AN ANALYSIS OF ERROR RECONCILIATION PROTOCOLS FOR USE IN QUANTUM KEY DISTRIBUTION

THESIS

James S. Johnson, 1st Lieutenant, USAF

AFIT/GCE/ENG/12-06

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCE/ENG/12-06

AN ANALYSIS OF ERROR RECONCILIATION PROTOCOLS FOR USE IN QUANTUM KEY DISTRIBUTION

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Engineering

James S. Johnson, BS

1st Lieutenant, USAF

February 2012

AFIT/GCE/ENG/12-06

AN ANALYSIS OF ERROR RECONCILIATION PROTOCOLS FOR USE IN QUANTUM KEY DISTRIBUTION

James S. Johnson, BS

1st Lieutenant, USAF

Approved:

Jeffrey W. Humphries, Lt Col, USAF (Chairman)	Date
Michael R. Grimaila, PhD, CISM, CISSP (Member)	Date
Gerald Baumgartner, PhD (Member)	Date

AFIT/GCE/ENG/12-06

Abstract

Quantum Key Distribution (QKD) is a method for transmitting a cryptographic key between a sender and receiver in a theoretically unconditionally secure way. Unfortunately, the present state of technology prohibits the flawless quantum transmission required to make QKD a reality. For this reason, error reconciliation protocols have been developed which preserve security while allowing a sender and receiver to reconcile the errors in their respective keys. The most famous of these is Brassard and Salvail's Cascade protocol, which is effective, but suffers from a high communication complexity and therefore results in low throughput. Another popular option is Buttler's Winnow protocol, which reduces the communication complexity over Cascade, but has the disadvantage of introducing errors, and has been shown to be less effective than Cascade. Finally, Gallager's Low Density Parity Check (LDPC) codes have recently been shown to reconcile errors at rates higher than those of Cascade and Winnow with a large reduction in communication, but with greater computational complexity. This research seeks to evaluate the effectiveness of these LDPC codes in a QKD setting, while comparing real-world parameters such as runtime, throughput and communication complexity empirically with the well-known Cascade and Winnow algorithms. Additionally, the effects of inaccurate error estimation, non-uniform error distribution and varying key length on all three protocols are evaluated for identical input key strings. Analyses are performed on the results in order to characterize the performance of all three protocols and determine the strengths and weaknesses of each.

Acknowledgments

This thesis would not have been possible without the support and mentorship of my thesis advisor, Lt Col Jeffrey Humphries, and I am grateful that I was given the opportunity to work on such an interesting the challenging project. I would also like to thank Dr. Michael Grimaila for his never-ending enthusiasm, optimism and programming expertise, and the rest of the QKD team for being a supportive sounding board for all of my peculiar dilemmas. Finally, and most importantly of all, I would like to thank my wife and son for their boundless love and support throughout this entire process, and in all my endeavors.

James S. Johnson

Table of Contents

Page

List of Figures

List of Tables

Page

AN ANALYSIS OF ERROR RECONCILIATION PROTOCOLS FOR USE IN QUANTUM KEY DISTRIBUTION

I. Introduction

The field of cryptography is the art and science of securing a message by transforming it into a new message, through an encryption algorithm or cipher, such that interception of the cipher text reveals little useful information about the original message. Cryptography has a long and rich history. It was used in Roman times to pass messages between military leaders, and famously during World War II by the German government. Current uses of cryptography vary, and include but are not limited to protection of data in transit or in storage, proof of identity, and validation of information. Regardless of the purpose, the goal of cryptography has nearly always been to ensure that no unauthorized user can extract meaningful information from encrypted data.

Cryptography can, in general, be subdivided into two categories. Symmetric-key cryptography refers to an encryption/decryption process where both the sender and receiver share a common encryption/decryption key (or keys). Through some secure method, a sender and receiver exchange key material, which is then later used by either side to encrypt or decrypt messages. Asymmetric-key cryptography, on the other hand, refers to an encryption/decryption process where it is not necessary for the sender and receiver to share a common key. A simple example of this is public key cryptography, where a publicly known encryption key is used to encrypt messages, while only a privately held decryption key is capable of retrieving the original message. Since a sender does not need to be able to decrypt the message, they would simply use the public key to encrypt a message before sending it to the receiver, who is the only one capable of decrypting it. Both symmetric and asymmetric key cryptography are in wide spread use today.

An important characteristic of cryptographic ciphers is theoretical security. In information theory, the highest level of security possible is unconditional security. Here an adversary is assumed to have unlimited resources and computing power, but even so is unable to retrieve the original message by observation of cipher text alone. In 1946 a researcher by the name of Claude Shannon proved that a cipher can only achieve unconditional security if the key used is truly random and the key length is equal to or greater than the length of the message to be encrypted (Shannon, 1949). In addition, no key (or part of a key) should ever be used more than once. The simplest and most well-known cipher that meets Shannon's criteria is the one time pad (OTP), a symmetric key algorithm. The weakness of the OTP lies in its implementation, since generating and securely distributing truly random, unique keys tends to be difficult in practice.

The next level of security therefore is known as provable security, where the security of the cipher is equivalent to the difficulty of another known problem, which itself is thought to be computationally infeasible. An example of this is the well-known Rivest, Shamir, and Adleman (RSA) asymmetric key algorithm which relies on the difficulty of factoring the product of two large prime numbers (Rivest, Shamir, & Adleman, 1978). RSA is built into most major operating systems, is used in online banking as well as many internet protocols, and is currently incorporated directly into physical devices such as smart cards. Still, the difficulty of factoring large integers has not been proven to be a difficult problem, and a polynomial time solution may exist. In fact, if quantum computers are ever fully realized, Peter Shor's quantum algorithm already presents a solution for factoring large numbers in polynomial time (Shor, 1994).

An ideal cipher would need to offer unconditional security of symmetric key protocols, but without the key distribution difficulties. For this reason, many current protocols use an asymmetric key cipher (such as RSA) to exchange a key for use in a symmetric key cipher, which offers better efficiency in terms of throughput. Still, the security of a system can only be as high as the security of its weakest link. Assuming the use of a one time pad, in this case the security would rest on the asymmetric key cipher, and therefore the problem of factoring of large numbers.

Quantum cryptography offers another alternative for exchanging a symmetric key without compromising security. Quantum cryptography is the use of quantum mechanics to perform cryptographic tasks (Trappe & Washington, 2005). A subset of quantum cryptography is Quantum Key Distribution (QKD), where properties of quantum mechanics are leveraged in order to ensure unconditional security of a transmitted key, which can then be used in a classical symmetric key cipher such as a one time pad.

Research Objectives

Although QKD in theory offers unconditional security for key exchange, in reality, due mostly to technical limitations, practical systems cannot achieve the flawless quantum transmission required by an ideal QKD protocol. This, as well as potential interference by eavesdroppers, leads to errors in a transmitted key which must be resolved prior to applying any cryptographic cipher. Efficiently reconciling these errors is the focus of this research. Specifically, this thesis will seek to evaluate the suitability of Low Density Parity Check (LDPC) codes for QKD error reconciliation, as well as characterize (theoretically and empirically) and scrutinize LDPC versus two of the most well-known error reconciliation protocols for QKD, namely Cascade and Winnow.

Research Questions

The following questions highlight areas critical to this effort:

- Are LDPC codes decoded using the Sum Product algorithm viable for use in the error reconciliation phase of QKD protocols in terms of complexity, effectiveness, throughput, runtime, and information leakage?
- How do the metrics of the three protocols (Cascade, Winnow and LDPC) compare to one another? Specifically, given identical sifted keys and ideal error estimation, how do the protocols compare in terms of effectiveness, throughput, runtime, and information leakage?
- How robust are the three protocols with respect to one another in the case of non-ideal error estimation? And in the case of non-uniformly distributed (burst) errors?
- Does key length affect the performance of the three error reconciliation protocols (beyond obvious increases in processing time)? Is there an ideal key length with regard to overall throughput?

The primary goal of this research is to evaluate the suitability of LDPC codes for QKD error reconciliation. LDPC codes are a relative newcomer to QKD, and therefore have not been thoroughly studied. Though LDPC codes themselves were introduced in the 1960's (Gallager, 1962), only recently has technology evolved to the point where implementing LDPC codes has become computationally feasible. While the Winnow and Cascade protocols were specifically developed for use with QKD, LDPC codes represent a class of Forward Error Correcting (FEC) codes originally developed for classical communications. Acceptable error rates in QKD systems are typically much higher than in classical communications, and therefore it is desirable to determine whether LDPC codes can efficiently operate in the error range of QKD. LDPC Codes also require significant computational power due to their iterative nature and complex calculations involved with their decoding algorithms. Since throughput is a key factor in any communication protocol, characterizing the key rate of the LDPC code for different error distributions is significant. This research will provide a theoretical evaluation of LDPC with respect to all the above characteristics, as well as an empirical analysis versus the well-known Cascade and Winnow error reconciliation protocols.

This thesis is organized as follows. Chapter 2 presents background material on QKD as well as the three error reconciliation protocols to be studied. Chapter 3 describes the experiments conducted to evaluate this author’s implementation of the Sum Product algorithm as a decoder for LDPC codes, as well as an empirical analysis of this LDPC decoder as an error reconciliation protocol versus the Cascade and Winnow protocols, also written by the author. Chapter 4 gives the results of the experiments outlined in Chapter 3. Finally, Chapter 5 presents a summary of conclusions and their impact on quantum key distribution, as well as recommendations for future research.

II. Literature Review

In this chapter, a brief overview of quantum key distribution (QKD) is presented as well as a review of literature and current research relevant to the topic at hand (QKD error reconciliation).

2.1. Quantum Key Distribution

Quantum key distribution involves encoding information in *qubits*, or quantum information bits, and exchanging those bits between a sender and receiver. While in classical communications a bit can represent either a 0 or a 1, in quantum communications a qubit can represent a 0, 1 or a superposition of both states (Trappe & Washington, 2005). Furthermore, due to properties of quantum mechanics, performing any kind of measurement on a qubit causes it to collapse into a deterministic state, and the original superposition is non-recoverable. The method of measurement is correlated with this new state, and forms a basis which can be associated with the values 0 and 1 for encoding information. The qubit will assume one of these values (0 or 1) with a probability based on the original state of the qubit.

The security of QKD centers on the Heisenberg uncertainty principle, which roughly states that the act of measuring a quantum system interferes with that system, and thus prohibits collection of information on the state of the system prior to measurement (Brassard, A Bibliography of Quantum Cryptography, 1993). Thus an eavesdropper would be incapable of monitoring communications without irrevocably altering the information in transit, which would be detectable by a sender and receiver. Additionally, the no-cloning theorem of quantum physics forbids any eavesdroppers from creating replicate photons with the intent of performing multiple measurements (Wootters & Zurek, 1982). In this way, the security of QKD is based on fundamental laws of physics, rather than that of problems thought to be computationally infeasible.

The origin of QKD can be traced to the 1960's with a graduate student name Stephen Wiesner (Wiesner, 1983). As a student of Columbia University, Wiesner authored a manuscript describing two applications for quantum coding: a method for the creation of fraud-proof banking notes (quantum money), and, more significantly for our purposes here, a method for the transmission of two or three messages in such a way that reading one of the messages destroys the others (quantum multiplexing). In quantum multiplexing, Wiesner proposed utilizing photons polarized in multiple conjugate bases in order to pass information. In this manner, if the receiver measures the photons in the correct polarization basis, they will receive a correct result with high probability. Measuring the photons in the incorrect basis however, results in an ambiguous result, and a total loss of all information about the original basis. In his original paper, Wiesner suggests linear and circular polarization bases for two messages, but extends this method to three messages utilizing a third, 45° offset polarization.

Unfortunately, Wiesner had some difficulty getting his work published, and it was not until nearly a decade later, in 1983, that his manuscript was widely recognized. Shortly thereafter the first QKD protocol was proposed by Charles H. Bennett and Gilles Brassard, and is today known as BB84 for Bennett and Brassard 1984 (Bennett & Brassard, Quantum Cryptography: Public Key Distribution and Coin Tossing, 1984). This polarization-based protocol is arguably the most well-known and is a direct extension of Wiesner's quantum multiplexing theorem. Bennett and Brassard realized that quantum coding could be used to ensure secure distribution of random key information between two parties who share no secret information initially.

In the BB84 protocol, Bennett and Brassard propose the use of photons to realize qubits, similar to Wiesner. However, Bennett and Brassard propose the use of two bases (eliminating the circular basis included in Wiesner's manuscript), corresponding to four polarizations states. The *rectilinear* basis is comprised of 0° and 90° polarizations, and the *diagonal* basis of 45° and 135° polarizations. Therefore, a qubit polarized in one basis yields a random measurement in its conjugate basis. For example, a qubit with a polarization of 90°, when measured in the diagonal basis would yield a 50% probability of producing either 45° or 135° result, but when measured in the rectilinear basis, the result would produce a 90° result with high probability. By associating these two bases with classical bit values, Bennett and Brassard were able to develop a method for passing a message between two parties in such a way that any eavesdroppers would always be detected.

The physical setup of the BB84 protocol consists of a sender (subsequently referred to as Alice) and receiver (subsequently referred to as Bob), as well as two channels. A quantum channel is used to communicate qubits (photons), and it is assumed that only active eavesdropping may take place on this channel, since passive eavesdropping is not possible due to the no-cloning theorem. Additionally, a classical channel is used for authentication and to pass setup and message verification information. It is assumed that only passive eavesdropping may take place on the classical channel as long as some initial key material is shared for authentication. In their paper, Bennett and Brassard propose using a Wegman-Carter authentication scheme in order to secure the classical channel. Alice and Bob are asymmetric; Alice must maintain the appropriate equipment to generate and transmit qubits, and Bob must maintain matching equipment capable of receiving and measuring the single photon qubits polarized by Alice.

After initialization, the first step in the protocol is for Alice to generate an appropriate length, random bit string for use as a key in an encryption cipher such as a one time pad. Alice then randomly chooses a polarization basis (rectilinear or diagonal), and transmits one polarized photon to Bob for each bit of the key string. Bob receives these qubits and randomly chooses a basis in which to measure them. If Bob chooses correctly, he will receive the same bit value that Alice transmitted (assuming perfect transmission and no interference). If he chooses incorrectly, then Bob has a 50% probability of obtaining the correct value.

After Bob receives all of the qubits, he and Alice move onto the next phase of the protocol, known as sifting. In sifting, Bob communicates, on the classical channel, his choice of basis for each qubit, and Alice communicates those bit positions for which Bob chose correctly. Then Alice and Bob both discard any bits for which their measurement basis differed. At this point, Alice and Bob should have identical copies of a message, which is a subset of the original message transmitted by Alice. To confirm this, they systematically select a random subset of bits from the sifted key and compare them over the open channel. If all the bits agree, then it is likely that Alice and Bob have the same version of the key. They definitively confirm this with a randomly chosen (public) hash function before using the key in a symmetric cipher. Naturally, the bits exposed during the public comparison are discarded from the final key before the hash is applied. Once this key is used up, Alice and Bob simply repeat the process to generate a new one. The

BB84 protocol is summarized in Figure 1 (Bennett & Brassard, Quantum Cryptography: Public Key Distribution and Coin Tossing, 1984).

Quantum Transmission															
Alice's random bits	0	1	1	0	1	1	0	0	1	0	1	1	0	0	1
Random sending bases	D	R	D	R	R	R	R	R	D	D	R	D	D	D	R
Photons Alice sends	↗	↑	↘	→	↑	↑	→	→	↘	↗	↑	↘	↗	↗	↑
Random Receiving bases	R	D	D	R	R	D	D	R	D	R	D	D	D	D	R
Bits as received by Bob	1		1		1	0	0	0		1	1	1		0	1
Public Discussion															
Bob reports bases of received bits	R		D		R	D	D	R		R	D	D		D	R
Alice says which bases were correct			✓		✓			✓				✓		✓	✓
Presumably shared information			1		1			0				1		0	1
Bob reveals some key bits at random					1									0	
Alice confirms them					✓									✓	
Outcome															
Remaining shared secret bits			1					0				1			1

Figure 1. BB84 protocol summary

If there were an Eve present, even with infinite computational power, the best she could do in an ideal system would be to intercept transmissions from Alice, randomly select a basis for measurement, and retransmit those qubits in the basis that she selected. Since Eve does not know the basis Alice transmitted in, she would be correct, on average, only 50% of the time. When she retransmits the qubits to Bob he will also select a random basis for measurement, and will be correct, on average, 50% of the time. The combination of these steps will introduce 0.5 * 0.5 = 25% error into the transmitted raw key. Therefore, if Alice and Bob detect an error rate of 25% or higher in their sifted key, they conclude that there is an Eve present and they abandon that key.

A BB84 QKD protocol implemented in this fashion does in fact present unconditional security, since Alice and Bob can always detect the presence of an Eve. Unfortunately, in practice many of the assumptions required to implement BB84 are not achievable. Reliable single photon generation and detection technology is still mostly theoretical. Though significant progress has been made in recent years, in most cases superfluous photons are generated, and not all are detected. In addition, even excluding the presence of an Eve, physical characteristics of the quantum channel can introduce errors which affect the polarization of the photons while in transit. The result of these real world technical limitations is that errors are introduced into the sifted key, even though no malicious Eve is present. These errors must be resolved before any cipher is applied, since any cipher would require Alice and Bob to have identical copies of the key.

In order to reconcile the errors in their sifted key, Alice and Bob must perform several additional steps. These steps are commonly known as Error Estimation, Error Reconciliation, and Privacy Amplification.

2.2. Error Estimation

The purpose of error estimation is to determine the percentage of errors in the key after quantum transmission and sifting have occurred. The percentage of errors is known as the Quantum Bit Error Rate, and is hereafter referred to simply as the error rate. Traditionally, this has been accomplished by Alice and Bob publicly disclosing a random selection of bits from the sifted key. For a truly random sampling, the error rate should be representative of the overall sifted key. Unfortunately, this now requires Alice and Bob to discard the publicly disclosed bits from their sifted key, which reduces the size even further. An improvement to this method was proposed in 1999 (Ardehali, Chau, & Lo, 2005) that aims to double the efficiency of the method described in the BB84 protocol. In their paper, Ardehali, Chau, & Lo propose that Alice and Bob do not choose their polarization bases with equal probability, but instead favor one basis over the other. This presents an advantage to any Eve's involved, since they could gain more than 50% of qubits in the correct basis, so the authors suggest compensating for this advantage by calculating two error rates, one for each basis used. The errors introduced by Eve would then be detectable in the basis she was measuring in. In this way bits would still have to be sacrificed in order to accomplish error estimation post-sifting, but the overall key rate could be increased by decreasing the number of bits where Alice and Bob's measurement basis differed.

Error estimation has been studied in depth elsewhere and therefore for the purposes of this research it will be assumed that error estimation can be accomplished to within a reasonable bound. However, the tolerability of different error reconciliation algorithms to imprecise error estimation will be examined.

2.3. Error Reconciliation

Error reconciliation is the process of resolving all errors in the key over the classical channel, after quantum transmission and sifting have occurred, without revealing an unreasonable amount of information about the key to a potential Eve. The three protocols that will be presented here are Cascade, Winnow and a method based on Low Density Parity Check (LDPC) codes, hereafter simply referred to as the LDPC protocol.

2.3.1. Cascade

The Cascade error reconciliation protocol was first suggested in a paper published in 1994 by Gilles Brassard and Louis Salvail (Brassard & Salvail, 1994). Cascade represents a refinement of an earlier protocol known as BBBSS (for its authors Bennett, Bessette, Brassard, Salvail and Smolin), developed in 1991 as part of the first QKD channel experiment (Bennett, Bessette, Brassard, Salvail, & Smolin, 1991). It was in fact two of the authors of BBBSS who developed the refined Cascade protocol.

BBBSS takes place after quantum transmission, sifting and error estimation have occurred. The first step in the protocol is for Alice and Bob to agree on a random permutation, in public, over the classical channel. They perform this permutation on their respective sifted keys in order to attempt to evenly distribute any errors. Alice and Bob then divide their sifted key into blocks of size k, where k is defined such that each block is likely to have no more than one error, based on the error rate obtained during error estimation. The authors of Cascade empirically determined the ideal block size to be approximately $\frac{.73}{p}$, where p represents the estimated error rate (Brassard & Salvail, 1994). After this step, the single bit parity of each block is calculated and shared publicly. If Alice and Bob's parities agree, then they assume that there are no errors in that block, and they move on. On the other hand, if the parity of a block disagrees between Alice and Bob, then they perform a binary search on that block in order to identify the single bit error, which they then correct. In this way, a maximum of 1 + ceil($\log_2 k$) parity bits are exchanged for each block in error, and 1 parity bit is exchanged for those blocks not in error. In order to account for these *leaked* bits over the public channel and minimize the information gained by any eavesdroppers present, the authors suggest discarding the last bit of each block and sub-block for which a parity bit was exchanged. This is referred to in the literature as *privacy maintenance*.

After this process is completed and Alice and Bob are in agreement for all of their block parities, they can be confident that all blocks contain either zero or an even amount of errors. This is due to the fact that a parity check alone cannot identify an even amount of errors in a block. Therefore Alice and Bob must again permute their new key before the next pass. Additionally, after each pass the block size is increased to account for the fact that fewer errors remain.

When Alice and Bob are reasonably certain that all but a few errors have been corrected, they adopt a new approach for error reconciliation. The reason for this is that the information leaked for the parity check-binary approach is too great when the percentage of errors is small, since most blocks would not contain any errors and the parity of those blocks would match. The new strategy consists of randomly choosing a subset of bits from the corrected key string to form a block for parity comparison, and performing the same binary search routine if the parity bits do not agree. In this way, Alice and Bob do not have to discard as many parity bits as if they were to perform a full pass of the protocol as described earlier, though they still discard the last bit from each block and sub-block for which a parity bit was exchanged.

At some point Alice and Bob will find that all their parity comparisons are in agreement. When this occurs for a number of passes (the authors suggest 20), Alice and Bob conclude that their reconciled keys are identical, and they move on to privacy amplification.

The differences between BBBSS and Cascade are minimal, but significant. Like BBBSS, in Cascade the first pass is accomplished by dividing the sifted key into block sizes of length *k* based on the estimated error rate, and parity bits for each block are exchanged. And like BBBSS, a binary search is performed in order to identify single bit errors on blocks that have mismatched parities. Unlike BBBSS however, no bits are discarded during this first pass. Instead, the block errors are corrected, a permutation is applied, the block size is increased to $2 \cdot k$, and another pass is performed identical to the first. It is at this point Cascade deviates most from BBBSS. For any errors corrected in the second pass, there must be at least one matching error that resided in the same block in the previous pass, since neither error was found or corrected in that pass. For this reason, for each correction made in any pass after the first pass, a binary search is rerun on the block containing that corrected bit in all previous passes, in order to identify any potential matching errors. Any time a new error is identified, it reveals the potential to have masked another error in a previous pass, so the process is repeated, and the error detection and correction *cascades* through all previous passes. This process is illustrated in Figure 2.

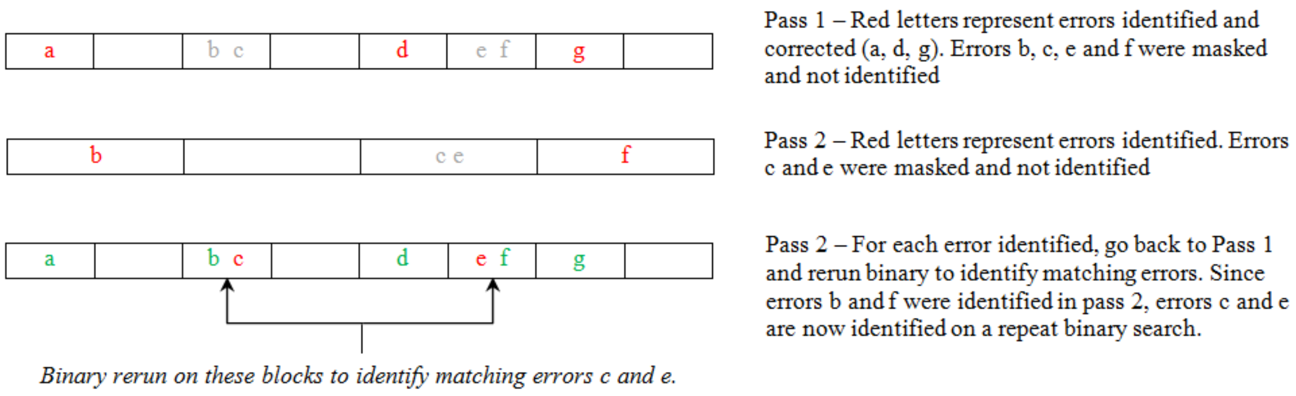

a
b c
d
e f
g
Pass 1 – Red letters represent errors identified and corrected (a, d, g). Errors b, c, e and f were masked and not identified
b
c e
f
Pass 2 – Red letters represent errors identified. Errors c and e were masked and not identified
a
b c
d
e f
g
Pass 2 – For each error identified, go back to Pass 1 and rerun binary to identify matching errors. Since errors b and f were identified in pass 2, errors c and e are now identified on a repeat binary search.
Binary rerun on these blocks to identify matching errors c and e.

Figure 2. Cascade protocol summary

In every pass after the first pass, on average two errors will be corrected for every bit detected, therefore the amount of errors present in the reconciled key decreases exponentially for each pass of Cascade. Empirically, for realistic error rates, four passes is generally considered sufficient to correct all errors, as alluded to in the original paper.

Cascade has been thoroughly studied since it was published, and several enhancements of note have been suggested (Calver, 2011). The biggest concern with Cascade is typically the amount of interaction required. To minimize the information exchanged, Sugimoto and Yamazaki suggest switching to the alternative block strategy of BBBSS after two passes; since after two passes the majority of errors have most likely been corrected (Sugimoto & Yamazaki, 2000). The authors showed how by using this method they could leak fewer bits and perform even closer to the theoretical limit proposed by Shannon (Shannon, 1949). Additionally, a dynamic block size selection technique has been studied (Rass & Kollmitzer, 2009) as well as a method of enhancing the permutation function used between passes (Bellot & Dang, 2009).

2.3.2. Winnow

Though Cascade has been proven to be a very effective protocol that is capable of resolving all errors and will never introduce errors if implemented correctly, it does suffer from a high rate of interactivity due to the necessary parity exchanges. Additionally, the amount of information leaked by Cascade is dependent on the distribution of the errors, and therefore the throughput of Cascade can be unpredictable in practice. In 2003 a new error reconciliation protocol for QKD was proposed that offers better throughput and lower interactivity, with a similar efficiency as Cascade (Buttler, Torgerson, Nickel, Donahue, & Peterson, 2003). In order to adequately understand the protocol, known as Winnow, a short background on Hamming codes is necessary (Trappe & Washington, 2005).

Hamming codes are a set of linear error correcting codes, and can be represented by two related matrices; a Generator matrix, G and a parity check Matrix H, such that

$H \cdot G^T = 0$. If G is an $m \ x \ n$ matrix, then the rate of the code, *r* is defined as $\frac{m}{n}$. In standard form, the first *m* columns of G form the identity matrix. Hamming codes are named after their creator, Richard Hamming and are capable of detecting up to two errors and correcting one error.

In order to transmit a message *M* using a Hamming code, a sender Alice calculates the dot product of the generator matrix and the message, called a *code word*. Since the first *m* rows of the generator matrix consist of the identity matrix, the first *m* bits of the code word constitute the message. This code word is then transmitted to a receiver Bob over a noisy channel. When Bob receives the message, he computes the dot product of the code word and the parity check matrix, called a *syndrome*. If the resulting syndrome is a zero vector, Bob concludes that the message was received intact and did not contain any errors. On the other hand, if the syndrome is not zero, the message is likely to contain at least one error, and the syndrome can potentially be used to correct it. This process is shown in Figure 3.

Sender ***Receiver***

$M \cdot G = C$ Transmit codeword (C) $H \cdot C = S$

$$\underbrace{\begin{bmatrix}1&0&1&0\end{bmatrix}}_{\text{Message (M)}} \quad \underbrace{\begin{bmatrix}1&0&0&0&0&1&1\\0&1&0&0&1&0&1\\0&0&1&0&1&1&0\\0&0&0&1&1&1&1\end{bmatrix}}_{\text{Generator Matrix (G)}} \quad \begin{matrix}1\\0\\1\\0\\1\\0\\1\end{matrix} \longrightarrow \begin{matrix}1\\0\\1\\0\\1\\0\\1\end{matrix} \quad \underbrace{\begin{bmatrix}0&1&1&1&1&0&0\\1&0&1&1&0&1&0\\1&1&0&1&0&0&1\end{bmatrix}}_{\text{Parity Check Matrix (H)}} \quad \underbrace{\begin{bmatrix}0&0&0\end{bmatrix}}_{\text{Syndrome (S)}}$$

Figure 3. A [7, 4] Hamming code for message transmission

If his calculated syndrome is not zero, Bob will attempt to correct the received code word in such a way that the syndrome is equal to zero with the fewest amounts of changes to C. This is known as maximum likelihood decoding, and is based on the assumption that minimal errors occurred; therefore the nearest code word is likely the correct one. The *distance* (or Hamming distance) between two code words is the number of positions in which they differ. The minimum distance, $d_{min,}$ therefore is the lowest distance between two valid code words. If the number of errors in a given code word is less than d_{min}, they will always be detectable, since the code word will not contain enough errors to convert the code word into another valid code word. This faulty code word may not be decodable (correctly) however, since, if the number of errors is greater than $\frac{d_{min}}{2}$, it may more closely resemble another code word than the correct one. Hence, if the number of errors is less than this bound $\left(\frac{d_{min}}{2}\right)$ the code word will always be decodable, since the closest code word will be the correct one.

For a linear code, the minimum distance of a code is equal to the minimum weight of all the non-zero code words in that code. For the example in Figure 3, it can be seen that there are $2^3 = 8$ possible code words and therefore the minimum distance is 3 since the weight of the message is 2, and the minimum weight of all the code words is 1. Therefore this code is capable of detecting 2 errors and correcting 1.

Using Hamming codes in this way is known as Forward Error Correction (FEC), where redundant bits are sent along with the message in order to facilitate the resolution of any errors without an interactive communication process. For use with error reconciliation in QKD however, it is necessary to make one small modification. In QKD, much effort is put into transmitting the key string in a secure method; therefore transmitting it publicly along with error correction data would make Eve's job trivial. For this reason, Alice computes the syndrome for her key string ($H \cdot M = S$, where H is the parity check matrix, M is the key string and S is the syndrome), and sends it to Bob. If Bob has an identical key string, then when he computes his syndrome ($H \cdot M' = S'$, where M' represents Bob's received key string), he should find that $S = S'$, in other words that the syndrome he received from Alice exactly matches the one he calculated on his own. If Bob's syndrome does not match Alice's, then Bob can use his syndrome to identify the location of the error. This process is illustrated in Figure 4.

Parity Check Matrix (H)

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \qquad M_A = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \qquad S_A = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad S_B = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \qquad S_A \oplus S_B = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \begin{matrix} 0 \\ 1 \\ 2 \end{matrix}$$

$$M_B = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix}$$

Bob sees discrepancies in his calculated syndrome at positions 0 and 1, so he corrects the key bit at position $2^0 + 2^1 = 3$

Figure 4. Error correction in Hamming codes

By definition, multiplication by the parity check matrix H results in a vector consisting of parity checks on the message M. Therefore, transmitting this information publicly is similar to sending the parity of blocks as suggested in the Cascade algorithm, and makes recovering the original message from this information prohibitively difficult.

