Secure Distributed Backup: Erasure Codes and Anonymous Message Delivery

by

Christopher Val Studholme

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Secure Distributed Backup:
Erasure Codes and Anonymous Message Delivery

Christopher Val Studholme
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2007

We address the problem of backing up one's important data files in an efficient, secure and robust manner by suggesting that copies of such files be sent to untrusted remote peers over the Internet. To provide incentive for such peers to store copies of your files, you would be required to devote some of your surplus disk space to storing copies of their files. In this manner, your data would be distributed widely about the planet and would thus be immune to loss due to all but the most global of disasters. We address a few of the major challenges that must be overcome to make such a network a reality.

First, we note that since remote peers are inherently unreliable, your data files must have some form a redundancy introduced to ensure complete recovery when needed; however, since wholesale replication of the files is neither space nor bandwidth efficient, we suggest the use of an erasure correcting code. We evaluate the use of a few existing codes for this task (Chapters 2 and 3) and propose a new class of codes, called *windowed erasure codes*, which aim to minimize time complexity while maintaining the greatest probability of successful recovery (Chapter 4).

To help ensure the security of one's data and to avoid the problem of an adversary engaging in a selective denial of service attack against a single participant, we propose the use of an anonymous message delivery technique for distributing file data (Chapter 6). The protocol we propose aims to be efficient in its use of network bandwidth while remaining secure in the face of clients who choose to collude against the others. Also, as a

necessary primitive, we develop a multiparty protocol for generating secret permutations (Chapter 5).

Finally, a practical backup application will require a method of encoding a hierarchy of files and the changes to those files over time into a sequence of fixed length blocks to be used with the above protocols. For this, we propose an algorithm based on a rolling checksum which is capable of identifying and removing redundant blocks of data located anywhere within a directory hierarchy (Chapter 7).

Acknowledgements

There are so many people to thank for their help and support during the research and writing of this thesis that I will be brief and only list the most notable. To the others, know that I do deeply appreciate your contribution.

I wish to thank my committee members: Charlie Rackoff, Peter Marbach, and Kumar Murty for all of their comments and direction. Also, Daniel Panario, my external examiner, for his careful reading of the thesis and detailed comments.

Finally, to Ian Blake, my supervisor and co-author on all of the papers to come out of my research, I could not have asked for a better mentor and greatly appreciate all you have done to aid in the completion of this thesis.

Contents

Chapter 1

Introduction

As computer technology continues to advance, more and more of our daily experiences are being digitized and stored within a computer. At first it was just text documents, personal and professional correspondence for example, that was being stored, but as computers become more powerful and storage plentiful, we are starting to store all of our photos, music, videos and other large datasets in the computer. Also, as always-on high bandwidth Internet connections become commonplace, we are beginning to manage our finances and other retail errands in a paperless way using the computer.

While this trend is considered to be a net benefit to our society, there is at least one serious maintenance issue that is often overlooked: ensuring that all of this digitized data is properly backed up. For most companies, the loss of their records or creative output can potentially mean the end of the company, and for this reason, companies typically employ or outsource someone to maintain backup copies of all this critical data. For the average individual, the loss of email or digital photos may have an emotional impact, but the event would hardly be life threatening. However, in addition to the short term effect of such a loss, there can also be a long term impact. The loss of family photos, for instance, will have an affect on future generations of that family, and the loss of data documenting events of historical significance or recording an important part of our culture can have a detrimental impact on society as a whole.

Despite all this, many individuals are unwilling to put in the time and expense needed to safeguard their personal data. How does one backup today's multi-hundred gigabyte hard drives? They could purchase a tape drive at relatively great expense, often more than the cost of a second hard drive. Instead, many are just buying a second hard drive, but many of the failure modes effecting a single hard drive will also take out drives near by. Most computers come with a DVD-ROM burner and one could backup their data to optical media, but the time commitment here is considerable, especially when one considers that a typical hard drive holds hundreds of gigabytes while each commodity optical disc holds less than ten. The other problem with backing up one's data to tape or optical disc is where to store the backup copy? Storing it in a box right next to the computer it came from may be no better than making the backup to a second hard drive.

We mentioned earlier the increased availability of high bandwidth Internet connections. Given this, Martinian, author of DIBS (Distributed Internet Backup System) [28], motivates the idea of backing up one's data to the Internet:

> Since disk drives are cheap, backup should be cheap too. Of course it does not help to mirror your data by adding more disks to your own computer because a fire, flood, power surge, etc. could still wipe out your local data centre. Instead, you should give your files to peers (and in return store their files) so that if a catastrophe strikes your area, you can recover data from surviving peers.

The idea is straightforward; however, several challenging technical problems need to be overcome to ensure such a data backup is both reliable and secure.

We envision a network that functions as follows. Each user's directory hierarchy full of files is efficiently encoded as a series of blocks. This encoding must, to the greatest extent possible, remove duplicate files, duplicate blocks within files, and efficiently encode the changes in the directory hierarchy over time. Since the data being backed up is to be stored on untrusted remote machines, the data must be encrypted using a password or key that is also backed up but not backed up in this same network. Since such a key is small and static its backup is not expected to pose a serious problem.

The remote peers that are to be storing this data may come and go as they please so we will require data blocks to be stored in a redundant manner to compensate for the unreliability of these peers. The blocks of data will be distributed to as many remote peers as is practical with some mechanism provided to later find and retrieve these data blocks. Retrieval will probably entail performing some search on encrypted data [41] to find an index that can then be used to retrieve the complete set of blocks.

To ensure the security of the backup, we have already mentioned the need to encrypt the data blocks. We would also require some mechanism to prevent data blocks from being linked to their source. This is to protect against selective denial of service and can be accomplished either by redistribution of blocks or by the use of anonymous message delivery techniques. Finally, to ensure peers are maintaining the data blocks they have been entrusted to store, regular random retrieval of small numbers of blocks should be employed to test them.

Researchers studying peer-to-peer networks have developed systems similar to the one we have described. Most notably, Anderson describes the eternity service [2], a network that uses redundancy and scattering techniques to preserve data essential to our culture and history. Another related network is the PAST persistent peer-to-peer storage utility [11]. The goal of this network is to create a robust distributed file system that allows users to store and access their data from any location. The network that we describe differs in that it is specific to the backup of users' important private data.

In this dissertation we are unable to provide solutions to all of the necessary building blocks needed for a secure distributed backup network, but we do tackle three of the problems: efficiently encoding a collection of directories and files that change over time as fixed size blocks, encoding those blocks with an erasure code to ensure that the data is still recoverable in the event that some of the data sent to peers cannot be retrieved, and the use of anonymous message delivery and retrieval techniques to protect against data loss due to the actions of malicious adversaries.

1.1 Deduplication

To break a set of files into blocks, one might simply use the `UNIX` tape archive utility `tar`. This tool first serializes all of the files found in a directory hierarchy, starting at a specified root directory, along with all the metadata associated with these files (owner and permission information) and then splits this stream of data into fixed size blocks suitable for storage on magnetic tape.

Newer versions of `tar` have the ability to route the data through the compression tools `gzip` or `bzip2` as a means of reducing local redundancies (those found within blocks up to about 1 megabyte in size); however, these compression tools fail to find the duplication of entire files (larger than 1 megabyte) or significant portions of files. This is where *deduplication* comes in. Also referred to as *commonality factoring* [18], this technique identifies instances where entire files, or large portions of files, have been duplicated and stored in distinct locations within a directory hierarchy.

Content-addressable storage file systems implement this idea at the whole file level. Instead of referring to files by name, they are referred to by some digest of their content (typically a cryptographic hash). Since identical files will have identical digests, they are detected and only stored once. In this type of filesystem, directories are simply a mapping from file name to digest (and perhaps additional metadata).

To detect if a portion of a file can be found in some other file, one needs deduplication at the block level. The naïve approach is to simply split each file into fixed sized blocks (say 512 bytes) and store each block in a content-addressable storage system. The file then becomes a list of the digests of its blocks. The problem with this naïve approach is that duplicate blocks are only found if they are aligned on block boundaries.

The deduplication technique we develop makes use of a variant of Tridgell's `rsync` algorithm [44, 45] to recognize previously seen blocks regardless of their alignment within the file currently being processed. Such detection is critical as typical file formats are rarely block oriented and files often change subtly by the addition or removal of a few bytes, perhaps near the beginning of the file, changing the alignment of all blocks that follow.

Deduplication may not be considered important when one thinks of backing up a single person's home directory once; however, if one is responsible for backing up the files of multiple people at regular intervals over a long period of time, deduplication is critical to reduce the storage space needed. Multiple people who are related in some way (e.g. members of the same family or department) are likely to have files in common. Consider what happens when multiple people collaborate on the creation of a document, or when one person finds an interesting file on the Internet and decides it must be emailed to everyone else. Also, if one is backing up the entire hard drive of multiple computers, the software installations on those computers will have many duplicate or similar files between them.

While the typical person's home directory does change over time, only a small number of their files will change from one day to the next, and even among the files that do change, the typical change likely only affects a small portion of the file. With a good deduplication scheme in place, people's entire home directories can be backed up every day while the amount of new storage space required each day is minimal.

Since the deduplication technique we describe splits a file hierarchy into fixed size blocks, these blocks can then be encoded using an erasure code and distributed over the

Internet using the anonymous message delivery, techniques that will be described in the following sections.

1.2 Rateless Erasure Correcting Codes

If data files are to be distributed to numerous nodes around the world over the Internet, then when it comes time to recover this data we do not want the temporary unavailability or permanent failure of some nodes to prevent successful recovery of the data. The most effective way to protect against such failure is to encode the fixed size data blocks produced by the deduplication algorithm with an erasure code.

An *erasure correcting code* is a transformation that can be performed on a message of k blocks to produce a message of more than k blocks with the property that, from any k of the latter blocks (or k plus a few extra blocks), the original k block message can be recovered. We will refer to the original k message blocks as the *input blocks* and the resulting encoded message as the *output blocks* or *coded symbols*. Typically, input blocks and output blocks are the same size.

If exactly k of the output blocks are required to recover the message, we say the code is *optimal*. On the other hand, if a small number of extra blocks, beyond k, are required, the code is said to be *near optimal*. We will refer to the expected number of extra blocks required as the *overhead* of the code.

The *rate* of an erasure code is the ratio of the number of input blocks to the number of output blocks. For some erasure codes (e.g. tornado codes), the rate is specified as a parameter of the encoder; however, with a *rateless* erasure code, also known as a *fountain code*, the encoder can produce an extremely large (practically infinite) number of output blocks. It is this latter class of erasure codes that we will be discussing in this dissertation.

The output blocks of a simple *random code* can be constructed by merely choosing input blocks at random (i.e. by flipping a fair coin) and summing (using sum modulo 2 — bitwise exclusive or) their contents. The resulting blocks of data along with metadata describing which input blocks were summed are assembled to form a binary matrix within the decoder and reduced using Gaussian elimination. Such a code is rateless and near optimal with an overhead of approximately 2 extra coded symbols (independent of k). The downside of such a code is the high decoder complexity of $\mathrm{O}\left(k^2\right)$ data block sums.

Luby greatly reduces decoder complexity with his *LT codes* [24], but this comes at the expense of overhead. Such a trade-off is justified, however, when one has a decoder implemented in an embedded device with limited computation and memory resources that can wait around for the additional output blocks necessary to recover the data. A good example of this is the transmission of updated map data through satellites to vehicles with on board mapping systems. Each vehicle will switch to the updated maps as soon as they have had enough exposure to the satellite data being streamed down to them.

While LT codes reduce decoder complexity to $\mathrm{O}(k \log k)$, *Online codes* [30] and *Raptor codes* [38] reduces complexity to linear time. Both of these codes employ an outer code to first expand the message by adding a few auxiliary blocks, creating an augmented message. Then, using a variant of LT codes as their inner code, they encode the augmented message. The decoder need not recover the entire augmented message as the outer code can make use of auxiliary blocks to recover any missing input blocks. The reduction in decoder complexity that results comes at the further expense of overhead.

Table 1.1 summarizes the important properties of these erasure codes, including our contribution, a new erasure code named *Windowed codes*. Described in Chapter 4, this new class of codes provides an improvement in decoder complexity over random codes without sacrificing the low overhead such codes achieve. While low decoder complexity is important to ensure the recovery of backed up data is possible with commonly available desktop and server class computer systems, we believe that maintaining the lowest possible overhead is also critical to ensure the data can be recovered even in the case where an absolute minimum number of output blocks are available.

1.3 Anonymous Message Delivery and Retrieval

When distributing blocks of data to peers it is important to ensure that the blocks are widely and uniformly distributed according to multiple regional classifications simultaneously. That is, the blocks should be sent to a variety of different geographic regions, countries, companies, *good guys*, *bad guys*, and through many different network carriers. We mention good guys and bad guys to illustrate that one might not be able to avoid sending blocks to their adversary (regardless of which one that is). Because of this, one must attempt to distribute the blocks uniformly to ensure that few blocks are distributed to an adversary and that the damage that can be done by that adversary is limited.

Erasure Code	Encoder Complexity	Decoder Complexity	Overhead
Random	$\mathrm{O}(\log k)$	$\mathrm{O}(k^2)$	$\mathrm{O}(1)$
Windowed	$\mathrm{O}(\log k)$	$\mathrm{O}\left(k\sqrt{k}\right)$	$\mathrm{O}(1)$
LT	$\mathrm{O}(\log k)$	$\mathrm{O}(k \log k)$	$\mathrm{O}\left(\sqrt{k}\right)$
Online/Raptor	$\mathrm{O}(1)$	$\mathrm{O}(k)$	$\mathrm{O}(k)$

Table 1.1: Complexity and overhead for various classes of erasure codes on k input blocks. Encoder complexity is per coded symbol while decoder complexity is overall. Overhead is the expected number of additional coded symbols required to successfully recover the input blocks.

Using anonymous message delivery techniques to distribute blocks of data helps prevent other parties from linking those blocks to each other and to their source. If an adversary can determine such links, the adversary may attempt to redirect or collect all blocks produced by an individual, thus nullifying any attempts at uniform distribution and reducing the probability of successful recovery.

Additionally, if the links between blocks and their source are to remain hidden even after blocks are fetched by the source (to recover from some sort of loss) or by some other party (perhaps as part of a content distribution network), some form of *private information retrieval* (PIR) must be used. In Chapter 6 we describe some of the existing PIR schemes.

For anonymous message delivery, we consider two existing techniques, mixnets and DC-nets, both introduced by Chaum in the 1980s. A *mixnet* [7] is a network of *mix nodes*, each of which engages in the activity of receiving messages, transforming (i.e. decrypting) and mixing the messages, and then sending the messages to the next node on the way to their destination. Obtaining anonymity from a mixnet requires that the user trust at least one mix node to not reveal the mapping from input to output. Because of this, a message is typically routed through many mix nodes, and the only way to really ensure it passes through at least one trusted node is for the user to operate a node themself. Passing messages through a mixnet has a high cost in terms of bandwidth with the total bandwidth being the size of the message multiplied by the length of its path through the mixnet. If every one of n users wants to send a message and every one of them also wants to be a node along the path, the total cost is n^2 times the size of a message.

The alternative for anonymous message delivery is a dining cryptographers network (DC-net) [8]. The motivation for this type of protocol is a group of cryptographers out at dinner. They are informed that an anonymous benefactor has paid for the dinner, and the group wishes to determine if this benefactor was one of them or an adversary. In this situation, the message can either be thought of as being a single bit, everyone submits a message and the sum modulo 2 is revealed, or the message is 1 and either exactly one of them has sent this message or none of them has. The latter interpretation is more insightful when one considers extending the protocol to multi-bit messages.

If exactly one party sends a message in a particular round, the message is received by all, but if more than one party attempts to disseminate a message, a collision occurs and the messages are garbled. To support multiple messages to be sent by multiple parties, the parties must first order themselves, but if this ordering is known to all parties, then any one of them can reveal the ordering of the transmitted messages. To deal with this problem, we develop, in Chapter 5, a multiparty protocol to generate a secret permutation of n parties. The result of the protocol is a permutation that is distributed among the parties is such a way that each party knows their position, but no others. In addition to being a useful component of a DC-net protocol, we also describe how secret permutation generation can be of use in online gaming.

With this problem solved, we address, in Chapter 6, the issue of communication complexity of a DC-net protocol. Existing protocols are no better than using a mixnet with a bandwidth requirement of n times the size of the message per message. If all n parties have a message to send, the total cost is n^2 times the size of a single message. Our protocol is aimed at reducing this requirement to the optimal cost of n times the size of a message (i.e. the total size of the messages). Our protocol has two phases, a setup phase and a message sending phases. If one considers only the communications complexity of the message sending phase, we achieve our goal; however, this comes at the expense of high setup cost (at least n times the total size of the messages to be transmitted). We propose that the remaining problem of high setup cost can be solved with a practical trapdoor discrete logarithm primitive. With such a primitive, setup cost becomes independent of the number of messages to be transmitted and our goal can be achieved.

1.4 Outline

This dissertation covers three major topics.

The next three chapters describe the various erasure codes introduced above. In Chapter 2 we show how to construct a random code, summarize the theory describing the rank properties of random matrices, and present our experimental results verifying this theory. Chapter 3 describes LT codes and our efforts to reduce overhead with an enhanced decoder. We conclude the discussion of erasure codes in Chapter 4 with a description of our windowed erasure codes and the theory governing them.

Chapters 5 and 6 deal with anonymous message delivery. The former describes our multiparty protocol for secret permutation generation and presents proofs of security for various security properties, while the latter demonstrates how to construct an efficient DC-net using a variant of the Pallier public key cryptosystem and what is required to make this protocol optimal.

The final topic covered, in Chapter 7, is the encoding of a set of files and directories that change over time into fixed size blocks by performing deduplication at the block level.

Chapter 8 further discusses how we believe these disparate topics fit together and what future work will be needed to make a secure distributed backup system a reality.

Chapter 2

Random Codes and the Rank Properties of Binary Matrices

2.1 Trivial Random Code

The construction of a rateless random erasure code is quite simple. Given a message consisting of k input blocks, we construct each output block by simply choosing a random subset of the input blocks and adding them together. As is typical, we will assume the blocks are each a fixed length string of bits and addition is the bitwise exclusive or (xor) operation. To choose a random subset, one flips a fair coin for each input block and then considers each block for which heads, say, is flipped to be included in the subset. The output block consists of the sum of the blocks in the subset along with whatever information is required to indicate the elements of the subset. Note that for large k, the probability that an empty subset is chosen is so small that we do not concern ourselves with this possibility.

To decode, at least k output blocks must first be collected. Each output block is considered to be a column of a matrix whose elements are chosen from $GF(2)$ and whose k rows are each associated with an input block. A '1' in a particular row of a column indicates that the corresponding input block is included in the sum for that output block. While this k row matrix is binary, we will consider the matrix to have an additional row (row $k+1$) that contains the data associated with each output block. Recovering the input blocks simply requires performing Gaussian elimination (via elementary column operations) to reduce the binary part of the matrix to the identity matrix. If successful, the $k+1$ row will hold the original input blocks. If, while reducing the $k \times k$ matrix, an all zero column is produced, that column must be discarded and replaced with some additional output block. If one is not available, decoding fails. The expected number of additional output blocks required is the overhead of the code, and, as we will soon calculate, is less than 2 for this trivial random code.

Clearly, the computational complexity of the encoder is $\mathrm{O}(k)$ per output block generated. Indeed, the expected number of data block additions required is $k/2$. For the decoder, Gaussian elimination requires $\mathrm{O}(k^2)$ elementary column operations in the general case, and each of these elementary operations involves adding two columns of length k. Furthermore, each elementary operation will require the addition of two data blocks. We will not concern ourselves with the cost associated with each elementary operation and simply quote decoder complexity in terms of the number of elementary operations required.

While the overhead of this type of random code is very low, both the encoder complexity and the decoder complexity are unacceptably high for most applications. Therefore, in the remainder of this chapter we will consider ways in which the encoder can be altered to reduce encoder complexity. This discussion will review known rank properties of random matrices and describe our experiments with alternate encoding strategies. In the following two chapters we show how decoder complexity may be reduced: first at the expense of overhead using LT codes and then without sacrificing overhead using windowed codes.

2.2 Rank properties of binary matrices

A random binary matrix is a matrix whose elements are chosen from $GF(2)$ via some random process. The matrices described in this chapter will typically have k rows and either k or n columns. If the matrix is $k \times n$, where $n \geq k$, we will often refer to $m = n - k$ as the number of *extra columns*.

One method of randomly generating such a matrix involves choosing each matrix element independently with some probability p being the probability of selecting a 1. The probability p will have one of the following forms: a constant such as 0.5, $p = (c \log k)/k$ for some constant c, or $p = d/k$ for some constant d.

The other method of interest involves generating each column of the matrix independently via a *degree distribution*. This distribution is on the possible column weights ranging from 1 to k (inclusive) and is sampled once for each column. Note that we intentionally exclude weight 0 columns. With a degree (column weight) chosen, a column having exactly that weight is then selected uniformly from the set of all such columns. Throughout this dissertation, we will use the terms degree and column weight interchangeably.

2.2.1 Probability of Full Rank

The property of most interest to us is the rank of a matrix as a function of the mean column weight and the number of extra columns. In particular, we wish to determine how the mean column weight effects the probability of full rank in a $k \times k$ or $k \times n$ matrix. In addition to this, we are also interested in a related property, *overhead*. This is the expected number of extra columns required to achieve full rank (as a function of the mean column weight).

In this section we will only concern ourselves with matrices generated via the first method described above. Each matrix element is chosen independently with the probability of choosing a 1 being p.

If $p = 0.5$, the probability that a $k \times n$ matrix ($n \leq k$) has rank n is

$$\prod_{i=1}^{n} \left(1 - \frac{1}{2^{k-i+1}}\right) \quad . \tag{2.1}$$

The intuition behind this expression is as follows. The probability that a single column with k rows is non-zero is $1 - 1/2^k$. Assuming we have one non-zero column, the probability that a second column is neither all-zero nor a copy of the first column is $1-1/2^{k-1}$. With $n-1$ columns having rank $n-1$, the probability that the n^{th} column is not dependent on the previous columns is $1 - 1/2^{k-n+1}$. Notice that if $n = k$, this last probability is $1/2$. The expression above is simply the product of all these probabilities. For $n \geq k$, the asymptotic probability of full rank, as $k \to \infty$, is (from Kolchin [19])

$$Q_m = \prod_{i=m+1}^{\infty} (1 - \frac{1}{2^i}) \ , \tag{2.2}$$

where $m = n-k \geq 0$. In practice, when $k \geq 10$, the two results agree to within 1%. Note that $Q_0 = 0.288788\ldots$ is the (asymptotic) probability of full rank for a $k \times k$ random binary matrix. Figure 2.1 shows a comparison between simulation and theory for $k = 100$ and $k = 500$. Note that for the various choices of p used, no difference can be seen. This tells us that the probability of full rank is independent of the mean column weight for p above some threshold. We will soon see that this threshold is roughly $p = (2 \log k)/k$.

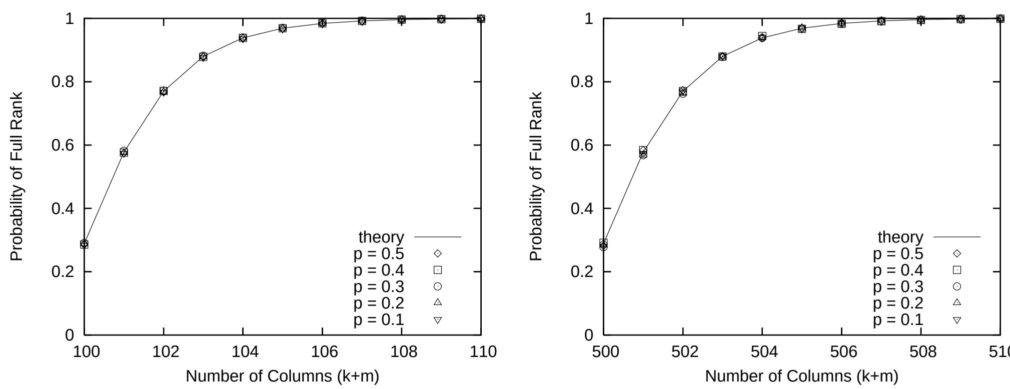

Figure 2.1: Probability of full rank for constant p as the number of extra columns increases. The theory line is given by (2.2). Each data point is the result of simulation using $50,000$ matrices (for $k = 100$) and $5,000$ matrices (for $k = 500$).

As p decreases (below $(2 \log k)/k$), all-zero columns and rows are increasingly likely and the presence of such a column or row is not accounted for in the above expressions. To see how all-zero rows might form, consider that when tossing k identical balls into an equal number of bins, one expects to have to throw $k \log k$ balls to ensure each bin has at least one ball with high probability. While our problem with matrices is not identical, we estimate that to ensure each row of our $k \times k$ matrix has at least one 1, we expect to have to "toss" $k \log k$ 1's at the matrix. This corresponds to $p = (\log k)/k$, indicating that this probability is, in some way, critical.

If we assume Q_m is the probability of full rank given $k + m$ non-trivial columns with no all-zero rows and $1 - (1 - p)^k$ is the probability that a random column is non-trivial, then

$$\sum_{j=0}^{m} \binom{k+m}{j} (1-p)^{kj} (1 - (1-p)^k)^{k+m-j} Q_{m-j} \tag{2.3}$$

is the probability that a $k \times (k + m)$ matrix with no all-zero rows has full rank. Further, since $(1 - (1 - p)^{k+m})^k$ is the probability that there are no all-zero rows, we get

$$(1 - (1-p)^{k+m})^k \sum_{j=0}^{m} \binom{k+m}{j} (1-p)^{kj} (1 - (1-p)^k)^{k+m-j} Q_{m-j} \tag{2.4}$$

as the probability of full rank in a $k \times (k + m)$ matrix. Note here that we have assumed there is no correlation between instances of all-zero columns and all-zero rows. This assumption is reasonable provided k is large. Also note that (2.4) applies even when p

is large; however, (2.2) is easier to evaluate. Figure 2.2 compares simulation results to theory.

Equation (2.4) also holds for very low values of p; however, as p approaches $1/k$, and for lower values of p, a simple approximation to the above expression is possible. When $p = 1/k$ the expected number of 1's per column is 1. If we have a matrix with exactly one 1 per column and no all-zero rows, that matrix is a permutation matrix and is guaranteed to have full rank. Also, if we have at most one 1 per column, then "no all-zero rows" implies "at least k non-trivial columns", and thus, full rank. Therefore, we can estimate the probability of full rank as simply

$$\left(1 - (1-p)^{k+m}\right)^k \ , \tag{2.5}$$

the probability of no all-zero rows. Figure 2.3 demonstrates the effectiveness of this approximation for $p = d/k$ and $d \leq 2$.

2.2.2 Overhead

For $p \geq (2 \log k)/k$, Q_m is a good approximation for the probability of full rank. From this, we can compute the expected number of extra columns needed to achieve full rank, the overhead, as

$$\bar{m} = \sum_{i=0}^{\infty} (1 - Q_i) = 1.60669515\ldots \ . \tag{2.6}$$

By substituting (2.4) in place of Q_m in the above equation, the overhead as a function of p and k can be calculated, say $\bar{m}_{p,k}$. The resulting equation, however, is quite difficult to simplify, but despite this, we make a few observations:

- for fixed k, $\bar{m}_{p,k} \to \infty$ monotonically as $p \to 0$,
- for $p = (c \log k)/k$ and $c < 1$, $\bar{m}_{p,k} \to \infty$ as $k \to \infty$, and
- for $p = (c \log k)/k$, $c \geq 2$ and $k > 10$, $\bar{m}_{p,k} < 2$.

The fact that overhead remains less than 2, and independent of k, for mean column weights as low as $2 \log k$, suggests that the encoder complexity can be reduced to $\mathrm{O}(\log k)$ without sacrificing overhead. Simply select each input block with probability $p = (2 \log k)/k$. Unfortunately, despite the low weight matrix generated, decoder complexity with Gaussian elimination will still be $\mathrm{O}(k^2)$. In the following two chapters we address methods to reduce this complexity. For the remainder of this chapter, we explore an alternate method for generating columns.

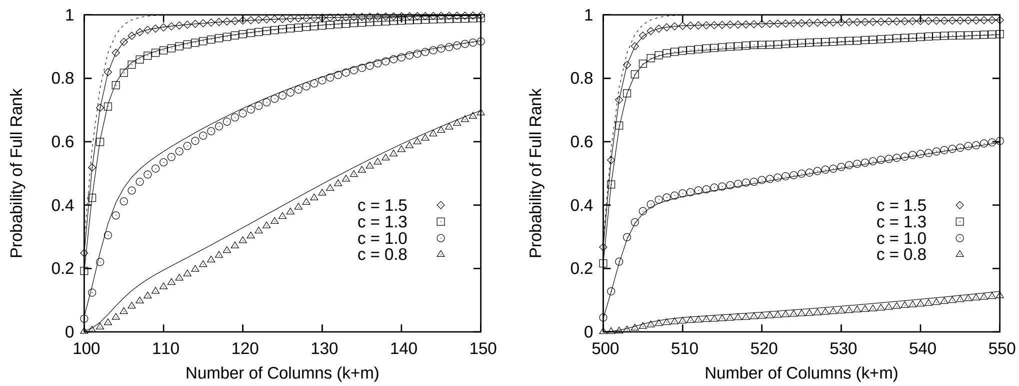

Figure 2.2: Probability of full rank for $p = (c \log k)/k$ as the number of extra columns increases. Each data point is the result of simulation using $50,000$ matrices (for $k = 100$) and $5,000$ matrices (for $k = 500$). Each solid line is computed using (2.4). For comparison, the dotted line was calculated using (2.2).

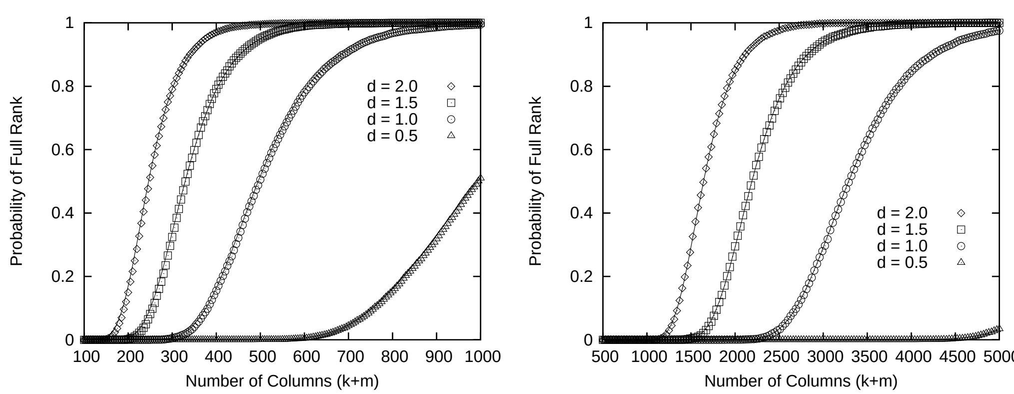

Figure 2.3: Probability of full rank for $p = d/k$ as the number of extra columns increases. Each data point is the result of simulation using $10,000$ matrices (for $k = 100$) and $5,000$ matrices (for $k = 500$). Each solid line is computed using (2.5).

