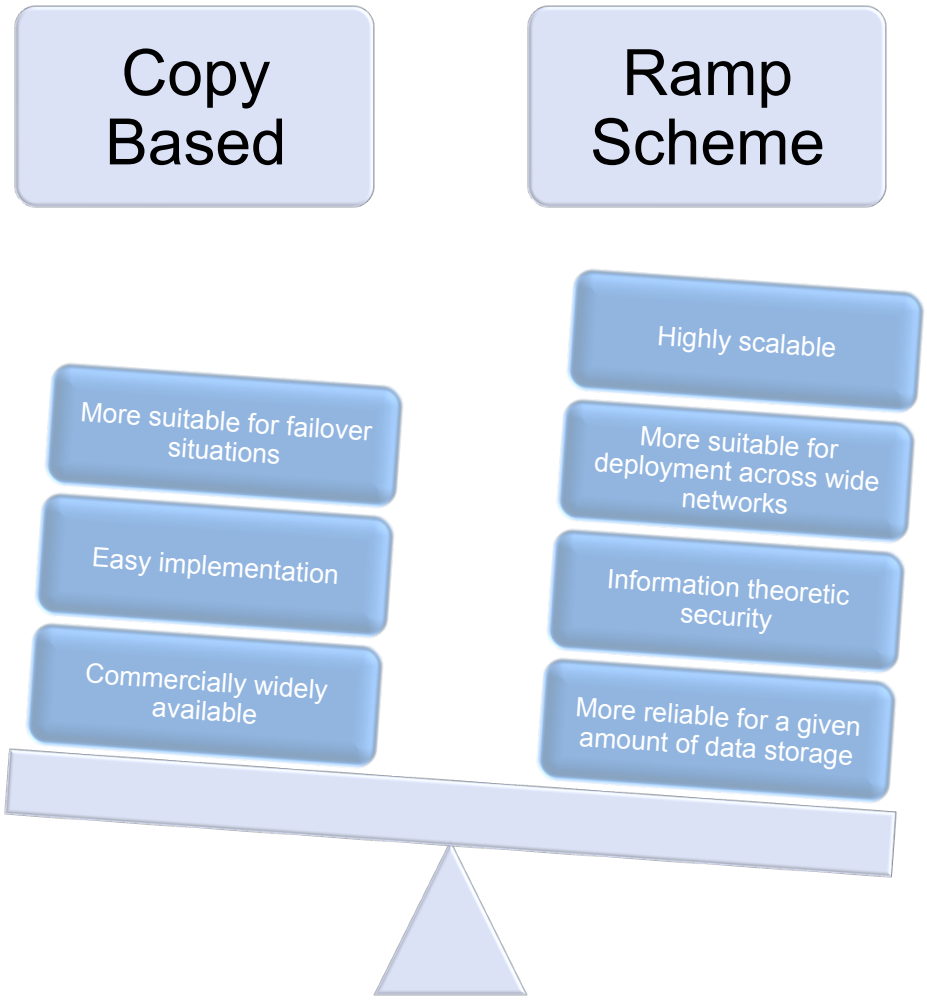


Figure 5 – Advantages of Ramp Schemes over Copy based systems.

We have shown through examples that a given amount of system reliability can be obtained using lesser data space using a Ramp Scheme when compared to a

conventional 1-N copy based systems. Furthermore Ramp Schemes provide Information Theoretic security, which is very valuable when managing sensitive information. We also presented our implementation of Ramp Schemes library in C programming language on Fedora Linux and discussed the performance of the implementation. We went on and established the importance of randomness in Ramp Schemes.

In all, using ramp schemes to secure data is a very cost effective way to achieve information theoretic security to confidentially disperse and store data.

6.2 Future Work

Current implementation of the Ramp Scheme library provides a synchronous interface to split and merge data, but when the data size is too large (in terms of GB's) splitting and merging would take a lot of time and the caller would be blocked till the operation is complete. Asynchronous interfaces for split and merge should be provided which would perform the operation and call back the caller to hand over the results.

The current implementation doesn't provide the user to use his/her own random number generation function. It would be a nice addition to the library to be able to accept a random number generator function as input to split.