It is this syndrome exchange process that Buttler suggests in his protocol which he calls Winnow (Buttler, Torgerson, Nickel, Donahue, & Peterson, 2003). In Winnow, much like Cascade, Alice and Bob first divide their key strings into blocks of length k (Buttler suggests starting with k = 8). Alice and Bob then compare parities and note any discrepancies, just as in Cascade. The difference is in what happens next. Instead of performing a binary search, Alice and Bob construct a parity check matrix for a Hamming code, given by $H_{i,j} = \left(\frac{j}{2^{i-1}}\right) mod\ 2$, and calculate syndromes S for each block B not in agreement, where $S = H \cdot B$. Alice then sends her syndromes to Bob, and Bob, by comparing these syndromes to his own, can correct any single bit errors.

In order to account for the information exposed by parity and syndrome exchanges, Buttler suggests performing privacy maintenance throughout the reconciliation phase. Buttler argues that this method is superior to the method used in Cascade, since some of the bits discarded along the way may actually be bits in error. Therefore, one bit is removed for every parity/syndrome bit exchanged, though the selection of the syndrome bits is not random. Buttler suggests removing the bits in each block at position 2^j for $j \in \{0, \ldots, m-1\}$, since these bits are independent in syndrome calculations and therefore are most exposed.

Since parity checks used in this fashion can only detect an odd amount of errors, after one pass it is likely that some blocks still contain an even amount of errors. For this reason, Alice and Bob then permute their reconciled key strings and repeat the process, although the block size is increased and the parity check matrix is regenerated to account for the new block size. The optimal number of passes and schedule of block sizes for each pass is subject to some debate and has been studied elsewhere (Lustic, 2011).

The downfall of Winnow lies with its reliance on Hamming codes. Where Cascade will not detect an even amount of errors and will only correct one error in blocks containing more than two errors (with odd parity), Winnow may actually introduce errors if the error count per block is too high. The reason for this is that as the number of errors grows past d_{min}, the probability of decoding the code word into an incorrect code word also increases, as mentioned earlier. For this reason, accurate error estimation and error distribution are critical parts of the Winnow protocol, and Winnow does not traditionally fare well in the face of burst errors. Fortunately, the latter problem is easily avoided by performing an initial permutation, as suggested by the original author.

2.3.3. Low Density Parity Check Codes

Cascade and Winnow are both well-established protocols that have been studied extensively for use in QKD systems. However, they are not without their respective weaknesses. Consider, for example, the communication complexity of Cascade. For each pass, Alice and Bob must exchange one parity bit for each block of the message, and an additional $\log_2 k$ bits for each k-bit block containing an odd amount of errors. For a 1000 bit key string with a 5% error rate, or 50 errors, and assuming a block size of 15 bits with half the errors detected in the first pass and the rest corrected in the second pass, Alice and Bob would exchange approximately 350 parity bits. Assuming the 95 block parity bits were contained in 4 single messages (one message for each pass), Alice and Bob would still need to exchange 255 additional messages during the binary search routine. Even with conservative estimates on packet size and network latency, this communication adds up. The Winnow protocol greatly reduces this overhead, but does not eliminate it completely, and has the disadvantage of possibly introducing errors.

Perhaps the greatest restriction of Cascade and Winnow is that in both protocols only one error per block is corrected. This fact necessitates complicated shuffling routines that must be done in the same way on both sides between passes, which only adds to the communication overhead.

Another alternative is presented with the recently rediscovered Low Density Parity Check (LDPC) codes. Originally introduced by a researcher named Robert Gallager in his doctoral dissertation at MIT in 1960 (Gallager, 1962), LDPC codes (alternatively known as Gallager codes) went mostly ignored for nearly 40 years. It was not until 1999, when Turbo Codes were gaining in popularity that LDPC codes were rediscovered by McKay (MacKay D. J., Good Error-Correcting Codes Based on Very Sparse Matrices, 1999) and shown to have similar performance near the theoretical Shannon limit.

LDPC codes are named for the sparseness (and therefore low density) of their parity check matrices. This feature is desirable, since, as we will see later on, the low density of the parity check matrix contributes to a near linear increase in the complexity of the decoding algorithm as the length of the message grows. In utilizing LDPC codes, the efficiency of the iterative decoding algorithm is an essential parameter.

Like Hamming Codes, LDPC codes are a FEC code defined by a parity check matrix H and a generator matrix G. In LDPC codes, as in Hamming codes, the minimum distance of the code is an important parameter, since it determines the decoding limit of the code. Unfortunately, for large codes finding this limit is not straightforward, and recent research has indicated that it may not be solvable in polynomial time (Otmani, Tillich, & Andriyanova, 2007). Therefore, a simpler approach for individual codes is to determine the decodable error range empirically.

In their original form, Gallager describes LDPC codes as having a fixed number *j* of 1's in each row, and a fixed number *k* of 1's in each column. Along with the block length, *n*, such a code is known as an (n, j, k) low density code. The number of 1's can be dispersed randomly, subject to the constraint:

$$n\ x\ k = m\ x\ j \tag{2.1}$$

However, a random distribution does not always prove to be optimal, as it may contain a short cycle.

The rate of the code r ($0\ \leq r\ \leq 1$) is typically decided beforehand, and will have a significant impact on the correcting power and efficiency of the code. The dimensions of the generator and parity check matrices are given by $m \times n$, where m is defined as:

$$m = n(1 - r) \tag{2.2}$$

Therefore, the size of the parity check matrix, *H*, shrinks as *r* grows, and *H* grows *quadratically* with increases in *n*, since $H = m\ x\ n = n(1 - r)\ x\ n$. However, the number of 1's and hence, the number of parity checks, only grows linearly with increases in *n*. This is evident from (2.2) above. The importance of these characteristics of LDPC codes will become evident when discussing the implementation of decoding algorithms in section 2.3.3.2.

Codes defined in this way would later become known as *regular* LDPC codes, since the definition of LDPC codes was later expanded to include *irregular* codes. An irregular code is defined such that the number of 1's in each column and row is not fixed, but rather governed by a degree distribution, as introduced by Luby (Luby, Mitzenmacher, Shokrollahi, Spielman, & Stemann, Practical Loss-Resilient Codes, 1997). Irregular codes are generally thought to have slightly better performance characteristics than regular codes (Luby, Mitzenmacher, Shokrollahi, & Spielman, 1998).

A convenient way of visualizing a LDPC code is by viewing the parity check matrix as a *Tanner graph*, which is a bipartite graph made up of nodes and edges as first described by Tanner (Tanner, 1981). Each row of the parity check matrix represents a *check node*, and each column a *variable node*. Each check node represents the parity check performed during the syndrome calculation, and each variable node represents a single bit of the incoming message. Therefore, a *m x n* parity check matrix would have *m* check nodes and *n* variable nodes. For each non-zero entry at position [i, j] in the parity check matrix, an edge is placed connecting check node *i* and variable node *j*. The number of these edges connected to a given node is called the *degree* of that node. An example of a Tanner graph is shown in Figure 5.

$$H = \begin{bmatrix} 0&1&1&1&1&0&0 \\ 1&0&1&1&0&1&0 \\ 1&1&0&1&0&0&1 \end{bmatrix}$$

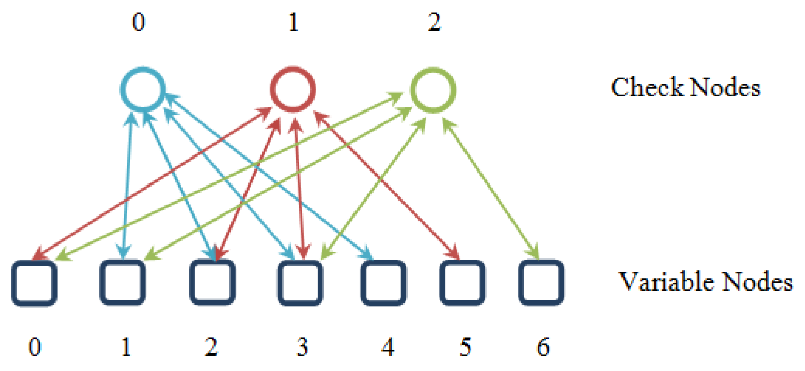


Figure 5. Tanner graph for a parity check matrix

In a graph, a *cycle* is a path that begins at a node n and traverses one or more different edges before arriving back at n. In the graph in Figure 5, a cycle of length 4 can be seen (V0 → C1 → V3 → C2 → V0), and there may be others. Furthermore, the *girth* of a LDPC code is defined as the shortest cycle present in the graph of the parity check matrix. Is it important to generate codes with a large girth, for reasons that will become evident in section 2.3.3.2.

2.3.3.1. Generating LDPC Codes

In his original dissertation, Gallager presented an algorithm for generating the LDPC matrices using a pseudo random number generator, taking care to ensure that no two parity check sets contain more than one digit in common. This procedure results in codes with a sufficiently wide girth, however it is computationally inefficient for large key lengths and only works for regular codes. Other attempts have also used similar random approaches (Kasai, Matsumoto, & Sakaniwa, 2010; Elkouss, Leverrier, Alleaume, & Boutros, 2009). McKay in his 1999 paper proposed a method based on pseudo random number generation, and he offered several variations aimed at improving the girth of the graph.

More recently, a less random approach to code construction has been suggested. In (Hu, Eleftheriou, & Arnold, 2001), the authors propose an algorithm which they call Progressive Edge Growth (PEG) construction. The algorithm is named for the fact that it progressively establishes edges between variable and check nodes. Edges are evaluated such that placement of the new edge has the least impact on the overall girth. Next, the edge with the least impact is selected, the graph is updated, and the procedure is iterated. Exhaustively searching through the list of possible edges is computationally infeasible for most realistic key lengths, so the PEG algorithm takes a best-guess approach described by:

- If the edge to be added is the first such edge for variable node V, randomly choose a check node C from the list of check nodes currently containing the smallest degree. Add an edge from V to C.
- Else define $\boldsymbol{N_s^l}$ as the set of check nodes reached by a tree spreading from variable node V to a depth l, and define $\boldsymbol{N'^l_s}$ as the complement set of $\boldsymbol{N_s^l}$, such that $\boldsymbol{N_s^l}\ \boldsymbol{U}\ \boldsymbol{N'^l_s} =$ the set of all check nodes. Expand a tree from V up to depth l such that $\boldsymbol{N'^l_s} \neq \boldsymbol{0}$ but $\boldsymbol{N'^{l+1}_s} = \boldsymbol{0}$ or the cardinality of $\boldsymbol{N_s^l}$ stops increasing, but is less than m. Then, randomly select a check node from the subset of check nodes in $\boldsymbol{N'^l_s}$ having the smallest degree.

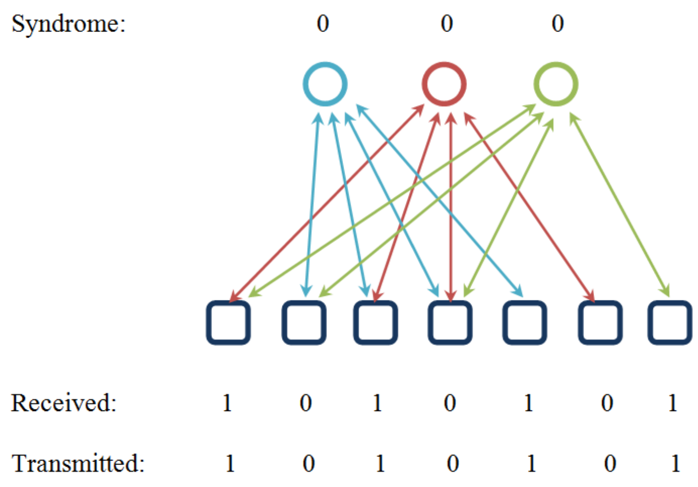
Since it was published in 2001, the PEG algorithm has been studied and improved in numerous papers. In (Richter, 2005) the author was able to lower the error floor and improve performance of the resulting code in the waterfall regions and here (Lin, Chen, & Chang, 2008) in order to improve the structure of the resulting codes to reduce the Very Large Scale Integration (VLSI) implementation complexity. Even without these relatively minor enhancements, the original algorithm is capable of generating very good codes with wide girth and low error floors.

Finally, a method for determining the decoding limit for ensembles of LDPC codes is presented in (Richardson & Urbanke, 2001). Known as *Density Evolution,* the technique tracks the probability distribution of the messages through individual iterations of the decoding algorithm, in order to determine the expected density and generate an overall picture of the best-case performance. Density evolution can be used to find the degree distribution that maximizes the threshold of an LDPC code such that the probability of error tends to zero as the number of iterations tends to infinity (Chung, Forney, Richardson, & Urbanke, 2001).

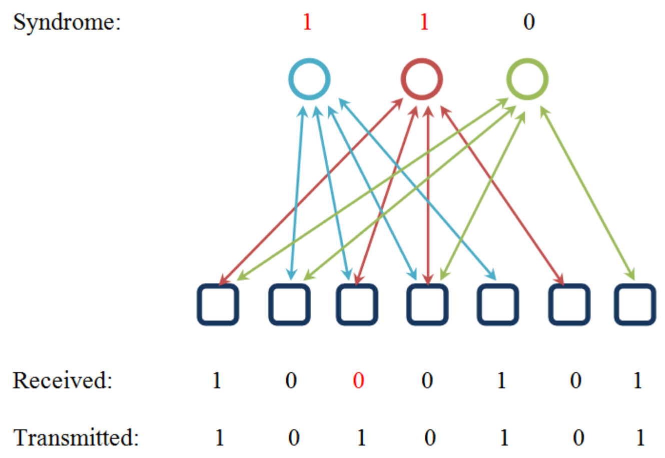
2.3.3.2. Decoding algorithms

Encoding messages using LDPC codes works in much the same way that encoding using Hamming codes does. A sender Alice would multiply the message and the Generator matrix together to form a code word, which would be sent to a receiver Bob. Bob would then take the received code word, multiply it by the parity check matrix, and if the resulting syndrome was a zero vector, he would be confident that he had received Alice's transmission intact. The difference with LDPC codes lie in how Bob decodes the code word into the original message, particularly in the presence of errors.

In his original paper, Gallager proposed two decoding algorithms. The first algorithm is relatively straightforward, but is only effective for small parity-check sets, where each set contains one or zero errors. This algorithm, which Gallager calls the *Gallager A* algorithm, involves computing all the parity checks for a given message, and then flipping any bits contained in more than a fixed number of unsatisfied parity checks. This algorithm is illustrated in Figure 6.



The message was received intact; therefore the syndrome is the zero vector.



The message was received with an error at position three. After the parity checks, the syndrome reveals the error. The bit at position 3 is involved with the most incorrect parity checks (2), so it is flipped.

Figure 6. The Gallager A algorithm

In fact, the Gallager A algorithm can be successfully used to decode Hamming codes, since it is effective when there are only one or zero errors present in each parity check set. Gallager offered a second decoding algorithm based on probabilistic decoding however, that is far more powerful and would later become known as Belief Propagation.

Belief propagation (also known as the Sum Product algorithm) is a *message passing*, iterative protocol. In a message passing protocol, messages are passed between nodes, along the edges of the graph. The messages contain information that influences the nodes; however, outgoing messages cannot be influenced by other messages coming in along the same edge. Therefore, each node in the graph calculates separate messages for each edge, based on the information received along all other edges. By exchanging information in this fashion, Gallager showed how it was possible to efficiently correct all of the errors in a transmitted message, up to a certain bound.

In the Gallager A algorithm, *hard* messages are passed between nodes. This means that the information contained in the message is not probabilistic, and the updates are definitive. In Belief Propagation, rather than hard messages, *soft* messages are passed; probabilities that the transmitted digit is a 0 or 1 conditional on the received value along with the channel error probability. The Belief Propagation algorithm was originally proposed by Gallager, famously applied to Bayesian belief networks by Pearl (Pearl, 1982) and was reinvented by McKay (MacKay D. J., Good Error-Correcting Codes Based on Very Sparse Matrices, 1999). The version presented here will follow McKay in his 1999 paper.

Belief propagation starts with a received vector and a corresponding LDPC parity check matrix. For each variable node V_i corresponding to one bit of the received vector, a series of messages is initialized, two for every check node, C_j, that V_i is connected to. Message $q^0_{i\rightarrow j}$ corresponds to the message sent to C-node j from V-node i, and represents the belief that the received bit value at position i is a 0. Similarly, message $q^1_{i\rightarrow j}$ represents the belief that the received bit value at position i is a 1. For the first iteration of the protocol, these values are initialized to the probability *P* that a sent bit is a 0 or 1 given that the received bit is a 0 or 1, respectively.

$$p_i^1 = P(sent\ bit = 1 \mid received\ bit) = p + (received\ bit) \cdot (1 - 2p) \quad (2.3)$$

$$p_i^0 = P(sent\ bit = 0 \mid received\ bit) = 1 - p_i^1 \quad (2.4)$$

The value *p* here represents the channel error probability, or the probability of a bit flip in transit.

Following this initialization step is the *Horizontal Step*, where the parity checks defined by the parity check matrix are evaluated and for each variable node that a check node is connected to, two messages are prepared. Message $r_{j \to i}^0$ represents the message sent from check node C_j to variable node V_i, containing the probability that the bit value at position *i* is a 0. This probability is calculated based on the messages received by check node C_j in the initialization step (not including the message received from variable node V_i). Similarly message $r_{j \to i}^1$ contains the corresponding probability that the bit value at position *i* is a 1. Mathematically, these two messages are given by:

$$r_{j \to i}^0 = \sum_{\{x \, \epsilon \, V \backslash V_i\}} \prod_{\{y \, \epsilon \, V \backslash V_i\}} q_{y \to i}^x \quad (2.5)$$

$$r_{j \to i}^1 = 1 - r_{j \to i}^0 \quad (2.6)$$

Where $\{x \, \epsilon \, V \backslash i\}$ is the set of all variable nodes connected to check node C_j, excluding variable node V_i for which the message is being prepared. In this way, each check node C_j compiles messages to all the variable nodes, V_i to which it is connected, consolidating all of the information it received from all the nodes except Vi. A useful extension on equations (2.5) and (2.6) is presented by McKay (MacKay D. J., Good Error-Correcting Codes Based on Very Sparse Matrices, 1999):

$$r^0_{j\to i} - r^1_{j\to i} = (-1)^{m_i} \cdot \prod_{y \,\epsilon\, V\setminus V_i} (q^0_{y\to i} - q^0_{y\to i}) \qquad (2.7)$$

Where m_i represents the observed value of the received bit at position *i*. Following this step, after the messages are composed and sent out from the check nodes, is the *Vertical Step*. In this step, each variable node updates its respective *Q* messages based on the input from the check nodes. This update is given by:

$$q^0_{i\to j} = \alpha \cdot p^0_i \cdot \prod_{\{z \,\epsilon\, C\setminus j\}} r^0_{z\to i} \qquad (2.8)$$

$$q^1_{i\to j} = \alpha \cdot p^1_i \cdot \prod_{\{z \,\epsilon\, C\setminus j\}} r^1_{z\to i} \qquad (2.9)$$

Where $\{z \,\epsilon\, C\setminus j\}$ denotes the set of check nodes *C* that variable node V_i is connected to, excluding check node C_j for which the message is being prepared. The variable α is a correction factor meant to ensure $q^0_{i\to j} + q^1_{i\to j} = 1$. In this way, the *Q* messages are updated at each pass to include all information from all of the check nodes with the exception of the check node which is receiving the information, in order to prevent nodes from influencing themselves.

It is at this point that the importance of a wide girth becomes evident. For a graph with a minimum cycle, *C*, there will be precisely *C – 1* iterations before the information exchanged between nodes will begin to cycle back and influence those same nodes. Once this happens, the quality of the information passed degrades, and the effectiveness of the protocol suffers.

Additionally in the Vertical step, the *pseudo-posterior probabilities* are computed. Similar to the *Q* message updates, but including all check nodes, these probabilities are defined by:

$$q^0_i = \alpha \cdot p^0_i \cdot \prod_{\{z \,\epsilon\, C\}} r^0_{z\to i} \qquad (2.10)$$

$$q_i^1 = \alpha \cdot p_i^1 \cdot \prod_{\{z \,\epsilon\, C\}} r_{z\to i}^1 \tag{2.11}$$

These probabilities represent the most recent likelihood that the received bit value $x_i = x$ at any given pass. It is from these probabilities that a *hard decision* will be made. If $q_i^0 > q_i^1$, then the probability that x_i is equal to 0 is greater than the probability that x_i is equal to 1, therefore x_i is set to 0, otherwise, x_i is set to 1. The strength of the *belief* that the new value of x_i is correct is correlated with the difference between q_i^0 and q_i^1, where larger differences represent stronger beliefs. After updating all of the bits in the received message this way, the syndrome is recalculated. If the parity checks are all 0, the message is assumed to contain no errors and the error reconciliation is complete. Otherwise, the algorithm iterates with another horizontal step, vertical step and hard decision. The algorithm only terminates when the corrected message passes all parity checks or a set number of iterations is reached, indicating failure.

2.3.3.3. Log Likelihood Ratio

A simplification to the Sum Product algorithm based on Log Likelihood Ratios (LLRs) is possible which significantly reduces the computational complexity. In statistics, a LLR is a method of combining test statistics by computing a ratio. This method, combined with the fact that in the log domain products become sums, allows the following simplifications:

$$P_i = \ln\left(\frac{p_i^0}{p_i^1}\right) \qquad R_{j\to i} = \ln\left(\frac{r_{j\to i}^0}{r_{j\to i}^1}\right) \tag{2.12}$$

$$Q_{i\to j} = \ln\left(\frac{q_{i\to j}^0}{q_{i\to j}^1}\right) \qquad Q_i = \ln\left(\frac{q_i^0}{q_i^1}\right) \tag{2.13}$$

Subsequently, the q message update and hard decision equations can be similarly simplified:

$$Q_{j\to i} = \ln\left(\frac{q^0_{i\to j}}{q^1_{i\to j}}\right) = \ln\left(\frac{\alpha\cdot p^0_i\cdot\prod_{\{z\,\epsilon\, C\setminus j\}} r^0_{z\to i}}{\alpha\cdot p^1_i\cdot\prod_{\{z\,\epsilon\, C\setminus j\}} r^1_{z\to i}}\right) = P_j + \sum_{\{z\,\in\, C\setminus j\}} R_{j\to i} \qquad (2.14)$$

$$Q_j = \ln\left(\frac{q^0_i}{q^1_i}\right) = \ln\left(\frac{\alpha\cdot p^0_i\cdot\prod_{\{z\,\epsilon\, C\}} r^0_{z\to i}}{\alpha\cdot p^1_i\cdot\prod_{\{z\,\epsilon\, C\}} r^1_{z\to i}}\right) = P_i + \sum_{\{z\,\in\, C\}} R_{j\to i} \qquad (2.15)$$

Note that the normalization factor, α is no longer needed. The *R* message update equation can be simplified as well, however it requires a bit more effort. First, observe the following identity applied to the LLR equation for *R* messages:

$$R_{j\to i} = \log\left(\frac{r^0_{j\to i}}{r^1_{j\to i}}\right) \to r^0_{j\to i} = \frac{e^{R_{j\to i}}}{1+e^{R_{j\to i}}}\ and\ r^1_{j\to i} = \frac{1}{1+e^{R_{j\to i}}} \qquad (2.16)$$

And hence equation (2.7) becomes:

$$r^0_{j\to i} - r^1_{j\to i} = (-1)^{m_i}\left(\frac{e^{R_{j\to i}}-1}{1+e^{R_{j\to i}}}\right) = (-1)^{m_i}\left(\tanh\left(\frac{R_{j\to i}}{2}\right)\right) \qquad (2.17)$$

Applying a similar procedure to $q^x_{i\to j}$ yields the following expression:

$$(-1)^{m_i}\left(\tanh\left(\frac{R_{j\to i}}{2}\right)\right) = \prod_{y\,\epsilon\, V\setminus V_i} tanh\left(\frac{Q_{j\to i}}{2}\right) \qquad (2.18)$$

And solving for $R_{j\to i}$ yields the LLR, *R* message update rule:

$$R_{j\to i} = (-1)^{m_i}\cdot 2\cdot\tanh^{-1}\left(\prod_{y\,\epsilon\, V\setminus V_i}\tanh\left(\frac{Q_{i\to j}}{2}\right)\right) \qquad (2.19)$$

The benefits of converting to the LLR domain are obvious, since in the horizontal step, a sum of products is reduced to simply a product, and in the vertical step, a product becomes a simple sum. A hyperbolic tangent and inverse hyperbolic tangent function have been introduced, which does adversely affect computational complexity, but the need for calculating separate messages for 0 and 1 bit value likelihoods has been eliminated, which significantly reduces complexity.

An important aspect of (2.19) is that the inverse hyperbolic tangent function is asymptotic at ±1. Therefore, it is important to bound the input to this function at a reasonable value close to 1, such as $\pm(1 - 10^{-12})$, otherwise the result may tend to infinity as the number of iterations increases.

The main complexity of the algorithm does in fact now lie with the large amount of hyperbolic tangent functions. One way to cut down on this overhead is through the use of the Max-Product (otherwise known as the Min-Sum) algorithm, which seeks to approximate the hyperbolic tangent evaluations in equation (2.19). However, this approximation comes at the cost of decreased efficiency, and may require extra iterations in order to recover the message. Additionally, utilizing previously calculated values for the hyperbolic tangent function through the use of a lookup table has been suggested, since the range required here is relative small (Gudmundsen, 2010).

Finally, as with Winnow and Hamming codes, one small modification is required in order to apply the Sum Product algorithm to QKD. In QKD, the quantum transmission is responsible for securely transmitting the key. Therefore, after the quantum transmission takes place, Alice computes a syndrome for her version of the message using a parity check matrix that she has previously shared with Bob. Bob then uses this syndrome in order to correct the errors in his version of the key string. Only one small adjustment is necessary to the Sum Product algorithm, to account for the fact that the correct syndrome may contain non-zero values. In the event that the correct syndrome contains a 1, the sign of the equation in 2.12 would need to be flipped, in order to become:

$$R_{j\to i} = -\ln\left(\frac{r_{j\to i}^{0}}{r_{j\to i}^{1}}\right) \tag{2.20}$$

Even with suggested enhancements taken into account, decoding LDPC a code through Belief Propagation requires larger computational and memory requirements than either the Cascade or Winnow algorithms. However, it has the potential benefit of being able to correct all the errors in a key with only one information exchange. This tradeoff offers potentially large gains in secrecy, as well as overall runtime when network latency is taken into effect.

2.4. Privacy Amplification

The final step in correcting the errors in a transmitted key is known as Privacy Amplification. The purpose of this step is to account for any information exposed during the error reconciliation phase and ensure that any eavesdroppers present do not gain sufficient information to the point where they are able to reconstitute a significant part of the key.

As mentioned earlier, the Winnow protocol discards bits during error reconciliation, a method known as *privacy maintenance*. The Cascade and LDPC protocols do not. The reason for this is that in Cascade, the algorithm must maintain the ability to go back to earlier passes and rerun a binary search on those blocks. Discarding bits along the way would change the message length and make this process difficult, though doing so has been examined (Boughattas, Iyed, & Rezig, 2010). In LDPC, only one pass is necessary, so there is no need to discard bits before the error reconciliation is complete.

Therefore, for Cascade (typically) and LDPC (always), the number of bits exposed is tracked and then subtracted from the final reconciled key at random. In

Cascade the maximum number of bits exposed is $B + n \cdot \lceil \log_2 k \rceil$ per pass, where B is the number of blocks, n is the number of errors identified by mismatched parities, and k is the block size of that pass. In actuality either $\lceil \log_2 k \rceil$ or $\lfloor \log_2 k \rfloor$ bits are leaked for each binary search, depending on the location of the error in a block. In LDPC the number of bits exposed is the size of the syndrome exchanged, or m in the case of a $m\ x\ n$ matrix.

After error reconciliation and privacy maintenance, Alice and Bob can be sure that they have the same version of the key, and that the bits gained by Eve have been reduced to a minimum. As one final security measure, Alice and Bob discuss over the classical channel the selection of a random hash function. This hash function, which reduces the size of the final key even further, is then applied to the reconciled key. In this way, even if Eve was a very efficient eavesdropper and had a reasonable copy of the key, performing this hash would *amplify* any errors in Eve's version, since small changes in the input of a good hash function result in large changes in the output. Therefore, even if Eve only had a few errors in her version of the key, after privacy amplification her errors would be amplified to the point where even a brute force search would be infeasible.

Assuming Cascade, Winnow and LDPC expose a similar number of bits for similar error rates and therefore produce similar key lengths after privacy maintenance, the privacy amplification phase of the BB84 protocol is similar for all three protocols. Therefore, the implementation of privacy amplification will not be examined in detail here.

III. Methodology

3.1. Overview

This chapter presents an overview of experiments and research conducted for this thesis. Quantum key distribution is a powerful idea capable of exchanging a key between two parties in a theoretically unconditionally secure way. However, real world limitations on technology for the present and foreseeable future prohibit the perfect quantum transmission necessary to achieve the unconditional security offered by QKD. It is precisely this reason why error reconciliation is such a critical factor in QKD. Error reconciliation allows a sender Alice and a receiver Bob to correct implementation errors up to a certain bound with high certainty. However, performing this error reconciliation efficiently in order to produce useful key rates is paramount if QKD is ever to gain success commercially. These motivations warrant a deeper look and a comparison of existing error reconciliation algorithms.

The Cascade algorithm has been the standard for QKD error reconciliation in the past, though its high communication requirements make Cascade highly dependent on network performance and highly susceptible to denial of service attacks. The Winnow algorithm significantly reduces the communication complexity in comparison with Cascade, with comparable computational complexity; however, Winnow has the significant detriment of introducing errors and is traditionally not as effective as Cascade for error rate ranges under 10%. The LDPC algorithm requires the least amount of communication with theoretical error correcting capability at least as good as Winnow and Cascade, but LDPC is the most computationally complex and therefore suffers the most in terms of processing time.

While existing literature abounds with research studying Cascade and Winnow and variations to both algorithms, there are very few examples of research comparing the two directly. Even more infrequent is an empirical comparison to determine metrics such as error correction, runtimes and efficiency for identical input strings. With regard to LDPC codes, a large amount of research has been performed since McKay's foundational paper revived interest in them. The majority of this research has dealt with classical communications; however there have been a limited number of publications that dealt specifically with LDPC with respect to QKD (Elkouss, Leverrier, Alleaume, & Boutros, 2009; Mesiti, Delgado, Mondin, & Daneshgaran, 2010; Elkouss, Martinex, Lancho, & Martin, 2010). McKay himself addressed the issue in 2004 (MacKay, Mitchison, & McFadden, 2004), and Matsumoto more recently published several problems that LDPC codes will have to overcome in order to gain widespread acceptance (Matsumoto R. , 2009).