2.3 Dependence on Distribution

With our understanding of matrices generated from a probability p, we now turn to matrices generated from a degree distribution. As mentioned above, these matrices are generated by randomly and independently choosing columns. First a degree is chosen from the desired degree distribution, and then a column with that number of 1's is chosen uniformly from the set of all columns with that degree.

Again, we are interested in the probability of full rank as a function of the mean column weight. Here we test the hypothesis that, with minor exceptions, the particular degree distribution used does not matter. Only the mean column weight has a significant influence on the probability of full rank.

Note that *mean column weight* and *mean degree* are interchangeable and we will label this quantity σ. Also note that $\sigma = pk$, or $p = \sigma/k$, relates following experiments to those of previous sections.

2.3.1 Wedge Distribution

If σ is an integer, one might consider using a degree distribution that results in all columns having the same weight σ. This will work fine if σ is odd; however, if all columns have even weight, it can be proven that the matrix cannot have full rank (regardless of how many columns it has).

To handle non-integer choices of σ, one might consider using only two degrees, d and $d+1$, where $d = \lfloor \sigma \rfloor$. This approach is similarly flawed in the case where σ is very close to an even integer.

To avoid these problems with all even degree, we define the wedge distribution as follows. Three degrees are used, $d-1$, d, and $d+1$. The center degree d is chosen to be σ rounded to the nearest integer. If σ is an integer plus 0.5, either choice for d will work. Probabilities are assigned to these degrees subject to the constraints: the mean degree must be σ and the resulting columns are 50% even weight and 50% odd weight. There is a unique distribution satisfying these constraints.

In general, all three degrees will have non-zero probabilities; however, when σ is an integer plus 0.5, only two of the degrees are used. The probability assigned to degree d is always 0.5. Obviously, the probabilities assigned to $d-1$ and $d+1$ sum to 0.5.

See Figure 2.4 for simulation results using the wedge distribution.

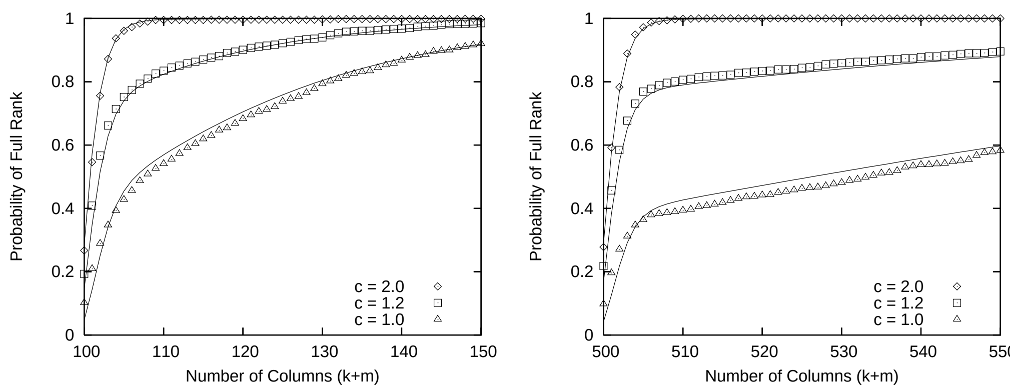

Figure 2.4: **Wedge distribution.** Probability of full rank for $p = (c \log k)/k$ as the number of extra columns increases. Each solid line is computed using (2.4).

2.3.2 Uniform Distribution

For a uniform distribution we want every degree to have the same probability of being chosen. Assuming $\sigma \leq k/2$, we assign the degrees 1 to roughly 2σ a common probability that is roughly $1/(2\sigma)$. To ensure the mean degree is exactly σ, we will need to assign the last degree some other non-zero probability.

To formalize this, we find the largest integer e such that there exists probability p_e with $0 \leq p_e \leq 1$ and

$$\sigma = ep_e + \frac{1-p_e}{e-1}\sum_{i=1}^{e-1} i = \frac{e\,(1+p_e)}{2} \quad . \tag{2.7}$$

Figure 2.5 shows simulation results for the uniform distribution.

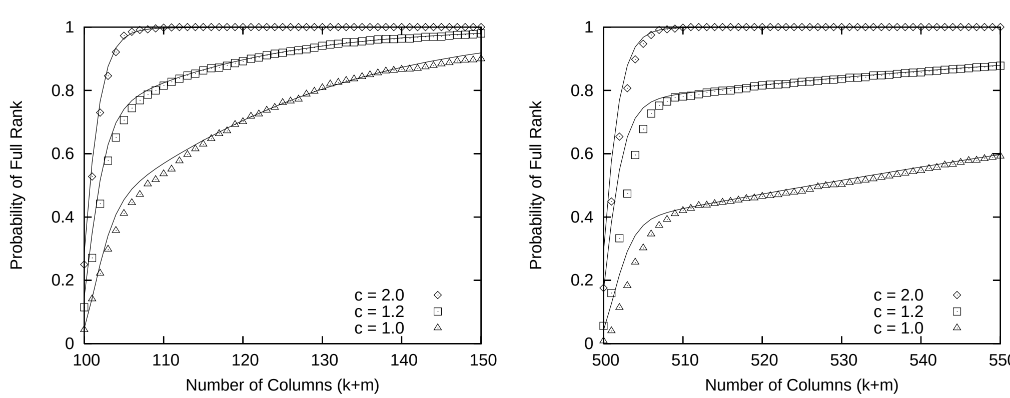

Figure 2.5: **Uniform distribution.** Probability of full rank for $p = (c \log k)/k$ as the number of extra columns increases. Each solid line is computed using (2.4).

2.3.3 Horseshoe Distribution

For a horseshoe distribution our goal is make half of the columns have a low degree (degree 2 or 3) and the other half a high degree (roughly 2σ); however, to ensure the mean is exactly σ and that we don't have problems with all of the columns having even degree, we will need to choose two degrees for the high degree columns.

There are two variants of the horseshoe distribution. The degree 2 variant has 50% degree 2 ($p_2 = 0.5$) and the other 50% split between two high odd degrees (d and $d+2$). We choose d and the probabilities such that

$$\begin{aligned} \sigma &= 2p_2 + dp_d + (d+2)p_{d+2} \\ &= 2 + d/2 - 2p_d \ . \end{aligned} \tag{2.8}$$

We notice in Figure 2.6 that for $c = 2$ there is a noticeable deviation from theory for m up to about 5 or 6. The reason for this is as follows. There are $\binom{k}{2} \approx k^2/2$ possible degree 2 columns, and, by the birthday paradox, within a random selection of columns whose number is approximately the square root of this value, two identical columns are likely to be found. What this means is that the probability that $k/2$ randomly selected degree 2 columns are not independent is actually quite high. For $k = 500$, this probability is 0.622316, and this accounts for the one or two extra columns needed to achieve full rank.

For the degree 3 horseshoe variant, we don't have to worry about the possibility of all even degree columns so we choose two high adjacent degrees (d and $d+1$). Here, d

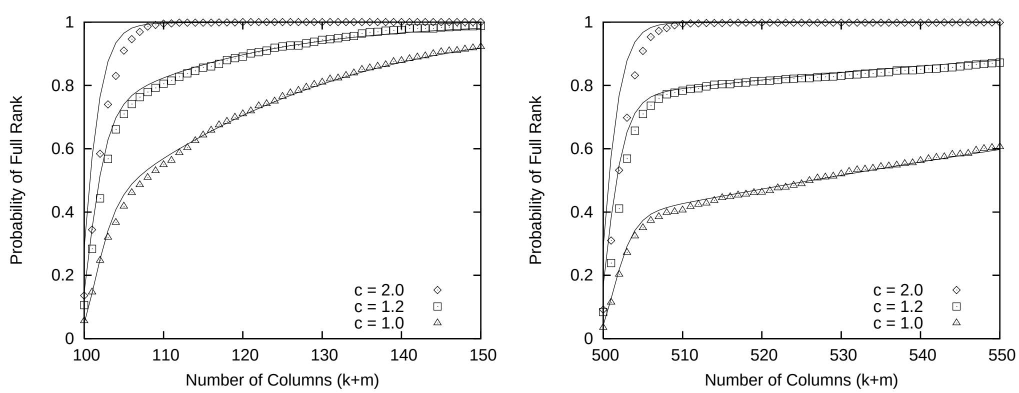

Figure 2.6: **Horseshoe (degree 2) distribution.** Probability of full rank for $p = (c \log k)/k$ as the number of extra columns increases. Each solid line is computed using (2.4).

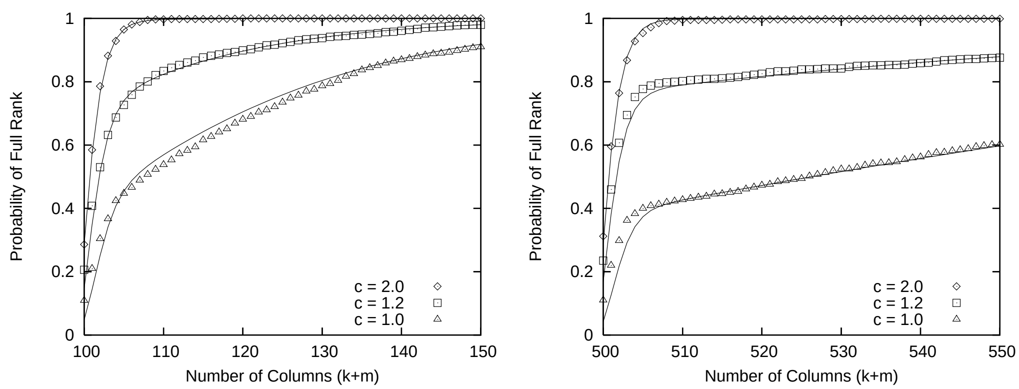

Figure 2.7: **Horseshoe (degree 3) distribution.** Probability of full rank for $p = (c \log k)/k$ as the number of extra columns increases. Each solid line is computed using (2.4).

and the probabilities are chosen such that

$$\begin{aligned} \sigma &= 3p_3 + dp_d + (d+1)p_{d+1} \\ &= 2 + d/2 - p_d \ . \end{aligned} \tag{2.9}$$

We also notice, in Figure 2.7, that this variant does not have the deviation from theory seen with the degree 2 variant.

2.3.4 Other Distributions

We have also tested a few other distributions, including a semi-uniform distribution that is a mixture of the horseshoe distribution (for the degree 2 or degree 3 columns) and the uniform distribution (for all of the other columns), and found that these experiments also give results that are consistent with the theory. We conclude from these experiments that the rank properties of these matrices have very little dependence on the specific distribution used.

In the next chapter, however, we will encounter a distribution that yields rank properties that vary significantly from theory. Then, in Chapter 4, we develop a distribution based on windowed columns whose rank properties only match theory as long as the critical parameter exceeds a specific lower bound.

Chapter 3

LT Codes

LT codes are a class of rateless erasure correcting codes developed by Luby [24] with the goal of reducing decoder complexity. In this chapter we explore the possibility of using LT codes as the erasure code for our distributed backup application. While LT codes has the same encoder complexity ($\mathrm{O}(\log k)$ per output symbol) as the random codes discussed in the previous chapter, LT codes have a decoder complexity, for complete decoding, that is much lower at $\mathrm{O}(k \log k)$ (compared to $\mathrm{O}(k^2)$ for a random code). We will also briefly describe subsequent work by Shokrollahi [38] and Maymounkov [30] which further reduces these complexities to $\mathrm{O}(1)$ and $\mathrm{O}(k)$, respectively.

The price to be paid for such low decoder complexity is higher overhead. With LT codes, overhead is $\mathrm{O}\left(\sqrt{k}\right)$, and with Raptor/Online codes, overhead is $\mathrm{O}(k)$. For a distributed backup application this high overhead may be unacceptable to those who need to recover their data, even if it is at a high cost. After describing the construction of these codes, we present our analysis of the rank properties of the matrices that are generated by the LT encoder and describe an enhanced version of the decoder which, while being more complicated, reduces expected overhead without increasing decoder complexity.

3.1 Soliton distribution

In [26], Luby *et. al.* describe the construction of rateless erasure codes in terms of a bipartite graph where the left vertices are identified with the k input blocks and the right vertices are thought of as parity checks on the input blocks (coded symbols). These are generated as follows: for a given degree distribution ρ on the integers $1, 2, \cdots, k$, the distribution is sampled to give an integer d, $1 \leq d \leq k$. The corresponding coded symbol is formed by choosing d input symbols uniformly at random and summing them (via bitwise exclusive or). To decode, if a sufficient number of coded symbols are obtained, the process starts by choosing a coded symbol of degree 1 (i.e. a coded symbol corresponding to a right vertex of degree 1). The value of the coded symbol is transferred to the corresponding input block, whose value is then transferred to all coded symbols containing it, and all the corresponding edges are removed from the graph. If a new right vertex is now of degree 1, the process continues.

Decoding continues until either it completes with all input blocks recovered or there are no right vertices of degree 1 at some stage, in which case decoding fails. To minimize the probability of the latter, the distribution ρ is chosen carefully. Luby suggests [24] that, in theory, the best distribution to use is the *soliton* distribution given by

$$\rho(i) = \begin{cases} 1/k & i = 1 \\ 1/i(i-1) & 2 \leq i \leq k \end{cases} . \tag{3.1}$$

As required, $\sum \rho(i) = 1$. The theoretical reasoning for this distribution is sound and one can easily see how the first few steps of the decoder will proceed. Since $\rho(1) = 1/k$, the decoder expects to see one degree 1 coded symbol within the first k coded symbols collected. This coded symbol provides the data for one input block. Also, since $\rho(2) = 1/2$, it is expected that this one input block will be included in one of the expected $k/2$ degree 2 coded symbols collected. This means we expect one degree 2 coded symbol to be reduced to degree 1. Note that each degree 1 coded symbol will also cause reductions in the degree of the higher degree coded symbols, thus producing more degree 2 code symbols. This *ideal soliton* distribution is designed such that at each stage of the decoding process, the expected number of degree 1 coded symbols available is exactly one.

In practice, if the expected number of degree 1 coded symbols is one, the probability that this number is less than one at some particular stage is quite high (on the order of 0.5), and when this happen decoding fails. To remedy the situation, Luby describes a *robust* soliton distribution designed to ensure that the expected number of degree 1 symbols at each stage (the *ripple*) is greater than one. The robust soliton distribution is $\mu(i) = (\rho(i) + \tau(i))/\beta$ where

$$\tau(i) = \begin{cases} R/ik & 1 \leq i < \lfloor k/R \rfloor \\ R\log(R/\delta)/k & i = \lfloor k/R \rfloor \\ 0 & i > \lfloor k/R \rfloor \end{cases} , \tag{3.2}$$

$R = c\log(k/\delta)\sqrt{k}$ and $\beta = \sum(\rho(i) + \tau(i))$. The constant δ is the allowable failure probability of the decoder when given $K = k + \mathrm{O}\left(\log^2(k/\delta)\sqrt{k}\right)$ coded symbols and $c > 0$ is a suitable constant. These two constants determine the expected number of extra symbols needed (the overhead) and the variance in the number of symbols. For details of this mapping, see [24] and [27]. All of our experiments were done using $\delta = 0.5$ and $c = 0.03$; parameters suggested in [27] as striking a good balance between overhead and variance.

The complexity of the encoder and decoder are easy to compute. The mean degree of the robust soliton distribution is $D = \log(k/\delta)$, and since the number of additions the encoder is required to perform is 1 less than the degree, encoder complexity is $D - 1$ additions per coded symbol. For the decoder, each data block addition reduces the degree of some coded symbol by 1. With k symbols, the sum of the degrees is kD, and therefore, the complexity is $k(D - 1)$. In practice, decoder complexity may be slightly higher because more than k coded symbols are required to decode and some block additions may be needlessly performed.

3.1.1 Linear Time Codes

Shokrollahi and Maymounkov, independently, with their Raptor [38] and Online [30] codes, respectively, reduce both encoder and decoder complexity by eliminating the $\log k$ factor in each. These two codes are very similar and both utilize a distribution that is similar to the ideal soliton distribution as a part of their construction, so we will describe briefly how they achieve this remarkable result.

These codes can be considered to be split into two (nested) erasure codes: an *outer code* and an *inner code*. The outer code is a fixed-rate erasure code applied to the input k block message to produce a small number of *auxiliary blocks* which, along with the k input blocks, are fed as input to the inner code. The construction of these auxiliary blocks is performed as a pre-computation before any coded symbols are output and requires time $\mathrm{O}(k)$. The inner code is rateless and very similar to the LT codes described above with the only major difference being that the degree distribution has a mean that is constant.

For Online codes, the following distribution is suggested for the inner code. Given parameters δ and ϵ, first compute $F = (\log\delta + \log(\epsilon/2))/\log(1-\delta)$ and then define

$$\rho(i) = \begin{cases} 1 - \frac{1+1/F}{1+\epsilon} & i = 1 \\ \frac{1-\rho(1)}{(1-1/F)\; i(i-1)} & 2 \le i \le F \end{cases} . \tag{3.3}$$

With this distribution, the decoder (of the inner code) will be able to recover a fraction $1 - \delta$ of the k input blocks in time proportional to $\log(1/\epsilon)k$. The outer code ensures that the remaining input blocks can be recovered with a linear time computation on the auxiliary blocks that have also been recovered by the inner code.

The summarize this result, we reproduce here, using our terminology, Theorem 2 from [30]:

Theorem 3.1.1 *(Maymounkov): For any message of size k input symbols, and any parameter $\epsilon > 0$, there is a rateless locally encodable code (right distribution) that can recover the input symbols from any $(1 + \epsilon)k$ coded symbols with high probability in time proportional to $k \log(1/\epsilon)$.*

3.2 Implementation and Overhead

We have implemented LT codes in `C++` and tested them for both performance and overhead. Figure 3.1 shows net overhead (the mean number of extra columns as a ratio of k) as a function of k for both an encoder using the ideal soliton distribution (left) and one using the robust soliton distribution with $\delta = 0.5$ and $c = 0.03$ (right). We have also included in these figures the overhead that results if one uses Gaussian elimination (GE) as the decoder and the overhead that results from a purely random code as described in the previous chapter. Recall the latter being an overhead of approximately 1.606 extra coded symbols.

Comparing the overhead that results from the standard LT decoder versus Gaussian elimination as the decoder is interesting for the following reasons. First, Gaussian elimination will always recover the file if sufficient information (in an information theoretic sense) is present, while the LT decoder may fail early. The curves in Figure 3.1 show how much lower the overhead could be if the LT decoder could run to completion in all cases where the information needed to recover the file is present.

Also, the shapes of the Gaussian elimination curves are of interest. In the left graph, we see that the GE curve almost parallels the LT decoder curve. In the next section we will see that this is because, contrary to the conjecture stated in the previous chapter, using the ideal soliton distribution to generate a binary matrix results in a matrix that behaves quite different from a random matrix with the same mean weight. Hence the need for the robust soliton distribution.

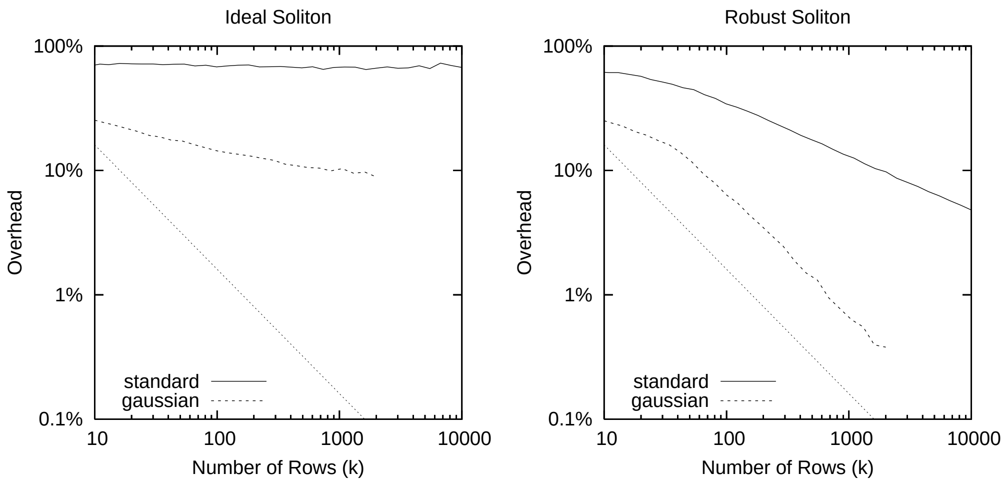

Figure 3.1: Overhead as a function of the number of input blocks. The left graph gives results achieved using the ideal soliton distribution as the encoder, while the right graph is for the robust soliton encoder. In both graphs, results achieved by the standard decoder and a decoder using Gaussian elimination are given. Note that the curves for Gaussian elimination end at $k = 2,000$ because beyond this value the combination of high complexity and the matrix no longer fitting in our processor's cache makes the time required to generate each data point quite excessive. The bottom broken line in each graph is $1.606/k$, the lowest overhead we expect from a random code.

In the right graph we see that the GE curve parallels the line representing an overhead of 1.606. We estimate that this GE curve is asymptotic to approximately $7/k$, giving an overhead of 7 extra coded symbols. This leaves a considerable gap between the overhead achieved with the standard LT decoder and GE, and later in this chapter we describe our attempt at improving upon the standard decoder. Interestingly, as we will see, while our improved decoder does not close the gap for the case of the robust soliton encoder, it does close the gap for the ideal soliton case.

3.2.1 The Ideal Soliton Mystery

In the previous chapter we generated binary matrices using a variety of distributions and found that all of the resulting matrices have rank properties that are almost identical to random matrices with an equivalent weight. The ideal soliton distribution; however, does not follow this pattern. Figure 3.2 compares the probability of full rank for matrices

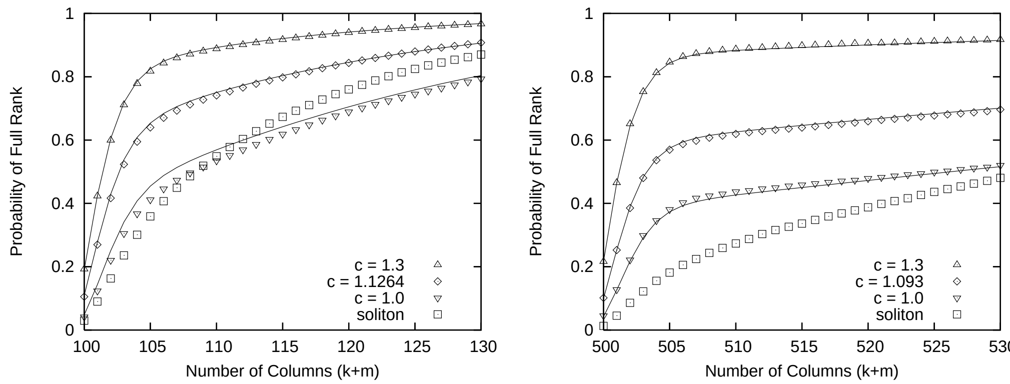

Figure 3.2: Probability of full rank in matrices generated using the ideal soliton distribution and random matrices with $p = c\log(k)/k$ as the number of extra columns increases. The solid lines are computed using (2.4). Note that the mean degree of the soliton distribution is approximately $1.1264 \log k$ when $k = 100$ and $1.093 \log k$ when $k = 500$.

generated using the ideal soliton distribution versus random matrices with an equivalent weight (probability p).

As can be seen, the probability of full rank is significantly lower for the matrices generated using the ideal soliton distribution. We assume that, if these graphs were extended to the right, one would see that the soliton curve (boxes) is asymptotic to the corresponding random curve (diamonds). Also, since the overhead is equal to the area above the curve (up to the line $y = 1$), we can see that the overhead of a code based on the ideal soliton distribution will be much higher than that of the corresponding random code.

In an attempt to understand why a matrix generated using the ideal soliton distribution differs from a random matrix, we have performed two different experiments.

In the first, we show a progression from random matrix to ideal soliton by first starting with the degree 2 horseshoe distribution described in Chapter 2. The results can been seen in Figure 3.3. In this graph, the *semi2* data set is essentially the same as the horseshoe distribution shown in Figure 2.6. Each *semi-j* distribution is defined as

$$\text{semi-j}(i) = \begin{cases} 1/k & i = 1 \\ 1/i(i-1) & 2 \le i \le j \\ \sigma_0 & i = d \\ \sigma_1 & i = d+1 \\ 0 & \text{otherwise} \end{cases}, \tag{3.4}$$

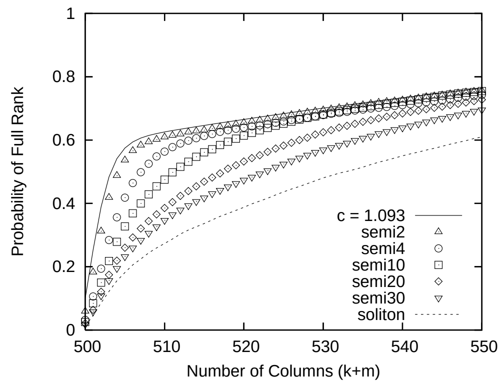

Figure 3.3: Progression from random matrix to one generated using the ideal soliton distribution. The solid line was calculated using (2.4) (with $p = c\log(k)/k$). The soliton distribution has a mean degree of $1.093 \log k$ when $k = 500$. Each data set is a semi-soliton distribution. It is soliton-like for degrees up to and including the number specified, and then has two consecutive large degrees to ensure the correct mean. Note that the *semi2* distribution is almost identical to the horseshoe distribution (degree 2 variant).

where $d > j$, σ_0 and σ_1 are chosen such that the distribution sums to 1 and has the correct mean ($1.093 \log k$ in Figure 3.3).

The other experiment we performed was to test the hypothesis that having a large number of columns with a weight below the mean is the reason for the deviation. For this experiment we define a biased horseshoe distribution. For this distribution, instead of having 50% low degree and 50% high, we have a fraction w of low degree columns and $1 - w$ high degree columns. Figure 3.4 shows the results of this experiment. Note that when $k = 500$ the ideal soliton distribution results in 83.5% of the columns having a degree less than the mean. When comparing the soliton distribution to the degree 2 variant of the horseshoe distribution we see that with approximately 70% to 80% low weight columns the results match quantitatively with the soliton data, but with a curve of significantly different shape (poor qualitative match). With the degree 3 variant, we need about 94% low degree to get this same sort of quantitative match.

Unfortunately, neither of these experiments provide much insight into the mystery of the ideal soliton distribution. The first experiment appears to show that no individual degree value is responsible for the observed performance. The distribution *semi30* agrees with the ideal soliton distribution over a large fraction of the overall weight and yet we still observe a noticeable difference in its performance. The second experiment shows

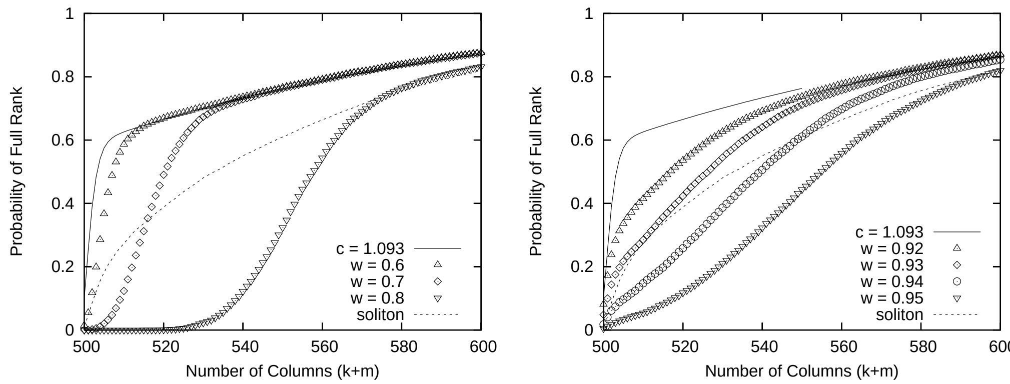

Figure 3.4: Probability of full rank in matrices generated using the ideal soliton distribution versus the biased horseshoe distribution. In both graphs $k = 500$. In the graph on the left, the degree 2 variant of the horseshoe distribution was used with w being the probability of a degree 2 column. The graph of the right uses the degree 3 variant. In both graphs, the solid line is computed using (2.4). Note that the mean degree of the soliton distribution is approximately $1.093 \log k$ when $k = 500$.

that while having a large fraction of the overall weight placed on low degree columns can adversely effect the probability of full rank, the qualitative differences between this scenario and the results for the ideal soliton distribution indicate that this is not the reason for the latter's performance.

3.3 Improving the LT Decoder

The standard LT decoder, while very simple and fast, is limited by its reliance on degree 1 coded symbols (right vertices in the bipartite graph mentioned earlier). If at any point during decoding, a degree 1 coded symbol is not available, decoding fails. Furthermore, a degree 1 coded symbol is required to begin the decoding process, and with the ideal soliton distribution's weight on degree 1 being only $1/k$, only one such coded symbol is expected after k have been received. For this latter reason, the robust soliton distribution puts considerable extra weight on the generation of degree 1 symbols.

Our motivation for developing an enhanced LT decoder is the observation that both the ideal soliton and robust soliton distributions put half of their weight on the generation of degree 2 coded symbols. Any special use that can be made of these degree 2 coded symbols is likely to be quite effective given the large number of such symbols.

To see how we might make use of these symbols, consider a graph similar to the one depicted in Figure 3.5. In this graph, the vertices correspond to the input blocks (the left vertices in the bipartite graph) and the edges correspond to degree 2 output symbols (a pair of edges in the bipartite graph connected to a single degree 2 right vertex). We will call this graph the *degree two* graph.

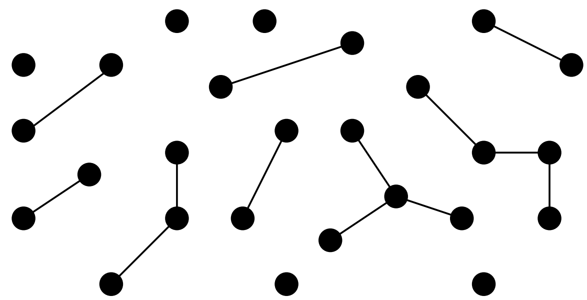

Figure 3.5: An example of a degree two graph. Each vertex corresponds to an input block and each edge corresponds to a degree 2 output symbol.