Splitting in the current implementation is limited by the system's free memory (RAM). Meaning, if the system memory is 1GB, max size of data that can be split is around 500MB. This needs to be fixed by implementing split and merge using a pre-defined block size.

The Reliability analysis of Ramp Schemes and 1-N redundancy systems doesn't consider the repair and maintenance aspect of the individual nodes where the data is

housed. This brings in a whole new view from the point of Reliability Engineering and has been left as future work.

Bibliography

1. A. Shamir, *How to Share a Secret, Commun*. of the ACM, 22:612-613, 1979.

2. G. R. Blakley, *Safeguarding Cryptographic Keys*, AFIPS Conference Proceedings, 48:313-317, 1979.

3. HMAC-SHA1, Aaron Gifford, http://www.aarongifford.com/computers/sha.html

4. Carlo Blundo, Alfredo De Santis and Ugo Vaccaro, *Efficient Sharing of Many Secrets,* STACS 93. 10th Annual Symposium on Theoretical Aspects of Computer Science, Wurzburg, Germany, February 1993, Proceedings.

5. Douglas R, Stinson. *Cryptography Theory and Practice*, 3rd Edition, Chapman and Hall publication. ISBN: 1-58488-508-4

6. C. Blundo, A. De Santis and U. Vaccaro, *Randomness in Distribution Protocols*. Inform. Comput. pp 111-139, 1996.

7. C Blundo and B Masucci, *A note on the Randomness in Dynamic Threshold Scheme.* Journal of Computer Security, Vol 7, No. 1, 1999, pp 73-85.

8. James Middleton. *1024-bit encryption is "compromised"*. Vnunet.com, March, 26, 2002. http://www.vnunet.com/News/1130451

9. Robert M Gray, *Entropy and Information Theory*, Springer 1st Edition. ISBN: 978-0387973715.

10. Narjess Ayari, Denis Barbaron and Laurent Lefèvre, *On improving the Reliability of Internet Services through Active Replication*, 2008 Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies.

11. Nidhi Aggarwal, James E. Smith, Kewal K. Saluja Norman P. Jouppi, Parthasarathy Ranganathan, *Implementing High Availability Memory with a Duplication Cache, Microarchitecture*, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium.

12. Amos Beimel and Benny Chor, *Secret Sharing with Public Reconstruction*, IEEE Transactions on Information Theory, vol. 44, no. 5, sept 1998.

13. Igor A. Ushakov and Robert A. Harrison, *Handbook of reliability engineering*, American edition, New York : Wiley, c1994.

14. G. R. Blakley and C. Meadows, *Security of ramp schemes*, in Advances in Cryptology -- CRYPTO '84.

15. Wu, T C and Wu T S, *Cheating detection and cheater identification in secret sharing schemes*, IEEE Transactions on Computers and Digital Techniques 142(1995), 367-369.

16. D. R. Stinson and S. A. Vanstone, *A combinatorial approach to threshold schemes*, SIAM Journal of Discrete Mathematics**1** (1988), 230-236.

17. Narjess Ayari, Denis Barbaron, Laurent Lefèvre and Pacale Primet*Fault Tolerance for Highly Available Internet Services: Concepts, Approaches and Issues,* IEEE Communications Surveys, 2nd Quarter 2008, Volume 10, No. 2.

18. Hasard, Acryptographically secure Pseudo Random Number Gernerator. http://sourceforge.net/projects/freshmeat_hasard/

19. Aarona D. Gifford, *SHA 2* – An OpenSource implementation. http://www.aarongifford.com/.

20. P. Gutmann, *Software Generation of Random Numbers for Cryptographic Purposes,* Proceedings of 1998 Usenix Security Symposium, USENIX Association, 1998, pp. 243 257.

21. P. Zimmermann, *The Ofiicial PGP User's Guide,* MIT Press, 1995.

22. RSA laboratories, *RSAREF cryptographic library,* March 1994.

23. John Kelsey, Bruce Schneier and Niels Furguson, *Notes on Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator,*http://www.schneier.com/paper-yarrow.html.

24. Ziv Gutterman, Benny Pinkas, Tzachy Reinman, *Analysis of the Linux Random Number Generator*, http://www.pinkas.net/PAPERS/gpr06.pdf.