This QKD LDPC research has shown promise. One problem, Matsumoto feels, is the nonexistence of adequate parity check matrices. Consequently, a large amount of research has been devoted to developing ideal LDPC codes which perform ever closer to the theoretical bound, and indeed LDPC codes have been shown to resolve errors at rates comparable to Cascade (Gudmundsen, 2010; Elkouss, Leverrier, Alleaume, & Boutros, 2009). In addition, some researchers feel that maintaining a library of codes is a computationally wasteful process, and that it is more efficient to maintain only one *master* code, which is then modified in some manner, by puncturing and shortening for instance, for different code rates corresponding to different error rates (Kasai, Matsumoto, & Sakaniwa, 2010). In most real world applications, the error rate of the channel will be well characterized and will not fluctuate greatly under normal circumstances, therefore for the purposes examined here a library of codes corresponding to different error rates is considered sufficient.

An in depth analysis of an LDPC error reconciliation algorithm for QKD with respect to the well-known Cascade and Winnow algorithms could not be found in existing literature. Consequently, this research seeks to answer the following questions:

- Are LDPC codes decoded using the Sum Product algorithm viable for use in the error reconciliation phase of QKD protocols in terms of complexity, effectiveness, throughput, runtime, and information leakage?
- How do the metrics of the three protocols (Cascade, Winnow and LDPC) compare to one another? Specifically, given identical sifted keys and ideal error estimation, how do the protocols compare in terms of effectiveness, throughput, runtime, and information leakage?
- How robust are the three protocols with respect to one another in the case of non-ideal error estimation? And in the case of non-uniformly distributed (burst) errors?
- Does key length affect the performance of the three error reconciliation protocols (beyond obvious increases in processing time)? Is there an ideal key length with regard to overall throughput?

The primary goal of this research is to directly evaluate the use of LDPC codes as a viable alternative to the Cascade and Winnow protocols for QKD error reconciliation.

Throughput is an important parameter in any error reconciliation protocol, but since LDPC suffers from greater complexity than Cascade or Winnow, implementing the Sum Product decoding algorithm efficiently is paramount. In LDPC, in contrast to Cascade and Winnow, memory management is a crucial parameter. For a small, 1000 bit key string and a 0.1 rate code, the LDPC parity check matrix needs to be 900 x 1000 = 900 000 bits. For a 100 000 bit key string however, this number grows to 9000 000 000 bits. Similarly, in the vertical and horizontal steps of the Sum Product algorithm, the large amount of updated messages must be stored to a certain precision, which becomes even more costly than storing the parity check matrix. A direct comparison of a LDPC implementation along with the Cascade and Winnow protocols will determine the viability of LDPC as an error reconciliation protocol. Furthermore, since ideal, uniform error distribution cannot always be assumed, the susceptibility of the three protocols to non-ideal error rate estimation as well as several types of burst error distributions will be evaluated. Finally, the performance of the protocol at three different key lengths is considered. The required size of a final key is likely to vary depending on the application and required level of security, therefore it is advantageous to determine if the protocol behaves differently for different key lengths, and if there are tradeoffs in effectiveness and/or throughput.

The rest of chapter 3 is dedicated to a description of the simulation environment and implementations of the three error reconciliation protocols, as well as the approach taken to answer these research questions.

Finally, since error rate estimation must be done for any of the protocols, it can be abstracted and treated as a separate process. Therefore, the implementations presented here do not take into account any bits discarded in order to obtain an error rate estimate. The error rate estimation is assumed to have been completed beforehand, and is treated as an input parameter.

3.2. Common Criteria

For certain experiments, steps need to be taken in order to ensure continuity between different protocol evaluations. Consequently, the key strings used to evaluate the corrective power (effectiveness) of the protocols are generated randomly using a common seed for a Random Number Generator (RNG) based on the Mersenne Twister algorithm (Matsumoto & Nishimura, 1998). Any references made to a RNG from this point forward refer to this Mersenne Twister class. Similarly, the insertion of errors into the generated key string is performed *pseudo-randomly* using a different seed for a separate RNG instance. The common seeds are generated initially using a third RNG instance which uses the current time as a seed. In a real-world implementation, a cryptographically secure RNG would be preferred, in order to minimize the possibility of predictable output.

In addition, the runtime and throughput parameters discussed in the next section may potentially be influenced by environmental factors. For this reason, these parameters are evaluated on the same machine, at approximately the same time (within milliseconds) using a common driver function for all three protocols.

All experiments were conducted on a Custom PC with an AMD Athlon II X4 640 3.00 GHz processor and 8.00 GB of RAM running Windows 7 64-bit. All software was developed in the C++ language using Microsoft Visual Studio 2010.

3.3. Experiment 1: Evaluating the Sum Product LDPC decoding algorithm

3.3.1. Implementing the LDPC protocol

The Sum Product decoding algorithm is implemented as a single C++ class, with calls made to an external RNG class and driven by a main class. The principal challenges associated with the development of this algorithm are generating the parity check matrices, and efficiently managing the size of the various matrices needed for larger key string sizes.

The PEG algorithm C++ implementation used for this research to generate the parity check matrices is a derivative of an algorithm that was written by one of the creators of the PEG algorithm, Xia-Yu Hu, and was obtained from the website of David MacKay (MacKay & Hu, 2011). Only minor changes from the original version are made, mostly to eliminate unnecessary modes of operation, memory leaks and optimize output format for the purpose of this research. The underlying algorithm remains unchanged. LDPC codes are generated using one of the supplied degree distribution files (denEvl_15.deg) for a key length of 100 000 bits. LDPC codes are generated in rates varying from 0.005 to 0.15, in 0.005 increments. In the BB84 protocol, error rates above 0.15 would result in an abandoning of the key for fear of eavesdropper interference, so 0.005 to 0.15 represents a reasonable error rate operating range.

As discussed earlier in section 3.1, in order to perform operations on a key length of 100 000 bits, a more efficient method of storing sparse matrices is necessary. Key lengths above 100 000 bits are not considered due to the time required to generate the parity check matrices, however performance is not expected to differ considerably for key lengths above this range. The parity check matrix can be represented using only single bit variables since it is comprised only of bit values, however the matrices used to store the Q-messages and R-messages need to be represented by 8-Byte double variables in order to maintain the precision necessary for the decision step. These matrices would be prohibitively large if implemented in the traditional manner.

For this reason, the algorithm is implemented such that only the non-zero values of the necessary matrices are maintained. Since low density parity check matrices are by definition mostly zero, such an implementation is capable of substantial savings in memory. In order to efficiently store the matrices, two, one-dimensional arrays are maintained for the parity check matrix, where each entry in the array contains a C++ vector. The C_Node array represents all the check nodes, and each entry in the array contains a vector which itself contains the indices of all the variable nodes that each check node is connected to. In a similar way, a V_Nodes array represents all the variable nodes, and each entry contains a vector which itself contains the indices of all the check nodes the variable node is connected to. By storing the parity check matrix in this way, all the non-zero entries can be eliminated, as illustrated in Figure 7.

Parity Check Matrix (H)

	0	1	2	3	4	5	6
0	0	1	1	1	1	0	0
1	1	0	1	1	0	1	0
2	1	1	0	1	0	0	1

C_Nodes Array

0	1	2	3	4
1	0	2	3	5
2	0	1	3	6

V_Nodes Array

0	1	2	
1	0	2	
2	0	1	
3	0	1	2
4	0		
5	1		
6	2		

Figure 7. Sparse matrix storage procedure

Therefore no extraneous information is stored, and the two resulting matrices are very suitable for use in the horizontal and vertical steps. Since the degree distribution used in this research has a maximum row weight of 15, this method allows a potential 90 000 x 100 000 matrix (for a 0.9 rate code and 100 000 bit key length) to be reduced to a maximum of two 15 x 100 000 matrices, resulting in a savings of nearly 9 Gb of dynamic memory.

The other benefit of storing the parity check matrix in the method detailed above is that it is no longer necessary to traverse an entire row or column of the parity check matrix in order to determine which variable and check nodes are connected. Instead, it is only necessary to traverse the (considerably shorter) list of elements in each vector. This results in a significant savings in runtime.

One other notable implementation modification is employed in order to significantly improve the runtime of the decoding algorithm. In equation 2.19, multiple hyperbolic tangent products must be calculated for each R message update. If a given check node contains n parity checks, there will be $n \cdot (n-1)$ multiplications for that check node, since the product of the hyperbolic tangent of all other messages will need to be calculated for each message. An alternative is to pre-calculate the product of all of the messages for that check node, and simply divide the result by the message that should not have been included. Using this method, $n \cdot (n-1)$ multiplications for each check node can be reduced to one initial multiplication for all check nodes and n divisions per check node. Even though division is a much more costly operation with regard to clock cycles, the amount of savings is substantial.

A main function drives the simulation and abstracts operations unassociated with the error reconciliation. The main function first generates a random bit string before creating a simple Alice object and providing it with a flawless copy of this string. The only product output by the Alice object is the syndrome for the original bit string. This syndrome, along with a uniformly flawed copy of the bit string and an error rate estimate is given to a Bob LDPC object constructor. Bob first generates a syndrome based on his copy of the bit string, in order to determine if errors exist. If so, as indicated by differences in Bob's calculated syndrome and the syndrome he received from Alice, Bob proceeds to read in the parity check matrix corresponding to his given error rate from a text file. Bob then performs the necessary vertical, horizontal and decision steps, stopping after each decision step to recalculate his syndrome and check it against the copy he received from Alice. If the two syndromes match, Bob stops and declares that all the errors have been corrected. If not, Bob continues in an iterative manner with another vertical, horizontal and decision step. The number of iterations Bob will complete before deciding that he has failed is set to 200 for this research, which has been determined empirically to be sufficient. The Q-messages and R-messages are stored in a one dimensional array of vectors in the same way that the parity check matrix is stored, which reduces the required amount of dynamic memory to a manageable size, even for key lengths well in excess of 100 000 bits.

Information exposed by the LDPC protocol is very straightforward to quantify, since the only information exchanged between Alice and Bob is Alice's syndrome. Unfortunately, depending on the rate of the code, this syndrome can be quite large.

3.3.2. Parameters and factors

The parameters for which the Sum Product decoding algorithm is evaluated for Experiment 1 are summarized in Table 1:

Table 1. Sum product evaluation parameters

Parameter	Units	Description
Effectiveness	error percentage	Maximum correctable error percentage for a given code rate
Information	bits	Size in bits of the syndrome sent from Alice to Bob
Runtime	seconds	Runtime of algorithm not including message generation or error
Throughput	kilobits / second	Reconciled key bits produced per second
Iterations	count	The number of iterations required to correct all errors

3.3.3. Approach and methodology

The effectiveness of the Sum Product algorithm is defined here as the maximum correctable error rate for a given code rate. Parity check matrices are produced using the PEG algorithm for code rates between 0.1 and 0.9 in 0.05 increments for a key length of 100 000 bits. For each code rate the algorithm is run beginning with an initial error rate of 0.001. Success for a given error rate is determined in three ways, given by 1000 consecutive runs where 0, 50 and 100 failed runs are considered acceptable. For each run a new bit string is pseudo-randomly generated using a seeded RNG. After 1000 runs, the error rate is incremented by 0.001, and the algorithm begins another 1000 runs, with the same set of bit strings as before. This process continues until the algorithm is unable to correct all of the errors and a failed run occurs. The error rate at which the algorithm fails is recorded, and the next code rate is evaluated.

Information leaked is a trivial parameter to track with regard to LDPC codes, since the only information exposed over the classical channel is the transmitted syndrome, and the size of the syndrome is simply the m dimension of the $m\ x\ n$ parity check matrix. Therefore information leakage is directly correlated to the rate of the code $\left(r = 1 - \frac{m}{n}\right)$.

The runtime of the error reconciliation algorithm is measured using the *queryperformancecounter()* function native to the Windows C++ API. This function returns the value of the high-resolution performance counter, and can be used along with the *queryperformancefrequency()* function in order to calculate elapsed time to microsecond precision. After the messages are generated for both Alice and Bob, the clock is started. An Alice object calculates the syndrome for the flawless version of the message, which is then given to a Bob object along with the flawed message. The Bob object then attempts to correct all of the errors. As soon as Bob is finished, the clock is stopped, and the number of clock cycles that have elapsed are recorded. The number of clock cycles is then divided by the processor cycles per second retrieved from the queryperformancefrequency() function to obtain a runtime in seconds. This process is repeated for each error rate at the maximum code rate as determined by the effectiveness parameter. Only successful runs are considered, since in the Sum Product decoding algorithm, failed runs are detectable and always take a set number of iterations. Each time 100 runs are performed, and the minimum, maximum and average runtimes are recorded.

The throughput of the LDPC protocol is defined here as the reconciled key length (raw key length - bits leaked) divided by the runtime, and therefore the units are bits/s or bps, although a conversion factor of $\frac{1}{1000}$ is applied to obtain a final result in kilobits/s or kbps. The throughput is calculated for each error rate and is presented as an average corresponding to the recorded runtime values. Additionally, in order to simulate network latency 20 ms are added to the runtime for every communication exchange between Alice and Bob. For LDPC there is one message sent from Alice to Bob for each instance of the protocol representing the initial syndrome. Therefore 20 ms are added to the runtime for each instance, except in the case where the size of the syndrome exceeds the size of a network packet. In this case 20 ms are added to the runtime for each network packet required to transmit the syndrome.

Finally, for the Sum Product algorithm an additional parameter is the number of iterations required to correct all of the errors in a given bit string. This is an important parameter, as it directly correlates to the runtime of the algorithm. For this reason, a minimum, maximum and average iteration count will be presented for each error rate.

3.3.4. Expected Results

The quantum channel in a QKD system can be modeled as a Binary Symmetric Channel (BSC). A BSC in information theory is a channel model where a transmitted bit can take on two values (0 or 1), with a probability, p, of *flipping* to the other value in transmission. Therefore the probability of receiving a bit in the correctly transmitted value is $1 - p$.

In 1948 Claude Shannon, often known as the father of modern information theory, published a foundational paper on his mathematical approach to information theory. In his paper, Shannon developed a formula for the amount of uncertainty associated with a random variable. This formula is given by:

$$H(X) = -\sum_{x \in X} p(x) \log_2 p(x) \tag{3.1}$$

H(X) here is referred to as the *Shannon Entropy*, where p(x) represents the probability distribution function of x. For a BSC, by observing that X can only take on two possible values with probability p and $1 - p$, this equation can be reduced to:

$$H(X) = -p \cdot \log_2 p - (1-p) \cdot \log_2(1-p) \tag{3.2}$$

Shannon goes on to state in his *noisy channel coding theorem* that a maximum capacity exists for every channel C, and that information transmitted at a rate, r > C will have a failure probability increasingly greater than zero. At rates less than C however, information can be transmitted reliably with high probability. This *Shannon Limit* for a BSC is defined by:

$$C = 1 - H(X) = 1 - (-p \cdot \log_2 p - (1-p) \cdot \log_2(1-p))$$

$$C = 1 + p \cdot \log_2 p + (1-p) \log_2(1-p) \tag{3.3}$$

A graph of the Shannon limit with respect to increasing values of p is shown in Figure 8 below.

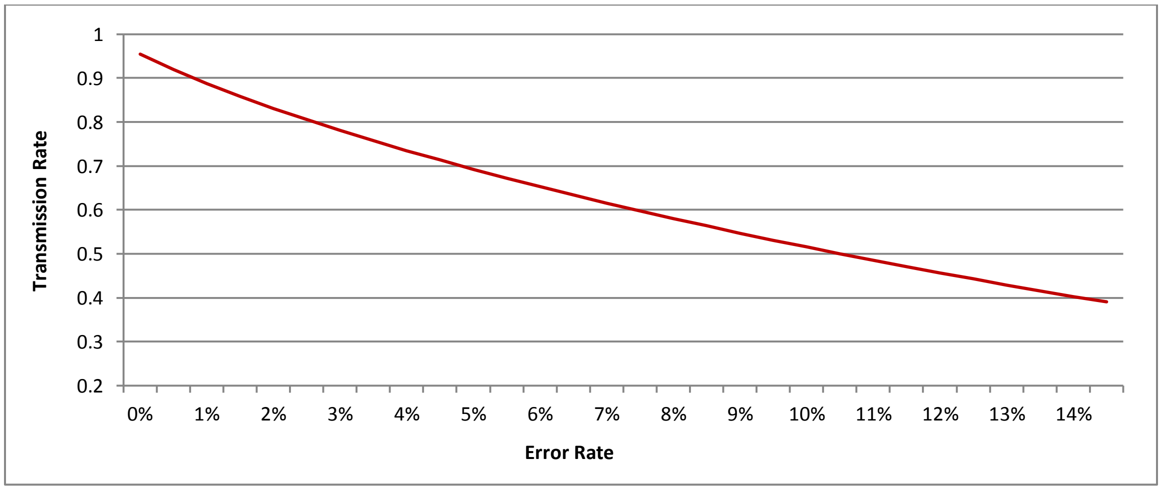


Figure 8. Shannon limit

With respect to QKD error reconciliation, the rate of a protocol is given by $r = \frac{length\ of\ reconciled\ key}{length\ of\ raw\ key}$. If one bit is discarded from the reconciled key for each bit exposed over the open channel, then the length of the reconciled key is simply the length of the raw key minus the number of bits exposed. For LDPC, the number of bits exposed is always simply the length of the syndrome exchanged.

Past research has shown that LDPC codes have the potential to operate very close to the Shannon limit (Elkouss, Leverrier, Alleaume, & Boutros, 2009; Elkouss, Martinex, Lancho, & Martin, 2010; Gudmundsen, 2010), and the Shannon limit is used as the benchmark for which the maximum effectiveness of the codes are evaluated.

The runtime of the LDPC protocol is expected to be high, and therefore the throughput is expected to be low. Previous research has shown runtimes for a 100 000 bit key length to be in seconds, which severely limits the reconciled key rate. Even still, the protocol requires very little information exchange, so it is expected that the overall throughput will be on the order of kilobits per second (kbps) even when network latency is taken into account.

3.3.5. Assumptions and Limitations

The largest assumption made in this implementation of the LDPC protocol is that the PEG algorithm with the chosen degree distribution produces good codes, in other words codes that have a wide girth and that perform reasonably close to the Shannon limit. However, the PEG algorithm is established and has been studied often since its foundational paper was published (Richter, 2005; Lin, Chen, & Chang, 2008), therefore this assumption is reasonable.

Furthermore, code rates in 0.05 increments are developed using the PEG algorithm. While smaller increments are possible, it is assumed that an increment of 0.05 represents a fine enough granularity in order to provide a reasonable estimate of the performance of the LDPC codes. Smaller increments would also increase number of codes and therefore the overhead associated with processing the codes for different error rates.

Finally, there have been numerous studies aiming to improve the runtime of the LDPC protocol (Gudmundsen, 2010; Chen, Dholakia, Eleftheriou, Fossorier, & Hu, 2005), most notably the implementation of the hyperbolic tangent operations as a lookup table. These improvements are not implemented here due to time constraints, and are not expected to dramatically decrease runtimes.

3.4. Experiment 2: Comparing LDPC to Cascade and Winnow

3.4.1. Cascade

3.4.1.1. Implementing Cascade

The Cascade protocol is also implemented as a single C++ class, with calls made to an external RNG class and a main driver class. Additionally, the Cascade implementation utilizes a permutation class that is responsible for the permutation performed after each pass. Since Cascade must maintain the ability to reverse permutations, a RNG alone would not suffice for the permutations. Therefore, the permutation class maintains an array of unique integers, randomly generated, in the range of 0 to the key string size, as well as a reverse array so that original indices can be obtained. An example is illustrated in Figure 9.

Permutation array:

0	1	2	3	4	5	6	7	8	9
7	5	0	8	2	3	9	1	4	6

Reverse Permutation Array:

0	1	2	3	4	5	6	7	8	9
2	7	4	5	8	1	9	0	3	6

Figure 9. Permutation arrays

In this way, a lookup of an index in the permutation array produces a random integer, and a lookup of the random integer in the reverse permutation array produces the original index. For later passes, in order to produce different permutations, multiple lookups are performed. For instance, indexing into position 0 produces random index 7, and indexing into position 7 produces random index 1. So for pass 2, index 0 would become index 1 (assuming no initial permutation was applied). All of this is transparent to the Cascade algorithm, which simply makes a call to a getIndex() or getReverseIndex() function and provides an index as well as the current pass number. In this way no explicit permutation between passes is needed, as index permutations are applied in line.

In the implementation presented here, a main function first creates a random bit string as well as an Alice object that it provides with a flawless copy of this key string. A Bob object is then created and given a copy of the key string with a certain percentage of uniformly distributed errors introduced. Alice and Bob are also given a common seed for the Permutation RNG, and an initial block size which corresponds to an ideal error rate. No initial permutation is performed. In addition, Bob is also passed a reference to the Alice object, which is only used to call the public functions required to calculate parity bits and can be equated with sending and receiving messages to Alice over the classical channel. At this point, Bob requests the parity bits for all Alice's blocks, and Bob initiates a binary search on any blocks that are in error, making calls to Alice's getParity() function to obtain parities for individual blocks. After each pass, both Alice and Bob double their block size and repeat the process until four passes have been completed. For any errors found in any pass after the first, Bob initiates a binary search on the blocks containing those errors in previous passes in order to identify potential matching errors.

In order to track the actual amount of information leaked, Bob maintains a bitsLeaked variable. In Cascade, the information exchanged is not constant and is actually a function of the error distribution, since the binary search will leak between $\lfloor \log_2 k \rfloor$ and $\lceil \log_2 k \rceil$ bits depending on the location of the error. To track actual bits leaked, Bob increments his bitsLeaked variable every time he requests a parity calculation from Alice.

3.4.1.2. Parameters (Cascade)

In order to evaluate the performance of the Cascade and Winnow protocol alongside the Sum Product algorithm, it is necessary to develop similar parameters. The parameters for which the Cascade protocol will be evaluated are summarized in Table 2:

Table 2. Cascade evaluation parameters

Parameter	Units	Description
Effectiveness	key rate	Key rate $\left(1 - \frac{reconciled\ key\ size}{sifted\ key\ size}\right)$ for a given error percentage
Information leakage	bits	Number of parity bits exchanged between Alice and Bob
Runtime	milliseconds	Runtime of algorithm not including message generation or error estimation
Throughput	kilobits / second	Reconciled key bits produced per second

3.4.1.3. Approach and Methodology (Cascade)

The effectiveness of the Cascade algorithm is defined here as a function of the reconciled key length for a given error percentage. Unlike LDPC, the key rate in Cascade is not static. Depending on the distribution of the errors, the reconciled key length, and therefore the key rate, fluctuates. In addition, starting block size is an important parameter in the Cascade algorithm. If the starting block size is too large, the algorithm may not be able to correct all of the errors. If the block size is too small, excess block parity bits may be exposed. The best starting block size therefore is the largest size that still results in all errors corrected.

The Cascade algorithm is run beginning with an error rate of 0.005 and increasing in 0.005 increments up to and including 0.15, similar to the method used for experiment 1. Since the key rate for Cascade is a continuous function, unlike the step function generated by LDPC, a 0.005 increment is considered sufficient. For each error percentage, the start block size is set at 5 bits. As before, success is determined three ways; as 1000 runs with 0 failures, 50 failures, and 100 failures respectively. After each run a new bit string is generated randomly using the same seeded RNG used for the LDPC experiment. When 1000 consecutive runs are completed with all errors corrected, the parameters are reset and the block size is incremented by 1 bit. This process is repeated with the same set of 1000 bit strings until the algorithm is unable to correct all of the errors. At this point the starting block size for the last successful set of runs is recorded, the error rate is incremented, and the process is repeated. In this way, the starting block size limit for each error percentage can be determined for a 100 000 bit key length.

Additionally, the maximum, minimum and average bits exposed are recorded for each error percentage. For this experiment it is only necessary to evaluate the bits exposed for the maximum starting block sizes, since the largest successful starting block size represents the optimal choice.

The runtime of the Cascade algorithm is measured using the queryperformancecounter() function, just as in the LDPC experiment. After the messages are generated for both Alice and Bob, the clock is started. Alice and Bob objects are provided with their respective message copies, and Bob initiates communication with Alice in order to correct all of the errors in his key string. After Bob is finished, the clock is stopped and the number of clock cycles that have elapsed are recorded. The number of clock cycles recorded is then divided by the processor cycles per second to obtain a runtime in seconds, and a conversion factor of 1000 is applied so that the final result is in milliseconds. This process is repeated for each error rate utilizing the maximum starting block size achieved in the effectiveness experiment. Even though the runtime of Cascade does not necessarily increase for failed runs, only successful runs are considered in order to ensure symmetry with LDPC. Each time 100 runs are performed, and the minimum, maximum and average runtimes are recorded.

The throughput of the Cascade algorithm is defined in the same way as the LDPC experiment in section 3.3.3, and is given by the reconciled key length divided by the runtime. The throughput is calculated for each error rate and is presented as a minimum, maximum and average corresponding to the recorded runtime values. Furthermore, in order to account for network latency in a uniform way, 20 ms are added to the runtime calculation for each message passed between Alice and Bob. For Cascade, this means one message for each block parity exchange, and two messages for each parity bit exchange (Bob must send his parity to Alice, and she must respond with match or mismatch). In order to track the number of messages exchanged, a separate, messageCount variable is maintained while the algorithm is running, and $20\ x\ messageCount$ milliseconds are added to the runtime of each of the iterations.

3.4.1.4. Assumptions and Limitations (Cascade)

The implementation of Cascade presented here is representative of the original algorithm as described by Brassard and Salvail (Brassard & Salvail, Secret-key Reconciliation by Public Discussion, 1994) and does not take advantage of any of the various enhancements that have been suggested in the literature since the original paper was published. Selecting which improvements offer the most benefit is outside the scope of this research. Therefore slightly better performance closer to the theoretical Shannon limit or with increased throughput may be possible.

Also, for this experiment, network latency is not considered empirically, and the experiment is conducted on one machine. The latency of the network is assumed to be 20 ms, which is a reasonable if not optimistic assumption given the current state of technology. Furthermore, the number of messages exchanged between Alice and Bob are highly dependent on the implementation. For instance, one implementation may have Alice and Bob exchange all their block parities in one message while another would separate the parities into individual messages. The method selected here takes a conservative measure, and assumes the minimum number of messages is exchanged.

3.4.1.5. Expected Results (Cascade)

The Cascade protocol has been shown previously to operate within a reasonable range of the Shannon limit (Brassard & Salvail, 1994; Elkouss, Leverrier, Alleaume, & Boutros, 2009), though not as closely as the LDPC protocol, and consequently the effectiveness of Cascade is expected to be lower than LDPC. The information exposed by Cascade and therefore the key rate is expected to vary due to the binary search, and perform slightly worse than the Sum Product algorithm, due mostly to the lower number of parity checks required by the LDPC protocol. The runtime however, even considering network latency, is expected to be lower than LDPC. This is due to the significantly lower level of computational complexity associated with the Cascade algorithm. As a result the overall throughput of Cascade is expected to be comparable to the LDPC protocol.

3.4.2. Winnow

3.4.2.1. Implementing Winnow

The implementation of Winnow used in the simulations presented here is derivative of an implementation created by another student (Lustic, 2011). The original C++ implementation was modified to use the same data types and inputs as the other protocols implemented; however the basic algorithm remains the same.

Similar to the Cascade implementation, the Winnow implementation consists of a main C++ Winnow class as well as the Mersenne Twister RNG class. Though in the case of Winnow it is not necessary to be able to reverse permutations; therefore the permutation is performed with a simple seeded RNG that randomizes the bit positions.

A main class creates a random bit string as well as Alice and Bob objects, and as before Alice is given the correct key string while Bob is given a faulty version with uniformly distributed errors. Alice and Bob are also provided a common seed for their permutation, as well as the error rate estimate. In addition, Bob is provided a handle to an Alice object, so that he can access her public parity and syndrome calculating functions, but not the key string.

At this point, Bob sends parity bits for each block to Alice, and instructs her to calculate the number of bad (mismatched parity) blocks as well as her syndromes for those bad blocks. Alice and Bob create the same parity check matrix using the method described in section 2.3.2, therefore their syndrome calculations are in sync. Alice sends the locations of the bad blocks and her syndromes to Bob, and Bob fixes the errors in his blocks using Alice's correct syndromes. At the end of each pass, Alice and Bob discard their required number of parity and syndrome bits (one for each bit exchanged) and apply a permutation randomly using the previously distributed seed before moving on to the next pass. The number of passes and the block size for the next pass are decided by the block size schedule, which varies depending on the key string size and error rate estimate.

The information leaked by Winnow is tracked using a bitsExposed variable which is incremented whenever a parity bit or syndrome is exchanged. If no errors are introduced, the value tracked by the bitsExposed variable represents the minimum number of bits that the algorithm can exchange in order to correct all of the errors in the bit string, since the only bits exchanged are one parity bit and one set of syndrome bits. However, if errors are introduced, even though they may be corrected in later passes and therefore have no effect on the effectiveness of the protocol, a syndrome must still be calculated and exchanged that would not have been had the error not been introduced. Therefore the number of bits exposed increases as new errors are introduced in the Winnow protocol, and the uniformity of the error distribution is paramount.

3.4.2.2. Parameters (Winnow)

The parameters for which the Winnow protocol will be evaluated are summarized in Table 3:

Table 3. Winnow evaluation parameters

Parameter	**Units**	**Description**
Effectiveness	key rate	Key rate $\left(1 - \frac{reconciled\ key\ size}{sifted\ key\ size}\right)$ for a given error percentage
Information leakage	bits	Number of parity and syndrome bits exchanged between Alice and Bob
Runtime	milliseconds	Runtime of algorithm not including message generation or error estimation
Throughput	kilobits / second	Reconciled key bits produced per second

3.4.2.3. Approach and Methodology (Winnow)

In the Winnow protocol the effectiveness is defined in a similar manner as the Cascade protocol, where the key rate is a function of the reconciled key length for a given error percentage. In the Winnow implementation used here, the starting block size is always set at 8 bits; however the block size does not always double after each pass. Instead, the block size is determined by a block size schedule and consequently so is the number of parity bits exchanged/exposed. The original Winnow algorithm acquired was written with a key length of 1000 000 bits in mind. Therefore, block size schedules for a key length of 100 000 bits are developed experimentally as part of this research effort.

The Winnow algorithm is run beginning with an error rate of 0.005 and increasing in 0.005 increments up to and including 0.15, identical to the method used for Cascade.

For each error percentage, a block size schedule is determined which corrects all errors while minimizing the number of bits exposed. Success for a given block size schedule is determined three ways; as 1000 runs with 0 failures, 50 failures and 100 failures, respectively. After each run a new bit string is generated randomly using a seeded RNG. When 1000 consecutive runs are completed with all errors corrected, the parameters are reset and the error rate is incremented by 0.005. This process is repeated until block size schedules are determined for each error rate. In keeping with the original author's implementation, the block schedules will be developed in error rate increments of 0.01. Therefore, each block schedule will cover two error rate increments.

Additionally, the maximum, minimum and average bits exposed are recorded for each error percentage. Since these values are dependent on the block size schedule, the bits exposed will vary for each error rate.