Two important observations are made. First, a cycle within this graph indicates that a redundant output symbol has been received. To see this, note that the data associated with any edge not currently found in the graph but for which some path exists between its end points can be generated by the decoder by simply summing the data blocks associated with each edge of this path. Because of this, the decoder need not bother adding any edge that will induce a cycle (such output symbols should be immediately discarded as redundant), and therefore, the graph will remain acyclic.

The second observation has to do with large degree coded symbols. Symbols with degree 3 or greater can be considered to be hyperedges within this graph. While we saw that a new degree 2 coded symbol with both vertices being part of the same connected component is redundant, a new degree 3 or greater coded symbol with two vertices that are both members of the same connected component can be reduced. For each pair of connected vertices found within a large degree coded symbol, the reduction consists of two steps: (i) the data associated with the coded symbol is adjusted by adding all of the data blocks associated with edges along the path between the two vertices and (ii) the two vertices (two edges in the bipartite graph) are removed, thus reducing the degree of the coded symbol by 2. It is this second observation that makes this degree two graph so useful.

This degree two graph has a few interesting properties. The graph can be considered to be a random graph where the probability of adding any particular edge is uniform, and random graphs are known to undergo a phase transition at a certain threshold edge count. The phase transition is the event where, for certain monotone-increasing properties, the random graph is very unlikely to have the property when the number of edges is slightly less than the threshold, and yet, very likely to have the property when the number of edges slightly exceeds the threshold. For our degree two graph an examination of the size and distribution of the connected components is of interest.

With zero edges, the graph consists of k distinct connected components, i.e. each vertex is its own component. With each non-redundant degree 2 coded symbol received and corresponding edge added to the graph, the number of distinct components is reduced by one as two previously distinct components are connected. Note that the maximum number of edges (not inducing a cycle) is $k-1$. A phase transition for this type of graph is known to occur at $k/2$ edges. With fewer than this threshold, the connected components are all roughly the same size and obviously small given that the mean component size is less than 2 vertices. Beyond the threshold, however, a single large connected component will certainly form. The ramifications of this large connected component are two fold: (i) new edges are far more likely to form a cycle (if α is the fraction of vertices found in the large component then α^2 is the asymptotic, as $k \to \infty$, probability that a new edge forms a cycle within this component) and (ii) coded symbols with degree 3 or larger are far more likely to be reduced as a result of having two or more vertices in this connected component.

In light of these graph properties, we find it interesting that the ideal soliton distribution assigns a weight of $1/2$ to the degree 2 symbols. Any higher probability will dramatically increase the overhead of the resulting code, while a lower probability would certainly reduce the effectiveness of the standard LT decoder and prevent our degree two graph from aiding the new decoder we are about to describe.

One more interesting result of our analysis is the effect of our decoder on even versus odd degree coded symbols. Since reduction of a large degree coded symbol by use of the degree two graph always results in a degree reduction of 2 or some multiple of 2, even degree coded symbols remain even degree. Likewise for odd degree symbols. While our new decoder no longer requires degree 1 coded symbols to get started nor a degree 1 coded symbol at every step of the decoding process, at least one odd degree coded symbol is required at some point. We mentioned above that only $k-1$ edges can be added to the graph without inducing a cycle and since even degree coded symbols can only be reduced to even degree, no more than $k - 1$ even degree coded symbols will be useful (non-redundant). In fact, it can be proven that a matrix whose columns (or rows) are all of even weight must be singular. This proves that not only can our decoder not decode when all of the coded symbols are of even degree, but no decoder (not even Gaussian elimination) could succeed at this task.

3.3.1 Implementation

We have implemented this new decoder as a highly recursive algorithm in `C++`; the core of which consists of three primary methods:

- **Process degree one:** given a degree 1 symbol for which the corresponding input block is not already known, copy the now known data and, using the degree two graph, compute the data associated with all input blocks that are part of the same connected component. All large degree (degree ≥ 3) coded symbols that have a vertex in this connected component are reduced and new degree 2 coded symbols may result.

- **Process degree two:** given a degree 2 symbol whose two vertices are not members of the same connected component, connect the two previously disjoint components by adding a new edge. If the data for one of the vertices is known, the coded symbol can be reduced to degree 1 and processed as such instead. All large degree coded symbols that have at least one vertex in this new component must be scanned to determine if any of them can be reduced as described next.

- **Reduce large degree:** New large degree coded symbols are scanned to determine if any of the vertices are already known, or if two or more vertices can be found in a single connected component. After any such reductions, the symbol may now be of degree 1 or 2 and, if so, can be processed as such.

The recursive nature of the algorithm is obvious and recursion can become quite deep during the later stages of decoding.

Figure 3.6 shows the improvements in overhead achieved by our new decoder. The curves for the standard decoder and Gaussian elimination (GE) are the same as those from Figure 3.1. As we mentioned earlier, this new decoder almost completely closes

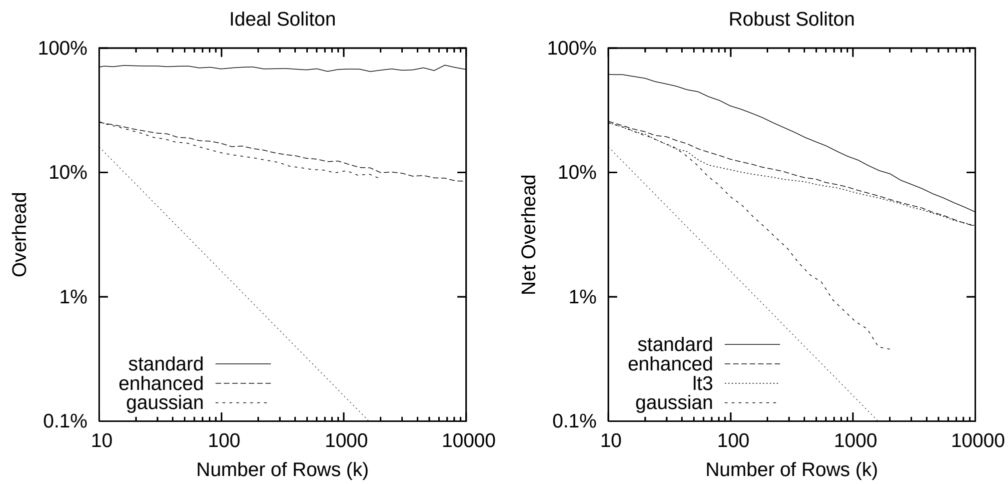

Figure 3.6: Overhead as a function of the number of input blocks. The left graph gives results achieved using the ideal soliton distribution as the encoder, while the right graph is for the robust soliton encoder. In both graphs, results achieved by the standard decoder, the enhanced decoder, and a decoder using Gaussian elimination are given. The bottom broken line in each graph is $1.606/k$, the lowest overhead we can expect from a random code.

the gap between the standard decoder and GE when decoding from the ideal soliton distribution; however, when decoding from the robust soliton distribution, there is a big improvement for lower values of k but unfortunately no improvement asymptotically. Also seen in the right graph is a curve labelled "lt3"; the decoder for which will be described below in Section 3.3.3.

3.3.2 Complexity

The complexity of the standard LT decoder is very easy to understand. At each step, a degree one coded symbol (known input block) is added to some higher degree coded symbol to reduce the degree of the latter by one. At the completion of decoding, all coded symbols will be of degree one. Therefore, to compute the decoder's complexity, one simply sums the degree of all coded symbols and then subtracts one per coded symbol. If the mean degree is $c \log k$, then the work done by the decoder is $(c \log k - 1)k$ block additions.

The only complication to what has just been said is the additional data block sums involving coded symbols that will eventually be found to be redundant. Since the number of such blocks is $O\left(\sqrt{k}\right)$, this additional work is $O\left(\sqrt{k}\log k\right)$ data block additions, which is asymptotically negligible. Furthermore, the decoder may "cheat" by first determining which blocks are redundant and disposing of them before beginning the data block additions. We say cheat because making this determination requires $O\left(\sqrt{k}\log k\right)$ work, and therefore, this strategy is only useful if the data blocks are quite large.

We now argue that the new decoder we have been describing has essentially the same complexity as the standard LT decoder. The new decoder has three types of operations, corresponding to the three methods described earlier.

- As with the standard decoder, when a degree 1 coded symbol is used to reduce other coded symbols, each data block addition results in a degree reduction of 1.
- Adding a new edge to the degree two graph can either require no data block additions or several, depending of which strategy discussed below is used.
- When a degree 3 or larger coded symbol is reduced by summing the path between two connected vertices, the number of additions required is equal to the length of the path and the degree is reduced by 2.

The two strategies mentioned are:

- **Lazy strategy:** no reductions are done to the degree two graph when new edges are added, and thus, no data block additions are required at this stage. The ramifications of this strategy are that when a degree 3 or larger coded symbol must be reduced by summing the path between two connected vertices, this path may be quite long (note Figure 3.7 left hand side) and the number of block additions required is equal to the length of the path. To prevent repeating these summations in the future, some method of caching previous results should be employed.

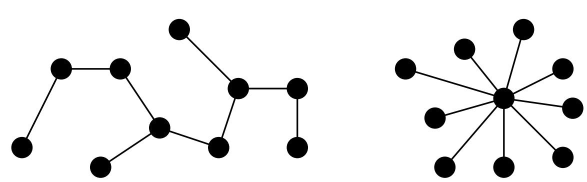

Figure 3.7: Two views of a connected component in the degree two graph. With the lazy new edge strategy, arbitrary trees like the one on the left may form, while the non-lazy strategy ensures that all trees have diameter 2 as in the right hand graph.

- **Non-lazy strategy:** each time a new edge is added to the degree two graph, the new connected component formed is reduced to the minimum diameter graph shown on the right hand side of Figure 3.7. This is possible because the decoder can compute any edge for which a path between the end points already exists. If both existing components are already in this form, the number of block additions required to transform the new connected component to this form is equal to the size of the smaller of the two original components (give or take 1 depending on the exact position of the new edge's vertices). The advantage of this approach is that all large degree coded symbol reductions require at most 2 block additions (resulting in a degree reduction of 2).

Our implementation of this new decoder utilizes the non-lazy strategy and Figure 3.8 shows the work required compared to the work done by the standard decoder. For small values of k, the fact that this new decoder is slightly faster is most likely due to the reduced overhead and not having to waste time processing some of the redundant coded symbols. From the graphs it is obvious that the new decoder is slower by only a very small constant amount. This slowdown may be eliminated by switching to some form of the lazy strategy, but we have not tested this.

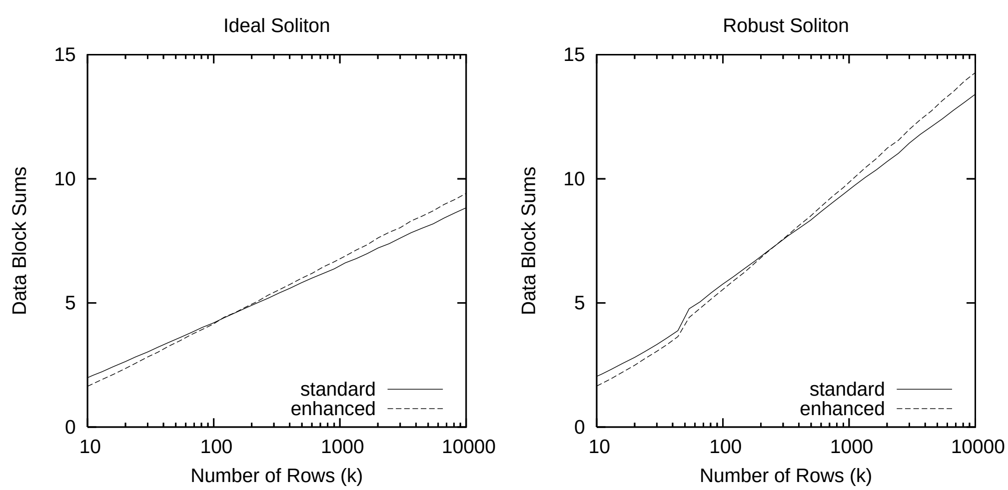

Figure 3.8: Number of data block sums (per block) required to decode as a function of the number of input blocks. The left graph gives results achieved using the ideal soliton distribution as the encoder, while the right graph is for the robust soliton encoder.

3.3.3 Beyond Degree Two

One might wonder, "if making special use of degree 2 coded symbols results in lower overhead, perhaps making special use of degree 3, 4 or higher coded symbols will result in a further reduction?" To answer this question for degree 3 coded symbols we have developed a decoder that makes special use of such symbols. Our decoder looks for opportunities to combine a degree 3 symbol with a degree $d \geq 3$ symbol to produce a new degree $d - 1$ symbol. To see how this works, consider the sum of a degree 3 with degree 5:

$$\begin{array}{ccccccccccc} & & B & & & & & + & E & + & F \\ A & + & B & + & C & + & D & & & + & F \\ \hline A & & & + & C & + & D & + & E & & \end{array}$$

The result is a new coded symbol of degree 4.

To find such opportunities for reduction we must compare each high degree coded symbol against all other such symbols to determine if any pair has at least two input blocks in common. For a newly received degree 3 coded symbol the process is as follows: (i) starting with one of the vertices of the degree 3, flag all coded symbols of degree 3 or greater that are connected to this vertex, (ii) then do the same for the other two vertices but before flagging check to see if the coded symbol is already flagged. If a coded symbol is seen connected to two (or more) vertices then an opportunity for reduction has been found. When processing a coded symbol with a degree greater than 3 the algorithm is the same except that only symbols with a degree of exactly 3 are flagged. The algorithm gets complicated, however, when one considers that rescanning is required anytime a large degree coded symbol is reduced to a degree of 3 or less.

The results of this even further enhanced decoder are found in the right hand graph of Figure 3.6 as the "lt3" curve. There is obviously no point in running this test for the ideal soliton distribution. Despite our test code reporting that large numbers of coded symbols were reduced in this way, the results show that the improvement in overhead is minor and probably not worth the effort. Certainly, any attempt to make special use of even larger degree coded symbols is not likely to result in further improvement.

We do consider, however, one possible avenue for further improvement. If one were to restrict the vertices included in a degree 3 or larger coded symbol to some small window of input blocks, the probability that two such coded symbols have two vertices in common can be increased. For example, suppose the input blocks are ordered, arranged in a circle, and windows of size $\sqrt{k}$ are chosen. Without this type of restriction, the probability that two randomly chosen degree 3 coded symbols have 2 vertices in common is roughly k^{-2}, but with this restriction the probability increases to $k^{-3/2}$. In the next chapter we explore another way in which such a window can be utilized to construct a low overhead erasure code.

Chapter 4

Windowed Erasure Codes

We now describe a new class of erasure correcting codes, called windowed codes, which have an overhead that is independent of the number of input blocks, down to as few as 2 extra coded symbols, and have a decoder complexity that is $O(k^{3/2})$ for k input blocks. This overhead is identical to that achieved by a general random code (as described in Chapter 2) and considerably better than that achieved by LT codes or any of its successors. On the other hand, decoder complexity is much better than that of a general random code but not as low as can be achieved by LT codes.

The only other work related to the problem of interest here is contained in the patent [25]. The problem of interest there is the encoding of very large files. To prevent the entire file being required to be in main memory at once, a sliding window approach is taken and thus has a similar view to this work. However, we are not aware of any performance or analytical results being available relating to that patent.

Windowed codes are easy to construct requiring only the uniform distribution during encoding and Gaussian elimination to decode. Note that while Gaussian elimination requires $O(k^2)$ column operations to invert a random matrix, when used to invert the windowed matrices introduced below this time is reduced to $O(k^{3/2})$.

These windowed codes will be of use in any application where achieving the lowest possible overhead is essential, even if it comes at the expense of requiring moderate processing power to decode. Note that for values of k up to $100,000$ decoding can be completed in a matter of minutes on a typical desktop machine, and that decoding codes of length $k = 1,000,000$ is feasible with higher end hardware or if one is willing to wait longer for the result.

4.1 Windowed Matrices

A binary windowed matrix is constructed from a set of windowed columns where each windowed column can be thought of as a random column with all of the 1's restricted to a small number of consecutive rows (the *window*). Recall from Chapter 2 that a random matrix with p as little as $p = (2\log k)/k$ will have similar rank properties to a random matrix with $p = 1/2$. With the former, the expected number of 1's per column is $2\log k$ which suggests it may be possible to confine the 1's to a window with as few as $4\log k$ rows; however, we have found that as a result of the random distribution of the windows within the columns, a larger window is required to maintain rank properties similar to that of a random matrix.

Figure 4.1 demonstrates the effectiveness of this restriction to a fixed size window for various window sizes of the order $\mathrm{O}\left(\sqrt{k}\right)$. The open squares and triangles show the results for window sizes of $2\sqrt{k}$, $1.5\sqrt{k}$ and $\sqrt{k}$. Clearly, $w = 2\sqrt{k}$ is necessary for good results. In the $k = 2,500$ and $k = 10,000$ graphs, the solid triangles show that for $w = 2\sqrt{k}$, a densely packed window with a mean column weight of $\sqrt{k}$ is unnecessary. The *low weight* data series (filled triangles) were generated using a mean column weight of $2\log k$ $(p = (\log k)/\sqrt{k})$.

In Chapter 2 we saw that, for a random matrix, the probability of full rank is Q_m, where m is the number of extra columns beyond k, and that, on average, only $1.606695\ldots$ extra columns are required to achieve full rank (rank k). Since our windowed matrices (with window length at least $2\sqrt{k}$) have a probability of full rank that is very similar to Q_m, these matrices will also have a high probability of full rank with only about 2 extra columns. This very low overhead is a significant advantage of the erasure code construction we give later in this chapter.

4.2 Rank properties of windowed matrices

In this section we first prove an upper bound on the probability of full rank for windowed matrices. This proof establishes a minimum window length of $\sqrt{k}$ to achieve a reasonably

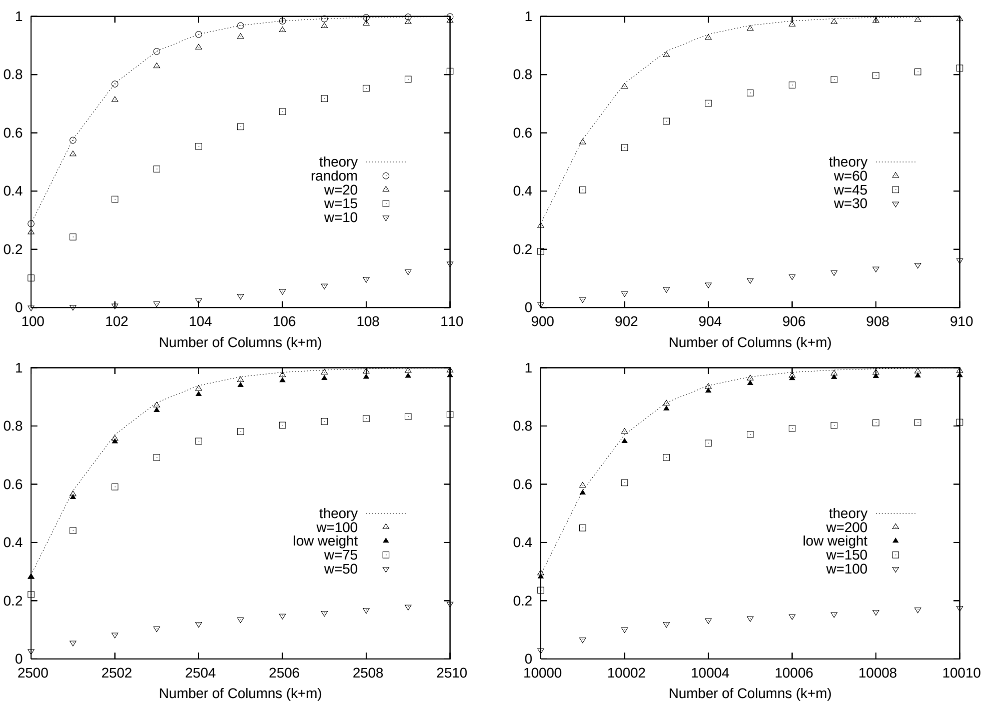

Figure 4.1: Probability of rank k for a $k \times (k+m)$ matrix and values of k ranging from $k = 100$ to $k = 10,000$. In all graphs, the broken line is Q_m, the open circles show results from random ($p = 1/2$) matrices, and the other symbols show results for windowed matrices (with the specified window size). For clarity, open circles are only shown in the $k = 100$ graph. In the $k = 2,500$ and $k = 10,000$ graphs, the *low weight* closed triangles are the results for a window size of $w = 2\sqrt{k}$ but with the mean column weight fixed at $2 \log k$. All data points are the results of testing at least $5,000$ matrices.

high probability of full rank. Later we show how one might go about calculating the exact probability of full rank as a function of window length.

Before stating our main theorem, we wish to be precise about how a windowed column is generated. First, a starting row is chosen randomly and uniformly from among the k rows. We will always place a 1 in this starting row and we refer to this 1 as the *initial 1*. Then, we will place 1's and 0's in the w rows immediately following the initial 1. In each of these rows, 1 is chosen with probability p. For the purposes of the theory given in this section, we assume $p = 0.5$. All other rows will simply contain 0. Note that if the initial 1 appears too close to the bottom of the column, the window will wrap back to the top.

$$\left.\begin{array}{c} 0 \\ \vdots \\ 0 \\ 1 \\ w\left\{\begin{array}{c} ? \\ \vdots \\ ? \end{array}\right. \\ 0 \\ \vdots \\ 0 \end{array}\right\} k$$

Figure 4.2: Column structure for windowed column.

Theorem 4.2.1 *For sufficiently large k, the probability that a $k \times k$ random windowed binary matrix with window length $w = \delta\sqrt{k}/2$ has rank k is at most $2\Phi(\delta)Q_0$, where $\Phi(z)$ is the normal distribution function and $Q_0 = 0.288788\ldots$ as given by (2.2).*

Proof: Suppose we generate k random windowed columns with k rows and window length $w = \delta\sqrt{k}/2$. Without loss of generality, we assume k is even. For each column, if the initial 1 falls in the first $k/2$ rows, we label the column a *top* column. Similarly, if the initial 1 falls in the last $k/2$ rows, we label the column a *bottom* column.

Let $\mathbf{t}$ be a random variable representing the number of top columns generated, and let $\mathbf{b}$ represent the number of bottom columns. Both $\mathbf{t}$ and $\mathbf{b}$ are sampled from the binomial distribution with the probability of generating each type of column being $1/2$. This tells us that the expected number of top columns is $k/2$ and the standard deviation is $\sqrt{k}/2$. This mean and standard deviation are the same for the bottom columns.

Now, consider what would happen if the number of top columns were greater than $k/2 + w$. If we look at only the top columns, the maximum number of rows containing at least one 1 is $k/2 + w$. Since the number of columns exceeds this, they cannot be independent, and since we require independence, we must have $t \leq k/2 + w$. A similar argument holds for the bottom columns, giving $b \leq k/2 + w$, but since $t + b = k$, we can instead express these two constraints as

$$k/2 - w \ \leq \ t \ \leq \ k/2 + w \ .$$

Recall that $w = \delta\sqrt{k}/2$ and that the standard deviation on the distribution of t is $\sqrt{k}/2$. The above constraint is satisfied as long as t is within δ standard deviations of the mean. If k is sufficiently large, the binomial distribution may be approximated using the normal distribution and $2\Phi(\delta)$ is the probability that t is within δ standard deviations of the mean.

Finally, for a non-windowed random binary $k \times k$ matrix, the probability that the matrix has rank k is $Q_0 \approx 0.288788$. Our windowed matrix is not going to have a probability of full rank that is greater than this. We assume that, as long as the windowed matrix has no columns that are dependent for the reasons described above, then its probability of full rank is Q_0. Therefore, the probability of full rank is at most $2\Phi(\delta)Q_0$.

■

For $\delta = 2$, $2\Phi(\delta)$ is roughly 0.95. The above theorem tells us that the probability of full rank for a matrix with window length $w = \delta\sqrt{k}/2 = \sqrt{k}$ will be at most $0.95Q_0$; however, Figure 4.1 shows that the probability is significantly less than this bound. Despite this, we also see from these experiments that with only a factor of 2 increase in the window length, the bound can be achieved.

We do not currently have proof that $w = 2\sqrt{k}$ is sufficient to achieve a high probability of full rank, but in the remainder of this section we will describe how the exact probability of full rank may be computed using a reduction to a balls in bins type argument.

Consider the distribution of k identical balls placed randomly into k bins. Suppose the bins are numbered from 0 to $k-1$ and let $\mathbf{b}_i$, $0 \le i < k$, be random variables whose values are the number of balls found in each bin.

We compute a partial sum of the balls found in t consecutive bins, starting at bin s as

$$\mathbf{x}_{s;\,t} = \sum_{i=0}^{i<t} \mathbf{b}_{s+i \bmod k} \quad .$$

Note that the bins are arranged in a circle so that bin 0 follows bin $k-1$. Also, for all s, $\mathbf{x}_{s;\,k} = k$.

This partial sum is related to a windowed matrix in the sense that there are $\mathbf{x}_{s;\,t}$ columns with an initial 1 found somewhere in the t rows starting at row s. Since we will be interested in how this partial sum compares to the number of rows involved, we define the following reduced partial sum

$$\hat{\mathbf{x}}_{s;\,t} = \sum_{i=0}^{i<t} \mathbf{b}_{s+i \bmod k} - t \quad .$$

Obviously, for all s, $\hat{\mathbf{x}}_{s;\,0} = \hat{\mathbf{x}}_{s;\,k} = 0$.

The required window size is the maximum that $\hat{\mathbf{x}}_{s;\,t}$ deviates from 0. It can be shown that

$$\max_{s,t} \hat{\mathbf{x}}_{s;\,t} = \max_{s,t} |\hat{\mathbf{x}}_{s;\,t}| = \max_{s,t} -\hat{\mathbf{x}}_{s;\,t} \ ,$$

so we define

$$\mathbf{w} = \max_{s,t} \hat{\mathbf{x}}_{s;\,t} = \max_{s,t} \left(\sum_{i=0}^{i<t} \mathbf{b}_{s+i \bmod k} - t \right) \ .$$

An equivalent, but much easier calculation is

$$\mathbf{w} = \max_{t} \hat{\mathbf{x}}_{0;\,t} - \min_{t} \hat{\mathbf{x}}_{0;\,t} \ .$$

The graphs in Figure 4.3 show a sample distribution of the random variable $\mathbf{w}$. In each graph, the results were compiled from $1,000,000$ tosses of either 100, 900, $2,500$ or $10,000$ balls, respectively, into an equal number of bins.

In Figure 4.4, random $k \times k$ matrices with the window length as specified on the x axis were tested for full rank. The probability of full rank is displayed along with the

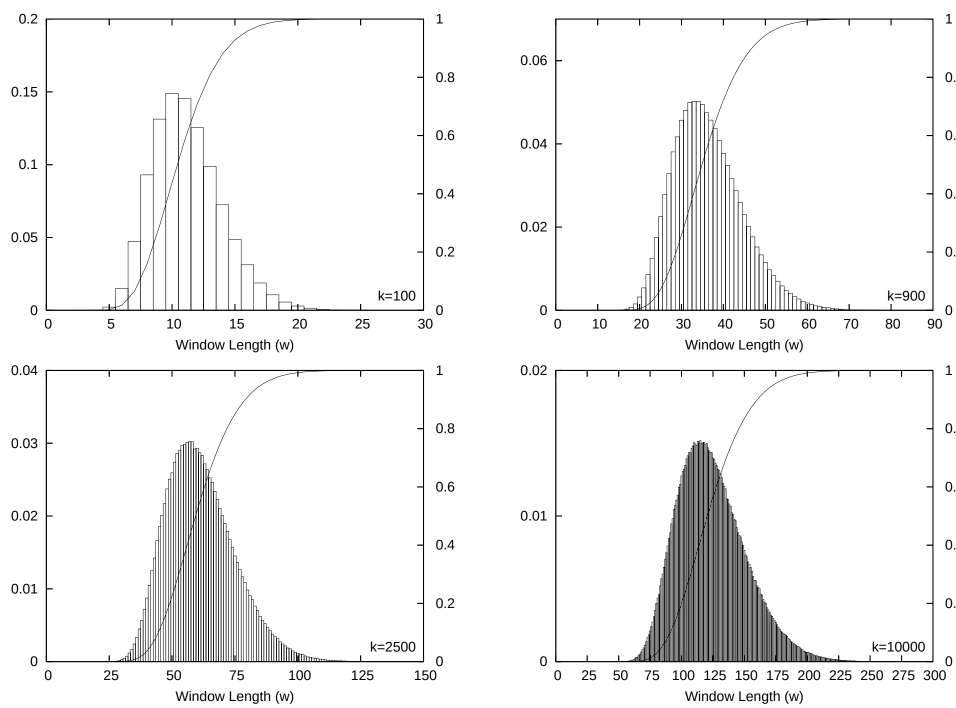

Figure 4.3: Distribution for $\mathbf{w}$ for $k = 100$, $k = 900$, $k = 2,500$ and $k = 10,000$. In all graphs the solid line is the cumulative distribution. The medians of the distributions are 11, 35, 60 and 121, respectively.