The runtime of the Winnow algorithm is measured using the queryperformancecounter() function just as in the LDPC experiment. After the messages are generated for both Alice and Bob, the clock is started. Alice and Bob objects are provided with their respective copies of the message along with the actual error rate, and Bob proceeds to communicate with Alice in order to correct all of the errors in his key string. After Bob is finished, the clock is stopped, and the number of clock cycles that have elapsed are recorded. The number of clock cycles recorded is then divided by the processor cycles per second to obtain a runtime in seconds, which is then converted to milliseconds for the final result. This process is repeated for each error rate. Each time 100 runs are performed, and the minimum, maximum and average runtimes are recorded.

In Winnow failed runs do not increase runtimes, however only successful runs are considered in order to maintain symmetry with the other two protocols.

The throughput of the Winnow algorithm is defined in the same way as the LDPC experiment in section 3.3.3, and is given by the reconciled key length divided by the runtime. The throughput is calculated for each error rate and is presented as a minimum, maximum and average corresponding to the recorded runtime values. Finally, 20 ms are added to the runtime for every message exchanged between Alice and Bob, in order to account for any network latency. For Winnow, the minimum communication possible is two messages for each pass; one for Bob to send Alice his block parities, and a second for Alice to respond with the number and location of bad blocks as well as syndromes for those bad blocks. Therefore $2\ x\ 20 = 40\ ms$ per pass is added to the runtime of each instance of the Winnow protocol. Additionally, for the first pass of the protocol for a 100 000 bit key string the number of block parities will exceed the size of a network packet. Therefore an additional 20 ms is added to the overall runtime to account for the additional message necessary in the first pass.

3.4.2.4. Assumptions and Limitations (Winnow)

In Winnow, due to fact that privacy maintenance occurs simultaneously with error reconciliation, the bit string will not always be evenly divisible by the current block size. The syndrome calculation, however, requires that the block size be of the same dimension as the parity check matrix. Therefore, in this implementation of Winnow, depending on the current block size, there may be an end block whose size is shorter than the block size. This block is ignored by the algorithm, and it is assumed that any errors that exist in this block will be shuffled back into another block in a later pass by the permutation applied between passes.

As with Cascade, the number of communication messages sent between Alice and Bob is dependent on the implementation. Here, a conservative estimate is taken, and it is assumed that Alice and Bob communicate with minimal message passing. Additionally, it is assumed that the latency is the same regardless of the size of the messages passed, and that the time to prepare or separate the message information is negligible.

Finally, for this experiment all errors are uniformly distributed, therefore no initial permutation is performed.

3.4.2.5. Expected Results (Winnow)

In the Winnow protocol, Hamming codes are generally small and are generated at runtime, and there are no overly complex mathematical operations. Therefore, it is expected that the runtime of Winnow will be lower and throughput will exceed that of LDPC. Also, due to the high quantity of binary searches required by Cascade, Winnow is expected to have a lower runtime and a higher throughput than Cascade. However, due to the high potential of introducing errors, it is expected that the effectiveness of Winnow will be lower than LDPC, and possibly the same as or lower than Cascade.

3.5. Experiment 3: Effects of inaccurate error rate estimation and Burst Errors

Up until this point only uniformly distributed errors with ideal error rate estimation have been examined. In the real world, neither of these criteria is likely to be the case. Therefore it is desirable to examine the performance of the three protocols with regard to inaccurate error rate estimation, as well as in the presence of burst errors.

Experiment 2 determined an upper bound for the correcting power (effectiveness) of each protocol. For this reason, it is meaningless to examine a situation where the error rate was underestimated, since the protocol would surely fail to correct all errors. However, if the error rate were overestimated, it is likely that the protocol would leak excess information to any eavesdroppers, and quantifying the amount of excess will provide a deeper understanding of the importance of error estimation and allow the development of an error rate estimation bound.

Burst errors are defined as errors that occur consecutively, for some burst length *b*. Burst errors are difficult to identify, particularly in protocols that utilize simple parity checks, since an even amount of errors will not be detected, and an odd amount of errors will register as only a single bit error. Burst errors are particularly treacherous in forward error correcting codes, since a large amount of errors increases the probability of decoding a code word into a valid, but incorrect code word. Many protocols rely on an initial permutation to distribute any burst errors, but depending on the quality of the permutation, this may not always be effective. With respect to the research conducted here, the Cascade and Winnow protocols utilize simple parity checks to identify errors, the Winnow protocol may actually introduce errors if the error rate is too high for an individual block, and LDPC and Winnow both utilize parity check matrices. Therefore it is highly desirable to measure the performance of these protocols in the presence of burst errors in order to determine whether effectiveness is adversely affected.

3.5.1. Parameters

The parameters examined in this experiment are given in Table 4 and in Figure 10:

Table 4. Experiment 3 evaluation parameters

Parameter	Units	Description
Error Rate	percentage	The estimated error rate of the quantum channel.
Uniform Distribution	bits	All errors are uniformly distributed
Single Large Burst	bits	All errors occur in a single burst of *b* bits
Single Small Burst	bits	50% of the errors occur in one burst of *b* bits. The remaining errors are distributed uniformly
Multiple Burst	bits per burst, burst count	Errors occur in *n* bursts of *b* bits each

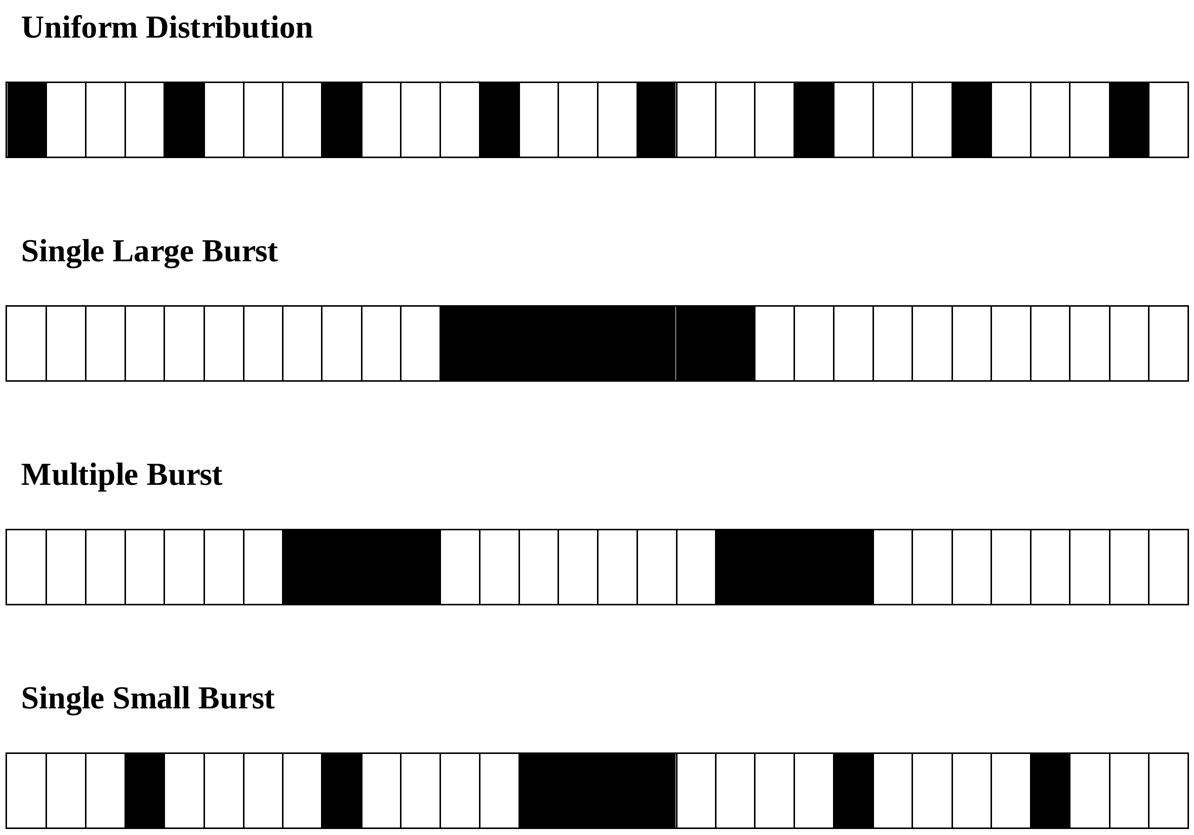


Figure 10. Burst types

3.5.2. Approach and Methodology

In order to evaluate the amount of excess information leaked by the different protocols in the presence of an overestimated error rate, each protocol is run 1000 times using the same seeded RNG used in Experiments 1 and 2. A static key length of 10 000 bits is evaluated. Beginning at the theoretical limit for a given error percentage, each algorithm is run 1000 times and the maximum number of bits leaked is recorded. The parameters corresponding to the error rate estimation for each protocol are then adjusted to equate to an estimate that is 0.005 higher than the actual error rate, and another 1000 runs are completed. Specifically, the protocol parameters are the starting block size for Cascade, the block size schedule for Winnow, and the code rate for LDPC. This process is repeated for higher and higher error rate estimations, until the error rate estimate is 2.5% higher than the actual rate. At this point, the actual error rate is incremented by 0.005 and the experiment is repeated beginning with an ideal error rate estimation until the full range of error rates (.03 - 0.15) has been covered. Finally, the maximum amount of information leaked is compared to the situation when the error rate is approximated ideally.

With regards to burst errors, each of the three error correction protocols are evaluated with ideal error rate estimation assumed. 1000 runs are performed for each parameter (Uniform Distribution, Large Burst, Small Burst and Multiple Burst) using the same seeded RNG as before. The minimum, maximum and average bits leaked as well as the runtime of the three protocols are recorded. After 1000 runs are performed for each parameter, the error rate is incremented by 0.005 and the process is repeated until the entire range of error rates (.005 - 0.15) has been covered. The results are compared to the data gathered earlier using a uniform error distribution, the goal being to determine if burst errors affect the information leaked, runtime or effectiveness of the protocols.

3.5.3. Assumptions and Limitations

It is assumed for this experiment that the lower bound on an error rate estimate is 2.5% greater than the actual error rate. For a relatively small error range of 0.005 to 0.15, 2.5% represents a reasonable bound on error rate estimation variance and estimates that vary outside this range likely represent a system that is highly unstable.

While burst errors could be evaluated in conjunction with declining error rate estimation accuracy, and additional burst distributions are certainly possible, these parameters are not evaluated here due to time constraints.

3.5.4. Expected Results

It is expected that as the error rate estimate drifts farther away from the actual rate, the information exposed will gradually rise with respect to the ideal estimation. This rise in information exposed is expected to be most significant in the Cascade protocol, since the number of bits exposed varies directly with the error rate. The rise should be moderate in Winnow, since block size schedules are in increments of 0.01, and therefore no change will be noticed until the error rate deviates from the ideal by 0.01 or more. In LDPC the rise should be least noticeable, since a single code rate is used across a modest range of error rates.

If the initial permutation is able to adequately distribute the burst errors, then no effect is expected to be evident in the effectiveness of the Winnow and Cascade protocols. In LDPC, no initial permutation is normally applied; however, if burst errors are found to adversely affect the algorithm, an initial permutation could be easily implemented. Burst errors are not expected to adversely affect the LDPC protocol since the parity checks are by nature widely dispersed.

3.6. Experiment 4: An analysis of key length and performance enhancements

Prior to this point, the key length has been held static for each experiment. In this experiment, the key length will be varied in order to determine whether the performance of the three protocols (Cascade, Winnow and LDPC) degrades for smaller key lengths. Larger key lengths will naturally degrade the performance of the algorithm with respect to runtime and may degrade the throughput; however it is less clear whether the effectiveness of the protocols will change for smaller key lengths. It may be beneficial to operate the protocols at smaller key lengths if shorter or partial keys are needed, or if throughput increases due to large differences in runtime. Furthermore, operating smaller implementations on equal parts of a large key could narrow the location of errors in an unresolvable key string, and allow a large percentage of the key to be retrieved correctly.

For the Cascade protocol, the initial block size is a function of the error rate and the bit string size. Therefore, regardless of the bit string size, the initial block size is decided such that, on average, one error remains in each block. Larger bit strings with the same percentage of errors would have the same initial block size and theoretically the initial block size should scale, so that no difference is observed in effectiveness for larger key lengths.

On the other hand, the Winnow protocol implemented here always begins with an initial block size of 8 bits, regardless of the size of the bit string. Even though the block size schedule is different for different bit string sizes, the initial size of 8 bits results in more blocks in the first pass of Winnow. An increase in the bit string size while maintaining an 8 bit initial block size and the same error percentage should result in an increase in effectiveness, since the likelihood of each block containing zero or one error increases.

Finally, with regard to LDPC codes, larger bit string sizes results in larger codes and more sparse parity checks, since the number of parity checks does not increase for larger codes, only different code rates. Greater sparseness of the parity checks for the same error rate results in a lower likelihood of two errors being connected to the same parity check, and therefore the effectiveness of the code should increase.

The goal of this experiment is therefore to evaluate the increase (if any) of the performance of each protocol with respect to different key lengths. This is accomplished by comparing the effectiveness, runtime and throughput of the protocols for key lengths of 1000, 10 000 and 100 000 bits.

3.6.1. Parameters

Table 5. Experiment 4 evaluation parameters

Parameter	**Units**	**Description**
Key length	bits	The length of the raw key string
Effectiveness	key rate	Key rate $\left(1 - \frac{reconciled\ key\ size}{sifted\ key\ size}\right)$ for a given error percentage
Information leakage	bits	Number of parity and syndrome bits exchanged between Alice and Bob
Runtime	milliseconds	Runtime of algorithm not including message generation or error estimation
Throughput	kilobits / second	Reconciled key bits produced per second

3.6.2. Approach and Methodology

Using the same methods detailed in experiments 1 and 2, the effectiveness, information leakage, runtime and throughput are obtained for all three protocols, for key lengths of 1000 and 10 000 bits. However, a static 5% error tolerance is examined which is expected to provide an accurate overall picture of the performance of the protocols.

For LDPC, parity check matrices are generated using the PEG algorithm with the same degree distribution as before. The maximum error rates achieved as well as the average, minimum and maximum numbers of iterations are recorded. For Cascade, the maximum starting block size that generates less than 5% failed runs is recorded, along with the average, minimum and maximum bits exposed. Finally, block size schedules are determined for Winnow, and the average, minimum and maximum bits exposed for the additional key lengths are obtained.

Additionally, the average, minimum and maximum runtimes of all three protocols are obtained by 100 runs at a 1000 and 10 000 bit key length. The throughput is calculated as $\frac{Reconciled\ key\ size - Bits\ exposed}{Average\ runtime + Latency\ correction}$, where the latency correction is 20 ms times the number of messages exchanged for each protocol.

The effectiveness, runtime and throughput of Cascade, Winnow and LDPC for key lengths of 1000, 10 000 and 100 000 bits are presented and conclusions are drawn about the overall best key length for use with each protocol.

3.6.3. Assumptions and Limitations

It is assumed that key length has an effect on the effectiveness of the protocols, even though that effect may be slight. Furthermore, it is assumed that the implementations used in this research function identically for different key lengths.

Finally, different key lengths require different parity check matrices for Cascade and different block schedules for Winnow. Therefore, it is assumed that these matrices and block schedules offer similar performance to their counterparts.

3.6.4. Expected Results

It is expected that the Cascade algorithm will produce only marginally different results for different key lengths. The Winnow and LDPC protocols however, are expected to increase in effectiveness considerably. The runtime of the algorithms is expected to increase significantly as the key length increases as well, due to the number of parity calculations that must be performed. The throughput on the other hand is expected to remain constant or decline slightly, since increases in runtime will be balanced by increases in the number of bits output.

IV. Analysis and Results

In this chapter the results of the experiments conducted in Chapter III are presented and conclusions are drawn.

4.1 Experiment 1: Evaluating the Sum Product LDPC decoding algorithm

The Sum Product algorithm was utilized to reconcile the errors in a transmitted key string, using no information other than the syndrome for Alice's version of the key string. The threshold for the Decision Step was set at ±10, which was determined empirically to give the best performance. The input to the inverse hyperbolic tangent function was limited to $\pm(1 - 10^{12})$ to assure finite results, and the input to the hyperbolic tangent function was monitored to prevent zero values. In the case of zero values, the input was defaulted to the log likelihood ratio of the channel error probability. Finally, the upper limit on the number of iterations before determining a run had failed was set at 200. While a higher threshold admittedly provided slightly better performance, the average number of iterations for successful runs was far less than the 200 iteration threshold, and the performance gain was not significant enough to warrant the greatly increased runtime for failed runs.

4.1.1. Effectiveness and Information Leakage

A graph of the achieved effectiveness of the LDPC protocol for 1000 trials at a 100 000 bit key length for 0%, 5% and 10% error tolerances along with the theoretical limit is given in Figure 11. The x-axis represents the error rate of the channel and the y-axis represents the reconciled key rate of the protocol, which is defined as $\mathbf{1 - \frac{\mathit{information\ bits\ exposed}}{\mathit{key\ length}}}$. The number of information bits exposed for LDPC is simply the length of the syndrome exchanged. The graph appears as a step function, since an individual code maintains the same key rate until it reaches its error threshold value, at which point a new code must be used. The severity of the step function could therefore be reduced by utilizing more codes; however this would result in increased computational or dynamic memory requirements in order to process the additional codes. Alternatively, the use of a master code which is then modified for different error rates has been proposed, however the achievable error rates for this method is usually lower.

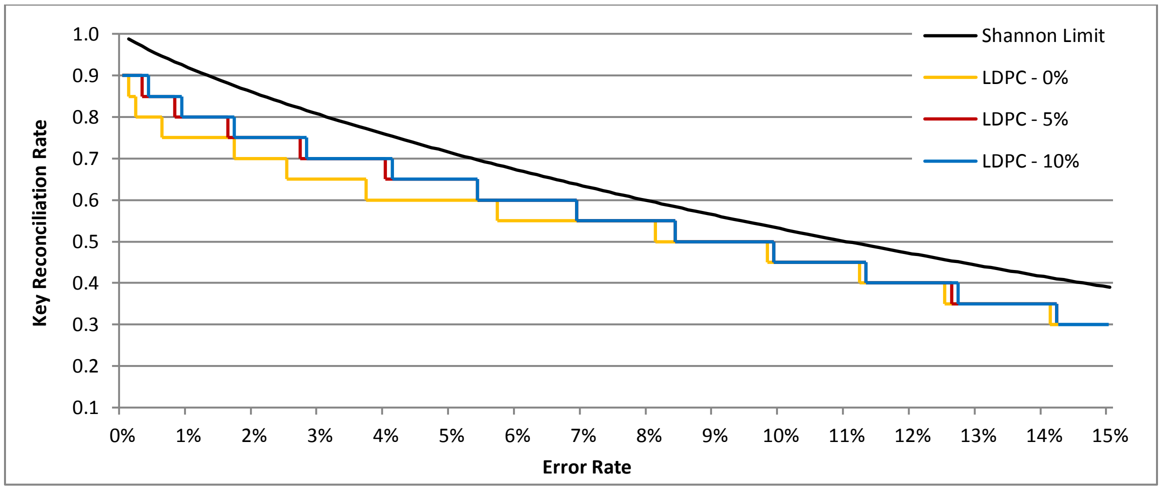


Figure 11. LDPC effectiveness

The yellow line in Figure 11 illustrates the case where no errors were observed. In this case, the LDPC codes performed reasonably close to the Shannon limit for error rates less than 5%, however a noticeable increase in performance was observed for error rates above 5%. In this range (5%-15%), the codes performed exceptionally well. The reason for the poor performance at low error rates may be related to the fact that higher rate codes have a much higher number of parity bits per check node, and therefore errors are more easily masked. Another possibility may be that the density distribution used to generate the LDPC codes used here was not optimal for lower error rates.

By allowing for a small percentage of errors, the maximum rate achieved increased across the entire range of error rates, as evident by the red line in figure 11 (in cases where only the blue line is visible, the blue overlaps the red). The effect is most noticeable in the range of 0% to 8% and less substantial for the higher end. This improvement was generally not as significant when transitioning from a 5% error rate to a 10% error rate, as indicated by the blue line.

The reason for the large improvement when allowing for a small percentage of errors is that while LDPC is an effective protocol, the sparse parity check nature contributes to bits combining in such a way that is nearly unpredictable, and consequently an occasional run occurs where not all errors are resolvable. Whereas in Cascade and Winnow a single bit error in a given block will always be correctable, in LDPC there are no block subdivisions, and depending on the error distribution certain errors may be masked to the parity checks. In general, this convergence of errors to elicit a failed run occurred infrequently, and consequently the increase for a 5% error tolerance is notable. By the time the 5% tolerance is reached, it appears that each code is very near its true limit, since tolerating higher levels of errors did not result in significant gains in maximum effectiveness.

4.1.2. Runtime, Throughput and Iteration Count

A graph of the number of iterations as well as the runtime of the LDPC protocol for a 100 000 bit key length is given in Figure 12. The runtime was recorded using a 5% error tolerance, which provided a good representation of effectiveness while limiting the number of failed runs. The unsuccessful runs were not considered in the runtime calculations. For this implementation a run is considered unsuccessful after 200 iterations; therefore the runtime for the failed runs is mostly static. Additionally, since unsuccessful runs always terminate after 200 iterations, failed runs are always detectable, though it may be that only a small number of errors remain. This detectability is advantageous in QKD, since failed instances can be detected without further communication with the sender, and LDPC can be combined with another protocol to resolve remaining errors.

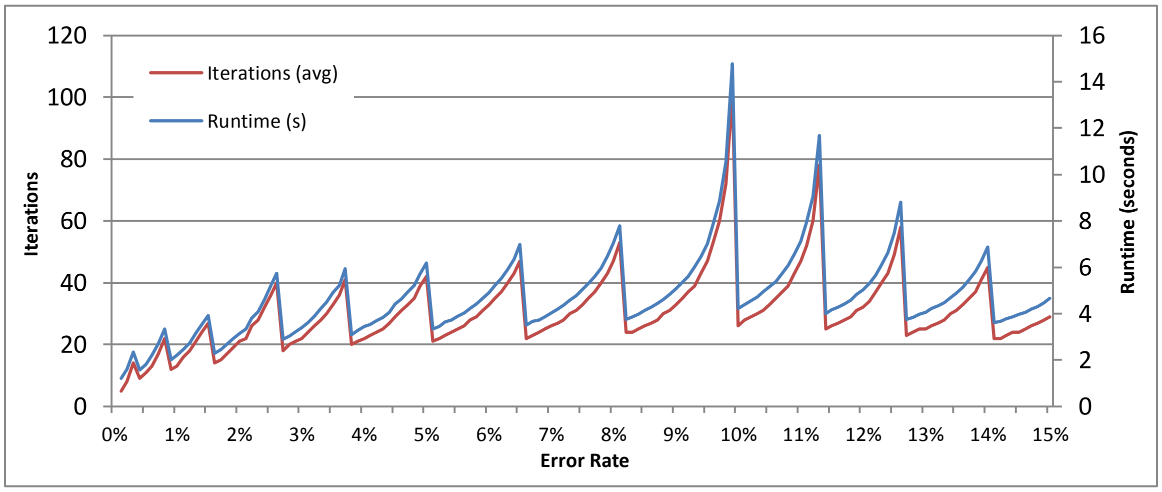


Figure 12. LDPC protocol runtimes and iterations

The runtime units are in seconds (s) and do not include any correction for network latency. The runtime of the protocol increases as the error rate increases, since it takes additional iterations in order to correct all of the errors. This increase reaches a maximum when the limit is reached for the current code. At this point a new code is utilized which is able to correct the same amount of errors in a lower number of iterations, resulting in a much reduced runtime at the cost of increased information exposure due to a larger syndrome. The largest peak is seen between an 8% and 10% error rate. This increase may be an indication that the step between codes is too large, and could almost certainly be reduced by the use of an additional code

Finally, the throughput of the LDPC protocol is shown in Figure 13. The throughput is calculated by $\frac{Reconciled\ key\ size - Bits\ exposed}{Average\ runtime + Latency\ correction}$ and the latency correction in the denominator is a correction factor used to account for the syndrome that must be exchanged between Alice and Bob. The time it takes to exchange a single network packet is assumed to be 20 ms. A standard Ethernet frame contains 1500 bytes or 12000 bits of data, therefore the latency correction was adjusted to correspond to the syndrome size by dividing the syndrome by 12 000 and multiplying the result by 20 ms.

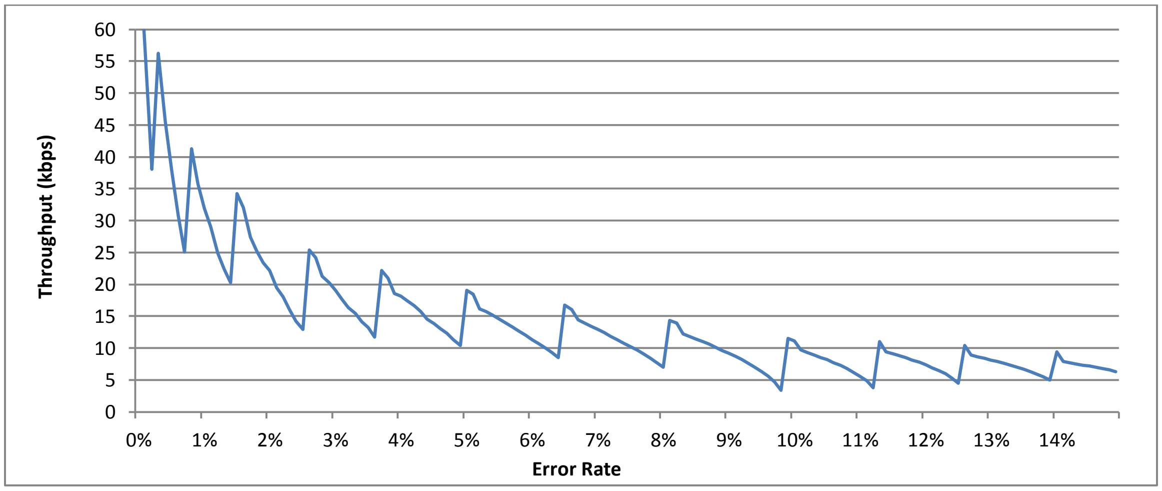


Figure 13. Throughput of the LDPC protocol

The throughput units are in kilobits per second (kbps). Naturally, higher rate codes perform faster (since the syndrome is smaller) and therefore have a higher throughput, and lower rate codes have a lower throughput, but can reliably correct a greater percentage of errors. The saw tooth shape of the graph is again attributable to the switch between codes. The average achieved throughput was 14.53 kbps. Even disregarding the early behavior above 25 kbps the average drops only slightly to 12.87 kbps.

Overall, the performance of the LDPC protocol is as expected. The achieved error rates and throughput are well within the range of a useful protocol for QKD. As anticipated the runtime is high, however since the protocol requires very little interaction the throughput is very respectable, and both parameters can be improved by faster or dedicated hardware, or by parallelization. In the next experiment the results of experiment 1 will be compared to similar parameters for the Cascade and Winnow protocols.

4.2. Experiment 2: Comparing LDPC to Cascade and Winnow

For this experiment the Cascade and Winnow protocols were run without an initial permutation, as all errors were uniformly distributed. The data types and external classes used were consistent across all three implementations, and all runtime measurements were made at approximately the same time in order to minimize environmental impacts such as system load, etc.

4.2.1. Effectiveness and Information Leakage

The effectiveness of the Cascade and Winnow protocols for 1000 trials at error tolerances of 0%, 5% and 10% as well as the values obtained for the LDPC protocol and the theoretical limit are given in Figures 14, 15 and 16. Results are examined for a static 100 000 bit key length. The x-axis represents the error rate of the channel and the y-axis represents the reconciled key rate of the protocol, which is defined as

$1 - \frac{\boldsymbol{information\ bits\ exposed}}{\boldsymbol{key\ length}}$. Information bits exposed for Cascade and Winnow were tracked as the algorithms ran and are presented as an average. Minimum and Maximum values were tracked as well and can be found in Appendix A.

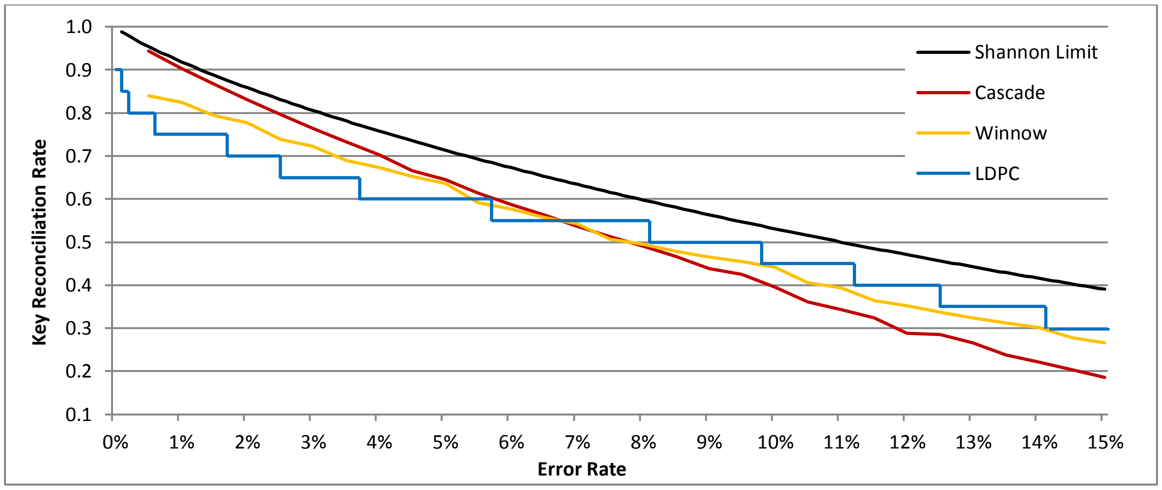


Figure 14. Protocol effectiveness - 0% Error Tolerance

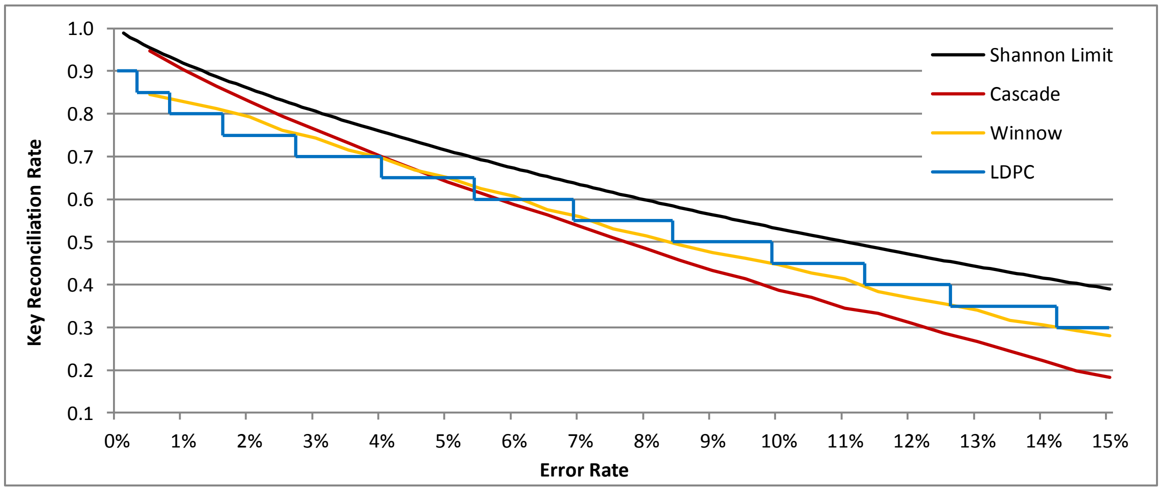


Figure 15. Protocol effectiveness - 5% Error Tolerance

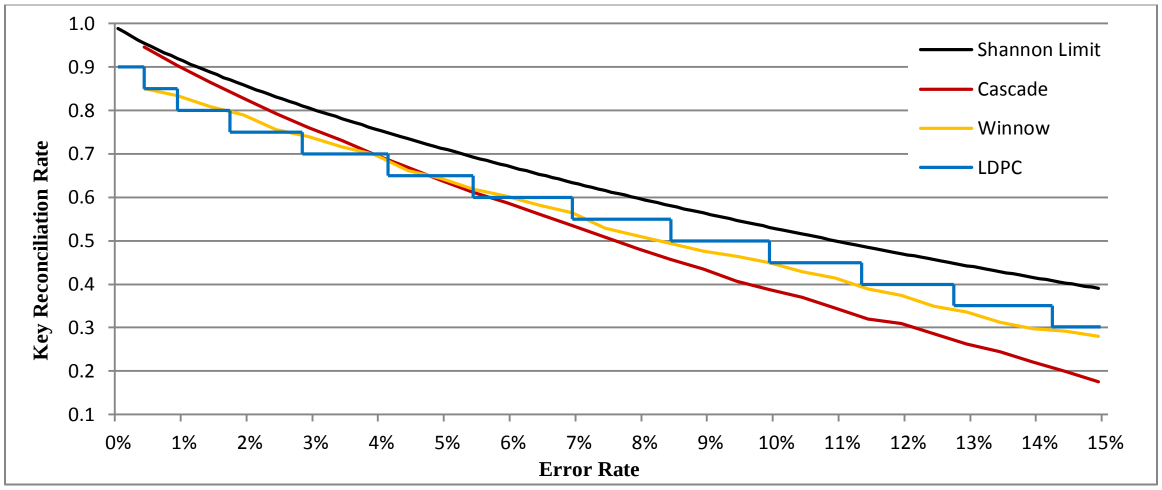


Figure 16. Protocol effectiveness – 10% Error Tolerance

For Cascade, the achieved effectiveness generally did not fluctuate significantly regardless of changes in error tolerance. Winnow performed equally consistently, though slight gains were noticed at error tolerances above 0%. LDPC performed the worst for a 0% error tolerance, and demonstrated significant gains from increased error tolerances, as outlined in the previous experiment.

In general, Cascade outperformed LDPC and Winnow for error rates less than 5%. At rates greater than 5%, Cascade diverged progressively farther from the theoretical limit, while Winnow continued at approximately the same distance. The achieved effectiveness of the LDPC protocol was lower than Cascade below a 5% error rate and roughly the same as Winnow. However, for error rates above 5%, LDPC achieved error rates exceptionally close to the theoretical bound, and significantly outperformed Cascade and Winnow.

The fact that the effectiveness did not increase for Cascade despite the introduction of a tolerance for errors is not unexpected, since the protocol is not aware of failed runs, and the operation of the algorithm does not generally change when a failed run occurs. The exception is the starting block size, which can be increased for even a small error tolerance. A graph of the starting block sizes for Cascade is shown in figure 17 below.

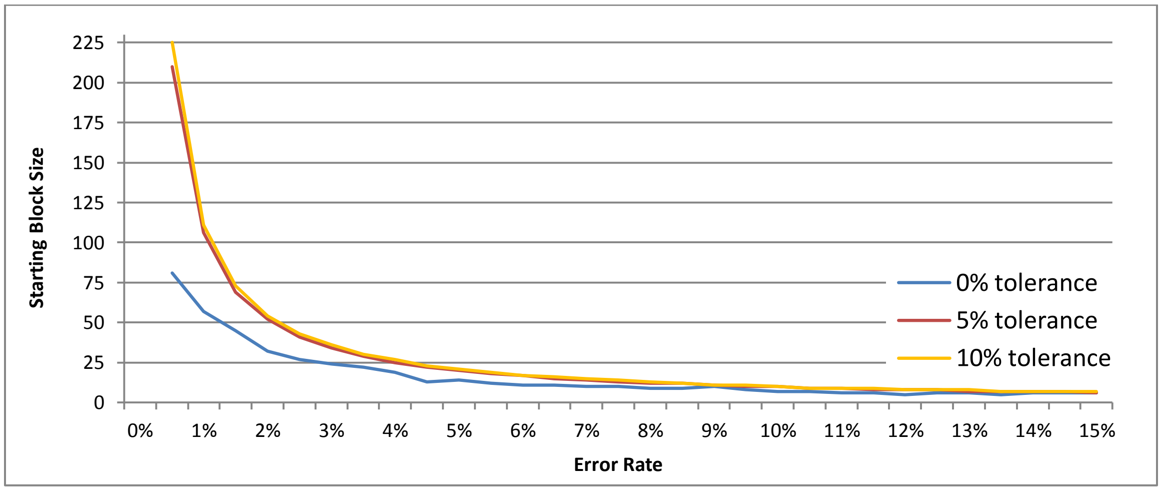


Figure 17. Cascade initial block size

Initially, the difference between starting block sizes is significant, though the effect is less noticeable as the error rate increases. In their original paper, Brassard and Salvail empirically determined a starting block size of $\frac{.73}{p}$ to be sufficient. Here, for a 0% tolerance, the maximum block size achieved was, on average $\frac{.71}{p}$. For a 5% tolerance the average increased significantly to $\frac{.99}{p}$ and for 10% a marginal gain of 0.04 to $\frac{1.03}{p}$ was seen. Despite these larger block sizes the effectiveness did not increase significantly. Though the number of block parities exchanged was reduced due to larger initial block sizes, the number of parity bits required to perform the binary search routine increased, and therefore the overall number of bits exposed was roughly the same.

For Winnow, a slight increase in effectiveness was noticed when a 5% error tolerance was introduced, but virtually no increase was noticed when the tolerance was increased to 10%. The block size schedules for Winnow are significantly different for varying error tolerances; therefore some compensation is possible when an error tolerance is introduced. However at a certain error percentage the Winnow protocol begins to introduce new errors and it is not much beyond that point that the protocol fails catastrophically.

The fact that the effectiveness did not fluctuate significantly for the Cascade and Winnow protocols stands in contrast to LDPC, where significant gains were noticed for even a small error tolerance.

Altogether in comparison to Cascade and Winnow the LDPC protocol achieved desirable maximum error rates when operated at the 0% error tolerance level, and performed exceptionally well for error rates greater than 5%. Allowing for even a small amount of errors increased effectiveness considerably, and in many cases LDPC performed closer to the theoretical bound than either Winnow or Cascade.

4.2.2. Runtime and Throughput

The runtimes for the three protocols for a static 5% error tolerance are shown in Figure 18, with no adjustment made for network latency. Due to the fact that the LDPC protocol runtime is so much greater than the runtimes of Cascade and Winnow, the graph is split, and two y-axes are displayed. The runtimes for Cascade and Winnow correspond to the left hand axis and LDPC to the right hand axis.

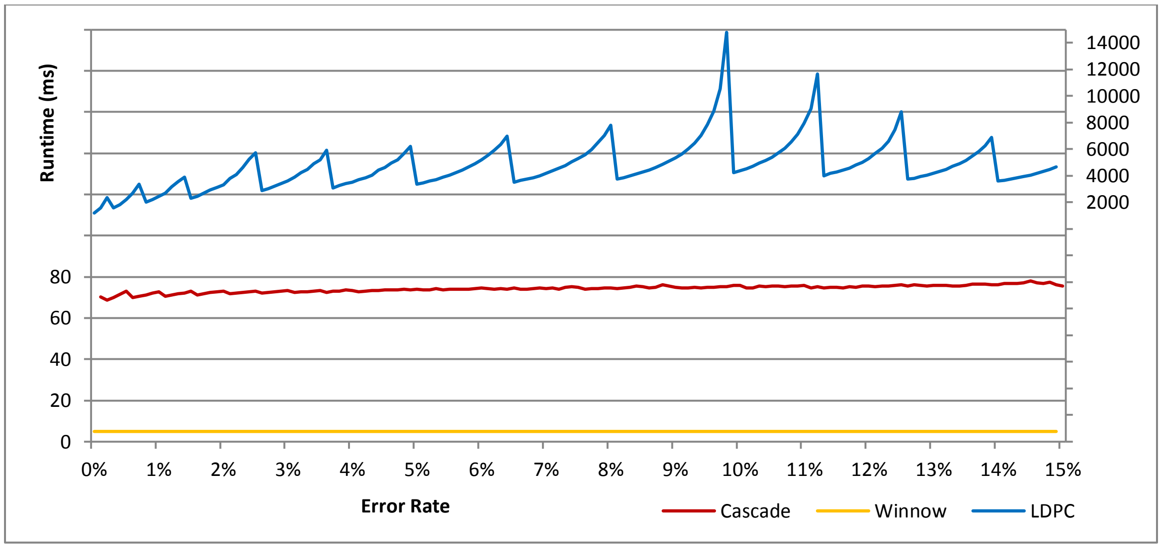


Figure 18. Runtimes for the Cascade, Winnow and LDPC protocols

The units for the runtimes are in milliseconds (ms). Notice that the runtimes for Cascade and Winnow do not increase significantly for larger error rates. Cascade always performs 4 passes, and the number of passes in Winnow does not deviate greatly for different block schedules. More errors do correspond to more parity calculations for both protocols, but that is a fairly trivial and efficient operation computationally. The runtime for LDPC, on the other hand, increases greatly as the error percentage increases, for reasons outlined in section 4.1.2. Winnow is particularly efficient and is noticeably faster than Cascade. This speed differential is most likely due to the binary searches required in the Cascade algorithm, and the fact that the cascading nature leads to an exponential rise in the number of binary searches for errors found in later passes.

In order to calculate the throughput for the Cascade protocol, the number of messages passed between Alice and Bob was tracked as the algorithm ran, and therefore the throughput is presented here as an average. For Winnow, the number of messages required is dependent on the number of passes and therefore the block schedule. Finally, for LDPC, only one message is exchanged between Alice and Bob, unless the syndrome is too large to fit in a single network packet. The throughput is then calculated by $\frac{Reconciled\ key\ size - Bits\ exposed}{Average\ runtime + Latency\ correction}$, where the latency correction this time is $20\ ms \cdot message\ count$. The throughput for all three protocols is shown in Figures 19 and 20.

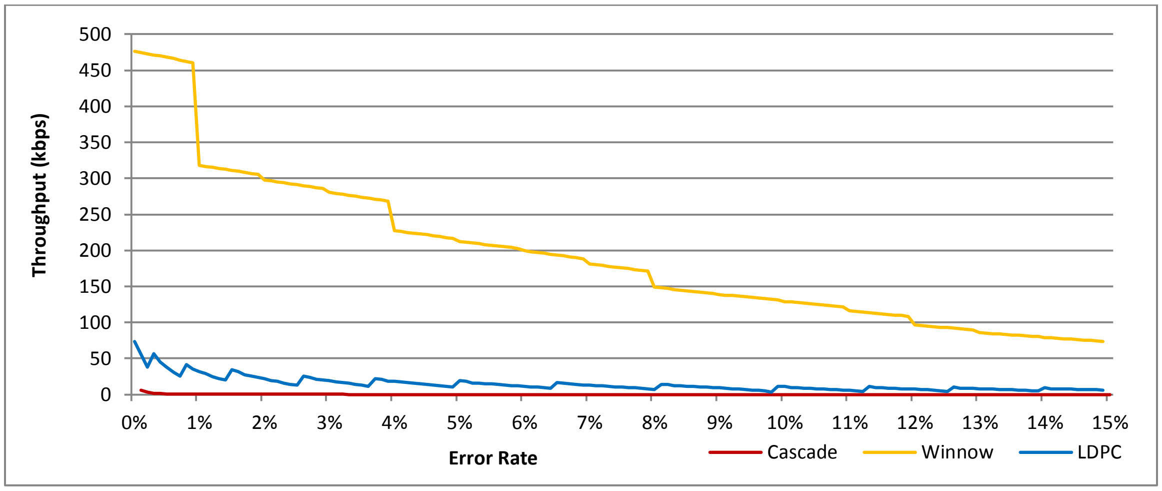


Figure 19. Throughput for Cascade, Winnow and LDPC

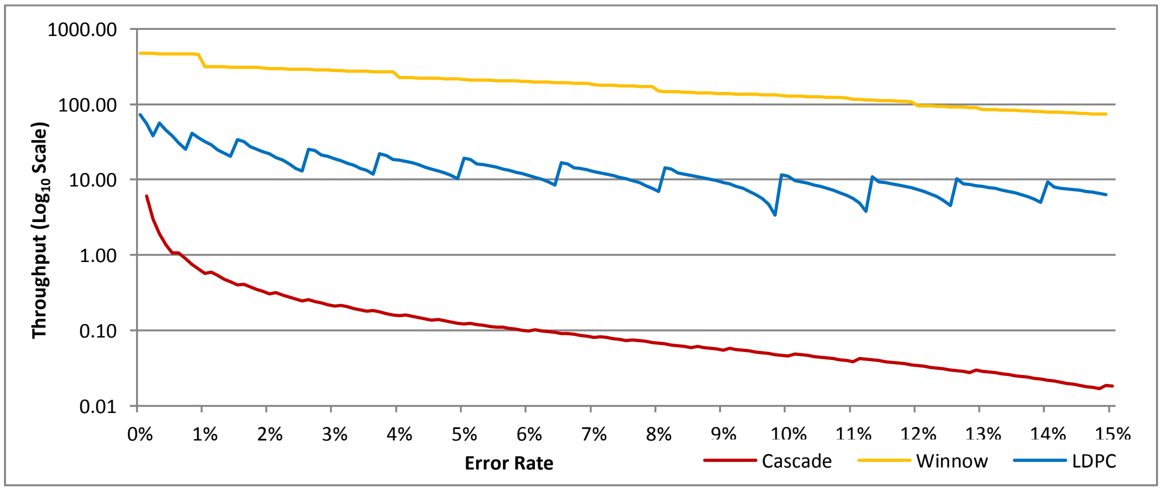


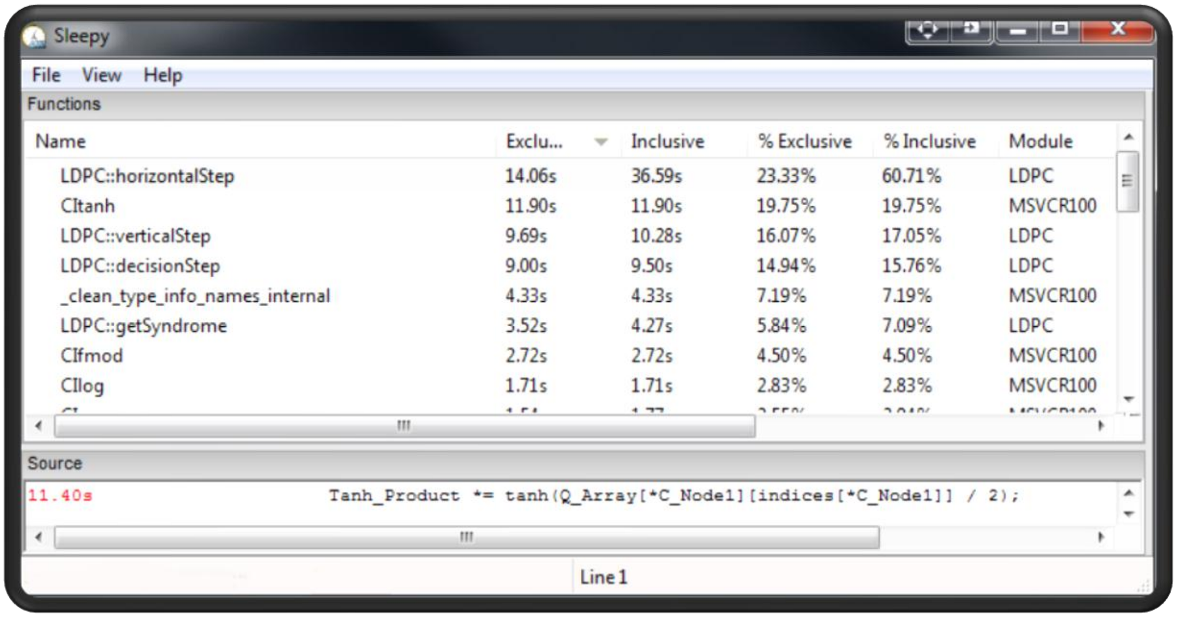
Figure 20. Throughput for Cascade, Winnow and LDPC (Log$_{10}$ scale)

The units of the throughput are again in kbps. For clarity a logarithmic scale graph is also presented, since the difference in throughput for the three protocols is large.

Clearly, any advantage that Cascade had in runtime is eliminated as soon as network latency is considered. The communication required between Alice in Bob for the binary search portion of the algorithm significantly reduces the throughput to an average of only 220 bps. It is also evident that Winnow dominates both protocols for the entire range of error rates with an average of 195 kbps. The reason for this is that Winnow is both computationally efficient and requires minimal communication. The steps that are visible in the Winnow throughput represent the change to a new block size schedule, and for an error rate of less than 1% the throughput increases significantly.

Although the Winnow throughput is significantly higher than that of LDPC, the rate achieved by the LDPC protocol is noteworthy at an average of approximately 15 kbps. Since in the LDPC protocol only one message is exchanged in most cases, the runtime of the overall algorithm and consequently the throughput are mostly independent of network speed. Therefore the throughput is easier to improve upon than a protocol which is dependent on a network.

In order to further investigate the behavior of the LDPC protocol, an instance of the Sum Product algorithm was profiled using a software profiling tool known as Very Sleepy (Chapman, 2012). The output of this tool is shown in Figure 21.



Name	Exclu...	Inclusive	% Exclusive	% Inclusive	Module
LDPC::horizontalStep	14.06s	36.59s	23.33%	60.71%	LDPC
CItanh	11.90s	11.90s	19.75%	19.75%	MSVCR100
LDPC::verticalStep	9.69s	10.28s	16.07%	17.05%	LDPC
LDPC::decisionStep	9.00s	9.50s	14.94%	15.76%	LDPC
_clean_type_info_names_internal	4.33s	4.33s	7.19%	7.19%	MSVCR100
LDPC::getSyndrome	3.52s	4.27s	5.84%	7.09%	LDPC
CIfmod	2.72s	2.72s	4.50%	4.50%	MSVCR100
CIlog	1.71s	1.71s	2.83%	2.83%	MSVCR100

```
11.40s          Tanh_Product *= tanh(Q_Array[*C_Node1][indices[*C_Node1]] / 2);
```

Figure 21. Profile of LDPC algorithm

The profiler was run for 60 seconds on a set of active LDPC runs for a 100 000 bit key length. The inclusive column includes the runtime of the function as well as any external calls, while the exclusive column refers only to operations performed within that function, with the runtime of any external calls excluded. Figure 21 is sorted descending by the exclusive column.

From figure 21, we can see that a large majority of the runtime (19.75%) is spent on hyperbolic tangent function calls. Therefore, reducing the time needed for those calculations could significantly improve the runtime and throughput of the algorithm. This could be accomplished through the use of previously calculated values stored in dynamic memory, parallelism, or a custom circuit designed to efficiently evaluate hyperbolic tangent functions.

For the LDPC protocol, improving the speed of the decoding algorithm implementation will nearly always result in improved throughput. This is not the case for

Cascade and Winnow, which are heavily reliant on the performance of the entire communication network. The achieved error rates for LDPC are significantly better than Cascade for the entire range of error rates examined here, and the overall throughput is notably better. Although the throughput for Winnow is higher, the LDPC algorithm achieves considerably higher error rates, and is therefore just as viable an option for the error reconciliation phase of QKD protocols. Furthermore, in situations where secure two-way communication is difficult, such as communications with moving objects such as satellites, LDPC offers a way to resolve transmission errors with only one message exchange.

In the next experiment, the effects of inaccurate error estimation as well as non-uniform error distributions on the Cascade, Winnow and LDPC protocols are examined.

4.3. Experiment 3: Effects of inaccurate error rate estimation and Burst Errors

4.3.1. Inaccurate error rate estimation

In this experiment the effects of inaccurate error rate estimation are examined. The Cascade and Winnow protocols were run 1000 times at estimated error rates that were increasingly greater than the actual error rate, in 0.005 increments. The maximum amount of information exposed for each protocol was then tracked. A static 10 000 bit key length was used, and the protocol parameters were set for a 0% error tolerance. For LDPC, only an analytic analysis was necessary, since information leakage in LDPC is a function of the code used, which is dependent solely on the error rate. The results of these runs and analyses are given in Figures 22, 23 and 24.

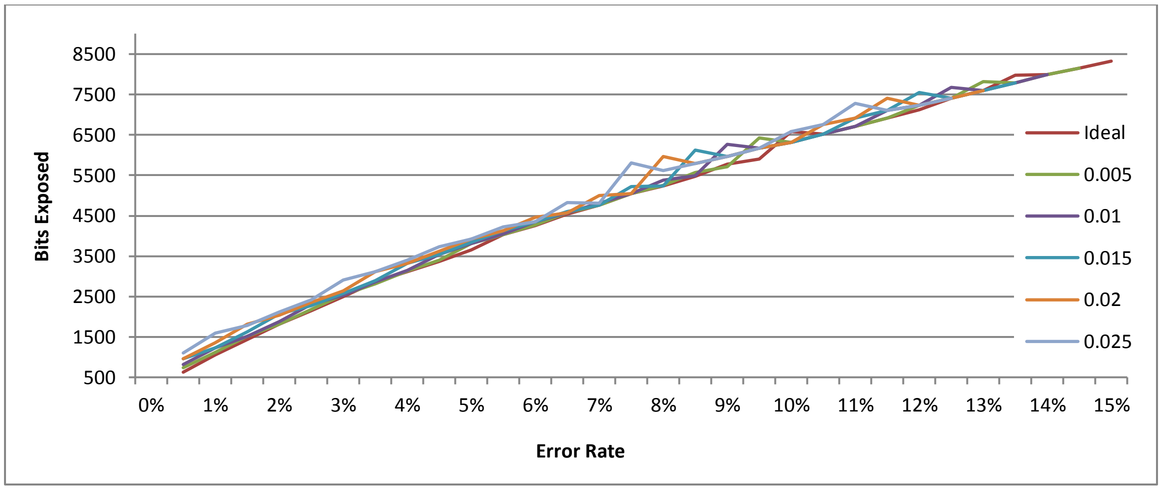


Figure 22. Effects of inaccurate error rate estimation - Cascade

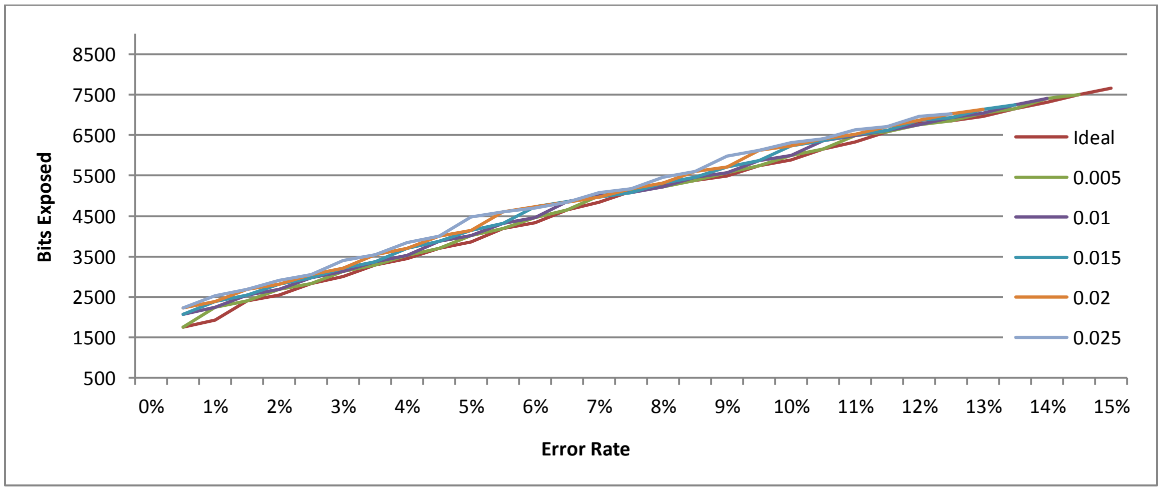


Figure 23. Effects of inaccurate error rate estimation - Winnow

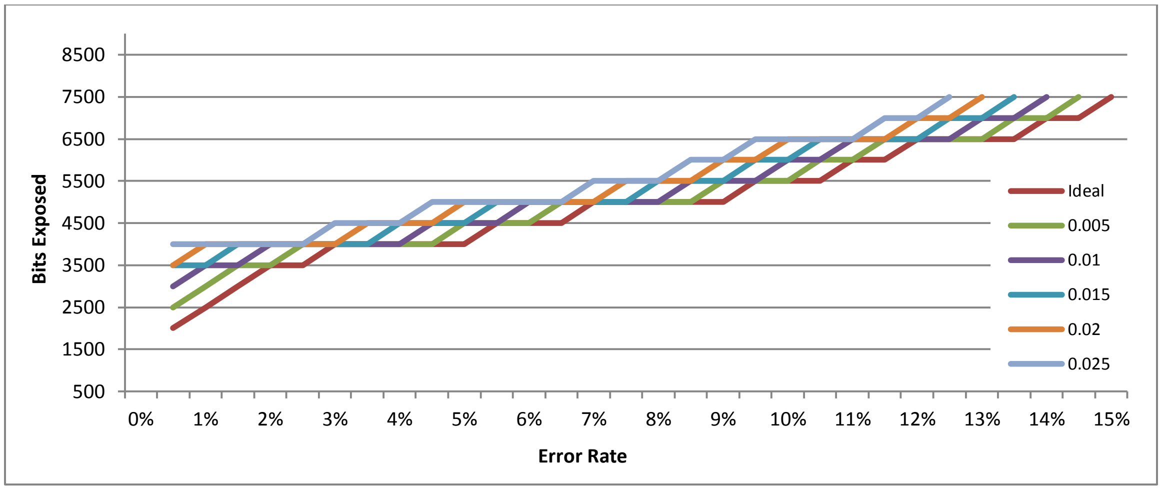


Figure 24. Effects of inaccurate error rate estimation - LDPC

For Cascade, the amount of information exposed varied only slightly for marginally overestimated error rates, however as expected much larger increases were experienced as the estimate diverged from the ideal value. Although it appears from the graph that a larger increase was experienced for larger error rates, this is erroneous, and is actually the result of a ripple effect experienced due to a spike in information leakage for ideal estimation. In general, the largest increase in the percentage of bits leaked was actually experienced for inaccurate estimation at lower error rates, with the maximum increase occurring at an error rate of only 0.5% overestimated to be 3%. This increase of nearly 77% in information leakage is largely due to the fact that for Cascade at smaller error rates the steps between block sizes are larger, and therefore an inaccurate error estimate is more costly.

In the case of Winnow, the information exposed varied relatively uniformly, regardless of the error rate. The peak gain in parity bits exposed was 31.6%, and occurred at a 1% error rate overestimated by 2.5% to be 3.5%. However, the average gain in exposure peaked at only 9% for an overestimate of 2.5%. Overall, the additional information exposed by overestimating the error rate was not significant for a rate that was within 1.5% of the correct value, at an average of less than 4% over ideal. The reason for these small increases was that the Winnow protocol as implemented here maintained the same block schedule for error rate increments of 1%, therefore in each case it would take two steps before the bits exposed increased over the ideal value. Additionally, the differences between consecutive block schedules were generally small, therefore it was not until an overestimate of 2% or greater that a block schedule was used that was significantly different than the correct one.

The shape of the LDPC graph in figure 24 is due to the fact that if an error estimate was too far off of the correct value, it would prompt the use of a code rated for a lower error rate which exposed more information. Hence, once an error estimate wandered too far, the information exposed jumped to the next level, in this case an additional 500 bits. This effect could be mitigated by the use of additional codes, but with tradeoffs in runtime and/or memory requirements as discussed earlier.

4.3.2. Non-Uniform Error Distributions

Since in real world systems it is unlikely that errors will always occur uniformly, this experiment sought to examine the impact of non-uniform or burst errors on the three error reconciliation protocols. For identical input strings, each protocol was run 100 times at each error rate. Initially, no permutations were performed before the beginning of the protocol. Again the key length examined was static at 10 000 bits.

In the first case, all errors were contained in a single burst. The location of the burst was generated using a seeded random number generator to assure uniformity. The number of errors remaining was recorded and is presented in Figure 25. Next, the errors were divided into 5 bursts of equal length, and the locations were again randomly generated, but steps were taken to assure that none of the bursts overlapped. The results for these runs are presented in Figure 26. Finally, the case was considered where a single burst containing half of the errors was generated and the remaining errors were distributed uniformly. These results are graphed in Figure 27.