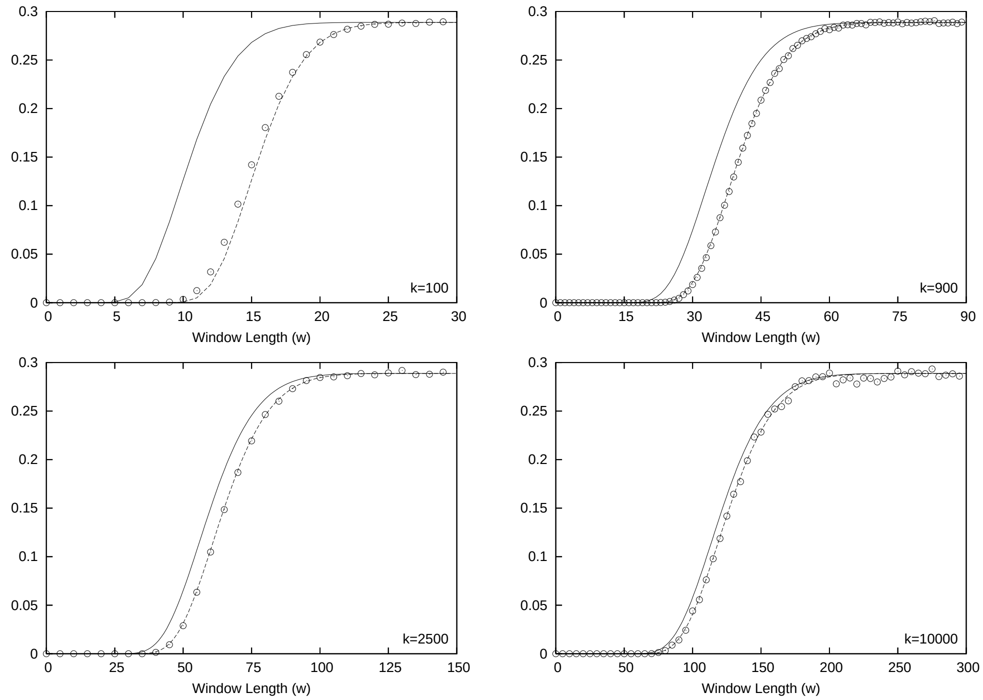

Figure 4.4: Probability of full rank for square matrix (symbols) compared to the cumulative probability distribution from Figure 4.3 scaled by Q_0 (lines). The solid line is the cumulative distribution from Figure 4.3 while the broken line is the distribution offset by 5.

corresponding cumulative distribution from Figure 4.3 scaled by 0.288788. We notice that the distribution does not exactly match the data; however, in all four cases the line appears to be offset from the data by about 5 (closest integer). This suggests that, asymptotically, the cumulative distribution line matches experiment.

4.3 Erasure code construction

These windowed matrices can be used as the basis for an efficiently encodable and decodable erasure code. The code we describe here has an encoder complexity of about $2\log k$ block sums per output block and a decoder complexity of $1.3k^{3/2}$ block sums. This complexity is achieved while keeping the overhead constant at approximately 2 extra blocks.

4.3.1 Encoding

The encoder has k data symbols (input blocks) that need to be transmitted to the decoder. To create an output block, the encoder first generates a windowed column as described in the previous section; however, we suggest that the encoder generate fixed weight columns with $\sigma = \lceil 2 \log k \rceil_{\text{odd}}$. This notation is used to refer to the lowest odd integer greater than or equal to $2 \log k$. Note that k columns each having even weight cannot have rank k, and therefore, σ must be odd or the decoder will always fail.

Along with generating a windowed column, the encoder sums (using bitwise exclusive or) the σ input blocks corresponding to the 1's in the column. The column (or some efficient encoding of it) along with the sum are then transmitted to the decoder.

Although setting $w = 2\sqrt{k}$ will work reasonably well, we suggest one instead set $w = 2(\sqrt{k} - 1)(\sigma - 1)/(\sigma - 2)$. This slightly larger value ensures that the expected number of rows between the first and last 1 (inclusive) in each column is $2\sqrt{k}$.

4.3.2 Decoding

The decoding algorithm is simply *Gaussian Elimination*; however, to ensure our discussion of decoder complexity is precise, we describe a specific decoding algorithm. Decoding has two phases:

1. **Column collection.** During column collection, matrix columns, along with their associated data blocks, are inserted into a hash table and reduced as necessary to ensure they are independent.

2. **Back filling.** At the conclusion of column collection, we have a lower triangular matrix with 1's along the diagonal. This matrix is non-singular. With a series of data block sums, the matrix is made diagonal and decoding is complete.

The hash table constructed during the first phase is to have exactly k bins and each column received hashes to the bin whose index is the index of the first row containing a 1. For the purposes of this algorithm, columns are not considered to wrap from bottom to top, and as a result, the first 1 in a column may not coincide with the initial 1. When this table is full, the k columns comprise a lower triangular matrix with 1's along the diagonal.

Hash collisions will occasionally happen during column collection. To resolve such a collision, simply add the two columns (and their associated data blocks) together to get a column that hashes to a different bin. A subtle but important detail of this algorithm is the choice of columns to keep after collision resolution. Obviously, the sum is to be kept. The other column to keep is the *shorter* of the two colliding columns. Here, the length of a column is the number of rows between the first 1 and the last 1 (inclusive). If the two columns are of equal length, either one may be kept. Two identical columns indicate a dependency; one is simply discarded and an extra column must be collected.

When the hash table is full, back filling can begin. Back filling is done starting at the last row and working up through the matrix. First, the data block associated with the 1 on the diagonal is added to all of the data blocks associated with 1's in the last row. Then, the second to last row is processed in a similar manner. At the completion of back filling, the data blocks will be the original input blocks.

Theorem 4.3.1 *Worst case decoder complexity is $\bar{\ell}k$ data block additions, where $\bar{\ell}$ is the mean column length. Column length, ℓ, as mentioned earlier, is the number of rows between the first* 1 *and the last* 1*, inclusive.*

Proof: During the column collection phase, one data block addition is required each time there is a hash table collision. If two columns, one of length x and the other of length y, $x \leq y$, collide, their sum will be a column whose length is no greater than $y-1$. Since $\bar{\ell}k$ is the sum of the lengths of the columns and each collision reduces this total length by at least 1, there can be at most $\bar{\ell}k$ collisions.

During the back filling phase, the number of data block additions needed is exactly the weight of the matrix (after column collection) less k. Also, the weight of the matrix is no greater than the total length, and the total length after column collection no greater than the total length before column collection less the number of collisions. Therefore, the sum of the weight of the matrix after column collection and the number of collisions resolved during column collection is at most $\bar{\ell}k$. ■

The average case complexity is $\bar{\ell}k/2$. This follows from the fact that when columns of length x and y, $x \leq y$, are added, the expected length of the resulting column is $y-2$. Furthermore, the expected weight of the matrix after column collection is half the total length.

To see how the complexity may be calculated from w, first notice that for columns that do not wrap, $\ell \leq w+1$; however, for columns that do wrap, ℓ may be as large as k. Since the probability of generating a column that wraps is w/k, the mean column length

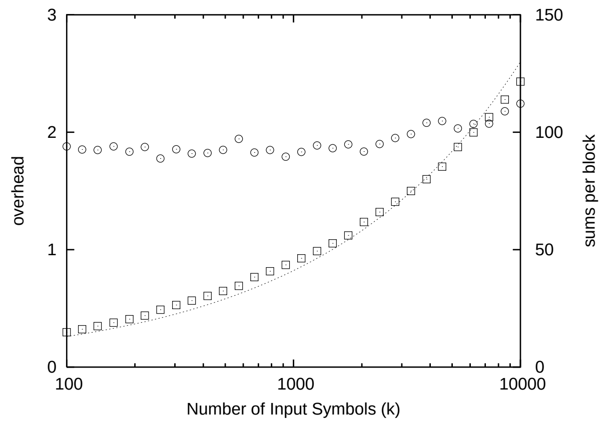

Figure 4.5: Overhead and decoder performance for a windowed erasure code. The circles show the mean number of extra columns needed (left axis) while the squares show the mean number of column operations needed per input symbol (right axis). Each data point is the mean from 10,000 runs. The broken line is $1.3\sqrt{k}$.

can be calculated as $\bar{\ell} = (w/k)k + (1 - w/k)(w + 1) \approx 2w$. When using $w \approx 2\sqrt{k}$ as suggested, expected decoder complexity is $wk \approx 2k^{3/2}$.

Figure 4.5 shows that for k between 100 and 10,000 we achieve a decoder complexity of $1.3k^{3/2}$ while maintaining an overhead of about 2 extra blocks. For an idea of real world performance we performed two tests involving a 32 MiByte file on an AMD Athlon XP 1800+. Decoding 8192 blocks (4 kiB each) requires about 10 seconds of CPU time, while decoding 65536 blocks (512 bytes each) takes 2 minutes.

4.3.3 Avoiding Wrapping

Columns that wrap from bottom to top are seen by the decoder as having length k and, as a result, require significantly more processing by the decoder. In fact, the approximately w columns that wrap require as much processing by the decoder as the $k - w$ columns that do not wrap, so we have developed a strategy to avoid wrapping in an attempt to reduce decoder complexity by a factor of 2. The strategy involves both a non-uniform distribution on the starting row selection and a variable window length. The calculations required are all based on the binomial theorem and are similar to the calculations performed in the proof of Theorem 4.2.1.

We start by considering the first row of the matrix formed by the decoder. To achieve full rank, we need at least one 1 in this row, and thus, we need at least one column whose window includes the first row. Let q_1 be the probability of generating such a column. After k columns have been received by the decoder, the binomial distribution tells us to expect $q_1 k$ columns starting in the first row; however, the probability that at least this many are received is only about 50%.

We must, however, guarantee at least one such column with arbitrarily high probability. This can be achieved by ensuring that the mean is some arbitrary number, say δ, standard deviations beyond 1. The standard deviation here is $\sigma_1 = \sqrt{kq_1(1-q_1)}$, so we want

$$q_1 k \geq 1 + \delta\sqrt{kq_1(1-q_1)}\ . \tag{4.1}$$

This equation will be solved in a moment.

Now consider the first two rows of the matrix. To get full rank, we will need at least two columns whose windows include these two rows. Of course, only one of these two columns need include the first row; the other may include only the second row. Let q_2 be the probability of generating such a column. Again, to ensure a high probability of generating the two columns needed, we require that

$$q_2 k \geq 2 + \delta\sqrt{kq_2(1-q_2)}\ . \tag{4.2}$$

If q_1 is the probability of generating a column whose window starts in row 1 and q_2 is the probability of generating a column whose window starts in either row 1 or row 2, then $q_2 - q_1$ is the probability of generating a column whose window starts in row 2.

To generalize, we consider the first i rows. We need at least i columns whose window's start in the first i rows. Let q_i be the probability of generating such a column. This probability must be constrained by

$$q_i k \geq i + \delta\sqrt{kq_i(1-q_i)}\ . \tag{4.3}$$

Assuming equality here we can solve for q_i to get

$$q_i = \frac{2i + \delta^2 \pm \delta\sqrt{\delta^2 + 4i - \frac{4i^2}{k}}}{2(k+\delta^2)}\ . \tag{4.4}$$

The correct choice of sign here is the + sign. Note that if $\delta = 0$, this equation reduces to $q_i = i/k$ as one might expect. Also, $q_k = 1$ as required for a cumulative probability distribution.

To compute the needed window length, we again consider the first row. If w_1 is the window length for columns whose initial 1 is in the first row, then we must ensure that the number of such first row columns received by the decoder does not exceed $1 + w_1$. To do this with high probability, we must ensure that the mean is at least δ standard deviations away, giving us the constraint

$$q_1 k \leq 1 + w_1 - \delta\sqrt{kq_1(1 - q_1)} \ . \tag{4.5}$$

This reasoning is easily generalized to

$$q_i k \leq i + w_i - \delta\sqrt{kq_i(1 - q_i)} \ , \tag{4.6}$$

which can be solved to give

$$w_i = 2\delta\sqrt{kq_i(1 - q_i)} \ . \tag{4.7}$$

To make use of (4.4) and (4.7) in the construction of a code, we must first consider a symmetry about the center row of the matrix. Just as we required at least one column with a 1 in the first row, we must have at least one column with a 1 in the last row. Furthermore, the probability of generating a column with a 1 in the first row should be equal to the probability of generating a column with a 1 in the last row. To make use of this symmetry, we construct a code which generates two types of columns: *top* columns and *bottom* columns. Figure 4.6 illustrates these two column types.

For i such that $q_i \leq 0.5$, the probability of generating a top column that starts in row i and has window length w_i will be $q_i - q_{i-1}$. Similarly, the probability of generating a bottom column with its bottom 1 in row $k - i + 1$ and window length w_i is also $q_i - q_{i-1}$. Note that we make no use of the q_i for which $q_i > 0.5$. This works perfectly if there exists some i for which $q_i = 0.5$. If this is not the case, as it most likely will not be, we can force the first q_i for which $q_i > 0.5$ to be exactly $q_i = 0.5$.

$$\left.\begin{array}{cc}
\begin{array}{c} 0 \\ \vdots \\ 0 \\ 1 \\ w_i\left\{\begin{array}{c} ? \\ \vdots \\ ? \end{array}\right. \\ 0 \\ \vdots \\ \vdots \\ 0 \end{array}
&
\begin{array}{c} 0 \\ \vdots \\ \vdots \\ 0 \\ w_i\left\{\begin{array}{c} ? \\ \vdots \\ ? \end{array}\right. \\ 1 \\ 0 \\ \vdots \\ 0 \end{array}
\end{array}\right\} k$$

Figure 4.6: Column structure for top and bottom columns (no wrapping).

We saw with Theorem 4.2.1 that while $\delta = 2$ seemed like it should yield good results, in practice $\delta = 4$ was required (giving $w = 2\sqrt{k}$). Assuming the same value will be required in this non-wrapping case, we can compute the window length for $q_i = 0.5$ (the center for the column) using (4.7) to be $w_i = 4\sqrt{k}$. Unfortunately, this factor of 2 increase in window length will counter the factor of 2 saving in decoder complexity we were hoping for.

Figure 4.7 shows the overhead and decoder complexity results for $\delta = 4$. As suspected, the work to be done by the decoder is virtually identical to that shown in Figure 4.5, but overhead is much worse. While our theory appears sound, obviously something more is going on and perhaps, with further experiments, the problem could be solved. However, even if the problem with overhead is solved, avoiding wrapping still provides no benefit over the simpler construction and therefore, in our opinion, is not worth the effort.

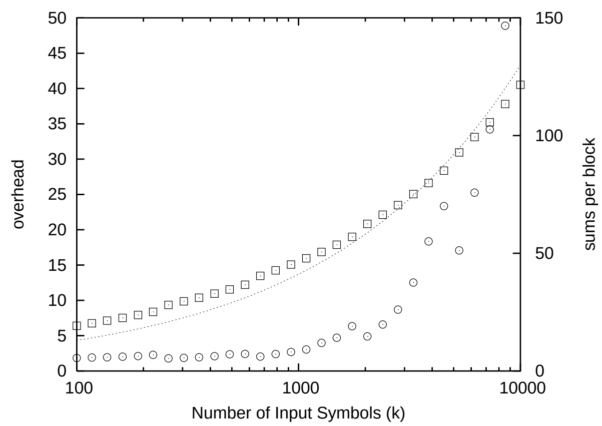

Figure 4.7: Overhead and decoder performance for a non-wrapping windowed erasure code. The circles show the mean number of extra columns needed (left axis) while the squares show the mean number of column operations needed per input symbol (right axis). The overhead value for $k = 10,000$ is off the graph at about 65. Each data point is the mean from 10,000 runs. The broken line is $1.3\sqrt{k}$.

Chapter 5

Multiparty Computation to Generate Secret Permutations

In this chapter we introduce a primitive for generating a secret permutation which will become useful in the next chapter where we introduce a secure and efficient anonymous message delivery protocol. This multiparty computation allows n clients to choose a random permutation, of themselves or some other set, and share that permutation among themselves in such a way that each client knows only part of the permutation. This can be considered a weak form of secret sharing. If a group of clients collude, they will learn each others' shares of the permutation and they can narrow down the possibilities for the remaining shares, but as long as at least two clients are honest, the coalition will be unable to learn the entire permutation.

In addition to being useful in the next chapter for anonymous message delivery, such a secret permutation can also be used for the ordering of players or in making other random choices in online games. Sweeney and Shamos consider this problem in the non-secret setting [43]. While not many games require a secret random ordering of the players, we imagine that with an efficient protocol for generating such orderings, perhaps such a game might be designed in the future. One common component of many games, however, the shuffling of cards, can be easily done with this protocol.

Other possible approaches to generating this sort of permutation include having each party choose some large number at random and then comparing those numbers using a multiparty greater than protocol. Care would need to be taken to ensure a party cannot unfairly influence their share of the permutation by making a non-uniform choice. To be successful, $\mathrm{O}(n \log n)$ instances of the greater than protocol would need to be performed, and in the end the permutation would not be entirely secret. A better approach might be to make use of a k^{th}-ranked element computation [1]. While each party would learn what the k^{th}-ranked element is, no one except the holder of that element would know who has it. Unfortunately, with each k^{th}-ranked element computation requiring $\log n$ rounds and $\mathrm{O}(n \log n)$ communication per round, the total complexity would be $\mathrm{O}\left(n^2 \log^2 n\right)$. We will show that our protocol has a communication and computation complexity of $\mathrm{O}(n \log n)$ per client.

In the next section we review the ElGamal cryptosystem and universal re-encryption. Then, in Section 5.2, we present our secret permutation sharing protocol and in Section 5.3 we prove several security properties. Finally, in Section 5.4, we discuss the application of our protocol to games.

5.1 ElGamal Cryptosystem

At the core of our secret permutation sharing scheme is a mixnet utilizing the ElGamal probabilistic public key encryption scheme [13]. Let G be some cyclic group $\langle g \rangle$ generated by $g \in G$ and let $q = |G|$. We use the operator $\in_R$ to denote a uniform random selection. The three essential algorithms provided by the ElGamal cryptosystem are:

- **Key generation**: Output $(PK, SK) = (y = g^x, x)$ for random $x \in_R \mathbb{Z}_q$.
- **Encryption**: Input comprises a message $m \in G$, a public key y and a random encryption factor $r \in_R \mathbb{Z}_q$. The output is a ciphertext $C = (\alpha, \beta) = (my^r, g^r)$.
- **Decryption**: Input is a ciphertext $C = (\alpha, \beta)$ under public key y and the corresponding secret key x. The output is plaintext $m = \alpha / \beta^x$.

In addition to these basic operations, a re-encryption algorithm is also commonly used. Re-encryption takes as input a ciphertext C and the public key the ciphertext was encrypted under y, and then outputs an alternate ciphertext C' that is an encryption of the same plaintext message under the same public key.

Golle, *et. al.*, extend the ElGamal cryptosystem to provide an algorithm for doing universal re-encryption [16]. Their new algorithms require a two-fold increase in ciphertext size but allow re-encryption without knowledge of the public key. To encrypt they choose a random encryption factor $r = (r_0, r_1) \in_R \mathbb{Z}_q^2$ and then form the ciphertext (4-tuple) $C = [(my^{r_0}, g^{r_0});\ (y^{r_1}, g^{r_1})]$. To re-encrypt the ciphertext $C =$

$[(\alpha_0, \beta_0);\ (\alpha_1, \beta_1)]$, choose a random re-encryption factor $r' = (r'_0, r'_1) \in_R \mathbb{Z}_q^2$ and compute $C' = [(\alpha_0 \alpha_1^{r'_0}, \beta_0 \beta_1^{r'_0});\ (\alpha_1^{r'_1}, \beta_1^{r'_1})]$. Decryption is the same as it is for standard ElGamal (using only the first 2-tuple), but it allows one to also verify that the ciphertext was encrypted using the correct public key. Such verification is done by decrypting the latter 2-tuple and checking for the identify element.

As will be seen in the next section, our protocol makes use of universal re-encryption; however, as will also be seen, we are only interested in recognizing ciphertexts encrypted under a certain key and not in sending messages. Therefore, all of our ciphertexts are encryptions of the identity element message. Since the latter half of a Golle ciphertext 4-tuple is actually an encryption of the identity element, we can define a universal re-encryption algorithm for use on standard ElGamal encryptions of the identity element, thus avoiding the two-fold increase in ciphertext size. We propose the following algorithm:

- **Universal identity re-encryption**: Input is a ciphertext $C = (\alpha, \beta)$ and a random re-encryption factor $r' \in_R \mathbb{Z}_q$. The output is an alternate ciphertext $C' = (\alpha^{r'}, \beta^{r'})$.

If this re-encryption operation is performed on a ciphertext that is not an encryption of the message $m = 1$, the re-encryption will corrupt the message — preventing successful decryption.

For secret key x, we define the set of all possible encryptions of the identity element under the public key g^x as $\mathcal{E}(x) = \{\ (\alpha, \beta) \mid \alpha = \beta^x\ \}$. The ciphertext $(1, 1)$ is an element of every such set, but without this element, the sets are disjoint. Furthermore, for every C with $\beta \neq 1$, there exists at most one x for which $C \in \mathcal{E}(x)$.

The final operations we need are a means of altering the key under which a ciphertext is encrypted. The following two algorithms accomplish this:

- **Key addition**: Input is a ciphertext $C = (\alpha, \beta)$ and an offset δ. The output is a ciphertext message $C' = (\alpha\beta^\delta, \beta)$, which is an encryption of the same plaintext message but now under public key $y' = yg^\delta$, where y is the public key C is encrypted under.

- **Key product**: Input is a ciphertext $C = (\alpha, \beta)$ and a coefficient c. The output is a ciphertext message $C' = (\alpha^c, \beta)$. If C is an encryption of m under the public key y, then C' is an encryption of m^c under the public key $y' = y^c$.

Note that while key addition can be used on any ciphertext, key product will alter the message if the ciphertext is not an encryption of the identity element. These operations can be performed without knowledge of the key a ciphertext is encrypted under, but they will not provide any information about said key. Also, if a message C is encrypted under g^x, then key addition with an offset of $-x$ will decrypt the message, yielding (m, β).

Key product will only be used to **negate the key** of a ciphertext. If a ciphertext $C \in \mathcal{E}(x)$, then performing key product with $c = -1 \bmod q$ will yield a ciphertext $C^- \in \mathcal{E}(-x)$. This operation can be used to turn any black box performing key addition with (secret) offset δ into a black box which performs key addition with offset $-\delta$. Simply negate the key of the ciphertext before input into the black box and again after output.

Note that the key addition operation can be thought of as a partial encryption or decryption operation and is closely related to the threshold decryption techniques described by Desmedt and Frankel [10].

5.1.1 Security of ElGamal

ElGamal is known to have *semantic security* if the group G is one for which the *Decisional Diffie-Hellman (DDH) assumption* holds. Semantic security is a property that limits an adversary's ability to derive information about a plaintext message from its corresponding ciphertext. Typically, instead of testing a cryptosystem for semantic security, one tests for *ciphertext indistinguishability*, a property that has been shown to be equivalent to semantic security [15]. This latter property asserts an adversary's inability to determine which of two plaintext messages, chosen by him, has been given back to him in the form of a ciphertext message.

The DDH assumption asserts that no computationally bounded adversary can distinguish between the distributions (g^x, g^y, g^{xy}) and (g^x, g^y, g^z), where $x, y, z \in_R \mathbb{Z}_q$. Alternatively, given (g^x, g^y, g^z), the adversary cannot determine if $z = xy$. By *computationally bounded* we mean an adversary who runs in time polynomial in κ, a security parameter. For this DDH assumption, and ElGamal, we choose the group G such that $q = \mathrm{O}(2^\kappa)$.

Semantic security limits an adversary's ability to derive information about a message from its corresponding ciphertext; however, since the ciphertexts we are interested in are all encryptions of the identity element, semantic security has little meaning in our setting. Instead, we provide a definition of *key indistinguishability* (KI). Informally, this is the inability of an adversary to distinguish between ciphertexts encrypted with distinct public keys.

To test an adversary, as defined by the algorithm $\mathcal{A}$, later referred to as an *adversarial algorithm*, we define an experiment for KI as follows. Two private ElGamal keys are generated along with two random encryption exponents. Then, an encryption of the identity element is formed using each key and the corresponding random exponent. The adversary is given these two ciphertexts in some randomly chosen order along with the two public keys and asked to guess the order of the ciphertexts. If the adversary guesses correctly the experiment terminates with an output of '1'. Otherwise the output is '0'. As mentioned above, ElGamal is parametrized under the security parameter κ.

Experiment 5.1.1 $Exp_{\mathcal{A}}^{KI}(EG, \kappa)$

$u_0, u_1 \in_R \mathbb{Z}_q$;
$r_0, r_1 \in_R \mathbb{Z}_q$;
$C_0 \leftarrow (g^{u_0 r_0},\ g^{r_0})$; $C_1 \leftarrow (g^{u_1 r_1},\ g^{r_1})$;
$\mathfrak{b} \in_R \{0, 1\}$;
$\mathfrak{b}' \leftarrow \mathcal{A}(g^{u_0}, g^{u_1},\ C_{\mathfrak{b}}, C_{1-\mathfrak{b}},\ \text{"guess"})$;
if $\mathfrak{b} = \mathfrak{b}'$ **then** output '1' **else** output '0' **fi**

Definition 5.1.1 *The ElGamal cryptosystem EG provides KI if for any adversary $\mathcal{A}$ with resources polynomial in κ the probability $pr[Exp_{\mathcal{A}}^{KI}(EG, \kappa) = \text{'1'}] - 1/2$ is negligible in κ.*

Theorem 5.1.1 *Under the DDH assumption, ElGamal provides KI.*

Proof: By contradiction, assume adversary $\mathcal{A}$ is successful in breaking KI. We will show how this adversary can be used to break the DDH assumption.

Consider the following slightly altered experiment:

$Exp_{\mathcal{A}}^{altki}(EG, \kappa)$
$u_0, u_1, u_1' \in_R \mathbb{Z}_q$;
$r_0, r_1 \in_R \mathbb{Z}_q$;
$C_0 \leftarrow (g^{u_0 r_0},\ g^{r_0})$; $C_1 \leftarrow (g^{u_1 r_1},\ g^{r_1})$;
$\mathfrak{b} \in_R \{0, 1\}$;
$\mathfrak{b}' \leftarrow \mathcal{A}(g^{u_0}, g^{u_1'},\ C_{\mathfrak{b}}, C_{1-\mathfrak{b}},\ \text{"guess"})$;
if $\mathfrak{b} = \mathfrak{b}'$ **then** output '1' **else** output '0' **fi**

The only difference between this experiment and Experiment 5.1.1 is that, in the latter, one of the public keys passed to the adversary is corrupt (alternatively, one of the ciphertexts is corrupt). The adversary may, however, still have sufficient information to determine $\mathfrak{b}$. This gives us two cases to consider.

Case 1: $pr[Exp^{altki}_{\mathcal{A}}(EG, \kappa) = \text{'1'}] - 1/2$ is negligible in κ. The adversary is not capable of handling corrupt parameters. We can construct an adversary $\mathcal{A}'$ successful against DDH as follows. Recall that the DDH test is to determine if $z = xy$.

<u>funct</u> $\mathcal{A}'(g^x, g^y, g^z, \text{"guess"}) \equiv$
  $u_0, r_0 \in_R \mathbb{Z}_q$;
  $C_0 \leftarrow (g^{u_0 r_0},\ g^{r_0})$; $C_1 \leftarrow (g^z,\ g^y)$;
  $\mathfrak{b} \in_R \{0, 1\}$;
  $\mathfrak{b}' \leftarrow \mathcal{A}(g^{u_0}, g^x,\ C_{\mathfrak{b}}, C_{1-\mathfrak{b}},\ \text{"guess"})$;
  <u>if</u> $\mathfrak{b} = \mathfrak{b}'$ **<u>then</u>** output '1' **<u>else</u>** output '0' **<u>fi</u>.**

If $z = xy$ then $\mathcal{A}$ will see parameters constructed as in Experiment 5.1.1; however, if $z \neq xy$ the ciphertext C_1 will not be a properly formed encryption under the key g^x and $\mathcal{A}$ will be unable to determine $\mathfrak{b}$.

Case 2: $pr[Exp^{altki}_{\mathcal{A}}(EG, \kappa) = \text{'1'}] - 1/2$ is not negligible in κ. In this case the adversary is clever enough to handle partially corrupt input, however, we can still construct an adversary $\mathcal{A}'$ successful against DDH.

<u>funct</u> $\mathcal{A}'(g^x, g^y, g^z, \text{"guess"}) \equiv$
  $u_1, u_1', r_1 \in_R \mathbb{Z}_q$;
  $C_0 \leftarrow (g^z,\ g^y)$; $C_1 \leftarrow (g^{u_1 r_1},\ g^{r_1})$;
  $\mathfrak{b} \in_R \{0, 1\}$;
  $\mathfrak{b}' \leftarrow \mathcal{A}(g^x, g^{u_1'},\ C_{\mathfrak{b}}, C_{1-\mathfrak{b}},\ \text{"guess"})$;
  <u>if</u> $\mathfrak{b} = \mathfrak{b}'$ **<u>then</u>** output '1' **<u>else</u>** output '0' **<u>fi</u>.**

If $z = xy$ then $\mathcal{A}$ will see partially corrupt parameters yet does have an advantage when guessing $\mathfrak{b}$. When $z \neq xy$, no information about $\mathfrak{b}$ is passed to $\mathcal{A}$, and therefore, the adversary cannot possibly determine $\mathfrak{b}$ with a probability significantly greater than 1/2.

These constructions prove that KI follows from the DDH assumption. ■

We also note that key addition with offset δ cannot be performed if one only has g^δ. This is obvious if one recalls that, for $C \in \mathcal{E}(u)$, key addition with offset $-u$ is equivalent to decryption and that g^{-u} can be easily computed from g^u. If one could do this, one could decrypt using only the public key.

Theorem 5.1.2 *Assuming KI and given a ciphertext $C \in \mathcal{E}(u)$ and a public key g^x, for any adversary $\mathcal{A}$ with resources polynomial in κ the probability that $\mathcal{A}$ outputs a ciphertext $C' = (\alpha', \beta')$ for which $\beta' \neq 1$ and $C' \in \mathcal{E}(u + x)$ is negligible in κ.*

Proof: Given any ciphertext C and a public key g^u, a successful adversary $\mathcal{A}$ can determine if $C \in \mathcal{E}(u)$ as follows. Choose $\delta \in_R \mathbb{Z}_q$ and compute $g^{-u+\delta}$. Then use $\mathcal{A}$ to compute key addition with offset $-u + \delta$. If this last step is successful, the resulting ciphertext will be an element of $\mathcal{E}(\delta)$ (which is easily checked) if and only if $C \in \mathcal{E}(u)$. Since $\mathcal{A}$ is not always successful but is assumed to be successful with a probability that is non-negligible in κ, the above steps must be repeated many, but no more than polynomial in κ, times to attain confidence in the result. If any of these repetitions results in an element of $\mathcal{E}(\delta)$, we assume that $C \in \mathcal{E}(u)$. A test of this form is sufficient to break KI, thus proving the theorem. ■

Corollary 5.1.1 *Given as input ciphertexts $C_1 \in \mathcal{E}(u_1)$ and $C_2 \in \mathcal{E}(u_2)$, for any adversary $\mathcal{A}$ with resources polynomial in κ the probability that $\mathcal{A}$ outputs a ciphertext $C' = (\alpha', \beta')$ for which $\beta' \neq 1$ and $C' \in \mathcal{E}(u_1 + u_2)$ is negligible in kappa.*

Finally, we define a generalized form of KI, called *generalized oracle key indistinguishability* (GOKI). KI is generalized in two ways: there are $h \geq 2$ keys and the adversary is provided with an oracle that can recognize ciphertexts encrypted under one of these keys. The following experiment tests an adversary $\mathcal{A}$ for GOKI. Note that $\pi(C_1, \ldots, C_h)$ produces a permuted list where C_i is located in position $\pi(i)$.