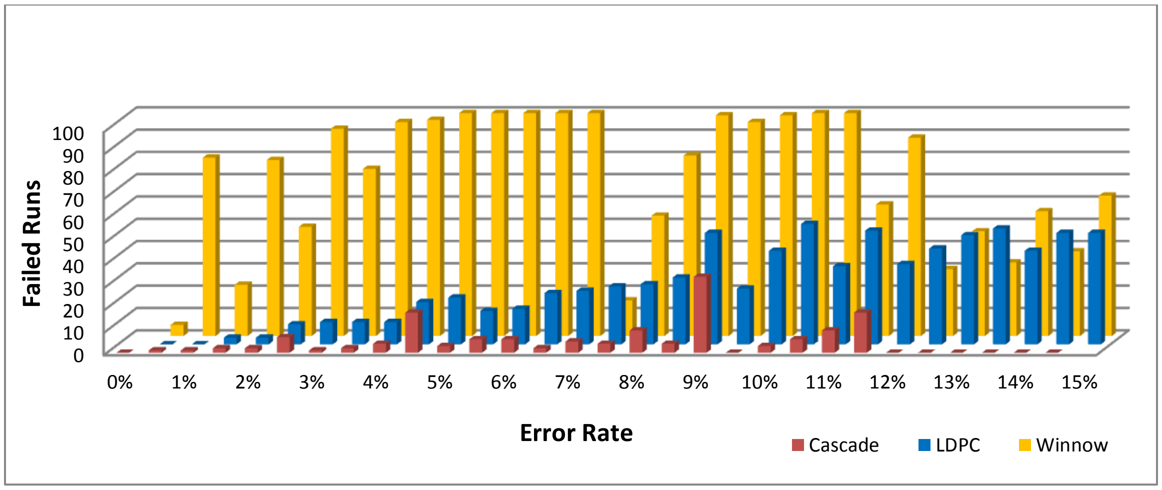


Figure 25. Error reconciliation results for a single burst of errors

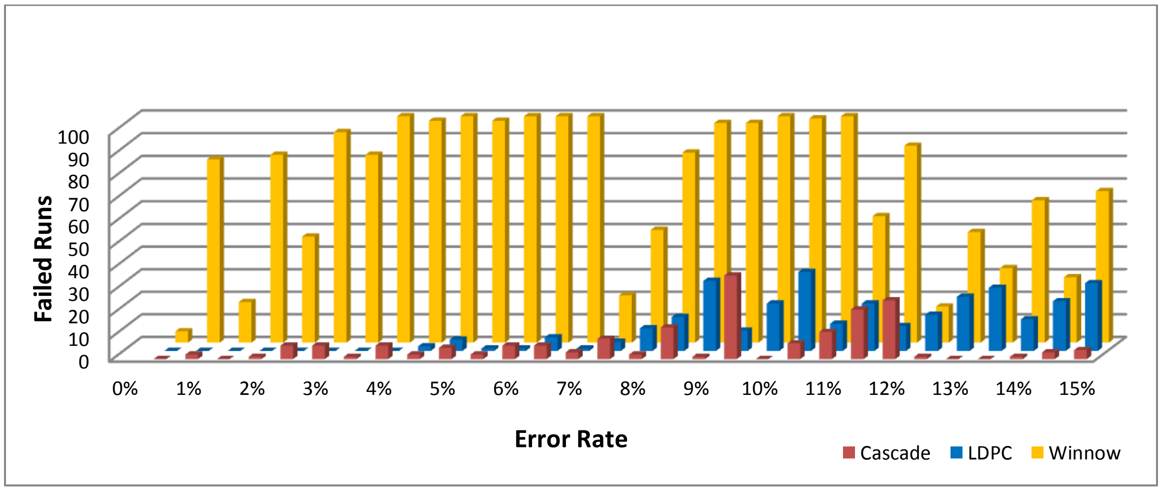


Figure 26. Error reconciliation results for five equal size error bursts

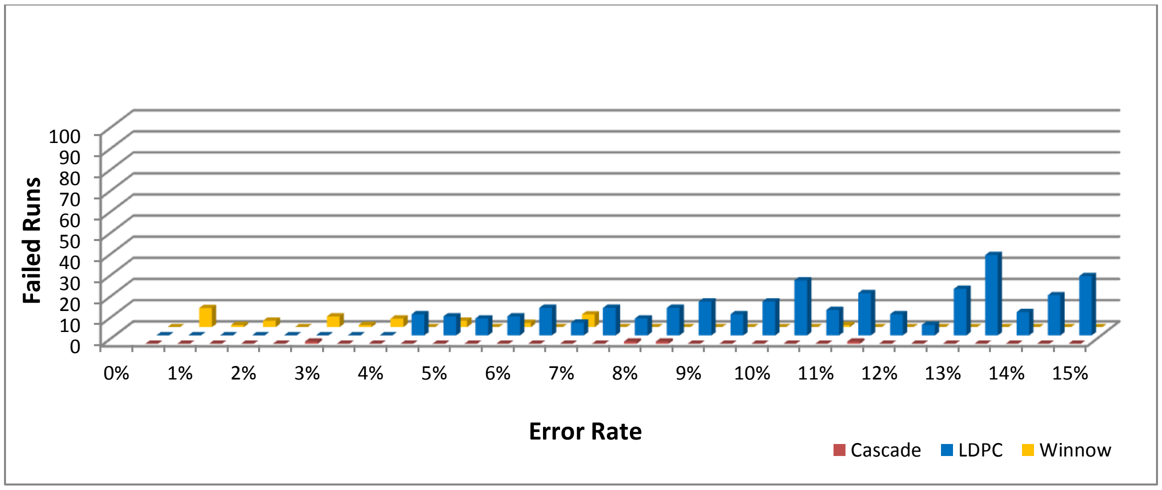


Figure 27. Error reconciliation results for one 50% burst and 50% uniform

The Cascade protocol was able to tolerate most conditions with little effect on overall performance, with the largest amount of failed runs occurring when five equally sized bursts were introduced. The average number of failed runs for Cascade was far less than Winnow however, which suffered catastrophic failure for a single large burst and for multiple smaller bursts. The fact that no errors are ever introduced in the Cascade algorithm and a permutation is performed between iterations contributes to Cascade's ability to withstand burst errors well. Winnow introduces errors when there is more than one error per block, so in all likelihood by the second pass, even with a permutation between passes, there were simply too many errors to correct. LDPC performed increasingly badly as the size of the burst(s) increased, but in general this performance was worse than Cascade, yet better than Winnow.

The third case, where a single 50% burst was introduced with the rest of the errors distributed uniformly elicited different behavior from the Winnow algorithm. Cascade still had very few failed runs, and seemed mostly unaffected by the single burst. LDPC also performed as before, with the number of errors increasing as the size of the burst increased. Somewhat unexpectedly however, Winnow performed well, with no error rate inducing more than 10 failures out of 100 runs. A possible reason for this is that the single burst caused Winnow to introduce additional errors into the key string; however the threshold of errors was not high enough that the protocol could not recover, and the permutation in between passes distributed the errors uniformly enough that all of the errors could be resolved.

After the data was collected for the different burst distributions, 100 runs were again performed using the same distributions; however this time an initial permutation was applied. A common seed was used across the three protocols to assure an identical permutation of the input string. With the initial permutation, all three protocols performed markedly better. In fact, only two of the 100 iteration runs resulted in any failed runs, and the number of failed runs was only 2 and 1 respectively. Therefore, if burst errors are probable in real world implementations, a simple permutation applied on the front end will allow any of the three protocols to operate normally.

In general, inaccurate error estimation and burst errors within a reasonable bound did not severely impact the performance of the three protocols. In the next experiment the effectiveness of the protocols for different key lengths as well as some potential performance enhancements will be considered.

4.4. Experiment 4: An analysis of key length and performance enhancements

Up until this point, the size of the raw key was a static parameter for all experiments. The purpose of this experiment was to examine the effect of a variable key length on the effectiveness, runtime and throughput of the various error reconciliation protocols. A static 5% error tolerance is presented, however results for all error tolerances and key lengths can be found in Appendix A.

Graphs of the maximum achieved error rates for all three protocols for key lengths of 1000 bits, 10 000 bits and 100 000 bits are presented in figures 28, 29 and 30. The x-axis represents the error rate, while the y-axis represents the reconciled key rate of the protocols.

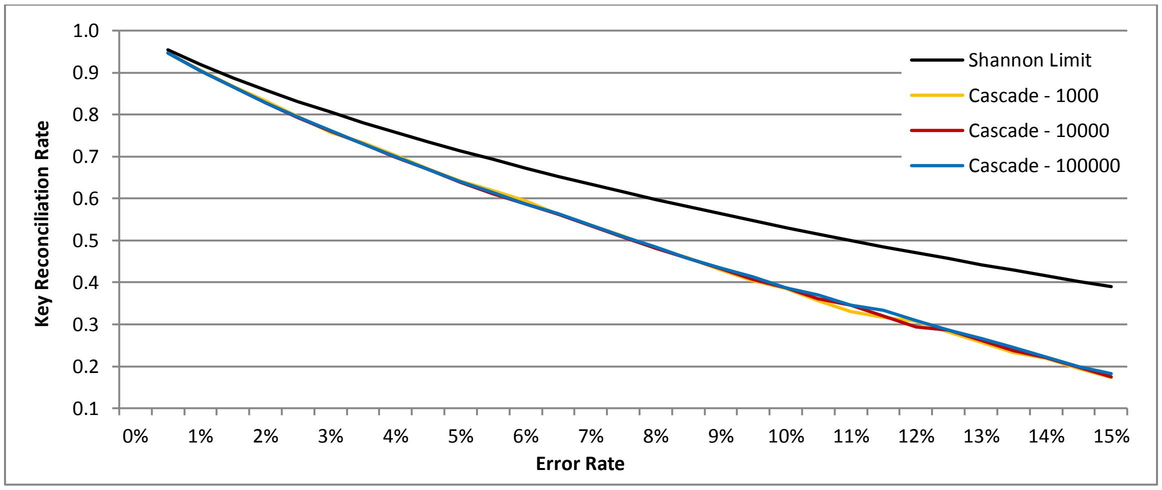


Figure 28. Achieved error rates for various key lengths – Cascade

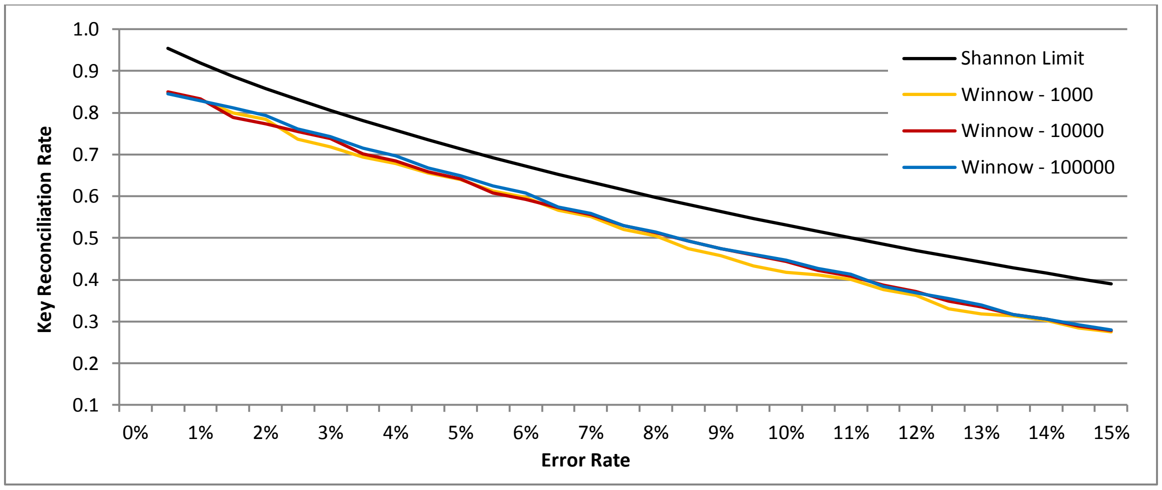


Figure 29. Achieved error rates for various key lengths - Winnow

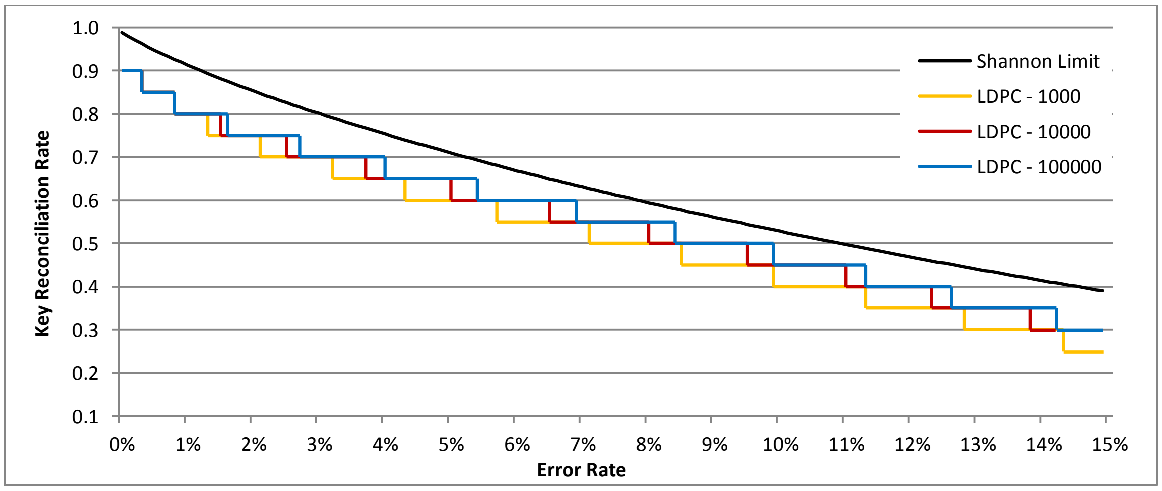


Figure 30. Achieved error rates for various key lengths - LDPC

As evident by the graph in figure 28, the effectiveness of the Cascade protocol remained remarkably stable for all three key lengths. Though larger key lengths permitted the use of a larger starting block size, the amount of information leaked and therefore the reconciled key rate fluctuated very little. In general, the Cascade algorithm appears to be very stable. As long as the algorithm is implemented properly and the initial block size chosen correctly, the performance is very predictable.

Similarly, the effectiveness of the Winnow protocol did not fluctuate significantly for increasing key length, though a slight increase was observed for 10 000 and 100 000 bit key lengths. The increase in effectiveness was expected due to the fact that the starting block size in Winnow is static and therefore the probability of multiple errors residing in any given block would decrease as the key length increases, however the increase in effectiveness was not as large as originally anticipated.

For the LDPC protocol, as expected, a large increase in the effectiveness was observed at each increase in the key length, and the most significant increase was noticed between a 1000 and 10 000 bit key length. The number of parity checks increases linearly with the key length in the LDPC protocol, however the number of bits per check remains the same. Therefore the result is more parity checks that are more widely spread out, and it becomes more unlikely that errors will be masked.

The same data is shown again in figures 31, 32 and 33 however this time the data is grouped by key length rather than protocol.

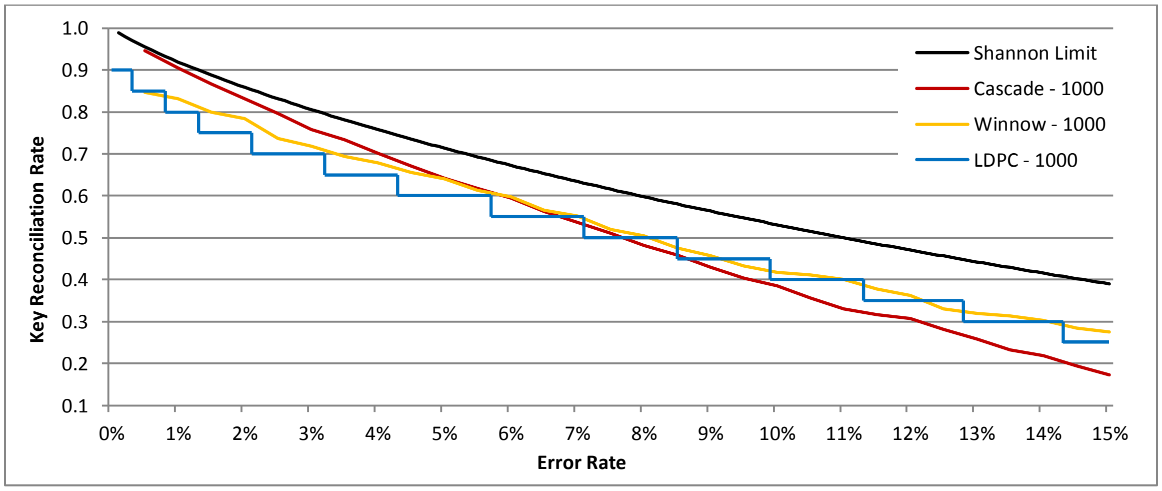


Figure 31. Achieved error rates for various key lengths - 1000 bit key

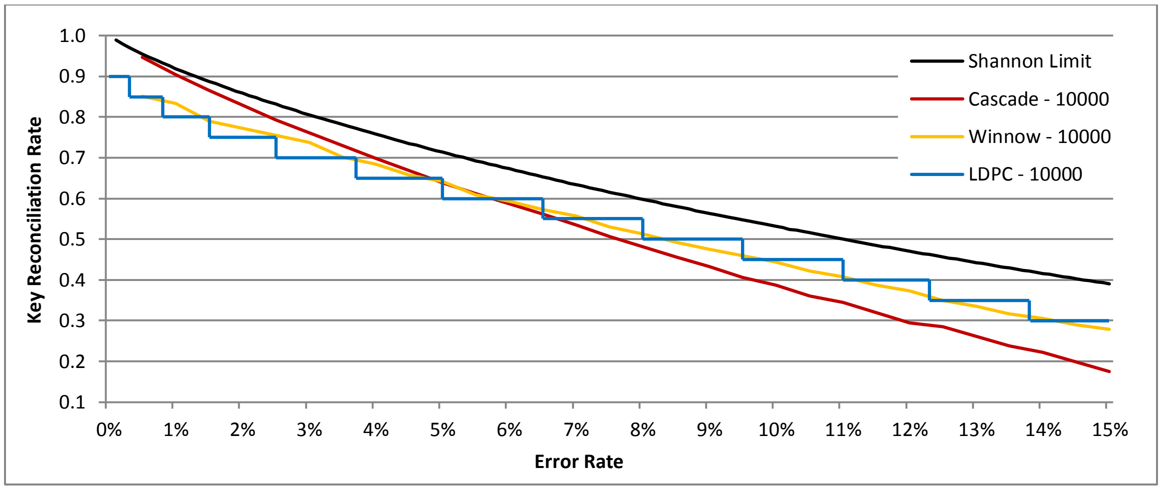


Figure 32. Achieved error rates for various key lengths - 10 000 bit key

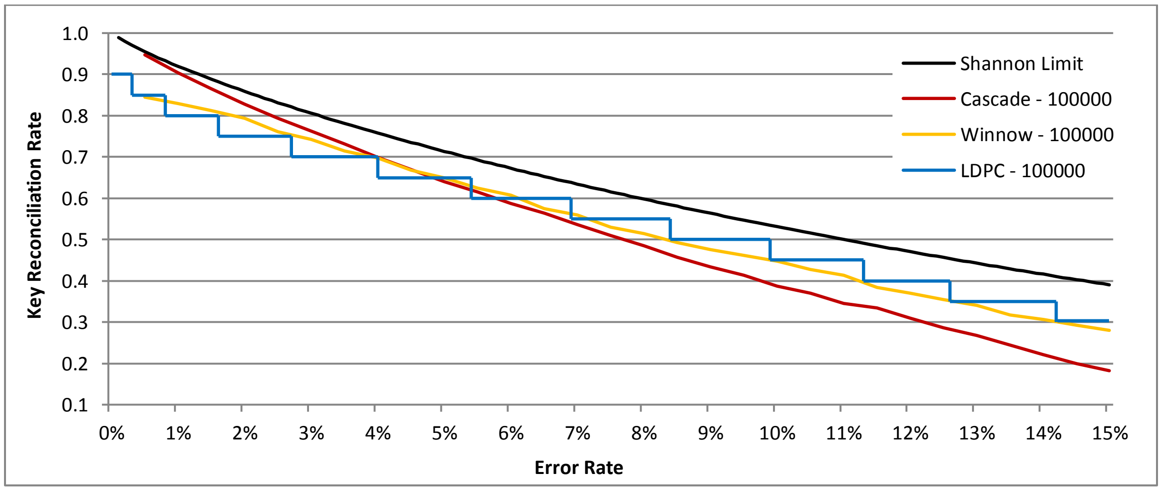


Figure 33. Achieved error rates for various key lengths 100 000 bit key

For all key lengths, it is evident that Cascade performs better than Winnow for error rates below 5%, and worse than Winnow for error rates above 5%. With respect to the theoretical bound, the effectiveness of the Cascade algorithm degrades as the error rate increases, while the effectiveness of the Winnow algorithm remains relatively static.

LDPC mirrors Winnow closely for a 1000 bit key length, though the step nature of the algorithm results in an overall lower average effectiveness. At a 10 000 bit key length, LDPC begins to outperform Winnow for error rates above 5%, and the effectiveness of LDPC progresses exceptionally close to the theoretical bound. At a 100 000 bit key length the results were similar, with the effectiveness of Cascade dominating for rates up to 5%, and LDPC performing the best from that point forward.

Figures 34, 35 and 36 illustrate the runtime of all three protocols. The units are once again in milliseconds.

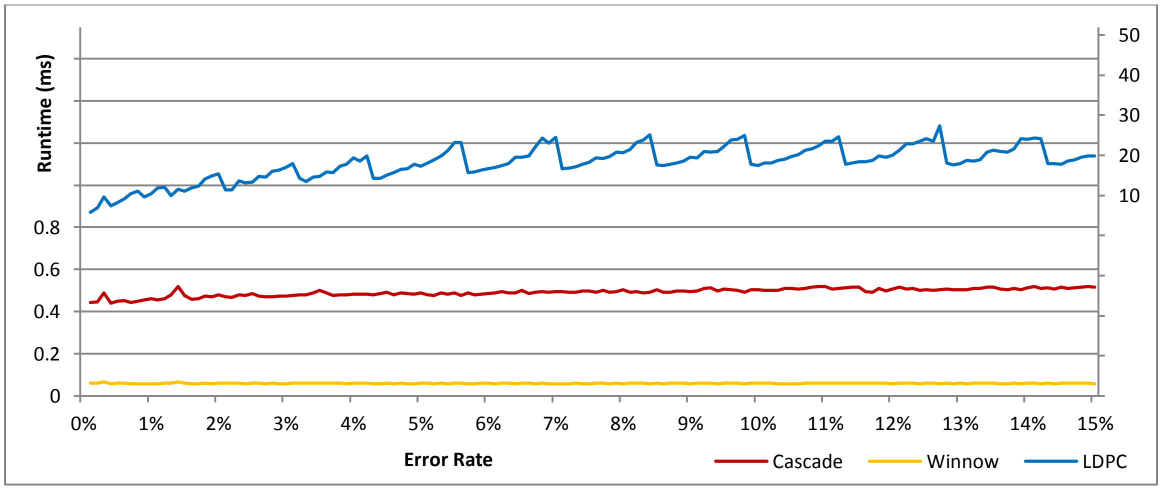


Figure 34. Runtime of Cascade, Winnow and LDPC - 1000 bit key

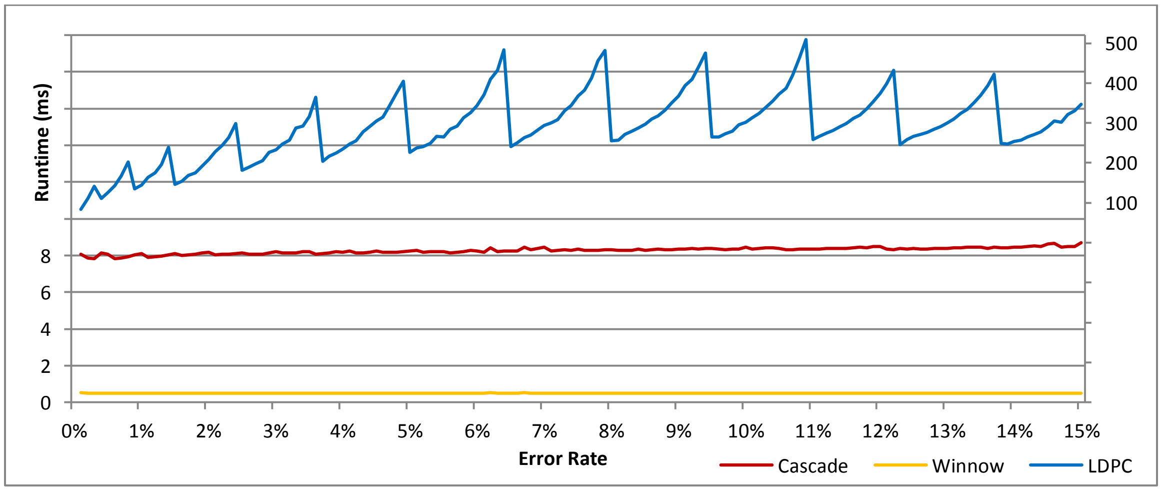


Figure 35. Runtime of Cascade, Winnow and LDPC - 10 000 bit key

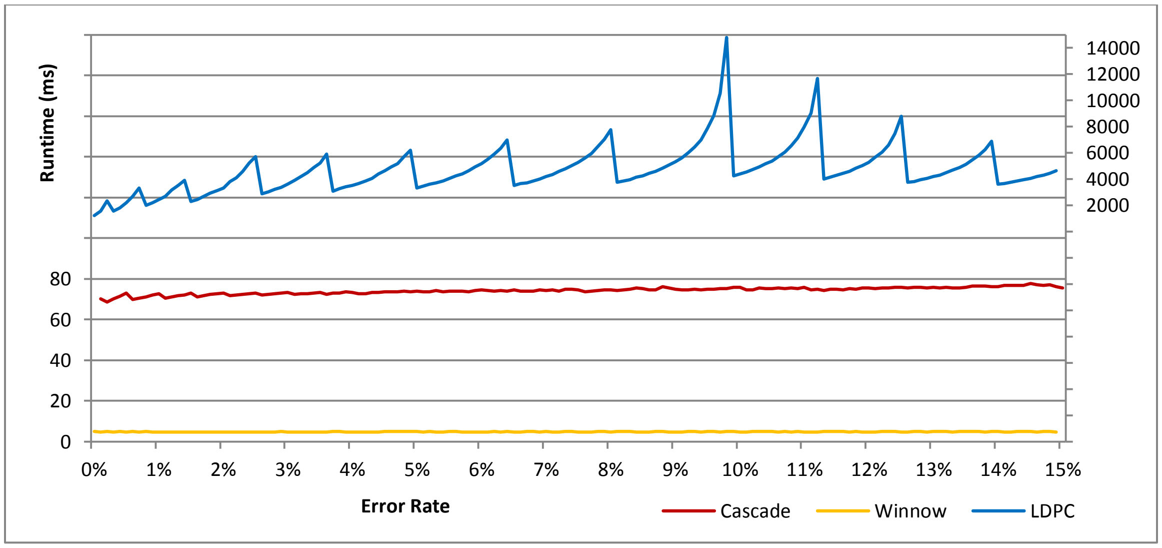


Figure 36. Runtime of Cascade, Winnow and LDPC – 100 000 bit key

Take note that once again the units for the LDPC line are represented by the right-hand side of the y-axis. The shape of the graphs are similar to the 100 000 bit runtime seen earlier, and the Winnow, Cascade and LDPC protocols show a near linear increase in runtime as the key length increases. For LDPC this linearity is significant, and is directly attributable to the sparse matrix storage utilized in the implementation. Otherwise, the size of the matrices required would scale quadratically with the key length, and therefore the runtime would as well.

Although the runtime of LDPC increases in a near-linear fashion, the shape of the graph indicates that as the key length increases, the runtime remains stable longer, with a sharper rise in runtime as individual codes near their decoding limit.

Generally the runtime is as expected, with Winnow exhibiting the best performance and LDPC presenting the worst. A more significant measure of performance is the overall throughput, which is displayed in figures 37, 38 and 39.

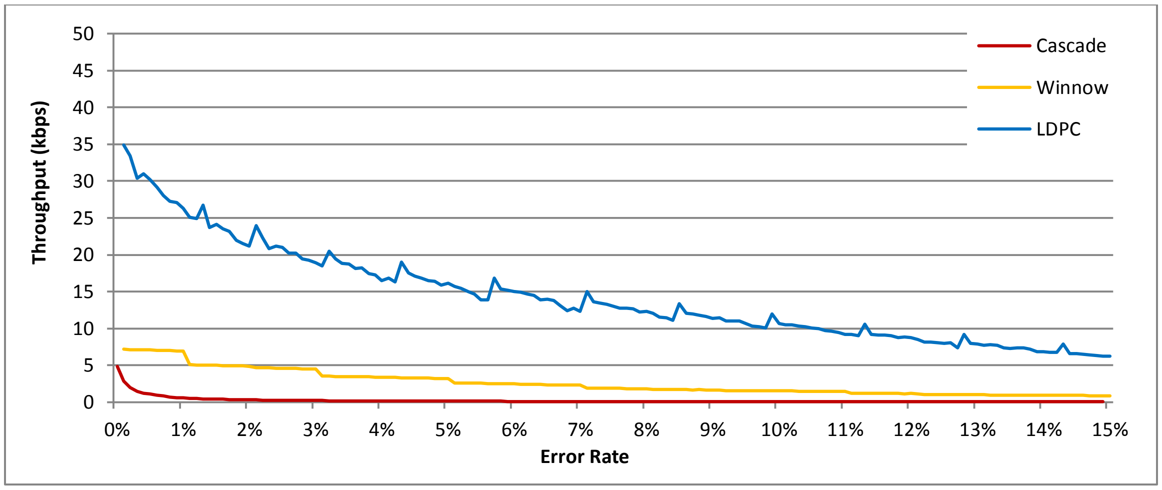


Figure 37. Throughput of Cascade, Winnow and LDPC - 1000 bit key

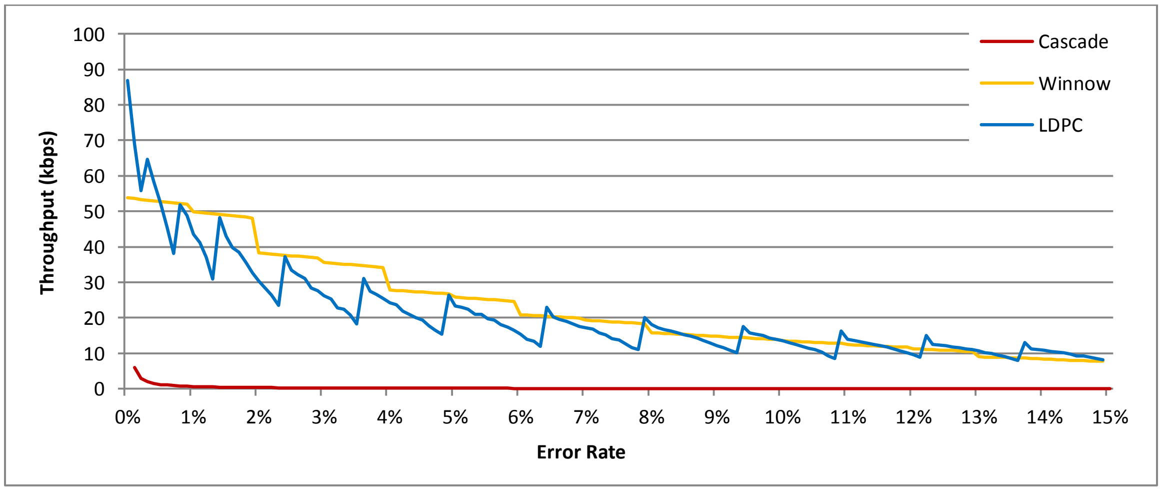


Figure 38. Throughput of Cascade, Winnow and LDPC - 10 000 bit key

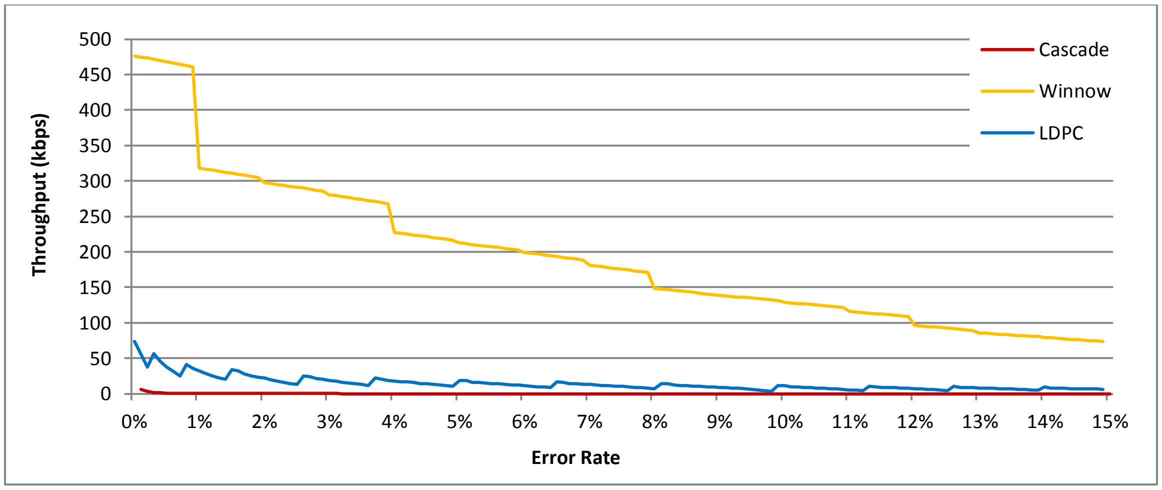


Figure 39. Throughput of Cascade, Winnow and LDPC - 100 000 bit key

While the throughput for the Cascade and LDPC algorithms fluctuates only marginally for varying key lengths, the throughput of the Winnow algorithm improves significantly as the key length is increased. The reason for this is that in Winnow the number of passes is generally the same, regardless of the key length. Therefore, although the number of parity checks increases, the amount of communication does not, and the network latency becomes less of a factor as the number of bits reconciled grows. The number of messages passed increases dramatically for Cascade with increases in key length, therefore even though more bits are produced, the throughput remains the same. In LDPC only one message is ever passed, so the throughput is mostly reliant on runtime and bits reconciled. Since both of these parameters increase with increases in key length, the throughput is mostly unchanged.