Experiment 5.1.2 $Exp_{\mathcal{A}}^{GOKI}(EG, \kappa, h)$

$u_1, \ldots, u_h \in_R \mathbb{Z}_q$;
$r_1, \ldots, r_h \in_R \mathbb{Z}_q$;
for $i := 1$ **to** h **do**
    $C_i \leftarrow (g^{u_i r_i},\ g^{r_i})$;
od
$\pi : [1 \ldots h] \rightarrow [1 \ldots h] \longleftarrow$ random permutation;

$(p, q) \leftarrow \mathcal{A}(g^{u_1}, \ldots, g^{u_h},\ \pi(C_1, \ldots, C_h),\ \mathcal{O}_{u_1,\ldots,u_h},$ "guess");
if $p = \pi(q)$ **then** output '1' **else** output '0' **fi**

The oracle $\mathcal{O}_{u_1,\ldots,u_h}$, on input a ciphertext $C = (\alpha, \beta)$, determines if $\alpha = \beta^{u_i}$ for some i. If index i exists, the oracle outputs '1', otherwise it outputs '0'. Obviously, the oracle does not reveal which key matched.

Definition 5.1.2 *The ElGamal cryptosystem EG provides GOKI if for all h and any adversary $\mathcal{A}$ with resources polynomial in κ, the probability $pr[Exp_{\mathcal{A}}^{GOKI}(EG, \kappa, h) = \text{'1'}] - 1/h$ is negligible in κ.*

Conjecture 5.1.1 *Under the DDH assumption, ElGamal provides GOKI.*

5.2 Secret Permutation Sharing

Before stating the protocol, it is useful to have in mind, at least informally, the security properties this protocol seeks to provide. They are:

- **Invalid permutation detection.** No coalition can manipulate messages in such a way that two honest clients will believe their respective shares of the permutation are the same.

- **Privacy.** No coalition of $n-2$ or fewer clients can learn the complete permutation. A coalition will know which shares are held by the honest clients, but as long as there are at least 2 honest clients, the coalition will be unable to determine the mapping of shares held by honest clients to those clients with a probability greater than that of random guessing.

- **Fairness.** The positions of the honest clients in the final permutation are distributed uniformly.

After stating the protocol we will define these properties formally and prove that they hold under suitable assumptions against an honest but curious (semi-honest) adversary. The protocol, as stated, is not secure against a malicious adversary, but we will discuss what is required to make it secure against such an adversary and describe our efforts to prove security in this case.

Assume $g \in G$ and $q = |G|$ are publicly known. The clients are initially in some order and numbered 1 to n. The protocol for generating a permutation of n items is as follows:

1. Each client chooses at random two private keys $x_i \in_R \mathbb{Z}_q$ and $u_i \in_R \mathbb{Z}_q$. The public key g^{u_i} is sent to Client 1.

2. Client 1 uses the n public keys to form an initial list of ciphertexts $[(g^{u_1}, g), \ldots, (g^{u_n}, g)]$. Note that each of these ciphertexts is an encryption of the message $1 \in G$ under the corresponding public key.

3. Each client in turn, starting with Client 1, will perform key addition with offset x_i on each ciphertext, re-encrypt each ciphertext and shuffle the list. Suppose Client i is processing the ciphertext $C_j = (\alpha_j, \beta_j)$. The client will, with random re-encryption factor $r_j \in_R \mathbb{Z}_q$, compute

$$\bar{C}_j = (\alpha_j^{r_j} \beta_j^{x_i r_j},\ \beta_j^{r_j}) \ . \tag{5.1}$$

   After processing each ciphertext in this manner, a random permutation is chosen and the list of ciphertexts is reordered. The new list is then sent to the next client for similar processing.

4. After Client n has completed the above step, she broadcasts the list of ciphertexts to all clients.

5. After receiving the list, each client broadcasts their x_i value to all clients.

6. Finally, each client will attempt to decrypt all messages using the secret key $w_i = u_i + \sum_j x_j$. If Client i locates exactly one ciphertext for which $\alpha/\beta^{w_i} = 1$, $\beta \neq 1$, then the client takes the position of this message within the list as his share of the secret permutation.

If one wishes a permutation on a larger number of items, say a multiple of n, each client can simply act as multiple clients; however, to improve efficiency slightly, each client only needs to choose a single x_i and perform Step 3 once.

5.2.1 Complexity

The computational complexity of the protocol is $\mathrm{O}(n\kappa)$ group operations per client ($\mathrm{O}(n)$ exponentiations), while the communication complexity is $\mathrm{O}(n\kappa)$ bits per client ($\mathrm{O}(n)$ group elements). This assumes that in Step 4 the final list is either efficiently broadcast or passed through the clients in a daisy chain manner (for example, from one client to the next in the reverse order of Step 3).

To ensure the probability that two clients choose the same key is negligible, the key space must have at least n^2 elements. This implies that the security parameter κ must be at least proportional to $\log n$, and thus, the communication complexity is $\mathrm{O}(n \log n)$ bits per client.

Consider the communication costs associated with either the clients deciding on a non-secret permutation or having a third party choose the permutation and send it to each client. Encoding a permutation of n items requires $\mathrm{O}(n \log n)$ bits. In this sense, our permutation generation protocol is optimal (to within a constant).

Of course, if a third party were to generate the permutation and only send each client his share of the permutation, the communication complexity would be $\mathrm{O}(\log n)$ per client, but $\mathrm{O}(n \log n)$ for the third party. In this sense, our protocol is not quite optimal.

5.3 Security

We consider 3 types of adversary:

1. **Global passive.** The global passive adversary has access to all communication and wishes to learn something about the generated permutation.

2. **Semi-honest client.** This client follows the protocol as stated but attempts to learn any additional information she can from the messages she sees and the secrets she has.

3. **Malicious adversary.** This adversary has complete control over one or more clients, and as such, may not follow the protocol. Of course, if any other client detects a problem with the messages sent out by the adversary, they may terminate the protocol. The goal of the malicious adversary is to break one of the security properties while avoiding premature termination.

All clients are computationally bounded and run in time polynomial in the security parameter κ. Furthermore, we assume that all communications are secure in the sense that if one honest client sends/broadcasts a message to some other honest client, the message will arrive intact.

Our protocol does not require that any of the communication be private. That is, all communication can be considered to be broadcast to all clients. Because of this, the above list is totally ordered on the strength of the adversary and proof of security against a malicious adversary implies security against the others.

We now formally define each of the three desired security properties, prove our SPG is secure against a semi-honest adversary, and discuss security against a malicious adversary.

5.3.1 Invalid permutation detection

Client i, assumed to be following the protocol, will take his position in the final permutation to be j if and only if the j^{th} ciphertext in the final list, say $C_j = (\alpha_j, \beta_j)$, has the property that $\alpha_j = \beta_j^{w_i}$ and $\beta_j \neq 1$. Recall that $w_i = u_i + \sum_k x_k$. An invalid permutation has been selected if Client i and Client i', $i \neq i'$, take the same final position j. This can occur if and only if one of the following holds: $w_i = w_{i'}$ or the two clients see a different ciphertext at position j, say $C_j \neq C'_j$, respectively.

The probability that $u_i = u_{i'}$ is negligible in the security parameter so we assume this is not the case. The sum of the x_k can only be different (for different clients) if when broadcasting x_k in Step 5 some client sends different values to different clients. This cannot happen in the case of a semi-honest adversary or in the case where a reliable broadcast channel is used. For the case where the adversary is malicious and uses point-to-point transmission of messages, we discuss below the changes necessary to ensure all clients receive the same x values, thus preventing the adversary from causing an invalid permutation to be generated in this manner.

We also note that all clients will see the same ciphertext in a given position, say ciphertext C_j in position j, both in the case of a semi-honest adversary and in the case where a reliable broadcast channel is used in Step 4. If a malicious adversary is present and the final list is distributed via point-to-point transmission (either directly from Client n or daisy chained through all clients) we suggest a simple verification of the final list. Given a collision resistant hash function[1] $\mathcal{H} : G^{2n} \to G$, broadcast in Step 5 both x_i and the hash value computed by applying $\mathcal{H}$ to the final list received in Step 4. Then, in Step 6, every client verifies that the $n-1$ hash values they received match the hash value they computed. If the values do not match, the permutation generation fails.

[1]For example, apply `SHA-512` to a bit-string representation of the element of the domain, and then map the result x to an element of G by treating x as an integer and computing g^x. To scale with κ, substitute a hash function with a sufficiently large range for `SHA-512`.

5.3.2 Privacy

To formally define privacy we first design an experiment which tests an adversary's ability to correctly determine an honest client's share of the permutation. We assume that either the adversary is semi-honest and may obtain the secret material held by an arbitrary subset of the clients or the adversary is malicious and may control an arbitrary subset of the clients. The following experiment could be simplified somewhat if only a semi-honest adversary were being considered, but we wish to use the same definition for both types of adversary and will consider the proof of privacy in the semi-honest case to be a sketch of the proof for the malicious adversary case.

The experiment that follows is intended to simulate the protocol but we have made a few simplifications which are justified as follows:

- Since the order in which the clients process the list in Step 3 does not affect correctness, we may assume that both Client 1 and Client n are adversary controlled. This gives the adversary the most power. In addition to this, we will assume that some number of adversary controlled clients go first, then all of the honest clients, and finally the remaining adversary controlled clients. Furthermore, the actions of these groups of clients may be aggregated. Note that even in the case where there are one or zero adversary controlled clients, we let the adversary process the list both first and last.

- In aggregating the actions of the honest clients in Step 3 we let $x = \sum x_i$.

- We force the adversary's x_i values to sum to zero. The adversary may initially perform key addition with some non-zero offset and then later undo this by performing key addition with the additive inverse. This does not weaken a semi-honest adversary as the adversary may also perform key addition with any desired offset on any private copies of ciphertexts as they desire.

- Forcing the adversary's x_i values sum to zero does weaken a malicious adversary against the protocol as stated. If such an adversary is able to choose his x_i values as a function of the honest client's x_i values, the adversary will have control over the sum $\sum_j x_j$ computed in Step 6 and can break privacy. To prevent this, we describe later how a non-mailable commitment scheme can be used to ensure that the value of the sum $\sum_j x_j$ is distributed uniformly. For the purposes of this definition, we simply assume this is the case.

- Finally, in aggregating the honest clients, we assume they all see the same final list. This list must have either been distributed using reliable broadcast or a hash function has been employed as described in the invalid permutation section above.

With these simplifications in mind, we now design an experiment for a probabilistic adversarial algorithm $\mathcal{A}$. Assume the secret permutation generator is parametrized by κ, the security parameter, n, the number of clients, and h, the number of honest clients. In the case of a semi-honest adversary, $n-h$ is the number of clients open to the adversary. We restrict the adversarial algorithm to running in time polynomial in κ.

The experiment proceeds as follows. First the honest clients choose their secret keys and send the public keys to the adversary. The adversary generates and returns a "first list" of ciphertexts along with two polynomial time algorithms to be invoked later. This first list includes the actions of the adversary in Step 3 of the protocol. The experimenter then performs Step 3 on behalf of the honest clients, using the aggregated x, and passes the resulting list to the adversary provided algorithm A which generates the "final list" of ciphertexts. This final list is tested to ensure it is valid, i.e. it contains exactly one ciphertext for each honest client. If the list is not valid, the experiment terminates with the special output value '$\perp$'. This output value distinguishes this outcome from a failed guess, which produces an output of '0'. Finally, after being given the value of x, the other adversary provided algorithm, A', guesses an honest client mapping, and if this guess is correct, the output is '1'. Note that $\pi(\bar{C}_1, \ldots, \bar{C}_n)$ produces a permuted list where $\bar{C}_i$ is located in position $\pi(i)$.

Experiment 5.3.1 $Exp_{\mathcal{A}}^{priv}(SPG, \kappa, n, h)$

> **<u>for</u>** $i := 1$ **<u>to</u>** h **<u>do</u>**
>
> $\quad u_i \in_R \mathbb{Z}_q;$
>
> **<u>od</u>**
>
> $(C_1, \ldots, C_n;\ A, A') \longleftarrow \mathcal{A}(g^{u_1}, \ldots, g^{u_h});$ // "first list"
>
> $x \in_R \mathbb{Z}_q;$
>
> **<u>for</u>** $i := 1$ **<u>to</u>** n **<u>do</u>**
>
> $\quad$ **<u>if</u>** $\alpha_i \notin G$ **<u>or</u>** $\beta_i \notin G$ **<u>or</u>** $\beta_i = 1$ **<u>then</u>** output '$\perp$' **<u>fi</u>** // $C_i = (\alpha_i, \beta_i)$
>
> $\quad r_i \in_R \mathbb{Z}_q;$
>
> $\quad \bar{C}_i \longleftarrow (\alpha_i^{r_i} \beta_i^{x r_i},\ \beta_i^{r_i});$
>
> **<u>od</u>**

$\pi : [1 \ldots n] \rightarrow [1 \ldots n] \longleftarrow$ random permutation;
$(C'_1, \ldots, C'_n) \longleftarrow \pi(\bar{C}_1, \ldots, \bar{C}_n)$;
$(C''_1, \ldots, C''_n) \longleftarrow A(C'_1, \ldots, C'_n)$; // "final list"
for $i := 1$ **to** h **do**
  $z_i \longleftarrow \left\{ j \mid \alpha''_j = (\beta''_j)^{u_i + x} \right\}$; // $C''_j = (\alpha''_j, \beta''_j)$
  if $|z_i| \neq 1$ **then** output '$\perp$' **fi**
od
$(p, q) \longleftarrow A'(x, C'_1, \ldots, C'_n)$; // "the guess"
if $\alpha''_p = (\beta''_p)^{u_q + x}$ **and** $\beta''_p \neq 1$ **then** output '1' **else** output '0' **fi**

Definition 5.3.1 *The secret permutation generator SPG with n parties, $h \geq 2$ of which are honest, has* privacy *if for any adversary $\mathcal{A}$ with resources polynomial in κ, the probability $pr[Exp^{priv}_{\mathcal{A}}(SPG, \kappa, n, h) = \text{'1'}] - 1/h$ is negligible in κ.*

A semi-honest adversary has additional restrictions not inherent in the above experiment that will be discussed in the proof of the following theorem. Also, notice that with a semi-honest adversary, the experiment will never terminate with an output of '$\perp$', and therefore, the tests that lead to this output could be removed if one is only interested in privacy against a semi-honest adversary.

Theorem 5.3.1 *Assume ElGamal has GOKI. If we consider only semi-honest adversaries $\mathcal{A}$ then our secret permutation generation protocol has privacy.*

Proof: By contradiction, we suppose there exists a semi-honest adversary $\mathcal{A}$ for which $pr[Exp^{priv}_{\mathcal{A}}(SPG, \kappa, n, h) = \text{'1'}] - 1/h$ is *non-negligible* in κ and use this algorithm to construct an adversary $\mathcal{A}'$ that can break GOKI.

Throughout this proof we will assume the h keys $g^{u_1}, \ldots, g^{u_h}$ provided by Experiment 5.1.2 are distinct. The probability that they are not is negligible in κ.

There exists a vector $(v_1, \ldots, v_n) \in \mathbb{Z}_q^n$ for which each $C_i \in \mathcal{E}(v_i)$ and $\bar{C}_i(x) \in \mathcal{E}(v_i + x)$. We denote $\bar{C}_i$ as $\bar{C}_i(x)$ to emphasis the dependence on x.

Since the adversary is semi-honest, it must be the case that there exists $\delta \in \mathbb{Z}_q$ such that for each j there exists a unique i for which $C_i \in \mathcal{E}(u_j - \delta)$ (i.e. $v_i = u_j - \delta$). Furthermore, the algorithm A must perform a key addition with offset δ on each input ciphertext, a re-encryption of each ciphertext, and some permutation (shuffling) of the ciphertexts. We note that the shuffling of the ciphertexts cannot help the adversary in any way and so, without loss of generality, we assume algorithm A does not shuffle its output.

We saw in Section 5.1.1 that an algorithm which performs key addition with offset δ can be transformed into an algorithm that performs key addition with offset $-\delta$ by simply negating the key of both the input ciphertext and the output ciphertext. In this way, we construct algorithm A^- from A which performs key addition with offset $-\delta$ (and re-encryption) on each of the input ciphertexts. Note that the composition of A and A^- in either order is an algorithm which, on input a list of n ciphertexts, simply re-encrypts each ciphertext and outputs them.

Now we describe the construction of the adversary $\mathcal{A}'$. In Experiment 5.1.2, the adversary is invoked with the line

$$(p, q) \longleftarrow \mathcal{A}'(g^{u_1}, \ldots, g^{u_h}, \hat{C}_1, \ldots, \hat{C}_h, \mathcal{O}_{u_1,\ldots,u_h}, \text{“guess”}) \ .$$

The first thing to do is simply pass the keys to the adversary

$$(C_1, \ldots, C_n;\ \ A, A') \longleftarrow \mathcal{A}(g^{u_1}, \ldots, g^{u_h}) \ ,$$

and then follow Experiment 5.3.1, choosing a x and π, until we have completed the computation of the “final list”

$$(C''_1, \ldots, C''_n) \longleftarrow A(C'_1, \ldots, C'_n) \ .$$

By performing key addition on each of these ciphertexts with offset $-x$ and passing the result to the oracle, the set $H = \{\, i \mid C''_i \in \mathcal{E}(u_j + x) \text{ for some } j \,\}$ can be computed. Since the adversary is semi-honest, this set must have exactly h elements.

Now, construct a new final list $(\hat{C}''_1, \ldots, \hat{C}''_n)$ as follows. For each i, if $i \notin H$ then set $\hat{C}''_i = C''_i$, otherwise $\hat{C}''_i$ is $\hat{C}_j$ with the offset x added to the key. Each $\hat{C}_j$ must be used exactly once so we choose a bijection $\mu : [1 \ldots h] \to H$ and set $j = \mu^{-1}(i)$.

With this new final list, apply A^- to compute a new intermediate list

$$(\hat{C}'_1, \ldots, \hat{C}'_n) \longleftarrow A^-(\hat{C}''_1, \ldots, \hat{C}''_n) \ .$$

Finally, the new intermediate list is simply passed to algorithm A' to obtain a mapping

$$(p, q) \longleftarrow A'(x, \hat{C}'_1, \ldots, \hat{C}'_n) \ .$$

If this mapping is an obviously incorrect guess, i.e. $p \notin H$, then $\mathcal{A}'$ is to output a random guess and terminate. Otherwise, the pair $(\mu^{-1}(p), q)$ is output to indicate that $\hat{C}_{\mu^{-1}(p)}$ is an element of $\mathcal{E}(u_q)$.

Guesses made by $\mathcal{A}$ fall into three categories: correct, wrong and obviously wrong ($p \notin H$). Since an obviously wrong guess is turned into a valid guess for $\mathcal{A}'$ we have that the probability $\mathcal{A}'$ outputs a correct guess is $P_{correct} + P_{bad}(1/h)$, where P_{bad} is the probability that a guess is obviously wrong. Since $P_{correct} \geq 1/h + k^{-\epsilon}$ for some $\epsilon > 0$ and infinitely many κ (the security parameter), we conclude that $\mathcal{A}'$ is correct with at least this same probability, thus breaking GOKI. ∎

Privacy against a malicious adversary

As mentioned, our protocol does not provide privacy against a malicious adversary without a few modifications. The most significant change here is to make use of a *non-mailable commitment scheme* to ensure that no client can compute his x_i value as a function of the other client's values. The commitment scheme is a method that allows a client to commit to their value while keeping it hidden. Later, the client can reveal his value and all other clients can verify that it is the one committed to. The non-mailable aspect of such a commitment is discussed below. The modifications required to the steps of the protocol are as follows:

Step 1: Using a non-mailable commitment scheme $\mathcal{C}$, each client broadcasts a commitment $\mathcal{C}(x_i)$ to x_i.

Step 3: Each client must verify that all of the ciphertexts in the list they receive are properly formed. This is done by checking that for all j, $\alpha_j \in G$, $\beta_j \in G$ and $\beta_j \neq 1$. If any of these checks fail, the protocol is terminated.

Step 5: In broadcasting x_i to all clients, the commitment $\mathcal{C}(x_i)$ is opened. Also, as described in the above section on invalid permutation detection, if the use of a hash function is required the hash value is broadcast in this step.

Step 6: In addition to the other checks described, the validity of the commitments is checked, and if a hash function was used, the hash values are also checked. If any of these checks fail, the protocol terminates with failure.

One must carefully choose the commitment scheme to use. If one client, say Client i, commits to x_i with $\mathcal{C}(x_i)$, some other client could commit to the same value, but since that client does not know x_i, that client would be unable to correctly perform Step 3. The result would be failure of the protocol. However, if a client, given the commitment $\mathcal{C}(x_i)$, can generate either $\mathcal{C}(-x_i)$ or $\mathcal{C}(x_j - x_i)$, for some x_j, the privacy of the protocol can be broken. For this reason using g^{x_i} as a commitment to x_i will not work. We believe committing to each bit in the binary representation of x_i should suffice. Other, more efficient, non-mailable commitment schemes may work as well. From this point on we assume a suitable commitment scheme has been selected.

With these modifications to the protocol in mind, we now prove our SPG has privacy against an almost malicious client. The adversary is malicious but has one restriction placed upon it; the algorithm $A(\bar{C}_1, \ldots, \bar{C}_n)$ is implemented as n algorithms $A_1(\bar{C}_1), \ldots, A_n(\bar{C}_n)$. We are working to remove this restriction.

Before stating our theorem, we prove two important lemmas. The first allows us to prove that for each public key u_j, at least one ciphertext in the "first list" is a function of that key and that key alone. The second is needed to prove that each of the h ciphertexts in the "first list" that are functions of the public keys have a specific form.

The following experiment tests an adversary's ability to produce two final ciphertexts from one intermediate ciphertext. We refer to this experiment as the *dual* experiment to emphasize the generation of two ciphertexts from one.

Experiment 5.3.2 $Exp_{\mathcal{A}}^{dual}(\kappa)$

$u_1, u_2 \in_R \mathbb{Z}_q$;
$(C_0, A) \longleftarrow \mathcal{A}(g^{u_1}, g^{u_2})$; // "initial ciphertext"
if $\alpha_0 \notin G$ **or** $\beta_0 \notin G$ **then** output '0' **fi** // $C_0 = (\alpha_0, \beta_0)$
$x_1, x_2 \in_R \mathbb{Z}_q$;
$r_1, r_2 \in_R \mathbb{Z}_q$;
$C'_1 \longleftarrow (\alpha_0^{r_1}\beta_0^{x_1 r_1},\ \beta_0^{r_1})$;
$C'_2 \longleftarrow (\alpha_0^{r_2}\beta_0^{x_2 r_2},\ \beta_0^{r_2})$;
$(C''_1, C''_2) \longleftarrow A(C'_1, C'_2)$; // "final ciphertext"
if $\beta''_1 = 1$ **or** $\beta''_2 = 1$ **then** output '0' **fi** // $C'' = (\alpha'', \beta'')$
if $\alpha''_1 = (\beta''_1)^{u_1+x_1}$ **and** $\alpha''_2 = (\beta''_2)^{u_2+x_2}$ **then** output '1' **else** output '0' **fi**

Lemma 5.3.1 *Assuming KI (as in Theorem 5.1.2), the probability* $pr[Exp_{\mathcal{A}}^{dual}(\kappa) = \text{'1'}]$ *is negligible in* κ.

Proof: By contradiction, suppose $\mathcal{A}$ is an adversary that is successful in the experiment with non-negligible probability. We show how this adversary can be used to break Theorem 5.1.2.

We are given as input a key g^x and a ciphertext $C \in \mathcal{E}(u)$, and we must show how to compute a ciphertext $C' = (\alpha', \beta')$ for which $\beta' \neq 1$ and $C' \in \mathcal{E}(u + x)$.

Choose $u_1 \in_R \mathbb{Z}_q$, compute g^{u_1}, and set $g^{u_2} = g^x$. These two public keys are given to $\mathcal{A}$ to get a ciphertext C_0. There exists some v_0 for which this ciphertext is an encryption under the key g^{v_0}.

During a "normal" invocation of A, the ciphertext C'_1, an encryption under the key $g^{v_0+x_1}$, is transformed into C''_1, an encryption under the key $g^{u_1+x_1}$. This is key addition with offset $u_1 - v_0$. Likewise, the ciphertext C'_2 is transformed into C''_2 via key addition with offset $u_2 - v_0$.

In Section 5.1 we noted that an algorithm that performs key addition with offset δ can be easily transformed into an algorithm to perform key addition with offset $-\delta$. Simply negate the key of both the input ciphertext and the output ciphertext. Let A^- be A transformed as described.

To break the lemma, we need two invocations of A. For the first, we provide C as input to A^-, first parameter, to obtain an element of $\mathcal{E}(u - u_1 + v_0)$. Using key addition, this ciphertext is transformed into an element of $\mathcal{E}(u + v_0)$. Finally, this latter ciphertext is given to A as the second parameter to obtain an element of $\mathcal{E}(u + u_2) = \mathcal{E}(u + x)$.

If each invocation of A yields an appropriate result with probability greater than $k^{-\epsilon}$, then our final probability of success at least $k^{-2\epsilon}$, which is non-negligible in κ. ■

We further note that this lemma holds even if the probability is only computed over experiments for which $x_1 = x_2$ as the proof does not require that these two values be independent.

Lemma 5.3.2 *Suppose h coloured balls are tossed (randomly) into n coloured bins with at most one ball allowed in each bin. The colours are chosen from some set and duplicates are allowed. Unless all of the balls are the same colour, the probability that each ball matches the colour of the bin it lands in is at most $1/h$.*

Proof: Suppose d of the balls are, say, red and $h - d$ are black. Also, to ensure the probability of a match is non-zero, we assume that at least d bins are red and at least $h - d$ are black. Some bins (up to $n - h$) may be painted some third colour. We consider a toss of the balls to have been a *success* if the colour of each ball matches the colour of the bin it landed in.

Let the bins be lined up in some order, toss the balls, and consider the h bins that have a ball in them. If any of these bins are a third colour, the toss was a failure. Assume this is not the case. Map the colours of the h bins to a string of bits with red mapping to 1 and black mapping to 0. If this bit string (of length h) is not of weight d, the toss was a failure, so we assume this is not the case. Label this bit string X.

Now consider the h balls and map their colours to the bits of a bit string labelled Y. The toss is only a success if the two bit strings match exactly (i.e. $X = Y$).

The bit string Y has weight d and was selected uniformly from the set of all weight d strings. This set is of size $\binom{h}{d}$. Therefore, the probability that $X = Y$ is $1/\binom{h}{d}$, and since we made some assumptions about X above, this probability is an upper bound. Notice that for $0 < d < h$, $1/\binom{h}{d} \leq 1/h$. Only if $d = 0$ or $d = h$ is a probability greater than $1/h$ possible.

Finally, we note that if there are more than two distinct ball colours, the probability of success is further reduced, thus proving the lemma. ■

Theorem 5.3.2 *Assume ElGamal has GOKI. If we consider only malicious adversaries $\mathcal{A}$ for which the adversary provided algorithm $A(\bar{C}_1, \ldots, \bar{C}_n)$ is implemented as n algorithms $A_1(\bar{C}_1), \ldots, A_n(\bar{C}_n)$, then our (modified) secret permutation generation protocol has privacy.*

Proof: By contradiction, we suppose there exists an adversary $\mathcal{A}$ for which $pr[Exp^{priv}_{\mathcal{A}}(SPG, \kappa, n, h) = \text{'1'}] - 1/h$ is *non-negligible* in κ and use this algorithm to construct an adversary $\mathcal{A}'$ that can break GOKI.

Throughout this proof we will assume the h keys $g^{u_1}, \ldots, g^{u_h}$ provided by Experiment 5.1.2 are distinct. The probability that they are not is negligible in κ.

There exists a vector $(v_1, \ldots, v_n) \in \mathbb{Z}_q^n$ for which each $C_i \in \mathcal{E}(v_i)$ and $\bar{C}_i(x) \in \mathcal{E}(v_i + x)$. We denote $\bar{C}_i$ as $\bar{C}_i(x)$ to emphasis the dependence on x.

Claim: If, for some i, j, k, and non-negligibly many x, algorithm A_k, on input $\bar{C}_i(x)$, outputs an element of $\mathcal{E}(u_j + x)$, then for all $j' \neq j$ and k', the number of x values for which algorithm $A_{k'}$, also on input $\bar{C}_i(x)$, outputs an element of $\mathcal{E}(u_{j'} + x)$, is negligible.

This follows from Lemma 5.3.1 and establishes that each ciphertext C_i can be used to generate elements of $\mathcal{E}(u_j + x)$ for at most one j. Therefore, each of at least h of the n ciphertexts output by $\mathcal{A}$ as the "first list" must have been constructed for the purpose of generating elements from a particular set $\mathcal{E}(u_j + x)$.

Each algorithm A_k is probabilistic and therefore its action is difficult to predict. Despite this, we will consider each of these algorithms, on any particular run, to be performing key addition with some offset; however, the offset may vary from one run to the next. Our justification for this is that the algorithm cannot (with non-negligible probability) output an element of one of the sets $\mathcal{E}(u_j + x)$ without performing key addition (even if it is with an offset of 0). Indeed, on any particular run, the probability that at least h of the algorithms are performing key addition must be at least $1/h$.

Claim: There exists $\delta \in \mathbb{Z}_q$ such that for all j there exists an i for which $C_i \in \mathcal{E}(u_j - \delta)$ (i.e. $v_i = u_j - \delta$).

Suppose, during some run of the algorithms, each algorithm A_k performs key addition with the offset δ'_k. Furthermore, each $C_i \in \mathcal{E}(u_j - \delta_i)$ for some j and δ_i. Actually, with the latter, for all j there exists some δ_i for which $C_i \in \mathcal{E}(u_j - \delta_i)$, but we assume the adversary had some j in mind when he generated C_i. If algorithm A_k is to process the input $\bar{C}_i(x)$ and output an element of $\mathcal{E}(u_j + x)$, then it is necessary that $\delta'_k = \delta_i$. Because of this we consider each ciphertext C_i to be coloured by δ_i and each algorithm to be coloured by δ'_k. A valid final list can only be produced if the permutation π maps h of the ciphertexts to similarly coloured algorithms.

Lemma 5.3.2 establishes that unless at least h of the ciphertexts are of the same colour, the probability that a valid final list is generated is at most $1/h$. Let δ be this colour. Furthermore, because of the specific requirements for a valid final list, it must be the case that for every j there exists an i for which $C_i \in \mathcal{E}(u_j - \delta)$.