In a real world implementation, it is reasonable to presume that different length keys may be of use, depending on the required level of security, the symmetric key algorithm the key is intended for, or the desired throughput. Based on the data presented here, the Winnow protocol offers the best performance in terms of throughput at the 100 000 bit level, and there is reason to believe that performance would increase for larger key sizes. Effectiveness remained static for key lengths larger than 1000 bits; therefore the largest feasible key length should be used in the Winnow protocol.

In the Cascade protocol, the effectiveness and throughput remained relatively constant regardless of the key size; consequently the length of the key that best suits the application should be used. Cascade, although slow, is a very robust protocol that is easily adaptable to different key lengths.

For LDPC, roughly the same effectiveness and throughput is seen at the 10 000 and 100 000 bit levels. Therefore it is likely that a 10 000 bit key length would be more advantageous, since partial keys may be useful, and any failed runs would be contained in a 10 000 bit block. Additionally, multiple instances of the 10 000 bit implementation could be performed in parallel in order to increase throughput.

To illustrate this, one final experiment was performed, where the previously developed LDPC implementation was modified so that 10 instances could run simultaneously using software multithreading. The same input strings were used, however 10 LDPC threads were created and each thread was given $\frac{1}{10}$ of a 100 000 bit key string, a corresponding syndrome for that string, and an accurate estimate of the overall error rate. 10 threads were chosen since 10 000 bit LDPC parity check matrices had already been developed for the previous experiment. The resulting throughput for this implementation is shown in figure 40, along with the previously obtained value for a 100 000 bit key string.

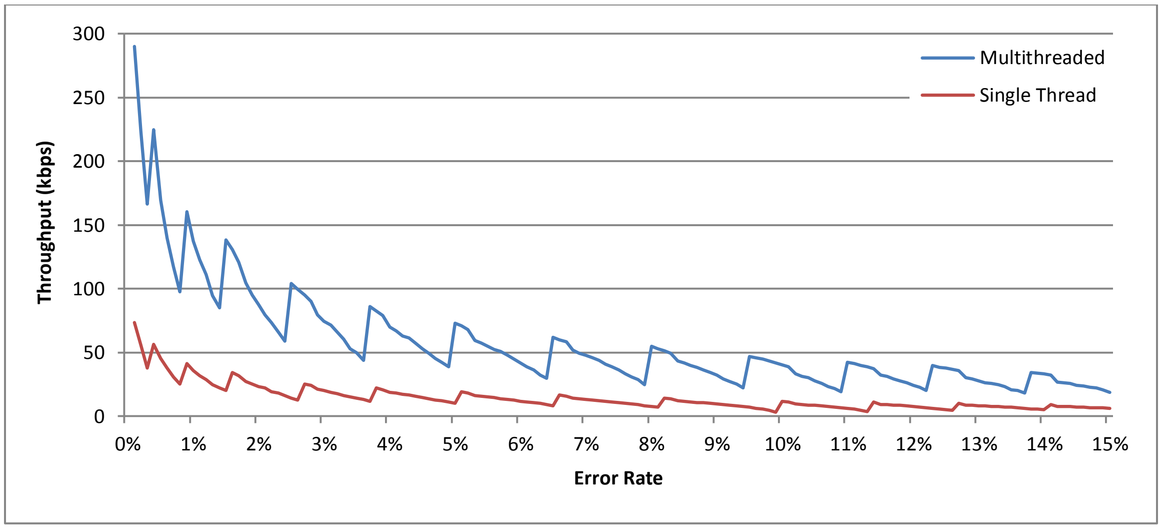


Figure 40. Throughput of a multi-threaded implementation of the LDPC protocol

Clearly, the LDPC protocol benefits significantly from a multithreaded implementation, and the throughput achieved showed an average increase of 300% (14 kbps vs. 56 kbps average). However, this increase came with a slight rise in the amount of failed runs. The reason for this may be that the overall error rate for a 100 000 bit string was not representative of the error rate for all of the individual pieces, depending on the error distribution. Still, the failed runs were contained to one or two of the 10 000 bit pieces, which means that 80-90% of the key string was decoded correctly. A similar approach could be taken in order to parallelize the larger, 100 000 bit implementation in order to produce an exceptionally large reconciled key.

Finally, with regard to communication efficiency, the LDPC protocol easily offers the best performance regardless of key size. The minimum amount of messages exchanged in the Cascade protocol for this experiment was 10, for a 1000 bit block size and a 0.1% error rate. For Winnow, the minimum was 6, also for a 0.1% error rate at a 1000 bit block size. However, for a reasonable 5% error rate and a 100 000 bit block size, the number of messages balloons to 26 217 for Cascade and 14 for Winnow. LDPC requires only one message exchange, regardless of error rate or key size.

V. Conclusions and Recommendations

5.1. Conclusions

Many present cryptographic implementations in use today utilize symmetric key ciphers, which offer efficiency as well as a high level of security. Unfortunately, symmetric key ciphers require the users to share a common key, so often less efficient asymmetric key ciphers are used in order to distribute a key for a symmetric cipher. Additionally, asymmetric key ciphers rely on problems which are thought to be, yet not proven to be computationally difficult, and therefore the level of their security is frequently questioned. Should quantum computers ever be realized, most asymmetric key algorithms will be rendered breakable in polynomial time.

Quantum Key Distribution offers an alternative for distributing a key for a symmetric cipher in an unconditionally secure way. However, the nature of the equipment, the medium, or the presence of eavesdroppers can introduce errors into the quantum transmission, which is destructive to any symmetric key cipher. Therefore, error reconciliation protocols have been developed that allow the resolution of errors without severely compromising security.

The most famous of these error reconciliation protocols was developed in 1994 and is known as Cascade. Although extremely efficient and capable of correcting large percentages of errors, Cascade is a highly interactive protocol that requires a large amount of communication and therefore suffers greatly in practical parameters such as throughput. In 2003 a new protocol was introduced based on Hamming codes which aimed to reduce the amount of communication needed. This protocol, known as Winnow, offers significantly reduced interaction when compared to Cascade with similar complexity and for comparable error rates. However Winnow is still an interactive protocol, and requires a sender and receiver to perform actions in a synchronized fashion. Additionally, in Winnow there is the possibility of introducing new errors, which leads to an increased probability of failure.

More recently, Low Density Parity Check Codes have been used in the error reconciliation phase of QKD. Developed by Gallager in the 1960's, LDPC codes went mostly ignored until McKay revived interest in them with his 1999 paper. LDPC codes offer a method of error correction that is not interactive, and only requires one initial communication. Moreover, LDPC codes are capable of correcting errors at rates that exceed that of Cascade and Winnow.

LDPC codes have traditionally been applied as forward error correcting codes in classical communications, and are now included as an option in the second generation digital video broadcasting and 802.11n Wi-Fi standards. Their application in QKD systems has been studied as well, though the lack of ideal codes and the high complexity of the decoding algorithms have prevented LDPC codes from achieving widespread adoption over Cascade and Winnow. Therefore, the purpose of this research was to evaluate the performance of LDPC codes generated using the Progressive Edge Growth algorithm, which is known to produce good codes, and decoded using the author's implementation of the Sum Product algorithm. This performance was compared empirically with that of Cascade and Winnow in order to determine if LDPC is truly viable for use in the error reconciliation phase of QKD.

The effectiveness of an error reconciliation protocol is a crucial parameter, since a protocol is not useful if it cannot successfully correct all the errors in a distributed key. The results obtained here show that for error rates up to 5% Cascade is highly effective and bests both of the other protocols, although Winnow and LDPC also perform well. Above the 5% threshold, the LDPC protocol performs extremely close to the theoretical bound, while the effectiveness of Winnow neither improves nor degrades. Cascade diverges significantly from the theoretical bound for higher error rates, and in general is not as effective as Winnow or LDPC beyond 5%.

Just as critical are the runtime and throughput of the protocols, since an effective protocol is unlikely to be used if it is prohibitively slow. It is in this area that Cascade suffered the most in this research, since its highly interactive nature makes it exceedingly dependent on network performance. For anything other than extremely low error rates (< 1%), the runtime and throughput of the Cascade protocol was shown to be vastly inferior to the Winnow and LDPC protocols.

The runtime of the Winnow protocol increases linearly for linear increases in key size, and as the key size grows the effects of the limited amount of interactions becomes negligible. In fact, of the three protocols, experiments performed here have shown that Winnow produces the highest average throughput for key sizes in excess of 10 000. At 10 000 bits the performance was comparable to the LDPC protocol, and below the 10 000 bit level the LDPC protocol was superior.

Due to some clever programming, the runtime of the Sum Product decoding algorithm for the LDPC codes was also shown to scale linearly for linear increases in the raw key size. However, since the runtime was still considerably higher than the Cascade and Winnow protocols, the overall throughput remained relatively static at a level in between Cascade and Winnow. Fortunately, since the amount of communication required by the LDPC protocol is extremely low, the algorithm was shown to be very well suited for enhancements in computational efficiency, such as multithreading, which resulted in a 300% increase in throughput. These improvements could also be implemented on the Cascade and Winnow protocols, but with less significant gains due to their interactive nature and hence their dependence on a network.

In order to perform error reconciliation in QKD protocols, an estimate of the number of errors in the key to be reconciled must be obtained, and the quality of this estimate will have an impact on the performance of the reconciliation algorithm. Experiments conducted showed that in the case of Cascade, the amount of information exchanged in order to reconcile a key with an inaccurate error estimate varies depending on the actual error percentage and the level of inaccuracy. At lower actual error rates an inaccurate estimate was demonstrated to be more costly, since the starting block size of the Cascade algorithm is determined based on the error estimate, and at lower error rates the steps between block sizes are more severe.

In Winnow the amount of interaction is determined by a block size schedule. Therefore the number of messages exchanged will not increase until the error estimate is sufficiently far from the true value that it triggers a switch to a new schedule. As implemented for this research the steps between block schedules tended to be large enough and the differences between schedules small enough that the increase in the amount of information exchanged for inaccurate estimates was minimal. Still, estimates that fluctuated significantly from the true value resulted in sizeable gains in information exchanged, and in general point to an error estimation threshold of +1.5% for the Winnow protocol.

The information exchanged in the LDPC protocol is constant for a given code rate, since the only information exchanged is the syndrome for the key string. Therefore, as long as the error estimate is not severe enough to cause a less efficient code to be used, the information will remain the same. If the wrong code is used however, the information exchanged will increase by the difference between the syndrome sizes of the two parity check matrices for the different code rates, which was 5000 bits for a 100 000 bit key length in the research conducted here. Additionally, the steps between code rates can be as small or as large as desired, with large steps resulting in more drastic differences between the lengths of the syndromes.

Finally, in a practical application it is unlikely that all of the errors in a given key string will be uniformly distributed, therefore the effect of non-uniform or burst errors was examined. Consecutive or burst errors were shown to be devastating to the Winnow protocol, which tends to introduce errors which are not adequately spaced. Cascade responded to burst errors well, due to the fact that no errors are ever introduced and a permutation is applied between passes. LDPC was neither tolerant of nor devastated by burst errors, and the probability of decoding failure increased steadily as the size of the burst grew. Regardless of the burst distribution, in nearly all cases a simple random permutation of the key string before error reconciliation begins was sufficient to distribute the errors in a uniform fashion.

The best error reconciliation method for a given QKD implementation is going to depend on many factors including but certainly not limited to security, ease of communication and error rate of the transmission. In general all three protocols examined here (Cascade, Winnow and LDPC) showed respective strengths and weaknesses and are suitable for the error reconciliation phase of Quantum Key Distribution. However, as reflected in the research results presented here, the LDPC protocol effectiveness is, on average, better than that of Cascade and Winnow, all the while requiring only a single message exchange. These traits, along with the fact that the throughput of the protocol can be significantly improved by parallelization or advances in hardware, makes the LDPC protocol the best choice for most applications.

5.2. Limitations

Several limitations are present in the research conducted here which are worth noting. The Mersenne Twister random number generation algorithm is not considered cryptographically secure, and although the output is considered to have very good random properties, it should not be used in a real-world implementation. Similarly, all achieved error rates were measured using the same set of 1000 randomly generated key strings. This was done for symmetry when comparing protocols, so an implementation utilizing different message generation techniques may result in slightly different maximum error rates.

5.3. Recommendations for Future Research

1. *Alternate Low Density Parity Check Codes*

Although the codes examined here exhibited performance exceptionally close to the theoretical limit, due to time constraints the interval between codes was left somewhat large, and all codes were generated using the PEG algorithm with a single degree distribution. The performance of this implementation of the Sum Product algorithm with different codes, more codes or perhaps even a single code that is modified for different error rates would be useful, and may result in better performance.

2. A Deeper Investigation into Performance Enhancements

Multithreading was only casually examined in this research; however the results demonstrate that a multithreaded implementation of the LDPC protocol could achieve much higher throughput. Specifically, implementing the protocol on an FPGA utilizing multiple embedded cores or a module designed specifically to efficiently evaluate hyperbolic tangent functions could result in markedly better performance. Since the LDPC protocol is complex computationally, the use of a dedicated processor is recommended.

3. A more realistic measure of network performance

The implementations studied here treated network performance as an ideal, static variable in order to assure uniformity when comparing the parameters between protocols. A study that examined the performance of the various protocols with respect to actual network performance could provide a better picture of realistic throughput rates. Additionally, such a comparison may reveal other difficulties not considered here, such as problems with variable delays in message transmission, and complications associated with keeping a sender and receiver in sync throughout the error reconciliation process.

4. Comparison with other protocols

Though Winnow and Cascade are two of the most well-known protocols for QKD error reconciliation, many others do exist. Turbo codes, for instance, are very closely related to LDPC codes and have been shown to achieve similar performance. A similar comparison would contribute to the overall characterization of QKD reconciliation protocols, so that the best protocol for a given situation can be selected.

VI. Appendix A Experimental Data

Contained in the tables below are comprehensive results gathered for protocol testing experiments.

- Cascade
    - Block size – Maximum starting block size achieved for a given error tolerance
    - Error rate – Error rate for a given set of 1000 runs
    - Average/Min/Max – Average, minimum and maximum bits exposed for a given set of 1000 runs
- LDPC
    - Code rate – the rate of the code for a set of 1000 runs, defined by $\mathbf{1-\frac{m}{n}}$
    - Error rate – Maximum error rate achieved for a given error tolerance
    - Average/Min/Max – Average, minimum and maximum number of iterations for a given set of 1000 runs
- Winnow
    - Error rate – Error Rate for a given set of 1000 runs
    - Average/Min/Max – Average, minimum and maximum bits exposed for a given set of 1000 runs

* Optimal block schedules for Winnow are contained in the source code in appendix B

1000 Bit Key Length, 0% Error Tolerance

Cascade				
Block size	Error Rate	Average	Min	Max
25	0.005	99	95	104
23	0.010	130	124	163
19	0.015	169	161	187
16	0.020	203	198	232
12	0.025	255	245	287
11	0.030	289	279	304
11	0.035	306	292	333
12	0.040	318	305	349
11	0.045	348	333	390
9	0.050	387	369	421
10	0.055	397	378	443
6	0.060	489	472	528
7	0.065	475	459	507
9	0.070	467	446	529
7	0.075	510	491	563
7	0.080	529	507	565
8	0.085	533	513	583
6	0.090	591	566	643
5	0.095	636	614	668
5	0.100	651	628	700
4	0.105	711	690	744
4	0.110	724	706	758
6	0.115	679	643	737
4	0.120	750	728	790
5	0.125	731	703	785
4	0.130	777	758	826
5	0.135	764	726	830
4	0.140	781	782	856
4	0.145	802	796	858
4	0.150	817	808	878

LDPC				
Code Rate	Error Rate	Average	Min	Max
0.90	0.001	2	1	3
0.85	0.002	2	1	4
0.80	0.003	3	2	7
0.75	0.011	6	4	15
0.70	0.016	6	4	20
0.65	0.023	7	5	17
0.60	0.035	8	6	23
0.55	0.037	7	6	14
0.50	0.063	11	8	21
0.45	0.069	10	8	16
0.40	0.076	9	7	13
0.35	0.113	14	11	32
0.30	0.127	14	11	27
0.25	0.143	14	11	35
0.20	0.156	13	11	26
0.15	0.174	14	11	123
0.10	0.179	11	10	21

Winnow			
Error Rate	Average	Min	Max
0.005	185	180	199
0.010	199	193	213
0.015	269	260	283
0.020	282	274	308
0.025	343	334	370
0.030	359	345	383
0.035	395	384	416
0.040	407	392	430
0.045	413	393	434
0.050	425	409	448
0.055	459	439	497
0.060	470	447	496
0.065	493	470	534
0.070	504	480	544
0.075	527	501	567
0.080	539	511	578
0.085	554	524	598
0.090	567	526	609
0.095	604	571	642
0.100	614	583	667
0.105	634	602	682
0.110	643	607	691
0.115	669	636	715
0.120	681	644	734
0.125	720	690	761
0.130	728	692	764
0.135	732	698	777
0.140	740	706	784
0.145	765	721	821
0.150	774	725	836

10 000 Bit Key Length, 0% Error Tolerance

Cascade				
Block size	Error Rate	Average	Min	Max
76	0.005	579	564	625
46	0.010	1006	981	1067
35	0.015	1374	1343	1446
27	0.020	1744	1708	1817
27	0.025	2035	1974	2152
22	0.030	2376	2316	2497
16	0.035	2728	2684	2858
17	0.040	2970	2893	3118
15	0.045	3260	3187	3374
14	0.050	3546	3456	3658
11	0.055	3906	3826	4038
11	0.060	4136	4040	4264
10	0.065	4403	4321	4543
9	0.070	4667	4590	4766
9	0.075	4884	4791	5048
8	0.080	5122	5045	5245
9	0.085	5335	5211	5480
7	0.090	5662	5564	5777
8	0.095	5752	5628	5910
5	0.100	6492	6423	6590
6	0.105	6396	6286	6525
6	0.110	6579	6473	6714
6	0.115	6764	6645	6912
6	0.120	6956	6823	7127
5	0.125	7286	7167	7412
5	0.130	7452	7331	7597
4	0.135	7867	7796	7978
5	0.140	7794	7654	7994
5	0.145	7970	7833	8151
5	0.150	8148	7986	8317

LDPC				
Code Rate	Error Rate	Average	Min	Max
0.90	0.001	4	3	29
0.85	0.002	4	3	8
0.80	0.005	5	4	15
0.75	0.01	7	6	13
0.70	0.015	8	6	14
0.65	0.029	11	9	18
0.60	0.051	19	14	69
0.55	0.067	21	17	31
0.50	0.091	33	27	49
0.45	0.106	33	28	47
0.40	0.119	29	25	47
0.35	0.135	30	24	78
0.30	0.149	27	22	59
0.25	0.164	26	21	54
0.20	0.179	25	21	67
0.15	0.193	23	19	42
0.10	0.21	23	19	68

Winnow			
Error Rate	Average	Min	Max
0.005	1742	1734	1762
0.010	1892	1877	1949
0.015	2357	2334	2397
0.020	2511	2476	2553
0.025	2791	2754	2838
0.030	2938	2886	3000
0.035	3224	3161	3286
0.040	3368	3304	3454
0.045	3593	3514	3700
0.050	3744	3663	3858
0.055	4062	3955	4187
0.060	4215	4101	4330
0.065	4500	4374	4658
0.070	4659	4518	4844
0.075	5006	4905	5106
0.080	5130	5035	5248
0.085	5228	5112	5375
0.090	5355	5211	5489
0.095	5574	5428	5749
0.100	5702	5529	5880
0.105	5956	5810	6155
0.110	6082	5917	6324
0.115	6427	6275	6580
0.120	6544	6349	6784
0.125	6685	6556	6845
0.130	6785	6607	6962
0.135	6985	6838	7158
0.140	7086	6933	7319
0.145	7280	7094	7505
0.150	7390	7225	7662

100 000 Bit Key Length, 0% Error Tolerance

Cascade				
Block Size	Error Rate	Average	Min	Max
81	0.005	5689	5634	5786
57	0.010	9627	9517	9798
45	0.015	13385	13215	13642
32	0.020	16906	16707	17159
27	0.025	20358	20145	20681
24	0.030	23619	23924	23347
22	0.035	26728	26342	27142
19	0.040	29672	29290	30093
13	0.045	33377	33142	33699
14	0.050	35440	35167	35875
12	0.055	38582	38323	38899
11	0.060	41335	41012	41705
11	0.065	43712	43317	44176
10	0.070	46314	45908	46707
10	0.075	48661	48251	49069
9	0.080	51022	50542	51436
9	0.085	53309	52906	53803
10	0.090	56115	55586	56866
8	0.095	57511	57097	58015
7	0.100	60481	60126	60839
7	0.105	63921	63568	64342
6	0.110	65740	65316	66079
6	0.115	67602	67165	68054
5	0.120	71201	70890	71625
6	0.125	71444	70942	71978
6	0.130	73414	72904	73996
5	0.135	76198	75819	76650
6	0.140	77926	77342	78449
6	0.145	79685	80237	79202
6	0.150	81463	80912	81980

LDPC				
Code Rate	Error Rate	Average	Min	Max
0.90	0.001	5	4	11
0.85	0.002	5	4	9
0.80	0.006	8	7	14
0.75	0.17	16	12	61
0.70	0.025	16	14	103
0.65	0.037	19	16	33
0.60	0.057	28	24	46
0.55	0.081	53	45	70
0.50	0.098	71	56	113
0.45	0.112	60	51	108
0.40	0.125	49	42	74
0.35	0.141	51	42	94
0.30	0.156	44	36	58
0.25	0.171	41	34	63
0.20	0.186	42	34	81
0.15	0.201	38	32	67
0.10	0.218	39	31	95

Winnow			
Error Rate	Average	Min	Max
0.005	15957	15920	16015
0.010	17538	17432	17650
0.015	20673	20572	20810
0.020	22290	22086	22515
0.025	26115	25930	26317
0.030	27757	27555	27983
0.035	31071	30815	31368
0.040	32655	32356	33012
0.045	34808	34523	35124
0.050	36412	35996	36757
0.055	40784	40448	41163
0.060	42300	41937	42718
0.065	44210	43782	44726
0.070	45814	45241	46289
0.075	49278	48849	49602
0.080	50506	50054	50940
0.085	52169	51769	52602
0.090	53425	52965	53955
0.095	54521	53996	55043
0.100	55847	55266	56386
0.105	59376	58854	59944
0.110	60610	60047	61170
0.115	63608	63025	64367
0.120	64841	64329	65472
0.125	66267	65650	66967
0.130	67551	66920	68302
0.135	68773	68215	69355
0.140	69890	69228	70594
0.145	72285	71670	72863
0.150	73369	72751	74093

1000 Bit Key Length, 5% Error Tolerance

Cascade				
Block size	Error Rate	Average	Min	Max
101	0.005	54	27	92
71	0.010	95	54	134
55	0.015	133	89	170
44	0.020	168	129	220
38	0.025	204	149	250
31	0.030	242	183	292
29	0.035	267	232	327
24	0.040	298	257	349
23	0.045	329	294	395
20	0.050	358	321	422
18	0.055	382	347	455
16	0.060	406	375	475
16	0.065	437	385	514
15	0.070	463	430	517
14	0.075	490	450	564
13	0.080	518	480	580
12	0.085	542	497	615
11	0.090	570	529	659
11	0.095	597	550	664
10	0.100	615	570	683
10	0.105	644	590	730
10	0.110	670	619	772
9	0.115	683	635	760
8	0.120	693	649	759
8	0.125	718	675	806
8	0.130	742	693	823
8	0.135	767	714	851
7	0.140	781	727	846
7	0.145	805	754	871
7	0.150	827	777	909

LDPC				
Code Rate	Error Rate	Average	Min	Max
0.90	0.003	5	2	37
0.85	0.008	8	5	179
0.80	0.013	11	6	166
0.75	0.021	13	7	181
0.70	0.032	16	9	198
0.65	0.043	18	11	183
0.60	0.057	21	13	179
0.55	0.071	23	14	169
0.50	0.085	23	15	196
0.45	0.099	22	16	177
0.40	0.113	22	16	198
0.35	0.128	21	16	170
0.30	0.143	21	15	194
0.25	0.158	21	15	172
0.20	0.171	20	14	165
0.15	0.187	20	13	159
0.10	0.202	18	13	162

Winnow			
Error Rate	Average	Min	Max
0.005	153	147	171
0.010	169	162	186
0.015	200	191	221
0.020	216	201	242
0.025	263	252	288
0.030	281	266	309
0.035	306	290	337
0.040	321	300	364
0.045	344	320	387
0.050	360	333	399
0.055	387	360	433
0.060	403	368	475
0.065	434	401	491
0.070	448	413	495
0.075	480	443	558
0.080	495	454	569
0.085	525	480	596
0.090	543	489	620
0.095	567	514	640
0.100	582	528	647
0.105	588	544	646
0.110	599	552	662
0.115	623	576	697
0.120	638	586	728
0.125	669	621	753
0.130	681	624	771
0.135	687	641	745
0.140	697	651	758
0.145	715	670	796
0.150	725	670	793

10 000 Bit Key Length, 5% Error Tolerance

Cascade				
Block size	Error Rate	Average	Min	Max
199	0.005	530	476	607
104	0.010	963	897	1154
69	0.015	1349	1267	1487
53	0.020	1717	1617	1853
42	0.025	2069	1969	2215
35	0.030	2387	2284	2527
31	0.035	2703	2586	2840
26	0.040	3015	2893	3196
23	0.045	3316	3193	3473
21	0.050	3617	3477	3796
19	0.055	3882	3729	4054
17	0.060	4126	3980	4332
16	0.065	4381	4226	4548
15	0.070	4648	4496	4833
14	0.075	4926	4771	5205
13	0.080	5185	5022	5359
12	0.085	5434	5288	5626
11	0.090	5669	5466	5864
11	0.095	5937	5755	6123
10	0.100	6134	5919	6368
10	0.105	6393	6219	6662
9	0.110	6550	6346	6764
9	0.115	6804	6610	7061
9	0.120	7057	6862	7306
8	0.125	7144	6954	7360
8	0.130	7388	7196	7622
8	0.135	7625	7430	7854
7	0.140	7786	7561	7971
7	0.145	8016	7831	8231
7	0.150	8251	8029	8468

LDPC				
Code Rate	Error Rate	Average	Min	Max
0.90	0.003	9	5	32
0.85	0.008	15	9	118
0.80	0.015	21	11	145
0.75	0.025	28	15	182
0.70	0.037	34	20	195
0.65	0.050	38	24	197
0.60	0.065	44	27	176
0.55	0.080	46	32	137
0.50	0.095	47	34	194
0.45	0.110	46	32	165
0.40	0.123	39	30	185
0.35	0.138	37	28	178
0.30	0.154	37	26	173
0.25	0.169	35	25	161
0.20	0.183	33	24	170
0.15	0.199	33	23	143
0.10	0.215	31	22	153

Winnow			
Error Rate	Average	Min	Max
0.005	1496	1485	1523
0.010	1660	1634	1720
0.015	2111	2083	2168
0.020	2270	2234	2321
0.025	2447	2397	2525
0.030	2617	2544	2705
0.035	2992	2914	3078
0.040	3155	3072	3263
0.045	3422	3339	3532
0.050	3578	3494	3713
0.055	3917	3821	4074
0.060	4068	3961	4180
0.065	4270	4151	4409
0.070	4435	4290	4654
0.075	4703	4570	4870
0.080	4866	4723	5068
0.085	5075	4889	5310
0.090	5250	5034	5497
0.095	5403	5205	5614
0.100	5565	5376	5877
0.105	5779	5625	6004
0.110	5920	5705	6160
0.115	6126	5924	6347
0.120	6276	6060	6587
0.125	6504	6310	6763
0.130	6644	6412	6961
0.135	6836	6651	7081
0.140	6945	6789	7193
0.145	7102	6892	7350
0.150	7219	7015	7485

100 000 Bit Key Length, 5% Error Tolerance

Cascade				
Block Size	Error Rate	Average	Min	Max
210	0.005	5342	5164	5655
106	0.010	9639	9391	9975
69	0.015	13496	13188	13882
52	0.020	17167	16844	17564
41	0.025	20638	20282	21026
34	0.030	23795	23391	24182
29	0.035	26920	26508	27307
25	0.040	30063	29638	30578
22	0.045	33057	32624	33653
20	0.050	35936	35329	36502
18	0.055	38563	38122	39112
17	0.060	41281	40721	41842
15	0.065	43609	43042	44102
14	0.070	46370	45817	47013
13	0.075	49032	45477	49697
12	0.080	51571	51074	52204
12	0.085	54323	53601	54936
11	0.090	56602	56020	57147
10	0.095	58675	59251	58096
10	0.100	61286	60691	62034
9	0.105	62961	62423	63644
9	0.110	65473	64812	66147
8	0.115	66637	66047	67149
8	0.120	69010	68305	69573
8	0.125	71400	70744	72033
7	0.130	73284	72721	73986
7	0.135	75543	74967	76215
7	0.140	77821	77174	78631
7	0.145	80127	79481	80741
6	0.150	81698	80891	82397

LDPC				
Code Rate	Error Rate	Average	Min	Max
0.90	0.003	21	12	193
0.85	0.008	22	14	108
0.80	0.016	33	21	142
0.75	0.027	47	29	186
0.70	0.040	63	40	192
0.65	0.054	75	52	196
0.60	0.069	87	59	194
0.55	0.084	91	67	194
0.50	0.099	92	69	199
0.45	0.113	76	57	167
0.40	0.126	58	46	126
0.35	0.142	63	45	184
0.30	0.158	60	45	156
0.25	0.173	56	41	194
0.20	0.187	49	36	137
0.15	0.203	53	37	166
0.10	0.219	46	32	105