Now we describe the construction of the adversary $\mathcal{A}'$. In Experiment 5.1.2, the adversary is invoked with the line

$$(p, q) \longleftarrow \mathcal{A}'(g^{u_1}, \ldots, g^{u_h}, \hat{C}_1, \ldots, \hat{C}_h, \mathcal{O}_{u_1,\ldots,u_h}, \text{“guess”}) \ .$$

The first thing to do is simply pass the keys to the adversary

$$(C_1, \ldots, C_n;\ A_1, \ldots, A_n, A') \longleftarrow \mathcal{A}(g^{u_1}, \ldots, g^{u_h}) \ ,$$

and then follow Experiment 5.3.1, choosing a x and π, until we have completed the computation of the “final list”

$$(C''_1, \ldots, C''_n) \longleftarrow (A_1(C'_1), \ldots, A_n(C'_n)) \ .$$

By performing key addition on each of these ciphertexts with offset $-x$ and passing the result to the oracle, the set $H = \{\, i \mid C''_i \in \mathcal{E}(u_j + x) \text{ for some } j \,\}$ can be computed. If this set does not have exactly h elements, then $\mathcal{A}'$ is to output a random guess and terminate. From the claims proven earlier, we know that with probability at least $1/h$, H has exactly h elements, for each j there exists an $i \in H$ such that $C''_i \in \mathcal{E}(u_j + x)$, and there exists a δ such that for $k \in H$, A_k, during this run, performed key addition with an offset of δ.

Now, let algorithm A^-_k be A_k but with both the input ciphertext and output ciphertext modified by key negation. Recall that the result of this is that if A_k performs key addition with offset δ, then A^-_k will perform key addition with offset $-\delta$.

To finish the construction of the adversary, we now compute a new intermediate ciphertext list $(\hat{C}'_1, \ldots, \hat{C}'_n)$. For each i, if $i \notin H$ then set $\hat{C}'_i = C'_i$, otherwise $\hat{C}'_i$ is $\hat{C}_j$ with the offset x added to the key and processed with A^-_i. Each $\hat{C}_j$ must be used exactly once so we choose a bijection $\mu : [1 \ldots h] \rightarrow H$ and set $j = \mu^{-1}(i)$. As above, with probability at least $1/h$, the h algorithms used here all performed key addition with offset δ.

Finally, the new intermediate list is simply passed to algorithm A' to obtain a mapping

$$(p, q) \longleftarrow A'(x, \hat{C}'_1, \ldots, \hat{C}'_n) \ .$$

If this mapping is an obviously incorrect guess, i.e. $p \notin H$, then $\mathcal{A}'$ is to output a random guess and terminate. Otherwise, the pair $(\mu^{-1}(p), q)$ is output to indicate that $\hat{C}_{\mu^{-1}(p)}$ is an element of $\mathcal{E}(u_q)$.

The probability that $\mathcal{A}'$ makes it to this last step and the list $(\hat{C}'_1, \ldots, \hat{C}'_n)$ is a well formed ciphertext list (i.e. there exists some π that would yield this intermediate list in Experiment 5.3.1) is at least $1/h^2$. If P_{guess} is the probability that $\mathcal{A}$ correctly guesses a mapping given that the final list is valid, then the probability that $\mathcal{A}'$ outputs a correct mapping is at least

$$\left(1 - \frac{1}{h^2}\right) \frac{1}{h} + \frac{1}{h^2} P_{guess} \ .$$

If $P_{guess} \geq 1/h + k^{-\epsilon}$ for some $\epsilon > 0$ and infinitely many κ (the security parameter), then the probability that $\mathcal{A}'$ is correct is at least $1/h + k^{-\epsilon}/h^2$ which is non-negligibly greater than $1/h$, thus breaking GOKI. ■

It is our intention to show, in future work, that the restricted form of the algorithm A does not in fact weaken the adversary. Not only does re-arrangement of the ciphertexts, using an intermediate ciphertext more than once, and duplicating output ciphertexts not provide the adversary with any advantage, we also note the infeasibility of creating an algorithm that takes as input two ciphertexts and combines them to create a single meaningful ciphertext as output. With this in mind, we state the following conjecture.

Conjecture 5.3.1 *Assuming ElGamal has GOKI, our (modified) SPG protocol has privacy against a malicious adversary.*

5.3.3 Fairness

We stated earlier that fairness is the property that the positions of the honest clients in the final permutation are distributed uniformly. If there are h honest clients and n positions, we have $\binom{n}{h}$ distinct arrangements of honest clients. It is from this set the positions are to be selected uniformly.

To test an adversary's ability to break fairness, we consider the following experiment. Let κ be the security parameter, n the number of clients, h of which are honest, and $H \subseteq \{1, \ldots, n\}$, $|H| = h$, the set of positions the adversary is aiming to put the honest clients into. The bulk of the following experiment is identical to Experiment 5.3.1 with the only difference being after the final list has been tested for validity. If the final list is valid the experiment simply checks to see if the honest clients are in the positions specified by the set H. If so, the experiment terminates with output '1'. If not, the output is '0'.

Experiment 5.3.3 $Exp_{\mathcal{A}}^{fair}(SPG, \kappa, n, h, H)$

> **for** $i := 1$ **to** h **do**
>
> $\quad u_i \in_R \mathbb{Z}_q$;
>
> **od**
>
> $(C_1, \ldots, C_n;\ A) \longleftarrow \mathcal{A}(g^{u_1}, \ldots, g^{u_h})$; // "first list"
>
> $x \in_R \mathbb{Z}_q$;
>
> **for** $i := 1$ **to** n **do**
>
> $\quad$ **if** $\alpha_i \notin G$ **or** $\beta_i \notin G$ **or** $\beta_i = 1$ **then** output '$\perp$' **fi** // $C_i = (\alpha_i, \beta_i)$
>
> $\quad r_i \in_R \mathbb{Z}_q$;
>
> $\quad \bar{C}_i \longleftarrow (\alpha_i^{r_i} \beta_i^{x r_i},\ \beta_i^{r_i})$;
>
> **od**
>
> $\pi : [1 \ldots n] \rightarrow [1 \ldots n] \ \longleftarrow\ $ random permutation;
>
> $(C'_1, \ldots, C'_n) \longleftarrow \pi(\bar{C}_1, \ldots, \bar{C}_n)$;
>
> $(C''_1, \ldots, C''_n) \longleftarrow A(C'_1, \ldots, C'_n)$; // "final list"

$\underline{\textbf{for}}\ i := 1\ \underline{\textbf{to}}\ h\ \underline{\textbf{do}}$
$\quad z_i \longleftarrow \{\, j \mid \alpha''_j = (\beta''_j)^{u_i+x} \,\}$; $\qquad // \ C''_j = (\alpha''_j, \beta''_j)$
$\quad \underline{\textbf{if}}\ |z_i| \neq 1\ \underline{\textbf{then}}$ output '$\perp$' $\underline{\textbf{fi}}$
$\underline{\textbf{od}}$
$H' \longleftarrow \{\, j \mid \alpha''_j = (\beta''_j)^{u_i+x}$ for some $i \,\}$;
$\underline{\textbf{if}}\ H = H'\ \underline{\textbf{then}}$ output '1' $\underline{\textbf{else}}$ output '0' $\underline{\textbf{fi}}$

Definition 5.3.2 *The secret permutation generator SPG with n parties, h of which are honest, is fair if for any adversary $\mathcal{A}$ with resources polynomial in κ and any $H \subseteq \{1, \ldots, n\}$, $|H| = h$, the probability $pr[Exp^{fair}_{\mathcal{A}}(SPG, \kappa, n, h, H) = \text{'1'}] - 1/\binom{n}{h}$ is negligible in κ.*

While our SPG does provide fairness against a semi-honest adversary, even the modified version does not provide fairness against a malicious one. There is a malicious adversary that can break fairness, but only with a high probability that the experiment fails with an invalid "final list". Such an adversary operates as follows. Each ciphertext C_i is an encryption under one of the public keys u_j, where each public key is used roughly the same number of times. Then, algorithm A is constructed to randomly choose exactly h of the intermediate ciphertexts and combine them with $n - h$ randomly generated ciphertexts. The chosen intermediate ciphertexts take the positions described by H in the final list. If the final list is valid, the honest client ciphertexts are in exactly the right positions.

We calculate that the probability the final list is valid is approximately $(n/h)^h/\binom{n}{h}$ (if n is a multiple of h, this expression is exact). Clearly, for all $h < n$, this probability is strictly greater than $1/\binom{n}{h}$. We conjecture, however, that in an environment where the adversary has a strong incentive to not cause the protocol to fail with an invalid final list, our SPG is fair.

Conjecture 5.3.2 *Assume ElGamal has GOKI. If we consider only malicious adversaries $\mathcal{A}$ for which the probability $pr[Exp^{fair}_{\mathcal{A}}(SPG, \kappa, n, h, H) = \text{'}\perp\text{'}]$ is negligible in κ, then our (modified) secret permutation generation protocol is fair.*

Furthermore, we suspect this restriction is too strong and we are exploring the possibility that a higher bound on the probability that the final list is invalid may be allowed.

Note that we are not very concerned with this weakness in the protocol for two reasons: first, our intended use of this protocol is for anonymous message delivery to be discussed in the next chapter and such use does not require an SPG protocol that is fair, and second, there is a straightforward way to add fairness to any SPG protocol.

Adding Fairness

The fairness property is not as essential to a SPG protocol as privacy because fairness can be easily added with one additional round of communication and a constant factor increase in communication complexity. The additional steps simply require that, after the secret permutation has been generated, the clients participate in a protocol to generate a non-secret fair permutation and then compose the two permutations to yield a secret and fair final permutation.

One example of a non-secret fair permutation generation protocol is as follows. Each client chooses an integer from $\mathbb{Z}_{n!}$. The clients then commit to and reveal their integers. The commitment is to ensure the integers are independent. Finally, the clients sum the integers modulo $n!$ and map the result to a permutation on n items.

5.4 Application to Games

A secret permutation sharing scheme can be used in a variety of games to either select a secret ordering of the players or to shuffle a deck of cards. Some games (e.g. the board game Civilization) require the players to draw numbers from a hat to determine an initial secret ordering. Many other games require a deck of cards to be shuffled and dealt. For games where an entire deck is to be dealt out (e.g. Bridge or Hearts), the application of our secret permutation sharing scheme is straightforward. Suppose 4 players are to be dealt 13 cards each. In Step 2, Client 1 forms an initial set of 52 ciphertexts from the 4 public keys received

$$[(g^{u_1}, g), \ldots, (g^{u_1}, g),\ (g^{u_2}, g), \ldots, (g^{u_2}, g),\ (g^{u_3}, g), \ldots, (g^{u_3}, g),\ (g^{u_4}, g), \ldots, (g^{u_4}, g)].$$

This list contains 13 copies of each of the 4 ciphertexts. Then, in Step 6, each client can expect to find exactly 13 ciphertexts for which $\alpha/\beta^{w_i} = 1$. The 52 list positions are associated with the cards of a standard deck and each player takes the cards associated with the 13 such ciphertexts as their hand. We are also working on an extension of our protocol to allow the cards to be revealed to players slowly over time.

Chapter 6

Anonymous Message Delivery

In this chapter we put the secret permutation generation protocol developed in the last chapter to good use in the development of a secure and efficient anonymous message delivery protocol. As discussed earlier, our reasons for wanting an anonymous message delivery protocol for our distributed backup system is to ensure an adversary cannot selectively deny service by hording the blocks output by a single user and then preventing their retrieval.

Anonymous message delivery has been an area of intense research in recent years and has been an area of study since the 1980's, when Chaum introduced the first primitives. In 1981 he introduced the idea of a *mixnet* [7] and then in 1988 the idea of *dining cryptographers* [8].

With a mixnet, a client who makes use of the mixnet to anonymously route messages must trust that at least one node of the mixnet is acting in an honest manner and that at least some of the other clients of the mixnet are honest and genuinely seeking anonymity. Of course, nothing prevents clients from participating in the mixnet as nodes themselves, but as we will discuss below, this approach can be inefficient.

In a dining cryptographers network (DC-net) the participants all broadcast their messages simultaneously. Since this is a broadcast network, there is no recipient *per se* nor any routing nodes to trust. One must simply trust that not all of the other clients are colluding against them.

Existing DC-net protocols are not particularly efficient as they typically impose an expansion factor of n, the number of participating clients, to each message broadcast. Furthermore, many DC-net protocols are not collusion resistant. If any one client decides to reveal their local information, the anonymity of all participants is compromised. Our goal is to address these two shortcomings.

In this chapter we will first describe the basic protocols used in the implementation of a mixnet and then discuss how one might implement a secure DC-net. Finally, we detail our efficient DC-net protocol and discuss its security. We will also introduce *private information retrieval* (PIR) as it is a necessary primitive used by our DC-net protocols and is of interest in its own right.

6.1 Mixnets

In its simplest form, a single mix node will first receive several equal length messages from some number of clients. The mix node then reorders the messages according to some random permutation chosen by the mix node and kept secret. Finally, the mix node will forward each message to the next hop along the way to its final destination; however, to ensure that an outside observer privy to both the messages going in to and out of the mix node cannot simply match the inputs to the outputs, the mix node must modify each message in some way. Typically the mix node will either decrypt, encrypt or re-encrypt each message.

Since the mix node knows the permutation it used to mix the messages, any anonymity gained through the use of the mix node can just as quickly be taken away. To reduce the impact of a dishonest mix, clients typically route their messages through several mix nodes. Doing this ensures that the client maintains at least some anonymity as long as at least one of the mix nodes is honest. In addition to trusting at least one mix node, the client must also trust that at least some of the other clients using the mix network are honest as well. The focus of this work is to eliminate the former, the client's need to trust the mix node or nodes.

The need for clients to trust at least one node of a mixnet can be rendered moot if the clients themselves form the mixnet. For each message block to be sent to the server, each client first prepares (encrypts) their message for entry into the mixnet and then sends the message to Client 1. Then, one by one, each client mixes the batch of n messages and sends the batch to the next client. Finally, Client n forwards the batch of messages to the server. Regardless of the specifics of how this done, it is obvious that total communication is at least $\mathrm{O}\left(n^2\left|m\right|\right)$, where $|m|$ is the size of a single message block. If the clients had simply sent their message blocks directly to the server, the total communication would be $\mathrm{O}(n|m|)$, so routing though the mixnet, in this case, imposes a factor of n increase in communication. We now discuss various ways to construct such a mixnet and provide a more accurate estimate of the complexity of each.

With a mixnet based on *onion routing*, each client encrypts their message block n times using the public keys of clients n through 1, in that order. Then, when mixing the batch of messages, each client can remove one layer of encryption by decrypting with their private key. After Client n has mixed and decrypted the messages, the plaintext message blocks are sent to the server. If it is undesirable to allow Client n to see the plaintext message blocks, the blocks can be pre-encrypted with the server's public key.

To prevent an adversary from simply encrypting the output of a particular mix node with that node's public key and then comparing the resulting ciphertext to the inputs of that node to determine the mixing permutation, a probabilistic public key encryption scheme must be used. Such schemes inevitably expand the message. At the very least, the ciphertext must be some constant number of bits larger than the plaintext. After the n encryptions needed to form the onion, each ciphertext will have size at least $|m| + \mathrm{O}(n)$. This makes the total communication cost of this scheme at least $\mathrm{O}(n^2|m| + n^3)$; obviously favouring large message blocks. Note that the pre-encryption with the server's public key need not be probabilistic as no one other than the server and the client that sent the message will have access to the plaintext. One advantage of this scheme is that each client must, necessarily, be involved in the mixing of the messages. Malicious clients cannot easily route around the honest ones.

Another technique for constructing mixnets is to use a probabilistic encryption scheme that allows re-encryption or universal re-encryption [16]. Re-encryption is, informally, the transformation of one ciphertext into another ciphertext that is an encryption of the same message under the same key, but which cannot be linked to the original ciphertext as being such (i.e. the new ciphertext looks entirely different). With non-universal re-encryption, each mix node must know the public key under which each ciphertext is to be re-encrypted. Because of this, all ciphertexts must be encrypted under the same public key. If this public key is that of the server, then any collusion between the server and a client will allow the two to decrypt all of the messages passing through that client. The use of universal re-encryption allows the various messages to be encrypted under different public keys as the public key under which a particular ciphertext is encrypted need not be known to the node doing the re-encryption. Even in this case, what public key does one use?

Another problem with the use of re-encryption is ensuring that every client has a chance to shuffle the messages. That is, malicious clients cannot route the batch of ciphertexts around honest clients. To combat this problem a *verifiable shuffle* may be used [5, 31, 33, 32, 37]. With a verifiable shuffle, each mix node provides some sort of proof that the outgoing ciphertexts are indeed a re-encryption of the input ciphertexts in some permuted order, thus allowing others to verify the correct operation of the node. In [40], a zero knowledge proof is given to convince others that a mix node shuffling secret shares has not tampered with the shares. Unfortunately, constructing and verifying these proofs can add considerably to the time and communication complexity of the mixnet.

6.2 Dining Cryptographers

In its most basic instance, at most one of the "dining cryptographers" has a message to transmit "I paid for the meal." The other cryptographers at the table are only interested in learning whether or not one of them paid for the meal, not which one. Chaum's story to motivate this problem is that the cryptographers have been informed that someone has paid the bill for the entire table. The cryptographers are both interested in preserving their anonymity, and hence cannot reveal who paid, but are also concerned that the U.S. National Security Agency (NSA) has secretly funded their dinner. If, upon execution of the protocol, they find that no one indicates that they paid the bill, then their NSA fear may be justified. More sophisticated variants involve multi-bit messages and a mechanism by which multiple parties may each have a message to transmit. The latter of these variants is where this work contributes.

In a dining cryptographers network (DC-net) at most one party can successfully transmit within each slot, where a slot is either in time or in space. If slots are spread out in time (the clients take turns) and the overall message rate is low, the clients may simply transmit when they want and, in the event of a collision, employ a random back-off and retransmit protocol. However, if slots are spread out in space, the message rate is relatively large, or if high efficiency is required, the clients must first agree on an ordering of the slots, i.e. a permutation of themselves, and then each transmit their message in the appropriate slot. Furthermore, to reduce the damaging effects of a collusion among players, such a permutation should be kept secret in the sense that each party should only know their slot number. We note that in "Dining Cryptographers Revisited", Golle and Juels [17] comment on the problem of generating such a secret permutation:

> "The problem can be avoided through techniques like secure multiparty computation of a secretly distributed permutation of slots among players, but this is impractical."

We hope the secret permutation generation primitive introduced in the previous chapter satisfies this requirement.

The Golle and Juels paper addresses the issue of robustness in a DC-net by allowing parties to detect those who seek to jam the network and recover from their actions. Their work is orthogonal to ours and we believe some combination of the techniques used there and here may be possible in future work.

6.2.1 Our Requirements

The problem we seek to solve is the following. Some number of parties, say n, referred to as *clients*, each have a message of some fixed length to transmit. The clients participate in a multi-round interactive protocol which utilizes only a broadcast channel, and at the end of the protocol anyone in possession of a complete transcript of the broadcasts, say a *server*, can run a reconstruction algorithm to recover the messages. Finally, as an unavoidable side-effect of reconstruction, the output messages are reordered according to some secret permutation relative to the ordering of the clients.

Formally, each client takes as input the parameters κ, n, i and M_i, where κ is the security parameter, n is the number of clients, $i \in \{1, \ldots, n\}$ is the index of the client and M_i is the client's message. All messages are the same length, say $|M_i| = \ell$.

A DC-net scheme must have, at least, the following two properties:

- **Correctness.** An algorithm exists which runs in time polynomial in κ and, on input the transcript of broadcasts, outputs the multiset of messages.

- **Permutation indistinguishability.** The server (now an adversary) learns nothing about the permutation from the transcript of broadcasts (beyond what may be guessed from the content of the messages). Specifically, if the server provides two message assignments (n-vectors) that are the same multiset, from which one is chosen as input to the clients, then the probability that the server can determine which message assignment was chosen will be negligibly (in κ) greater than 1/2. We assume the clients are honest and the server is computationally bounded (running in time polynomial in κ).

In addition to these properties, we are only interested in DC-net schemes that can be divided into two phases: a *setup phase* and a *message phase*. The setup phase is comprised of all rounds except the last and proceeds independently of the messages. The final round is the message phase, is obviously non-interactive, and is the only round that is dependent on the message.

In practice, the message may be broken into fixed size blocks and sent in a series of sub-rounds (steps) during the message phase. Despite this, we require that each client proceed independent of the others during this phase.

We are interested in creating a DC-net scheme that has some or all of the following additional properties:

- **Setup efficient.** The total size of the broadcasts made during the setup phase should be independent of the size of the messages. Furthermore, we prefer that the size of these broadcasts be proportional to a small polynomial in n, but we do not state a specific requirement.

- **Message efficient.** The total size of the broadcasts made during the message phase should be linear in the total size of the messages. We will refer to the ratio of the size of the messages to the size of the broadcasts as the *rate* of the DC-net. The DC-net is message efficient if this rate is constant.

- **Collusion resistance.** Suppose $h < n$ clients are honest and the remaining $n - h$ clients are chosen (in advance) to reveal all their secrets to the server (i.e. they are chosen to collude with the server). As with permutation indistinguishability, the server provides two message assignments that are the same multiset; however, these message assignments must agree on the messages provided to the $n - h$ colluding clients. If one assignment is chosen as input to the clients, the probability that the server can determine which message assignment was chosen will be negligibly greater than 1/2. Collusion resistance may be considered in two settings: the colluding clients are semi-honest or the colluding clients are malicious, they go last in each round, their input is irrelevant, and the honest clients may start sending garbage if things go wrong. As before, the server (and any malicious clients) are computationally bounded.

In this chapter we will describe two DC-net protocols. The first is setup efficient and collusion resistant but not message efficient, while the second is message efficient and either setup efficient or collusion resistant. The second protocol we describe can be made to have all of the above properties but only if a key-evolving primitive providing key independence for discrete logarithm based cryptosystems exists. This is currently an open problem and therefore we are unable to provide a DC-net protocol providing all of the desired properties at this time. We describe the needed key-evolving protocol and discuss this open problem in greater detail toward the end of the chapter.

Before describing our DC-net schemes, we must introduce the notions of *private information retrieval* (PIR) and *oblivious transfer* (OT) as such protocols are a required primitive in both of our DC-net schemes. Furthermore, PIR is a useful way to anonymously retrieve data that was stored anonymously using a DC-net.

6.3 Private Information Retrieval (PIR)

PIR, first introduced by Chor, *et. al.* in 1995 [9], is a method by which a user can request and later receive data from a database without revealing to the database the name, index or any other information about the data requested, including the data itself. The database can process the query and return the correct data item (in some encrypted form) without ever learning anything about the data requested.

PIR schemes come in two flavours: ones providing information theoretic privacy and ones providing computational privacy. With the former, privacy is maintained regardless of the servers' processing capabilities; however, to achieve this level of privacy with acceptable communication complexity, one must either replicate the database among multiple non-communicating servers or make use of auxiliary servers [14]. On the other hand, if one assumes one-way functions exist [3], along with certain common intractability assumptions, computational privacy can be achieved with only a single server running in polynomial time.

6.3.1 Information Theoretic Privacy

It has been proven that no PIR scheme sending less than the entire database to the user can have information theoretic privacy and be implemented with a single server [9]. Therefore, to achieve lower communication complexity, IT-PIR schemes require that either the database be replicated among multiple servers or auxiliary servers holding data that is independent of the database be employed [14]. To maintain the user's privacy, these servers must not be allowed to communicate during the online phase of the protocol. Furthermore, for each query the user wishes to make, the user prepares a request for each of the servers and then reconstructs the answer to the query from the replies received from the servers.

Two modifications to the basic schemes are possible. With a threshold scheme, the parameter t specifies the maximum number of servers that may conspire against the user without breaking the user's privacy. For the basic schemes, we can say $t = 1$. As t increases, the communication complexity of the scheme also increases to pay for this insurance. If t is equal to the number of servers, then the servers may be considered parts of a single server and they will necessarily need to return the entire database to the user in response to each query.

Robust PIR addresses the issue of server failure. Protocols have been developed to allow successful data reconstruction provided a sufficiently large fraction of the servers responded to the query. The definition of "sufficiently large" is a parameter set before hand. Beimel and Stahl [4] also give a protocol that can tolerate byzantine servers that maliciously corrupt their reply to a query.

6.3.2 Computational Privacy and Oblivious Transfer

Kushilevitz and Ostrovsky [22] describe the first single-server PIR scheme with sub-linear communication complexity. Their scheme relies on the hardness of computing quadratic residues modulo a composite and has communication complexity $\mathrm{O}\left(n^{\epsilon}\right)$, for any $\epsilon > 0$.

Chang [6] describes a single server PIR scheme that requires only logarithmic communication and is based on Paillier's cryptosystem [36]. Chang's scheme can also be directly used to implement 1-out-of-n oblivious transfer (OT).

OT is closely related to PIR in that it also involves a transfer of information from a sender (server) to a receiver (user) without the server knowing which information was sent; however, with 1-out-of-n OT, the privacy of the database is preserved as the user only receives the one item that was sent and no information about the other database items.

Since our DC-net schemes will require a single server OT solution, we recommend the use of Chang's scheme and briefly describe it here. This description of both Paillier's cryptosystem and the basic Chang scheme are duplicated from [6] with some minor changes in notation.

Paillier's Cryptosystem

Let $N = pq$ be an RSA modulus where p and q are two large safe primes, and let $g \in \mathbb{Z}^*_{N^2}$ be an element whose order is some multiple of N. Encryption of $a \in \mathbb{Z}_N$ is performed by choosing $b \in_R \mathbb{Z}^*_N$ and computing

$$w = \mathcal{E}_g(a, b) = g^a b^N \bmod N^2 \ . \tag{6.1}$$

At the heart of Paillier's cryptosystem is the fact that, for any given $w \in \mathbb{Z}^*_{N^2}$, there exists a unique pair $a \in \mathbb{Z}_N$ and $b \in \mathbb{Z}^*_N$ such that (6.1) holds. Furthermore, if one knows the factorization of N, one can compute

$$a = \mathcal{D}_g(w) = \frac{L[w^\lambda \bmod N^2]}{L[g^\lambda \bmod N^2]} \bmod N \ , \tag{6.2}$$

where $L[v] = (v-1)/N$ and $\lambda = \mathrm{lcm}(p-1, q-1)$. Once a is known, b can be found (if needed) by computing an N-th root mod N.

Paillier's cryptosystem is *additive homomorphic*, having the following two properties:

- $\mathcal{D}_g(\mathcal{E}_g(m_0, r_0)\mathcal{E}_g(m_1, r_1)) = m_0 + m_1 \bmod N$, and
- $\mathcal{D}_g(\mathcal{E}_g(m_0, r_0)^c) = cm_0 \bmod N$.

Basic PIR Scheme

Let $n = \ell^2$ be the size of the database held by the server (in bits) and $x(i, j)$, $0 \le i, j < \ell$, denote the bit $x[i\ell + j]$. Also let $I(t, t_0)$ be an indicating function such that $I(t, t_0) = 1$ if and only if $t = t_0$, otherwise $I(t, t_0) = 0$. We assume the user wants to learn bit $x(i^*, j^*)$. The basic scheme, called *PIR on 2-Hypercube*, is:

- **Initializing:** User sends $\alpha_t = \mathcal{E}_g(I(t, i^*), r_t)$ and $\beta_t = \mathcal{E}_g(I(t, j^*), s_t)$ to the server for $0 \le t < \ell$, where $r_t, s_t \in_R \mathbb{Z}^*_N$.
- **Filtering:** The server computes $\sigma_i = \prod_{t=0}^{\ell-1} {\beta_t}^{x(i,t)} \bmod N^2$ for $0 \le i < \ell$.
- **Splitting-and-then-filtering:** The server splits each σ_i by computing $u_i, v_i \in \mathbb{Z}_N$ such that $\sigma_i = u_i N + v_i$, and then sends $u = \prod_{t=0}^{\ell-1} {\alpha_t}^{u_t} \bmod N^2$ and $v = \prod_{t=0}^{\ell-1} {\alpha_t}^{v_t} \bmod N^2$ to the user.
- **Reconstructing:** The user computes $x(i^*, j^*) = \mathcal{D}_g(\mathcal{D}_g(u)N + \mathcal{D}_g(v))$.

With this scheme, server-side communication is 2κ bits and user-side communication is $2\kappa n^{1/2}$ bits, where $\kappa = 2\log N$ is the security parameter. The paper [6] goes on to describe PIR on a c-Hypercube, which has server-side communication of $2^{c-1}\kappa$ bits and user-side communication of $c\kappa n^{1/c}$. By choosing the constant c appropriately, and given that κ must be at least $\log n$, user-side communication can be made $\mathrm{O}\left(n^{\epsilon}\log n\right)$ for arbitrarily small ϵ while server-side communication remains $\mathrm{O}\left(\log n\right)$.

While the algorithm described above is for a database with single bit entries, extension of the algorithm to the case where each database entry is an element of $\mathbb{Z}_N$ is trivial.

As a simple test of this PIR scheme, we have implemented it in `C++` using Shoup's `NTL` library [39] for the finite field arithmetic. In testing on a machine with a 1 GHz processor, we found that about 5 minutes were required to query a single 512 bit entry from a database with 2^{16} entries. In this test both the client and the server processes were running on the same machine. The RSA modulus used was 512 bits in length and we chose $c = 4$.

6.3.3 Private Information Storage

Closely related to PIR, Ostrovsky and Shoup have described a protocol for achieving *private information storage* [35]. Their protocol provides information theoretic storage of individual bits in the multi-server database. To our knowledge, no protocols providing private information storage on a single server exist. We propose that a DC-net, combined with a PIR scheme, could be used to implement a form of single database private information storage. To make updates to the database, several clients will use the DC-net to send their updates to the server. To later query the database, single database private information retrieval can be used. The DC-net schemes we present in this paper favour long messages, so this form of private information storage will work best for databases containing a relatively small number of large files. Alternatively, many updates can be sent in a single DC-net session provided the server's ability to link several updates as coming from the same client does not compromise the privacy or anonymity of the clients.

6.4 Collusion Resistant DC-net

We now describe an inefficient (lacks message efficiency) but collusion resistant DC-net protocol with a communication complexity of $\mathrm{O}\left(n\ell\right)$ per client for setup, where ℓ is the per query complexity of the OT scheme, and a rate of $1/n$ during the message transmission phase. Note that this protocol has comparable complexity to previously known DC-net protocols but adds the collusion resistance property.

In addition to secret permutation sharing, the following standard primitives are required:

- **Single server 1-out-of-n oblivious transfer (OT).** Chang's OT scheme with logarithmic communication [6] should suffice and will result in a per client setup complexity of $\mathrm{O}(n \log n)$.

- **Non-mailable bit commitment.** To make the simultaneous revelation of a bit string possible.

- **Stream cipher or pseudo random number generator (PRNG).** Any keyed secure pseudo random number generator will suffice.

The scheme described here is essentially a simple secret sharing scheme. Each client splits his message into n shares and (in a sense) distributes these shares to the clients. The clients then send all of their shares to the server. From these n^2 shares, the server can recover the n messages, but does not learn the origin of each message.

During setup, each client distributes to every other client a key for a pseudo random number generator (PRNG). Then, the way each client splits their message is by having the other clients use the PRNG to generate their shares of the message, and the client with the actual message compute the sum of the message and the output of the PRNG keyed with each of the keys the other clients have. The security of this scheme relies on the fact that the server cannot distinguish between pure PRNG output and PRNG output that has been summed with the message.

1. **Setup phase:**

   (a) The clients engage in secret permutation sharing to generate a permutation of themselves $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$. Each client j will know the value $\pi(j)$ but no others.

   (b) Each client, say Client j, chooses $n - 1$ keys to use as seeds for the PRNG and one extra key. Call them $K_{i,j}$, where $1 \le i \le n$. The key $K_{\pi(j),j}$ is the one not used to seed a PRNG and is only present because a malicious client might request it in the next step.

(c) To properly encrypt each message block, Client i needs to know $K_{\pi(i),j}$, for all $j \neq i$; however, to ensure that Client i does not reveal $\pi(i)$ to Client j, Client i must request $K_{\pi(i),j}$ using OT.

(d) To ensure each key, except $K_{\pi(i),i}$, is fetched exactly once, the clients perform a simple parity calculation. After completing the OT queries, each client will have $2n-2$ keys (not including $K_{i,\pi(i)}$). Let c_i be the bitwise xor of the $2n-2$ keys held by Client i. The clients then simultaneously reveal their checksums. Since such simultaneity is difficult in practice, the clients will first commit to their c_i using the bit commitment scheme and then reveal the checksums. After all of the checksums have been revealed, each client can verify that they xor to the all-zero bit string. This result is expected because each key is held by exactly 2 clients.

2. **Message phase:**

(a) Client i has message m_i to send to the server. Let $r_{i,j}$ be the next output from the PRNG keyed with $K_{i,j}$. Client i will generate $2n-2$ such blocks of random bits, each having length equal that of the message block.

(b) Compute

$$e_i^{\pi(i)} = m_i \bigoplus_{j \neq i} r_{\pi(i),j} \ , \tag{6.3}$$

and set $e_i^j = r_{j,i}$ for all $j \neq \pi(i)$. The ciphertext will be a concatenation of n blocks $E_i = e_i^1 \| e_i^2 \| \cdots \| e_i^n$.

(c) Send the ciphertext E_i to the server.

3. **Reconstruction algorithm:**

(a) The server receives n ciphertext messages E_i. To decrypt the server simply computes the bitwise xor of all of these messages. The result will be n concatenated plaintext message blocks

$$m_{\pi^{-1}(1)} \parallel m_{\pi^{-1}(2)} \parallel \cdots \parallel m_{\pi^{-1}(n)} \ . \tag{6.4}$$

Message Integrity

Each message is essentially encrypted with $n-1$ instances of a stream cipher, and as such, are susceptible to bit manipulation by malicious clients. One can protect against such manipulation using any technique applicable to stream ciphers. For example, one might structure each plaintext message block as the concatenation

$$r \parallel m \parallel \mathcal{H}(r \parallel m) \ , \tag{6.5}$$

where r is a sufficiently large number of secret random bits, m is the actual message block, and $\mathcal{H}$ is some secure hash function. The server would reject any decrypted message that does not have this form.

6.5 An Efficient DC-Net

The DC-net scheme described in this section is loosely based on Paillier's homomorphic cryptosystem [36], and it can achieve both our efficiency requirements and collusion resistance if one has a key-evolving scheme for discrete logarithm based cryptosystems that supports key independence. Unfortunately, the existence of such a scheme is currently an open problem in cryptography. Without this primitive, the protocol we describe is message efficient and either setup efficient or collusion resistant, but not both. We will first describe a basic protocol that is efficient but not collusion resistant, and then later we discuss the changes needed to add collusion resistance.

The basic idea is that the clients will send the server a $n \times n$ permutation matrix corresponding to some randomly chosen permutation π. During each message sub-round, the server receives a column whose n rows are the ciphertext messages from the clients and then computes the "matrix product" of the permutation matrix and this ciphertext column to produce a column of plaintext messages (in the permuted order). The actual calculation is not exactly matrix multiplication as we do exponentiation in place of multiplication when computing the inner product of the column and each row of the matrix. Obviously, this permutation matrix cannot have the traditional 0's and 1's. Instead, each entry is encrypted in such a way that the server cannot distinguish 0's from 1's. Finally, "multiplication" with the permutation matrix does not only permute the messages, but decrypts them as well. If any of the ciphertexts are missing or corrupted, none of the decryptions will yield any information about the plaintext messages.

6.5.1 Composite Modulus Discrete Logarithm

Recall our brief introduction to the Paillier cryptosystem in Section 6.3.2. Paillier uses a partial discrete logarithm calculation as a decryption function, but for our DC-net protocol, we will use a similar discrete log calculation to encrypt each plaintext message block that is to be sent to the server. Each client will have their own RSA modulus and the server will need to combine calculations done using each of these moduli. To facilitate this, the server will choose an integer s and all messages blocks will be interpreted as elements of $\mathbb{Z}_s$. The only restrictions on s are that $s < N$ and $\gcd(s, N) = 1$. This latter restriction will most certainly be satisfied provided the party generating N is competent.

The problem we solve is, given $m \in \mathbb{Z}_s$, find e such that

$$m = (g^e \bmod N^2) \bmod s \ . \tag{6.6}$$

Note that this expression is not well defined if one considers the entity in the brackets to be an equivalence class. Instead, we must insist that from now on anytime we write $A \bmod B$ we mean the least non-negative residue (i.e. the common residue). We claim an e satisfying (6.6) can be found if one knows $\lambda = \mathrm{lcm}(p-1, q-1)$, but we take this one step further. Given any fixed integer x, we find y such that

$$m = (g^{x+y\lambda} \bmod N^2) \bmod s \ . \tag{6.7}$$

To see that such a y must exist, first note that the set of N distinct residues $\left\{ g^{x+y\lambda} \bmod N^2 \;\middle|\; 0 \le y < N \right\}$ is the same, but with a different element ordering, as the set $\{ (g^x \bmod N) + zN \mid 0 \le z < N \}$. This follows from $g^\lambda = 1 \bmod N$. Then, with $s < N$ and s prime to N, we have that this set must form a complete set of residues modulo s (and include $N - s$ duplicates). Note that for some m, more than one y in $0 \le y < N$ may satisfy the relation, but we only concern ourselves with finding one such y.

To find a satisfying y, we start by writing

$$m + us = g^{x+y\lambda} \bmod N^2 \ , \tag{6.8}$$

for some non-negative integer u. If this expression holds modulo N^2, it must also hold modulo N, but in this latter case we note that $g^\lambda = 1 \bmod N$ and write

$$m + us = g^x \bmod N \ . \tag{6.9}$$

Solving for u gives

$$u = (g^x - m)s^{-1} \bmod N \ . \tag{6.10}$$

This u is in the range $0 \leq u < N$, but there are some values of $u \geq N$ for which $m + us < N^2$. In particular, if $u' = u + N$, where u is as calculated above, is such that $m + u's < N^2$, then u' can be used to find an alternate solution for y. Since we do not care which value of y we find, we will not consider this possibility further.

Now, to calculate y, we just rewrite (6.8) as

$$(m + us)g^{-x} = g^{y\lambda} \bmod N^2 \ , \tag{6.11}$$

and make use the identity $y = L[g^{y\lambda}]/L[g^\lambda] \bmod N$. This identity is just (6.2) with $w = g^a$. This gives us

$$y = \frac{L[(m + us)g^{-x} \bmod N^2]}{L[g^\lambda \bmod N^2]} \bmod N \ . \tag{6.12}$$

6.5.2 Basic Protocol

As with the protocol given in Section 6.4, this protocol is broken into three phases.

1. **Setup phase:**

   (a) The clients choose a random permutation of themselves $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$.

   (b) Set $s = 2^\kappa$, where κ is the security parameter. Note that each message block will be interpreted as an element of $\mathbb{Z}_s$.

   (c) Each client must generate an RSA modulus. Client j chooses safe primes p_j and q_j to form the modulus $N_j = p_j q_j > s$ and then broadcasts N_j to all other parties. Let $\lambda_j = \mathrm{lcm}(p_j - 1, q_j - 1)$.

   (d) Each client then chooses random ring elements with maximal order $\alpha_j \in_R \mathbb{Z}^*_{N_j^2}$ and $\beta_j \in_R \mathbb{Z}^*_{N_j^2}$, $|\langle\alpha_j\rangle| = |\langle\beta_j\rangle| = \lambda_j N_j$. Also, for each $i \neq \pi(j)$, choose a random exponent $b_{i,j} \in_R \mathbb{Z}^*_{\lambda_j}$, and compute $\beta_{i,j} = \beta_j^{b_{i,j} N_j}$. Client j is responsible for generating column j of the permutation matrix to be used by the server:

$$\begin{pmatrix} \beta_{1,j} \\ \vdots \\ \beta_{\pi(j)-1,j} \\ \alpha_j \\ \beta_{\pi(j)+1,j} \\ \vdots \\ \beta_{n,j} \end{pmatrix} . \tag{6.13}$$

Notice that α_j is in row $\pi(j)$. Each client sends the column it generated to the server and the server will assemble these columns to form its permutation matrix. For reasons given below the server cannot distinguish between α's and β's.

(e) Each client j sends the exponents $b_{\pi(i),j}$ to the respective clients $i \neq j$.

(f) Finally, each client needs to choose a seed for a pseudo random number generator and use this PRNG to generate a sequence of exponents $1 \leq x_{j,t} < \lambda_j$, where t is the sub-round number. These seeds, along with β_j, need to be shared with all clients so they can compute the sequence $\gamma_{j,t} = \beta_j^{x_{j,t}N_j} \bmod N^2$; however, the values $x_{j,t}$ must be kept secret from the server.

2. **Message phase:**

(a) In sub-round t, Client i has message block $m_{i,t} \in \mathbb{Z}_s$, random ring elements $\gamma_{j,t} \in \mathbb{Z}^*_{N_j^2}$ and the exponent $x_{i,t}$.

(b) First compute $m'_{i,t} = m_{i,t} - \sum_{j \neq i} (\gamma_{j,t}^{b_{\pi(i),j}} \bmod N_j^2) \bmod s$.

(c) The ciphertext message Client i must output is an exponent $e_{i,t}$ which satisfies the equations

$$\begin{cases} m'_{i,t} &= (\alpha_i^{e_{i,t}} \bmod N_i^2) \mod s \\ \gamma_{i,t} &= \beta_i^{e_{i,t}N_i} \mod N_i^2 \end{cases} . \tag{6.14}$$

Since $\beta_i^{N_i}$ has order λ_i (or some factor of λ_i) and $\gamma_{i,t} = \beta_i^{x_{i,t}N_i}$, we can set $e_{i,t} = x_{i,t} + y_{i,t}\lambda_i$ and compute $y_{i,t}$ by solving the first equation. But this equation is just (6.7) so one can find $y_{i,t}$ by evaluating (6.10) and (6.12).

(d) Each client sends its ciphertext $e_{i,t}$ to the server.

3. **Reconstruction algorithm:**

   (a) The server has a $n \times n$ matrix that consists of elements α_j and $\beta_{i,j}$, $i \neq \pi(j)$; however, since the server cannot distinguish α's from β's, let us set $\beta_{\pi(j),j} = \alpha_j$ and say the server has the matrix $\{\beta_{i,j}\}$. The server can recover the plaintext message blocks $m_{i^*,t}$, for $1 \leq i^* \leq n$, by computing

$$m_{i^*,t} = \sum_j ({\beta_{i^*,j}}^{e_{j,t}} \bmod N_j^2) \bmod s \ . \tag{6.15}$$

   Note that $m_{i^*,t}$ is the message block sent by client $i = \pi^{-1}(i^*)$.

The communication complexity of the setup phase is $\mathrm{O}(n)$ per client and the rate for Client i is $\log s \,/\, \log N_i \lambda_i$, which is approximately $1/2$ if $N_i \approx s$.

6.5.3 Correctness

The server, when computing message block $m_{i^*,t}$, is really recovering Client i's plaintext via the equation

$$m_{i,t} = (\alpha_i^{e_{i,t}} \bmod N_i^2) + \sum_{j \neq i} ({\beta_{\pi(i),j}}^{e_{j,t}} \bmod N_j^2) \bmod s \ . \tag{6.16}$$

Since $\beta_{i,j}$ has order λ_j and $\beta_{i,j} = \beta_j^{b_{i,j} N_j}$, we can expand the above equation to

$$m_{i,t} = (\alpha_i^{e_{i,t}} \bmod N_i^2) + \sum_{j \neq i} ({\beta_j}^{x_{j,t} b_{\pi(i),j} N_j} \bmod N_j^2) \bmod s \ . \tag{6.17}$$

Now, $\gamma_{j,t} = \beta_j^{x_{j,t} N_j}$, so we simplify to get

$$m_{i,t} = (\alpha_i^{e_{i,t}} \bmod N_i^2) + \sum_{j \neq i} ({\gamma_{j,t}}^{b_{\pi(i),j}} \bmod N_j^2) \bmod s \ . \tag{6.18}$$

Finally, note that $m'_{i,t} = (\alpha_i^{e_{i,t}} \bmod N_i^2) \bmod s$, as dictated by the first equation of (6.14), so

$$m_{i,t} = m'_{i,t} + \sum_{j \neq i} ({\gamma_{j,t}}^{b_{\pi(i),j}} \bmod N_j^2) \bmod s \ , \tag{6.19}$$

which is equivalent to the equation evaluated in Step 2b of the protocol.

6.5.4 Security

The security of this scheme against an honest but curious server depends on two factors: the decisional composite residuosity assumption (DCRA) and the fact that the server does not know any of the exponents $x_{i,t}$.

The decisional composite residuosity assumption [36] simply states that "there exists no polynomial time distinguisher for N-th residues modulo N^2." This assumption ensures that, given columns of the form (6.13), the server cannot distinguish the α's from the β's.

Any party having knowledge of both $x_{i,t}$ and $e_{i,t}$ for some i and t can compute $y_{i,t}\lambda_i$, and given this multiple of λ_i very likely factor N_i. This fact is the primary reason why our efficient scheme is not also collusion resistant (and cannot be easily made collusion resistant, as discussed in the next section). Every client knows all of the exponents $x_{i,t}$ and the server knows the ciphertexts $e_{i,t}$. If any client colludes with the server, they can factor every other clients' moduli.

6.6 Towards Efficiency and Collusion Resistance

To bring collusion resistance to the efficient scheme, the steps 1a, 1e and 1f of the setup phase must be altered. For steps 1a and 1e, the changes are straightforward; however, without an appropriate key-evolving scheme the necessary changes to Step 1f are not possible.

Step 1a: The modification required to this step is to make use of a secret permutation sharing scheme to generate the permutation π.

Step 1e: Client j cannot send exponent $b_{\pi(i),j}$ to Client i as Client j does not know $\pi(i)$. Instead, Client i must request $b_{\pi(i),j}$ from Client j and to ensure that Client i does not reveal $\pi(i)$ to Client j, Client i must make this request using OT. Since OT is being used here, each Client j must generate a phony exponent $b_{\pi(j),j}$ even though no other client will request it.

We believe the OT scheme developed by Chang [6] is ideally suited to this task. This scheme is also based on Paillier's cryptosystem so the same RSA moduli can be used. Furthermore, if the exponents being fetched are less than the modulus, a single query returning the entire exponent can be performed. Assuming s is sufficiently large, choosing exponents that are less than s will ensure that each one is small enough to be returned in a single OT query. Finally, since the exponent being fetched is at the same index in each client's database, the same query can be broadcast to all clients. If an efficient broadcast channel is available, this latter point suggests one might choose a OT scheme where a small reply is returned in response to each relatively large query.

As with the collusion resistant scheme presented in Section 6.4, we must ensure that each exponent (except $b_{\pi(j),j}$) is fetched exactly once. Again we propose a simple checksum calculation. Each client first computes

$$c_j = \sum_{i \neq j} b_{\pi(j),i} - \sum_{i \neq \pi(j)} b_{i,j} \mod s \ , \tag{6.20}$$

and then they all reveal their checksums simultaneously. Since such simultaneity is difficult, the clients first perform some sort of bit commitment and then reveal their c_j. All clients verify that these c_j sum to zero (mod s).

Step 1f: Client i needs to generate a pseudo random sequence of exponents $\{x_{i,t}\}$ and all other clients need the pseudo random sequence $\{\gamma_{i,t}\}$, where $\gamma_{i,t} = \beta_i^{x_{i,t}N_i} \bmod N_i^2$. As discussed in the previous section, anyone knowing both $e_{i,t}$ and $x_{i,t}$, for one particular t, can likely factor N_i. Therefore, to achieve collusion resistance, clients other than i must not learn $x_{i,t}$, but still compute $\gamma_{i,t}$.

This problem is identical to the problem of constructing a key-evolving protocol for discrete logarithm based cryptosystems that provides key-independence. Lu and Shieh [23] describe two such protocols. Their first protocol, which works in $\mathbb{Z}_p^*$ but could be adapted for $\mathbb{Z}_N^*$, provides t-bounded key-independence. This protocol could be used to provide a collusion resistant DC-net provided at most t message/reconstruction subrounds are performed. Unfortunately, if one considers the additional overhead required in the setup phase, this DC-net protocol is no better than the inefficient collusion resistant protocol described in Section 6.4. Nevertheless, in applications where achieving a high rate in the message phase is more important than the complexity of the setup phase, this protocol may have value.

Protocol 2 of Lu and Shieh claims to provide key-independence in $\mathbb{Z}_N^*$ by making use of a trapdoor discrete logarithm technique first introduced by Maurer and Yacobi [29]. Unfortunately, this method has been discounted as insecure [21]. An effective trapdoor discrete logarithm technique would solve both the key-evolving problem and our collusion resistant DC-net problem, but is such a technique required? To our knowledge no one has proved that a trapdoor discrete logarithm is required to generate this sort of pseudo random sequence where the one party having secret information knows the discrete logarithms of the elements.

Chapter 7

Data Backup

Thus far we have discussed the erasure codes necessary to ensure backed-up data is not lost in the event that some of the encoded blocks become irretrievable, and we have discussed the use of anonymous message delivery and recovery techniques to prevent an adversary from selectively denying service to specific parties. The final component we develop is a technique for breaking one's files into fixed sized blocks and efficiently encoding the changing state of the files over time.

To break a set of files into blocks, one might simply use the `UNIX` tape archive utility `tar`. This tool is intended to be used when backing up files to tape, and newer versions have the ability to pipe the data through `compress` or `gzip` as a means of reducing local redundancies; however, these compression tools usually fail to find the duplication of entire files or large portions of files.

A more difficult problem is efficiently encoding the changes to a set of files over time. Most people have a large number of files that rarely change and only a few files that change frequently (e.g. from one day to the next). An efficient backup strategy will attempt to capture only the data that has changed during each backup period. Traditional methods employ a combination of full backups and incremental or differential backups. Obtaining an optimum mix of these backup types is tricky and infrequent full backups make recovery tedious.

In this chapter we propose a technique by which files can be stored as a set of fixed size blocks and, as additional files are added to this store, the presence of any previously seen blocks within these new files can be easily detected; thus reducing the number of new blocks that must be created. This block detection technique does not rely on the blocks being aligned on block boundaries. It can detect a block lying on any byte boundary.

This solves the problem of recognizing duplicate files or large sections of duplicate data within the initial set of files being backed up and the problem of detecting and encoding only the changes in a set of files that are backed up at regular intervals. Recovery is also simplified as the most recent backup is always, essentially, a full backup.

7.1 The rsync algorithm

The `rsync` algorithm, invented by Tridgell [45], provides a method by which one can update a local file to make it an identical copy of some remote file. If the remote file is largely similar to the local file, say a newer version of the local file, this update can be done using a low-bandwidth high-latency bi-directional communication link. The protocol requires as little as one round of communication (hence its suitability on a high-latency link), is capable of analyzing the differences between the two files without having them located on the same machine, and only transmits these differences from the server to the client.

Suppose the client has a file that can be broken into equal size blocks like so

A	B	C	D	E

,

and that the server's version of the file has been modified so that (in terms of these blocks) it looks like

x	C	D	E	y	A	z

.

The `rsync` algorithm provides a way for the server to determine that its version of the file contains the blocks A, C, D, and E, which the client already has, and that it must only send the partial blocks x, y, and z, along with instructions describing how to assemble these pieces to produce a copy of the server's file.

7.1.1 Rolling Checksum

To ensure the server can find the locations of blocks the client has within its file, the client first sends the server a set of checksums (or hashes) of the blocks it has. The key to the `rsync` algorithm is that there are two checksums computed from each block. One is a cryptographically strong hash function (such as `SHA-256`) which can be assumed to uniquely represent the block. The other is a weak "rolling" checksum.

The rolling checksum, for bytes $[X_s, X_{s+1}, \ldots, X_{t-1}]$, is defined as

$$\begin{aligned} a(s,t) &= \sum_{i=s}^{t-1} X_i && \mod M \\ b(s,t) &= \sum_{i=s}^{t-1} (t-i)X_i && \mod M. \end{aligned} \tag{7.1}$$

We note two well known checksum algorithms that make use of this general construction. Adler's `adler-32` uses $M = 65521$ (the largest prime number smaller than 2^{16}) and is considered to be almost as effective at detecting random errors as a cyclic redundancy check (CRC) but is much faster to compute in software. An even faster but not quite as effective checksum is `Fletcher's checksum`. This checksum uses $M = 65535$ to make the modular reduction slightly faster, but the real key to speed is doing the calculations using 16-bit words instead of 8-bit bytes. These changes can give the Fletcher's checksum calculation as much as a two-fold increase in speed over `adler-32`.

For `rsync`, $M = 2^{16}$ and the calculation is done on individual bytes. The latter is essential to ensure blocks can be matched regardless of their relative alignment. For our data backup application, we extend the rolling checksum beyond 32 bits by using larger values of M.

The "rolling" part of this checksum comes from the observation that if one knows the checksum for a particular block of the file starting at position s and ending at position $t-1$, one can, in constant time, compute the checksum for the block starting at position $s+1$ and ending at position t. This calculation is

$$\begin{aligned} a(s+1,t+1) &= a(s,t) - X_s + X_t && \mod M \\ b(s+1,t+1) &= b(s,t) - (t-s)X_s + a(s+1,t+1) && \mod M. \end{aligned} \tag{7.2}$$

Because of the rolling nature of the checksum, the server can efficiently compute the checksum for a block of a particular size starting at each and every possible byte boundary. As the server works its way through its file computing the rolling checksum at each byte boundary, it can check if the checksum matches any of the checksums provided by the client. If a match is found, the strong checksum of the block in question is computed and checked against the manifest provided by the client. This additional calculation is required to detect the false positives that occur because of the limited range of the rolling checksum and the ease by which collisions can be maliciously generated. If a block is found for which both checksums match, the server can be sure that the client already has a copy of that block and instead of sending the block to the client, the server indicates to the client that it should copy its existing block to the new file being constructed.

If the two files have many whole blocks in common, regardless of the positioning of these blocks, the `rsync` algorithm will greatly reduce the amount of communication required to transmit the new file to the client.

7.2 Using rsync for backup

The `rsync` algorithm can be used for backup to solve the following two problems:

1. achieving compression by only storing the changes in a directory tree from one backup session to the next and

2. achieving compression by recognizing similarities between files within a directory tree during a single backup session.

There have been other attempts at using `rsync` for backup (for example, [20, 46]); however, while these address the first problem to the extent that as long as a file has not changed or moved, it is only stored once, they do not address the second. Another similar tool is `rdiff-backup` [12] which uses *reverse diffs* to efficiently encode changes to files that have not moved, but again, this tool does not address the second problem.

Our proposal is to split the backup copy of a set of files into two parts: a *dictionary* of blocks and an *object store*. The dictionary is simply a collection of fixed size data blocks for which both the rolling (weak) checksum and the cryptographically strong checksum have been calculated. When backing up a file that is at least one block is length, the rolling checksum is computed at each byte offset to determine which, if any, of the blocks currently stored in the dictionary appear within the file being backed up. As such blocks are found, the file is compressed by replacing these blocks with some sort of reference to the block within the dictionary (either the strong hash or an index). Any whole blocks within the original file that are not found in the dictionary are added to the dictionary and similarly replaced with a reference in the original file. Of course, before adding a new block to the dictionary, the algorithm must scan at least one block length further into the file to ensure a partial block of data has not been inserted between two known blocks. In this way, the file being backed up is compressed to a mixture of block references and partial blocks of data. If the resulting file, after compression, is still at least one block in length, the compression algorithm is repeated recursively. The final result will be a file that is smaller than one block in size. We call this resulting data an *object* and store it in the object store.

The object store is simply a hash table where the keys are a cryptographically strong hash of each original file backed up and the values stored are the objects that result from compression of those files. When backing up a directory, a file is created that is the list of files contained in the directory along with the hashes of those files and any other metadata one wishes to preserve.

Our goal is to realize the following benefits:

1. identical files appearing in multiple locations within the directory tree are only stored once,

2. files that have not changed from one backup session to the next are not stored during subsequent backups, even if they have been moved,

3. any pair of files containing whole blocks with identical data will be efficiently stored as those identical blocks are only stored once,

4. files that have partially changed from one backup session to the next are stored efficiently as whole blocks that have not changed are not stored more than once, and

5. a file that has been split into multiple files or multiple files concatenated into one require minimal additional storage.

Although the above list appears to focus on individual files, the main benefit here is a de-emphasis on files in favour of (generally smaller) blocks. Once a block of data has been added to the dictionary, any time an identical copy of that block is observed anywhere within any file in the directory tree, during any later backup session, it will be replaced with a short reference and space will have been saved.

In the next section we discuss the impact of block size on the performance of the algorithm, and then in later sections we describe our implementation of these structures and our empirical results.

7.3 Choice of Block Size

In Tridgell's PhD thesis [44], he suggests the optimal block size for `rsync` is $\sqrt{24n/Q}$, where n is the length of the file and Q is the number of distinct differences (each separated by at least one full block) between the two files. This calculation assumes one is interested in balancing communication between the client and the server, and since our goal is quite different, this calculation is of little use.

Our goal, of course, is to minimize the total size of the dictionary and object store. Having smaller blocks will allow the changes in files to be identified and stored with greater efficiency; however, small blocks imply there will be a large number of them and this will introduce overhead as the blocks must be tracked. When compressing individual files, each block of data is reduced to a reference which, in a practical system, is likely to be at least 64 bits in size, and therefore, one component of this overhead is the ratio of the block size to the size of these references. Another significant component of the overhead is the ratio of the block size to the combined size of the weak and strong checksums (256 bits in our implementation); however, it is possible to avoid computing and storing the strong checksum if one is willing to directly compare each block for which the weak checksum matches to the corresponding blocks in the dictionary, an option that is not possible with `rsync`.

Unfortunately, knowledge of these ratios is not sufficient to compute an optimal block size. Ultimately, the optimal block size is a function of the data being backed up, the nature of its internal redundancy and the way in which the dataset changes over time. Furthermore, the optimal block size may change with the data over time. Therefore, we will take a different approach to determining an appropriate block size; we compute the effective maximum dictionary size as a function of block size.

In theory, the maximum number of blocks that can be stored in the dictionary is unlimited; however, in practice, there is a limit and it is related to how often the rolling (weak) checksum falsely matches a block in the dictionary. Recall that the rolling checksum is updated once for each new byte read from a file being backed up and after each update the checksum is used as an index into the dictionary. If any blocks with a matching checksum are found in the dictionary, either a strong hash is computed and compared to these blocks or the blocks themselves must be compared. Updating the rolling checksum is very fast but comparing blocks is not, so the number of false rolling checksum matches must be minimized.

We propose the following rule of thumb. The mean rate of false positives should not exceed one per block of data processed. Our justification for this rule is as follows. When processing a data file, the strong checksum is computed once per rolling checksum match or new block added to the dictionary. This amounts to computing the strong checksum over the entire file once. A false positive rate of one per block also amounts to computing the strong checksum over the entire file once (making it twice). Our implementation, described below, also does an initial hashing of the entire file to determine if it has already been seen or not. Therefore, depending on whether the file is normally hashed once or twice, this false positive rate increases the total computation by a factor of either 2 or 1.5 respectively. We believe this is a reasonable upper limit on this type of overhead.

With this rule of thumb, we can compute an upper bound on the size of the dictionary. Let 2^ℓ, for some positive integer ℓ, be the block size. From (7.1), ignoring the "mod$M$" parts, we can compute the maximum value for a and b as $a \leq 255 \cdot 2^\ell < 2^{\ell+8}$ and $b \leq 255 \cdot 2^{\ell-1}(2^\ell - 1) < 2^{2\ell+7}$, respectively. This tells us that the entropy to be found in the rolling checksum, i.e. the size of the weak hash, is at most $3\ell + 15$ bits. If we assume that collisions in this hash function are randomly and uniformly distributed, then the collision rate can be kept to less than $2^{-\ell}$ if the size of the dictionary is restricted to $2^{2\ell+15}$ blocks. Table 7.1 summarizes these calculations for a variety of choices of ℓ that may be used in practice.

7.4 Implementation

To test our ideas in a real world setting, we have created an implementation of this backup strategy written in `C++` and utilizing `Berkeley DB` [34] for the object store. This implementation consists of the following components:

- **Dictionary:** Each fixed size block of data to be stored is stored along with a 64 byte structure holding, primarily, the 8 byte rolling checksum, the 24 byte strong hash (`SHA-256` truncated to 192 bits), a 24 byte hash as a "reverse" reference to an object referencing this block, and for the few blocks that are referenced by multiple objects, a link to a linked list (stored in a separate file) of additional reverse references.

  Each block is tested for compressibility using the `deflate` algorithm of `gzip`. If the block compresses to a size at least 16 bytes less than the block size, it is deemed to

ℓ	10	11	12	13	14	15	16
block size (bytes)	1024	2048	4096	8192	16384	32768	65536
hash bits	45	48	51	54	57	60	63
false positive rate	2^{-10}	2^{-11}	2^{-12}	2^{-13}	2^{-14}	2^{-15}	2^{-16}
maximum blocks	2^{35}	2^{37}	2^{39}	2^{41}	2^{43}	2^{45}	2^{47}
maximum bytes[1]	32 TiB	256 TiB	2 PiB	16 PiB	128 PiB	1 EiB	8 EiB

Table 7.1: Maximum dictionary size as a function of the block size. The false positive rate is a maximum as per our rule of thumb. Also, the rolling checksum (weak hash) computation is assumed to produce values that are distributed uniformly within the range and since this assumption does not hold in practice, this maximum dictionary size should be considered an upper bound.

be compressible and is compressed along with up to 15 other compressible blocks. Compressing multiple blocks together can increase the compression ratio, and since files are processed sequentially, compressible blocks are likely to be compressed together with other blocks from the same file (likely increasing the compression ratio further). Although one could simply compress all blocks, we feel that not storing compressed versions of incompressible blocks will improve later read-back performance (thus compensating for the extra "test for compressibility" step we perform).

- **Object Store:** As mentioned, the object store is implemented using a hash table in `Berkeley DB` version 4.4. The keys are simply the 24 byte hash (again `SHA-256` truncated to 192 bits) of the original file and the value is the object data.

  No attempt is currently made to compress objects. For the tests described in our results section below where we report the total compressed size of the backup, we have used the `UNIX gzip` tool to compress the `Berkeley DB` data file and are reporting its compressed size. We believe an online database supporting compression would be of a similar size.

[1]In 1999, the International Electrotechnical Commission introduced a series of prefixes to be used when specifying binary multiples of a quantity and clarified the position that SI prefixes only have their base-10 meaning and never have a base-2 meaning. The IEEE has also adopted this standard and we use it here.

- **`store`**: When given a starting directory, this tool performs a depth first search of the directory hierarchy, first checking to see if each file is already stored in the object store, and then if it is not found, converting each file to an object and storing it. Since storing a file in the object store incurs an overhead of at least 48 bytes (a 24 byte hash stored both with the object and in the directory referencing it), any file of size less than 64 bytes is not stored as an object and is instead included directly in the directory where it was found.

  Directories are stored as `XML` files and are simply lists of file names along with their associated object store keys (or embedded data). `UNIX` style permission bits, user and group ids, and information about special files are also stored.

- **`stat`**: This program computes various statistics from the distribution of data in the dictionary and object store, including:

  - total blocks in the dictionary,
  - ratio of compressed to uncompressed and spaced saved by compression,
  - distribution of reverse references and spaced saved by not storing redundant blocks,
  - distribution of rolling checksum values showing the number of collisions,
  - number of objects, and
  - distribution of object size, mean and median size.

  Many of these statistics are shown in the next section.

7.5 Empirical Results

To assess the potential benefits of this algorithm, we have performed an experiment involving the backing up of 16 versions of my home directory taken at times ranging from February 2004 to January 2007. Furthermore, this experiment has been repeated for the 5 distinct values of the block size corresponding to $10 \leq \ell \leq 14$. For each experiment, the 16 versions of my home directory are backed up one-by-one in chronological order and a variety of statistics were recorded after each version had been processed.

Figure 7.1 shows how our backup strategy compares to the more traditional strategies of raw storage and strategies capable of identifying redundant (or unchanged) files. When

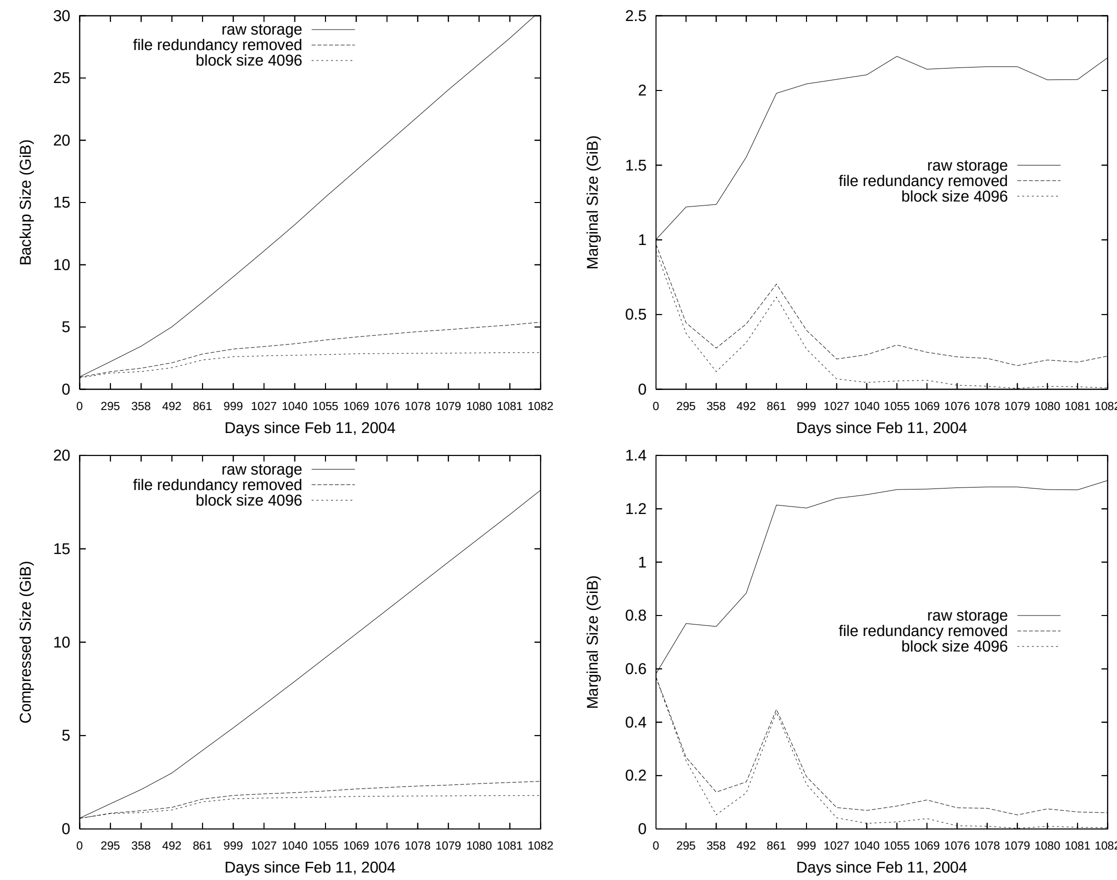

Figure 7.1: Size of backup and marginal rate for both uncompressed and compressed backups. These graphs compare the strategies: raw backup of files, removal of redundant files, and our rolling checksum based method with a block size of 4096. The top graphs are for uncompressed storage while the bottom graphs are the results with compression. With our rolling checksum based strategy, the daily marginal rate for uncompressed storage is less than 20 MiB and with compression it is less than 10 MiB.

looking at only the last week's worth of data (where the backups were made daily), we see that with a redundant file based strategy, the daily marginal rate is about 200 MiB uncompressed or 60 MiB compressed. Our strategy reduces this marginal rate to 20 MiB uncompressed or 10 MiB compressed. For the redundant file based strategy, the time required to double the size of the backup (starting at its current size) would be approximately 1 month, while with our strategy about 6 months are required.

Figure 7.2 compares the various block sizes tested. When the data is stored uncompressed the largest block size (16384) has the worst performance. This is to be expected as there are fewer opportunities for compression due to redundant blocks being found. On the other hand, when the data is compressed, the smallest block size (1024) had the worst performance. We suspect the reason for this is that with a smaller block size more

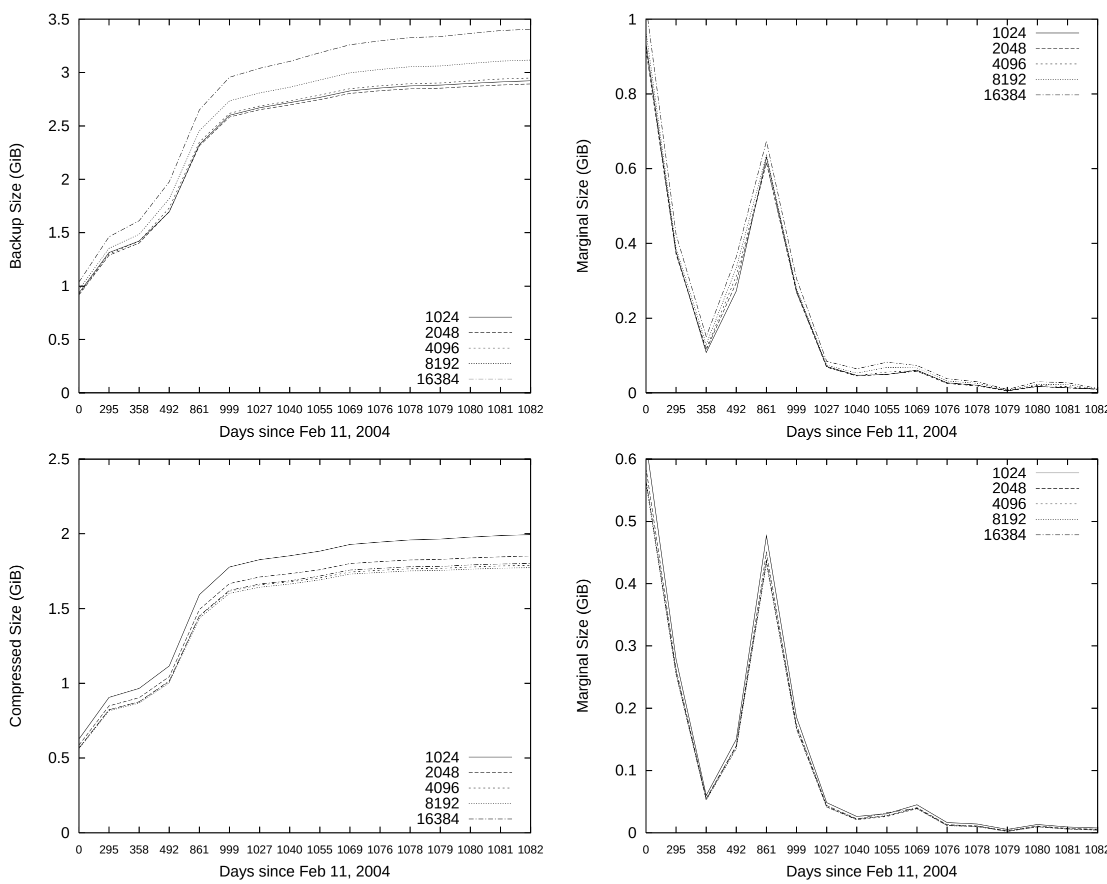

Figure 7.2: Size of backup and marginal rate for both uncompressed and compressed backups as a function of block size. The top graphs are for uncompressed storage while the bottom graphs are the results with compression.

of the data is in the dictionary where not all blocks are compressed; however, with the larger block size more data is in the object store, which we have compressed completely. Furthermore, the heuristic we have used to determine if a block is compressible favours larger block sizes.

Table 7.2 summarizes the final state of the dictionary and object store for each of the block sizes tested. Notice that, as expected, a smaller block size yields a greater reduction in storage due to redundant blocks, but a larger block size allows for more efficient compression of those blocks.

In Section 7.3 we suggested that the most significant factor limiting the size of the dictionary is false matches of the rolling checksum. Figure 7.3 compares the actual false positive rate to the expected (optimistic) value. Counter-intuitively, the false positive rate seems to go down as the dictionary is filled. We suspect this must be due to some peculiar data found in our data set, but we concede that this could also be further evidence that the rolling checksum is not a very good hash function. For a fully populated dictionary, Table 7.2 has data showing the number of blocks having a weak hash that has already been seen and the collision rate that can be calculated from this statistic. These observations suggest that, in practice, it might be best to ensure the size of the dictionary remains well below the maximum size given in Table 7.1.

Table 7.2 also gives the mean and median object size as a function of block size. The reasons these values are considerably less than half the block size (as would be expected) must be due to the distribution in size of the files in the directory hierarchy being backed up. In particular, having a large number of files with a size less than one block will skew the distribution. This effect is larger for the larger block sizes.

Finally, the time required to complete each experiment on an AMD Athlon XP 1800+ machine is the last statistic given in Table 7.2. As expected, having a larger block size reduces the total number of blocks to keep track of and increases I/O efficiency; however, beyond the 4096 byte block size, our software is I/O bound. From this data we recommend that the typical home user with at most a few tebibytes of data should use a 4kiB block size. For a corporate or industrial user a larger block size may be needed to ensure large datasets can be efficiently backed up. Alternatively, we are evaluating possible changes to the rolling checksum to increase its effective size (number of hash bits) while not significantly slowing it down. With such a modification one could choose a smaller block size and still store a large dataset provided their machine has enough RAM to store a complete table of weak hash values.

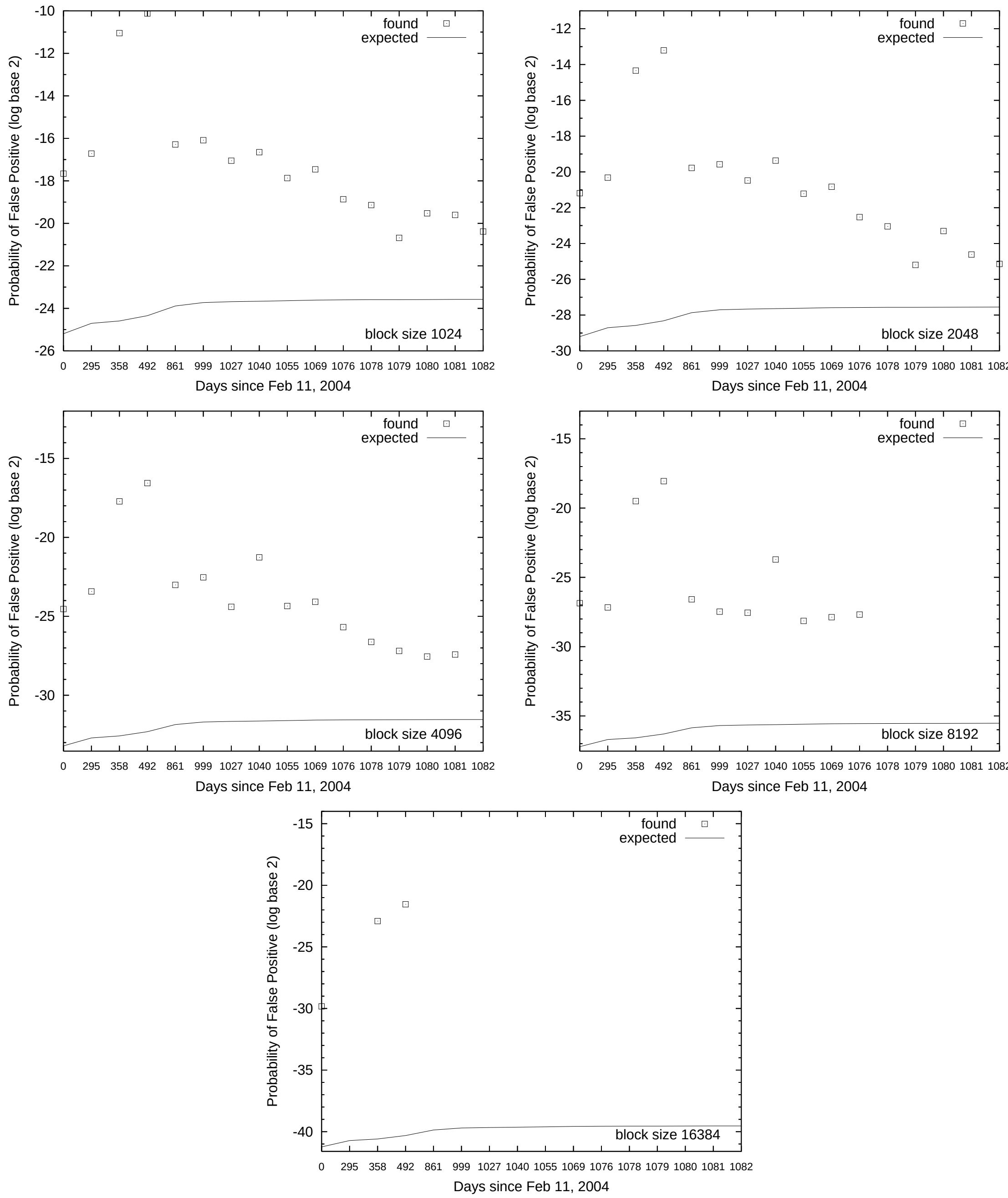

Figure 7.3: Rolling checksum false positive rate for various block sizes. The solid line is the expected rate as a function of the number of blocks currently in the dictionary. In cases where a data point is not present, zero false positives were observed. Note that the top edge of each graph is the maximum tolerable rate as per our rule of thumb.

block size	1024	2048	4096	8192	16384
blocks referenced	4849597	2425240	1207403	601039	297588
blocks stored	2789504	1418705	715928	360283	179383
storage reduction	42.5%	41.5%	40.7%	40.0%	39.7%
compressed blocks	1603328	859895	462052	250669	133856
reduction	34.3%	36.8%	38.2%	38.8%	39.2%
duplicate weak hash	1816	187	33	7	2
collision rate	$2^{-10.6}$	$2^{-12.9}$	$2^{-14.4}$	$2^{-15.7}$	$2^{-16.5}$
expected rate	$2^{-23.6}$	$2^{-27.6}$	$2^{-31.6}$	$2^{-35.5}$	$2^{-39.6}$
total objects	53890	53890	53890	53890	53890
mean size	492	869	1398	2120	3666
median size	474	774	1118	1427	1661
cpu time (hours)	2.4	2.0	1.7	1.5	1.3
clock time	3.1	2.7	2.4	2.4	2.4

Table 7.2: Summary of statistics collected during experiment. Blocks referenced does not include the additional references to blocks referenced more than once within a single object. All tests were performed on a machine with an AMD Athlon XP 1800+ processor. The difference between cpu time and clock time is the time spent waiting for I/O.

Chapter 8

Open Problems

This thesis addresses several of the challenges to be overcome in the development of an Internet based data backup application. We have proposed a new class of low overhead erasure codes that could be useful in such an application. Also, to avoid selective denial of service type attacks, we proposed an anonymous message delivery scheme to be used to prevent an adversary from determining which data blocks belong to each node. Finally, we have proposed a technique for efficiently dividing the data backed up by a user each day into fixed size blocks. Our work in these areas has suggested several avenues for future work.

8.1 Erasure Codes

Our windowed erasure codes have an overhead which is independent of the length of the code and very low (about 2 extra coded symbols), but the price paid for this low overhead is the higher decoder complexity $\mathrm{O}\left(k^{3/2}\right)$. For lower decoder complexity, LT codes [24] have the complexity of $\mathrm{O}\left(k \log k\right)$ but with a higher overhead of $\mathrm{O}\left(\sqrt{k}\right)$ extra coded symbols.

Does there exist a code having both low overhead and a decoder complexity of $\mathrm{O}\left(k \log k\right)$? We consider the following two pieces of evidence. First, if one substitutes $\epsilon = 1/k$ into Theorem 3.1.1 (Theorem 2 from [30]), the theorem is made to read "there is a rateless locally encodable code that can recover the input symbols from any k coded symbols with high probability in time proportional to $k \log k$". The second piece of evidence is Figure 3.1 (right graph) which shows that if the robust soliton distribution is decoded using Gaussian elimination, the overhead appears to be independent of k. This implies that LT codes can have both low overhead and low decoder complexity, but not both simultaneously.

One possibility was suggested at the end of Chapter 3; a combination of LT codes and windowed codes. The degree 2 coded symbols from the robust soliton distribution must be chosen uniformly from the set of $\binom{k}{2}$ possibilities (otherwise duplicates appear too often), but for coded symbols of degree 3 or greater, the chosen input blocks could be confined to a window of size $\mathrm{O}\left(\sqrt{k}\right)$. Assuming such a restriction does not increase the overhead, it may lead to a low complexity decoder capable of recovering the input blocks with an overhead similar to Gaussian elimination.

8.2 Anonymous Message Delivery

We have stated that to make our DC-net protocol both efficient and collusion resistant a key-evolving protocol for discrete logarithm based cryptosystems that provides key-independence is required. Such a protocol is trivial with a trapdoor discrete logarithm primitive, but the issue of whether such a primitive is essential is an open problem.

Another direction for further research is to determine how our efficient DC-net protocol could utilize the work of Golle and Juels [17]. They consider the issue of detecting malicious players who attempt to jam the DC-net. Furthermore, after such a detection and expulsion of the offending parties, they offer techniques for recovering from the attack. This type of robustness will be important to any practical DC-net construction, and therefore, we would want to consider the possibilities of applying these methods to our efficient DC-net protocol.

8.3 Block Based Backup

The only problems with our rolling checksum based backup strategy are technical in nature. While our goal was to encode the files stored in a directory hierarchy into fixed size blocks and the blocks in our dictionary are of fixed size, the objects created vary in size. Furthermore, if one wants to make use of a compression algorithm to further reduce the space needed, the resulting compressed blocks will also vary in size. Packing the resulting compressed blocks and objects into fixed size blocks will be necessary to utilize an erasure code and an anonymous delivery protocol.

Another issue is dealing with a dictionary containing millions or billions of blocks. One must ensure that the hash table used to lookup the rolling checksum as each byte of a file is processed is a fast as possible. We note that the vast majority of these lookups will produce a negative result so one might optimize for this case. Perhaps one could construct a compact in-core structure to be used only to determine if a lookup would result in a hit or not, and only in the case of a possible hit is a, presumably slower, lookup done in a hash table.

8.4 Additional Challenges

In addition to potential improvements to the primitives we have discussed in this thesis, we also note a couple of problems that have not been addressed.

One is to determine how best to spread the encoded data blocks around the Internet. The actual distribution need not be too difficult; merely attempt to spread blocks uniformly among the available nodes using a DC-net scheme to ensure nodes cannot link data blocks to their source. Problems, however, arise when one attempts to determine how to retrieve the data blocks. Usually, the node retrieving a block of data will be the same node that initially published that data block, and therefore, to continue to preserve the unlinkability between data block and source, some sort of private information retrieval technique may be needed.

Also, if one has lost all of their data, they will not know which nodes have each of their data blocks nor the indices of those blocks. Some sort of search on encrypted data may be required to boot strap the recovery process.

The final problem we mention is related to the retrieval of data blocks and is ensuring that nodes keep the blocks that have been entrusted to them. The erasure code allows for some loss, but nodes will likely need to be tested periodically to ensure they are maintaining the blocks. If a node is found to be cheating, sanctions may be necessary; however, if a node has suffered some sort of catastrophic failure, that node must be given a chance to retrieve their data before such sanctions begin to take effect.

Finally, while we believe the methods developed in this thesis will be useful in the construction of an online secure distributed backup protocol, we also note that our erasure code construction and rolling checksum based backup strategies can be used to improve the efficiency and robustness of a more traditional backup strategy. One could use our rolling checksum based backup strategy to minimize the space required to keep an online backup copy of one's data, useful for restoring files deleted accidentally, and from this online backup, one could then use our erasure code construction to improve the robustness of an offline copy of the data stored offsite on some stable media. While distributing one's data globally provides the best insurance against data loss, a local offsite backup copy is certainly preferred to no backup copy at all.

Bibliography

[1] Gagan Aggarwal, Nina Mishra, and Benny Pinkas. Secure computation of the k^{th}-ranked element. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT*, pages 40–55. Springer, 2004. LNCS 3027.

[2] Ross Anderson. The eternity service. In *Proceedings of Pragocrypt '96*, 1996.

[3] Amos Beimel, Yuval Ishai, Eyal Kushilevitz, and Tal Malkin. One-way functions are essential for single-server private information retrieval. In *Proc. of 31st Symposium on Theory of Computing*, pages 89–98, 1999.

[4] Amos Beimel and Yoav Stahl. Robust information-theoretic private information retrieval. In *Proc. of the 3rd Conference on Security in Communication Networks*, 2002.

[5] Dan Boneh and Philippe Golle. Almost entirely correct mixing with applications to voting. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 68–77, New York, NY, USA, 2002. ACM Press.

[6] Yan-Cheng Chang. Single database private information retrieval with logarithmic communication. In Huaxiong Wang, Josef Pieprzyk, and Vijay Varadharajan, editors, *ACISP*, pages 50–61. Springer, 2004. LNCS 3108.

[7] David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–88, 1981.

[8] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1:65–75, 1988.

[9] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *IEEE Symposium on Foundations of Computer Science*, pages 41–50, 1995.

[10] Yvo G. Desmedt and Yair Frankel. Threshold cryptosystems. In *CRYPTO '89: Proceedings on Advances in cryptology*, pages 307–315, New York, NY, USA, 1989. Springer-Verlag New York, Inc. LNCS 435.

[11] Peter Druschel and Antony Rowstron. Past: A large-scale, persistent peer-to-peer storage utility. In *HotOS VIII*, Germany, May 2001.

[12] Ben Escoto. rdiff-backup. `http://www.nongnu.org/rdiff-backup/`.

[13] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Proceedings of CRYPTO 84 on Advances in cryptology*, pages 10–18, New York, NY, USA, 1985. Springer-Verlag New York, Inc. LNCS 196.

[14] Yael Gertner, Shafi Goldwasser, and Tal Malkin. A random server model for private information retrieval or how to achieve information theoretic PIR avoiding database replication. 1998. LNCS 1518.

[15] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 28(2):270–299, 1984.

[16] Philippe Golle, Markus Jakobsson, Ari Juels, and Paul Syverson. Universal re-encryption for mixnets. In *RSA Conference Cryptographer's Track*, pages 163–178, 2004. LNCS 2964.

[17] Philippe Golle and Ari Juels. Dining cryptographers revisited. In *Proceedings of Eurocrypt 2004*, pages 456–473, May 2004. LNCS 3027.

[18] Jim Hamilton and Eric W. Olsen. Design and implementation of a storage repository using commonality factoring. In *MSS '03: Proceedings of the 20^{th} IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03)*, page 178, Washington, DC, USA, 2003. IEEE Computer Society.

[19] V.F. Kolchin. Random graphs. *Cambridge University Press*, 1999.

[20] Kevin Korb. Backups using rsync. `http://www.sanitarium.net/golug/rsync_backups.html`.

[21] Dennis Kügler and Markus Maurer. A note on the weakness of the Maurer-Yacobi squaring method. Technical Report TI-15/99, Fachbereich Informatik, Technische

Universität Darmstadt, 1999. Available at ftp://ftp.informatik.tu-darmstadt.de/pub/TI/-TR/TI-99-15.weaksquaring.ps.gz.

[22] Eyal Kushilevitz and Rafail Ostrovsky. Replication is NOT needed: SINGLE database, computationally-private information retrieval. In *IEEE Symposium on Foundations of Computer Science*, pages 364–373, 1997.

[23] Cheng-Fen Lu and Shiuh-Pyng Winston Shieh. Secure key-evolving protocols for discrete logarithm schemes. In *CT-RSA '02: Proceedings of the The Cryptographer's Track at the RSA Conference on Topics in Cryptology*, pages 300–310, London, UK, 2002. Springer-Verlag.

[24] Michael Luby. LT codes. In *the 43rd Annual IEEE Symposium on Foundations of Computer Science*, 2002.

[25] Michael Luby, Gavin Horn, Jeffrey J. Persch, John Byers, Armin Haken, and Mike Mitzenmacher. On demand encoding with a window. United States Patent and Trademark Office, Patent No. 6,486,803, November 26, 2002.

[26] Michael Luby, Michael Mitzenmacher, M. Amin Shokrollahi, Daniel Speilman, and Volker Stemann. Practical loss-resilient codes. *Proc. of Symposium on Theory of Computing*, pages 150–159, 1997.

[27] David MacKay. *Information Theory, Inference, and Learning Algorithms.* Chapter 50, http://www.inference.phy.cam.ac.uk/mackay/DFountain.html.

[28] Emin Martinian. Distributed Internet Backup System (DIBS). http://web.mit.edu/~emin/www/source_code/dibs/index.html.

[29] Ueli Maurer and Yacov Yacobi. A non-interactive public-key distribution system. *Designs, Codes and Cryptography*, 9(3):305–316, November 1996.

[30] Petar Maymounkov. Online codes, 2002. Technical Report TR2002-833, New York University, October 2002.

[31] C. Andrew Neff. A verifiable secret shuffle and its application to e-voting. In *CCS '01: Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 116–125, New York, NY, USA, 2001. ACM Press.

[32] Lan Nguyen, Reihaneh Safavi-Naini, and Kaoru Kurosawa. Verifiable shuffles: A formal model and a Paillier-based efficient construction with provable security. In Markus Jakobsson, Moti Yung, and Jianying Zhou, editors, *ACNS*, pages 61–75. Springer, 2004. LNCS 3089.

[33] Lan Nguyen, Reihaneh Safavi-Naini, and Kaoru Kurosawa. A provably secure and efficient verifiable shuffle based on a variant of the Paillier cryptosystem. *Journal of Universal Computer Science*, 11(6):986–1010, 2005. http://www.jucs.org/jucs_11_6/a_provably_secure_and.

[34] Oracle. Berkeley DB. http://www.oracle.com/database/berkeley-db/.

[35] Rafail Ostrovsky and Victor Shoup. Private information storage. In *Proc. of 29th Symposium on Theory of Computing*, pages 294–303, 1997.

[36] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, pages 223–238, 1999. LNCS 1592.

[37] Kun Peng, Colin Boyd, and Ed Dawson. Simple and efficient shuffling with provable correctness and ZK privacy. In Victor Shoup, editor, *CRYPTO*, pages 188–204. Springer, 2005. LNCS 3621.

[38] Amin Shokrollahi. Raptor codes, 2003. Available at: http://www.inference.phy.cam.ac.uk/mackay/DFountain.html.

[39] Victor Shoup. NTL: A library for doing number theory. http://www.shoup.net/ntl/.

[40] Marius-Cualin Silaghi. Zero-knowledge proofs for mix-nets of secret shares and a version of ElGamal with modular homomorphism. Technical report, Florida Institute of Technology, May 1 2005.

[41] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *IEEE Symposium on Security and Privacy*, pages 44–55, 2000.

[42] Chris Studholme and Ian Blake. Windowed erasure codes. In *International Symposium on Information Theory (ISIT)*, 2006.

[43] Latanya Sweeney and Michael Shamos. Multiparty computation for randomly ordering players and making random selections. Technical Report CMU-ISRI-04-126, Carnegie Mellon University, School of Computer Science, Pittsburgh, July 2004.

[44] Andrew Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, 1999.

[45] Andrew Tridgell and Paul Mackerras. The rsync algorithm. `http://samba.anu.edu.au/rsync/tech_report/`.

[46] Andreas kre Solberg. rsyncbackup: A backup solution. `http://rsyncbackup.erlang.no/`.