Winnow			
Error Rate	Average	Min	Max
0.005	15471	15424	15542
0.010	17095	16961	17225
0.015	18768	18555	19023
0.020	20657	20287	21064
0.025	23945	23734	24206
0.030	25695	25434	25972
0.035	28486	28186	28837
0.040	30295	29943	30740
0.045	33199	32812	33611
0.050	34998	34408	35492
0.055	37566	37162	38020
0.060	39246	38757	39755
0.065	42510	42102	43005
0.070	44117	43639	44675
0.075	46974	46449	47411
0.080	48614	48083	49217
0.085	50715	50035	51402
0.090	52482	51674	53537
0.095	53886	53341	54421
0.100	55271	54659	55920
0.105	57217	56567	57895
0.110	58647	58049	59511
0.115	61618	61016	62392
0.120	63070	62294	63910
0.125	64531	63791	65145
0.130	66007	65124	66926
0.135	68304	67812	69477
0.140	69417	67716	69018
0.145	70808	68752	70196
0.150	71988	71233	72996

1000 Bit Key Length, 10% Error Tolerance

Cascade				
Block size	Error Rate	Average	Min	Max
123	0.005	54	25	77
87	0.010	92	49	130
61	0.015	132	79	174
52	0.020	168	100	228
44	0.025	203	133	255
35	0.030	243	175	291
32	0.035	266	217	319
28	0.040	299	254	356
25	0.045	330	256	404
22	0.050	361	317	430
21	0.055	389	309	453
19	0.060	418	341	482
18	0.065	443	396	516
16	0.070	465	422	519
15	0.075	492	447	560
14	0.080	519	479	593
13	0.085	544	491	617
12	0.090	575	524	648
12	0.095	601	531	698
11	0.100	624	571	704
11	0.105	650	581	726
10	0.110	670	619	772
10	0.115	696	633	807
9	0.120	707	650	827
9	0.125	733	675	824
9	0.130	760	709	846
8	0.135	767	714	851
8	0.140	791	731	884
8	0.145	814	758	895
7	0.150	827	777	909

LDPC				
Code Rate	Error Rate	Average	Min	Max
0.900	0.004	6	3	170
0.850	0.009	11	6	145
0.800	0.015	13	7	156
0.750	0.023	15	8	173
0.700	0.033	17	9	175
0.650	0.046	21	11	193
0.600	0.059	23	14	193
0.550	0.073	24	15	175
0.500	0.087	24	16	141
0.450	0.101	24	16	141
0.400	0.116	24	17	199
0.350	0.130	23	17	188
0.300	0.145	22	15	198
0.250	0.160	21	15	198
0.200	0.173	20	14	181
0.150	0.189	21	14	190
0.100	0.205	20	14	170

Winnow			
Error Rate	Average	Min	Max
0.005	146	140	154
0.010	162	149	183
0.015	188	180	210
0.020	204	189	241
0.025	256	244	284
0.030	274	259	300
0.035	294	278	327
0.040	309	272	342
0.045	339	315	384
0.050	355	328	395
0.055	376	349	422
0.060	392	357	435
0.065	422	384	486
0.070	437	395	500
0.075	470	432	540
0.080	485	443	550
0.085	513	468	575
0.090	531	477	606
0.095	550	511	606
0.100	561	522	605
0.105	581	537	638
0.110	593	545	656
0.115	616	568	690
0.120	631	578	724
0.125	658	609	730
0.130	671	612	743
0.135	690	628	762
0.140	703	640	777
0.145	712	666	787
0.150	721	666	783

10 000 Bit Key Length, 10% Error Tolerance

Cascade				
Block size	Error Rate	Average	Min	Max
228	0.005	531	433	620
117	0.010	966	886	1094
76	0.015	1362	1268	1526
57	0.020	1724	1623	1904
45	0.025	2082	1968	2222
38	0.030	2414	2301	2588
32	0.035	2714	2582	2882
28	0.040	3028	2885	3221
25	0.045	3336	3197	3541
22	0.050	3632	3485	3782
20	0.055	3912	3767	4065
18	0.060	4162	4001	4383
17	0.065	4431	4273	4663
16	0.070	4681	4514	4878
15	0.075	4945	4793	5144
14	0.080	5215	5049	5390
13	0.085	5469	5290	5656
12	0.090	5712	5555	5934
11	0.095	5937	5755	6123
11	0.100	6208	6021	6423
10	0.105	6393	6219	6662
10	0.110	6658	6463	6879
9	0.115	6804	6610	7061
9	0.120	7057	6862	7306
9	0.125	7311	7083	7547
8	0.130	7388	7196	7622
8	0.135	7625	7430	7854
8	0.140	7870	7678	8124
7	0.145	8016	7831	8431
7	0.150	8251	8029	8468

LDPC				
Code Rate	Error Rate	Average	Min	Max
0.900	0.004	13	7	176
0.850	0.009	18	9	144
0.800	0.016	24	12	164
0.750	0.026	30	17	199
0.700	0.038	38	22	196
0.650	0.051	42	24	197
0.600	0.066	48	31	195
0.550	0.081	51	35	192
0.500	0.096	49	37	148
0.450	0.110	46	32	165
0.400	0.124	42	32	164
0.350	0.139	40	30	159
0.300	0.155	37	29	139
0.250	0.170	37	26	162
0.200	0.184	34	24	160
0.150	0.200	34	23	181
0.100	0.216	34	23	146

Winnow			
Error Rate	Average	Min	Max
0.005	1463	1451	1497
0.010	1633	1601	1705
0.015	1994	1957	2066
0.020	2164	2122	2241
0.025	2440	2390	2520
0.030	2610	2537	2702
0.035	2909	2831	3007
0.040	3076	2989	3193
0.045	3397	3313	3507
0.050	3555	3469	3671
0.055	3803	3688	3946
0.060	3965	3842	4107
0.065	4248	4128	4394
0.070	4415	4267	4622
0.075	4688	4554	4863
0.080	4850	4708	5020
0.085	5060	4869	5322
0.090	5240	5018	5495
0.095	5392	5186	5626
0.100	5559	5358	5901
0.105	5717	5558	5971
0.110	5867	5639	6108
0.115	6103	5901	6330
0.120	6254	6036	6528
0.125	6477	6281	6745
0.130	6617	6383	6924
0.135	6831	6644	7044
0.140	6940	6784	7162
0.145	7081	6869	7310
0.150	7201	6995	7482

100 000 Bit Key Length, 10% Error Tolerance

Cascade				
Block Size	Error Rate	Average	Min	Max
225	0.005	5358	5183	5578
111	0.010	9663	9362	9984
73	0.015	13594	13275	14002
54	0.020	17208	16857	17621
43	0.025	20736	20326	21208
36	0.030	23997	23636	24456
30	0.035	26946	26505	27376
27	0.040	30215	29663	30686
23	0.045	33186	32776	33731
21	0.050	36123	35621	36622
19	0.055	38842	38269	39333
17	0.060	41281	40721	41842
16	0.065	43825	43278	44485
15	0.070	46506	46014	47021
14	0.075	49232	58690	49829
13	0.080	51840	51278	52320
12	0.085	54323	53601	54936
11	0.090	56602	56020	57147
11	0.095	59298	58683	59849
10	0.100	61286	60691	62034
9	0.105	62961	62423	63644
9	0.110	65473	64812	66147
9	0.115	67978	67238	68693
8	0.120	69010	68305	69573
8	0.125	71400	70744	72033
8	0.130	73811	73161	74449
7	0.135	75543	74967	76215
7	0.140	77821	77174	78631
7	0.145	80127	79481	80741
7	0.150	82427	81699	83256

LDPC				
Code Rate	Error Rate	Average	Min	Max
0.900	0.004	21	12	193
0.850	0.009	28	16	161
0.800	0.017	41	22	190
0.750	0.280	58	35	195
0.700	0.041	75	48	197
0.650	0.054	75	52	196
0.600	0.069	87	59	194
0.550	0.084	91	67	194
0.500	0.990	92	69	199
0.450	0.113	76	57	167
0.400	0.127	72	52	180
0.350	0.142	63	45	184
0.300	0.158	60	45	156
0.250	0.173	56	41	194
0.200	0.188	59	41	192
0.150	0.203	53	37	166
0.100	0.220	55	37	188

Winnow			
Error Rate	Average	Min	Max
0.005	14973	14925	15046
0.010	16617	16478	16750
0.015	19143	18952	19373
0.020	20970	20639	21291
0.025	24383	24183	24607
0.030	26103	25867	26349
0.035	28416	28116	28773
0.040	30229	29878	30676
0.045	33837	33456	34294
0.050	35625	35116	36086
0.055	38071	37695	38492
0.060	39693	69265	40175
0.065	41859	41367	42442
0.070	43627	42954	44232
0.075	47084	46524	47651
0.080	48874	48263	49534
0.085	50673	50016	51368
0.090	52411	51622	53414
0.095	53537	52950	54096
0.100	55008	54352	55751
0.105	57175	56524	57854
0.110	58608	58008	59477
0.115	61163	60551	61949
0.120	62636	61849	63503
0.125	65056	64354	65609
0.130	66464	65637	67299
0.135	68765	67951	69535
0.140	70168	69332	71048
0.145	70753	70055	71501
0.150	71941	71176	72894

VII. Appendix B Source Code

Common Files:

```
//----------------- UNCLASSIFIED//FOR OFFICIAL USE ONLY ---------------------//
// Common.h
//
// Lt Jim Johnson
// Air Force Institute of Technology
//
// Created on: December, 2011
// Last Updated: January, 2012
//
// Description: A common file used to consolidate includes and global values.
//---------------------------------------------------------------------------//

#include <bitset>
#include <string>
#include <stdio.h>
#include <fstream>
#include <iostream>

#include "MersenneTwister.h"

#define messageSize 100000
#define zero_percent_tolerance
//#define five_percent_tolerance
//#define ten_percent_tolerance

typedef unsigned int uint;

using namespace std;

//----------------- UNCLASSIFIED//FOR OFFICIAL USE ONLY ---------------------//
```

```
//----------------- UNCLASSIFIED//FOR OFFICIAL USE ONLY ---------------------//
// Permutation.h
//
// Lt Jim Johnson
// Air Force Institute of Technology
//
// Created on: December, 2011
// Last Updated: January, 2012
//
// Description: A permutation class used to maintain an efficient method of
// permuting indicies while maintaining the ability to reverse the permutation.
//---------------------------------------------------------------------------//

#ifndef Permutation_h
#define Permutation_h

#include "Common.h"

class Permutation
{
        public:
                uint first;                     // permute first pass flag
                uint perm[messageSize];         // the permutation array
                uint reversePerm[messageSize];  // the reverse permutation array

                Permutation();

                // creates the permutation array
                void createPerm(uint seed);
                // returns a pseudo-random index from the permutation array
                uint getIndex(uint index, uint pass);

                // returns the original index from the reverse permutation array
                uint getReverseIndex(uint index, uint pass);
};

#endif

//----------------- UNCLASSIFIED//FOR OFFICIAL USE ONLY ---------------------//
```

```
//----------------- UNCLASSIFIED//FOR OFFICIAL USE ONLY ---------------------//
// Permutation.cpp
//
// Lt Jim Johnson
// Air Force Institute of Technology
//
// Created on: December, 2011
// Last Updated: January, 2012
//
// Description: A permutation class used to maintain an efficient method of
// permuting indicies while maintaining the ability to reverse the permutation.
//-------------------------------------------------------------------------//

#include "Permutation.h"

// Constructor
Permutation::Permutation()
{
        first = 0;

        for (int i = 0; i < messageSize; i++)
        {
                perm[i] = 0;
                reversePerm[i] = 0;
        }
}

// Input:
//              seed - a seed for the random number generator used to create the
//                     permutation array
// Purpose:
//              Creates an array of pseudo-random indices
void Permutation::createPerm(uint seed)
{
        MTRand random(seed); // create new RNG instance seeded with the seed
        bitset<messageSize> usedIndices; // tracks repeat values

        // generates random indices for the permutation array, taking care not to
        // repeat any values
        for(uint i = 0; i < messageSize; i++)
        {
                // generate a random index in the range of 0 to messageSize-1
                uint index = random.randInt(messageSize-1);

                // if we already generated this index, don't use it
                if (usedIndices[index] != 1)
                {
                        perm[i] = index;
                        reversePerm[index] = i;
                        usedIndices.set(index);
                }
                else i--; // decrement the counter if the index was a repeat
        }
}

// Input:
//              index - the original index
//              pass - the current pass
// Return value:
//              a random index from the permutation array
// Purpose:
//              Returns a random index by using the given index as a lookup into the
//              random integer array. Performs multiple lookups for higher pass values.
//              Note: if pass = 0, the original index is returned
uint Permutation::getIndex(uint index, uint pass)
{
        int newIndex = 0;
        int oldIndex = index;

        if (first == 1) pass++; // permutes the first pass if desired

        // no permutation for pass 0 if the first flag is not set
        if (pass == 0) return index;

        // Perform multiple lookups into the array for higher passes
        for (uint i = 0; i < pass; i++)
        {
                newIndex = perm[oldIndex];
                oldIndex = newIndex;
        }
        return newIndex;
}

// Input:
//              index - the permuted index
//              pass - the pass for the desired original index
// Return value:
//              the original index
// Purpose:
//              given a permuted index, a reverse lookup is performed in order to
//              retrieve the corresponding original index for the given pass
uint Permutation::getReverseIndex(uint index, uint pass)
{
        int newIndex = 0;
        int oldIndex = index;

        if (first == 1) pass++; // set if the first pass was permuted

        // if no initial permutation was applied and pass = 0, then the original
        // index is the same as the current index
        if (pass == 0) return index;

        // Perform multiple lookups into the array for higher passes
        for (uint i = 0; i < pass; i++)
        {
                newIndex = reversePerm[oldIndex];
                oldIndex = newIndex;
        }
        return newIndex;
}

//----------------- UNCLASSIFIED//FOR OFFICIAL USE ONLY ---------------------//
```

```
// MersenneTwister.h
// Mersenne Twister random number generator -- a C++ class MTRand
// Based on code by Makoto Matsumoto, Takuji Nishimura, and Shawn Cokus
// Richard J. Wagner  v1.1  28 September 2009  wagnerr@umich.edu

// The Mersenne Twister is an algorithm for generating random numbers.  It
// was designed with consideration of the flaws in various other generators.
// The period, 2^19937-1, and the order of equidistribution, 623 dimensions,
// are far greater.  The generator is also fast; it avoids multiplication and
// division, and it benefits from caches and pipelines.  For more information
// see the inventors' web page at
// http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html

// Reference
// M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-Dimensionally
// Equidistributed Uniform Pseudo-Random Number Generator", ACM Transactions on
// Modeling and Computer Simulation, Vol. 8, No. 1, January 1998, pp 3-30.

// Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
// Copyright (C) 2000 - 2009, Richard J. Wagner
// All rights reserved.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions
// are met:
//
//   1. Redistributions of source code must retain the above copyright
//      notice, this list of conditions and the following disclaimer.
//
//   2. Redistributions in binary form must reproduce the above copyright
//      notice, this list of conditions and the following disclaimer in the
//      documentation and/or other materials provided with the distribution.
//
//   3. The names of its contributors may not be used to endorse or promote
//      products derived from this software without specific prior written
//      permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
// AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
// ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
// LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
// CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
// SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
// INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
// CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
// ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
// POSSIBILITY OF SUCH DAMAGE.

// The original code included the following notice:
//
//     When you use this, send an email to: m-mat@math.sci.hiroshima-u.ac.jp
//     with an appropriate reference to your work.
//
// It would be nice to CC: wagnerr@umich.edu and Cokus@math.washington.edu
// when you write.

#ifndef MERSENNETWISTER_H
#define MERSENNETWISTER_H

// Not thread safe (unless auto-initialization is avoided and each thread has
// its own MTRand object)
#include <cmath>
#include <ctime>
#include <cstdio>
#include <climits>
#include <iostream>

class MTRand
{
        // Data
        public:
                typedef unsigned long uint32;  // unsigned integer type, at least 32 bits
                enum { N = 624 }; // length of state vector
                enum { SAVE = N + 1 };  // length of array for save()

        protected:
                enum { M = 397 };  // period parameter

                int left;          // number of values left before reload needed

                uint32 *pNext;     // next value to get from state
                uint32 state[N];   // internal state

        // Methods
        public:
                MTRand();  // auto-initialize with /dev/urandom or time() and clock()
                MTRand(const MTRand& o);       // copy
                MTRand(const uint32 oneSeed);  // initialize with a simple uint32

                // Do NOT use for CRYPTOGRAPHY without securely hashing several
                // returned values together, otherwise the generator state can be
                // learned after reading 624 consecutive values.

                // Access to 32-bit random numbers
                uint32 randInt();                    // integer in [0,2^32-1]
                uint32 randInt(const uint32 n);      // integer in [0,n] for n < 2^32

                // Re-seeding functions with same behavior as initializers
                void seed();
                void seed(const uint32 oneSeed);
                void seed( uint32 *const bigSeed, const uint32 seedLength = N );

        protected:
                void reload();
                void initialize(const uint32 oneSeed);

                uint32 hiBit(const uint32 u)  const { return u & 0x80000000UL; }
                uint32 loBit(const uint32 u)  const { return u & 0x00000001UL; }
                uint32 loBits(const uint32 u) const { return u & 0x7fffffffUL; }
                uint32 magic( const uint32 u ) const
                        { return loBit(u) ? 0x9908b0dfUL : 0x0UL; }
                uint32 mixBits(const uint32 u, const uint32 v) const
                        { return hiBit(u) | loBits(v); }
                uint32 twist( const uint32 m, const uint32 s0, const uint32 s1 ) const
                        { return m ^ (mixBits(s0, s1) >> 1) ^ magic(s1); }
                static uint32 hash(time_t t, clock_t c);
};

// Functions are defined in order of usage to assist inlining

inline MTRand::uint32 MTRand::hash( time_t t, clock_t c )
{
        // Get a uint32 from t and c
        // Better than uint32(x) in case x is floating point in [0,1]
        // Based on code by Lawrence Kirby (fred@genesis.demon.co.uk)

        static uint32 differ = 0;  // guarantee time-based seeds will change

        uint32 h1 = 0;
        unsigned char *p = (unsigned char *) &t;
        for( size_t i = 0; i < sizeof(t); ++i )
        {
                h1 *= UCHAR_MAX + 2U;
                h1 += p[i];
        }
        uint32 h2 = 0;
        p = (unsigned char *) &c;
        for( size_t j = 0; j < sizeof(c); ++j )
        {
                h2 *= UCHAR_MAX + 2U;
                h2 += p[j];
        }
        return ( h1 + differ++ ) ^ h2;
}

inline MTRand::MTRand()
{ seed(); }

inline void MTRand::seed()
{
        // Seed the generator with an array from /dev/urandom if available
        // Otherwise use a hash of time() and clock() values

        // First try getting an array from /dev/urandom
        FILE* urandom;
        fopen_s(&urandom, "/dev/urandom", "rb" );
        if( urandom )
        {
                uint32 bigSeed[N];
                register uint32 *s = bigSeed;
                register int i = N;
                register bool success = true;
                while( success && i-- )
                        success = (fread(s++, sizeof(uint32), 1, urandom) == 1)
                                               ? true : false;
                fclose(urandom);
                if( success ) { seed( bigSeed, N );  return; }
        }

        // Was not successful, so use time() and clock() instead
        seed( hash( time(NULL), clock() ) );
}

inline void MTRand::seed( uint32 *const bigSeed, const uint32 seedLength )
{
        // Seed the generator with an array of uint32's
        // There are 2^19937-1 possible initial states.  This function allows
        // all of those to be accessed by providing at least 19937 bits (with a
        // default seed length of N = 624 uint32's).  Any bits above the lower 32
        // in each element are discarded.
        // Just call seed() if you want to get array from /dev/urandom
        initialize(19650218UL);
        register int i = 1;
        register uint32 j = 0;
        register int k = ( N > seedLength ? N : seedLength );
        for( ; k; --k )
        {
                state[i] =
                state[i] ^ ( (state[i-1] ^ (state[i-1] >> 30)) * 1664525UL );
                state[i] += ( bigSeed[j] & 0xffffffffUL ) + j;
                state[i] &= 0xffffffffUL;
                ++i;  ++j;
                if( i >= N ) { state[0] = state[N-1];  i = 1; }
                if( j >= seedLength ) j = 0;
        }
        for( k = N - 1; k; --k )
        {
                state[i] =
                state[i] ^ ( (state[i-1] ^ (state[i-1] >> 30)) * 1566083941UL );
                state[i] -= i;
                state[i] &= 0xffffffffUL;
                ++i;
                if( i >= N ) { state[0] = state[N-1];  i = 1; }
        }
        state[0] = 0x80000000UL;  // MSB is 1, assuring non-zero initial array
        reload();
}

inline void MTRand::initialize(const uint32 seed)
{
        // Initialize generator state with seed
        // See Knuth TAOCP Vol 2, 3rd Ed, p.106 for multiplier.
        // In previous versions, most significant bits (MSBs) of the seed affect
        // only MSBs of the state array.  Modified 9 Jan 2002 by Makoto Matsumoto.
        register int i = 1;
        register uint32 *s = state;
        register uint32 *r = state;

        *s++ = seed & 0xffffffffUL;

        for(; i < N; ++i)
        {
                *s++ = (1812433253UL * (*r ^ (*r >> 30)) + i) & 0xffffffffUL;
                r++;
        }
}

inline void MTRand::reload()
{
        // Generate N new values in state
        // Made clearer and faster by Matthew Bellew (matthew.bellew@home.com)
        static const int MmN = int(M) - int(N);  // in case enums are unsigned

        register int i;
        register uint32 *p = state;

        for(i = N - M; i--; ++p)
                *p = twist(p[M], p[0], p[1]);
        for(i = M; --i; ++p)
                *p = twist(p[MmN], p[0], p[1]);
        *p = twist(p[MmN], p[0], state[0]);

        left = N, pNext = state;
}

inline void MTRand::seed(const uint32 oneSeed)
{
        // Seed the generator with a simple uint32
        initialize(oneSeed);
        reload();
}

inline MTRand::MTRand( const uint32 oneSeed )
{ seed(oneSeed); }

inline MTRand::MTRand( const MTRand& o )
{
        register const uint32 *t = o.state;
        register uint32 *s = state;
        register int i = N;

        for(; i--; *s++ = *t++) {}
        left = o.left;
        pNext = &state[N - left];
}

inline MTRand::uint32 MTRand::randInt()
{
        // Pull a 32-bit integer from the generator state
        // Every other access function simply transforms the numbers extracted here

        if(left == 0) reload();
```

```
	--left;

	register uint32 s1;
	s1  = *pNext++;
	s1 ^= (s1 >> 11);
	s1 ^= (s1 <<  7) & 0x9d2c5680UL;
	s1 ^= (s1 << 15) & 0xefc60000UL;
	return (s1 ^ (s1 >> 18));
}

inline MTRand::uint32 MTRand::randInt(const uint32 n)
{
	// Find which bits are used in n
	// Optimized by Magnus Jonsson (magnus@smartelectronix.com)
	uint32 used = n;

	used |= used >> 1;
	used |= used >> 2;
	used |= used >> 4;
	used |= used >> 8;
	used |= used >> 16;

	// Draw numbers until one is found in [0,n]
	uint32 i;

	do
		i = randInt() & used;  // toss unused bits to shorten search
	while(i > n);
	return i;
}

#endif  // MERSENNETWISTER_H

// Change log:
//
// v0.1 - First release on 15 May 2000
//      - Based on code by Makoto Matsumoto, Takuji Nishimura, and Shawn Cokus
//      - Translated from C to C++
//      - Made completely ANSI compliant
//      - Designed convenient interface for initialization, seeding, and
//        obtaining numbers in default or user-defined ranges
//      - Added automatic seeding from /dev/urandom or time() and clock()
//      - Provided functions for saving and loading generator state
//
// v0.2 - Fixed bug which reloaded generator one step too late
//
// v0.3 - Switched to clearer, faster reload() code from Matthew Bellew
//
// v0.4 - Removed trailing newline in saved generator format to be consistent
//        with output format of built-in types
//
// v0.5 - Improved portability by replacing static const int's with enum's and
//        clarifying return values in seed(); suggested by Eric Heimburg
//      - Removed MAXINT constant; use 0xffffffffUL instead
//
// v0.6 - Eliminated seed overflow when uint32 is larger than 32 bits
//      - Changed integer [0,n] generator to give better uniformity
//
// v0.7 - Fixed operator precedence ambiguity in reload()
//      - Added access for real numbers in (0,1) and (0,n)
//
// v0.8 - Included time.h header to properly support time_t and clock_t
//
// v1.0 - Revised seeding to match 26 Jan 2002 update of Nishimura and Matsumoto
//      - Allowed for seeding with arrays of any length
//      - Added access for real numbers in [0,1) with 53-bit resolution
//      - Added access for real numbers from normal (Gaussian) distributions
//      - Increased overall speed by optimizing twist()
//      - Doubled speed of integer [0,n] generation
//      - Fixed out-of-range number generation on 64-bit machines
//      - Improved portability by substituting literal constants for long enum's
//      - Changed license from GNU LGPL to BSD
//
// v1.1 - Corrected parameter label in randNorm from "variance" to "stddev"
//      - Changed randNorm algorithm from basic to polar form for efficiency
//      - Updated includes from deprecated <xxxx.h> to standard <cxxxx> forms
//      - Cleaned declarations and definitions to please Intel compiler
//      - Revised twist() operator to work on ones'-complement machines
//      - Fixed reload() function to work when N and M are unsigned
//      - Added copy constructor and copy operator from Salvador Espana
```

Cascade

```
//----------------- UNCLASSIFIED//FOR OFFICIAL USE ONLY ---------------------//
// Cascade.h
//
// Lt Jim Johnson
// Air Force Institute of Technology
//
// Created on: December, 2011
// Last Updated: January, 2012
//
// Description: An implementation of the Cascade protocol as defined by
// Brassard and Salvail.
//---------------------------------------------------------------------------//

#ifndef CASCADE_H
#define CASCADE_H

#define BOB // eliminates unnecessary Alice functions

#include <bitset>
#include "Common.h"
#include "Permutation.h"

typedef unsigned int uint;

using namespace std;

class Cascade : public Permutation
{
        public:
                Cascade(const bitset<messageSize> &initialMessage, uint seed,
                            double errorRate, Cascade *cascade = NULL);

                Cascade *alice; // Pointer to the alice object, if we're Bob

                uint bitsLeaked; // Actual bits leaked
                uint minBitsLeaked; // Minimum possible to leak
                uint maxBitsLeaked; // Maximum possible to leak

                bitset<messageSize> parityString; // contains block parities

                void findErrors(); // main function to start the error correction

                bitset<messageSize>& getMessage(); // retrieves the corrected message

        private:
                uint startBlockSize; // the starting block size

                bitset<messageSize> message; // the key string to be corrected
                // retrieves the block parities
                void buildParityString(uint blockSize, uint pass);

                // sets the starting block size based on the error rate
                void setStartSize(double errorRate);

                // performs binary on earlier passes for errors found after pass 0
                void doCascade(int currentPass, uint error,
                                   bitset<messageSize> (&errors)[5]);

                // retrieves the parity of an individual block
                uint getParity(uint start, uint end, uint pass);

                // given an index and a pass, returns the block containing that index
                // in an earlier pass
                uint getBlock(uint index, uint blockSize, uint pass);

                // binary search routine
                uint getError(uint block, uint pass, uint blockSize);
};

#endif;
```

```
//----------------- UNCLASSIFIED//FOR OFFICIAL USE ONLY ---------------------//
// Cascade.cpp
//
// Lt Jim Johnson
// Air Force Institute of Technology
//
// Created on: December, 2011
// Last Updated: January, 2012
//
// Description: An implementation of the Cascade protocol as defined by
// Brassard and Salvail.
//---------------------------------------------------------------------------//

#include "Cascade.h"

// Base 2 Logarithmic function
double Log2(uint n);

// Input:
//      initialMessage - the key string to be reconciled
//      seed - a seed for the permutation function
//      startSize - the inital block size, chosed based on the error rate
//      cascade (opitonal) - a pointer to an Alice object, in the case of Bob
Cascade::Cascade(const bitset<messageSize> &initialMessage, uint seed,
                      double errorRate, Cascade *cascade)
{
        if (alice != NULL) alice = cascade; // only Bob needs a pointer to Alice

        // initialize the global variables
        bitsLeaked     = 0;
        maxBitsLeaked  = 0;
        minBitsLeaked  = 0;
        startBlockSize = 0;

        setStartSize(errorRate);

        message  ^= initialMessage;

        // use if an initial permutation is desired
        // Permutation::first = 1;
        // creates the permutation array used between passes
        Permutation::createPerm(seed);
}

// Input:
//      blockSize - the block size of the current pass
//      pass - the current pass number
// Purpose:
//      builds a string of block parities
void Cascade::buildParityString(uint blockSize, uint pass)
{
        parityString.reset();

        // increment the minimum and maximum bits leaked counters. The actual bits
        // leaked is not updated here, because it is updated in the getParity
        // function.
        maxBitsLeaked += (uint)ceil((double)messageSize / blockSize);
        minBitsLeaked += (uint)ceil((double)messageSize / blockSize);

        // calls getParity on each block of the message
        for(uint start = 0, index = 0; start < messageSize; start += blockSize, index++)
        {
                uint end = start + blockSize - 1;

                // Adjust for message lengths that are not evenly divisible by the
                // block size
                if (end >= messageSize) end = messageSize - 1;

                if (getParity(start, end, pass)) parityString.set(index, 1);
        }
        return;
}

// Input:
//      start - start index of the block
//      end - end index of the block
//      pass - current pass number
// Return value:
//      the parity of a given block
// Purpose:
//      returns the parity of a given block of the message, applying
//      a permutation in line.
uint Cascade::getParity(uint start, uint end, uint pass)
{
        uint count = 0; // stores the number of 1's found

        for (uint i = start; i <= end; i++)
        {
                // Permutes inline by making calls to the getIndex function and
                // providing it with the current pass value
                if (message[Permutation::getIndex(i, pass)] == 1) count++;
        }

        // Increment the actual bits leaked for each parity bit exchanged
        bitsLeaked++;

        return count % 2; // 0 if even parity, 1 if odd
}

#ifdef BOB // functions past this point are not used by Alice

// Purpose:
//      Attempts to correct all the errors in the message
void Cascade::findErrors()
{
        uint blockSize = startBlockSize;
        // used to track errors found in each pass
        bitset<messageSize> errorsFound[5];

        // loop for four passes
        for (uint pass = 0; pass < 4; pass++)
        {
                uint paritySize = 0;

                // Set the parity string length, even if the origMessage length is not an
                // even multiple of the block size
                if (messageSize % blockSize == 0) paritySize = messageSize / blockSize;
                else paritySize = (uint)ceil((double)messageSize / (double)blockSize);

                buildParityString(blockSize, pass);

                // tell Alice to build her parity string
                alice->buildParityString(blockSize, pass);

                // loop through Bob's parity string
                for(uint i = 0; i < paritySize; i++)
                {
                        // if we find a mismatch between Bob's parity string and Alice's,
                        // call getError to perform a binary search and find a single error
                        if (alice->parityString[i] != parityString[i])
                        {
                                uint error = getError(i, pass, blockSize); // get the error
                                message.flip(error);                       // fix the error
                                errorsFound[pass].set(error, 1);           // track the error

                                // do Cascade for passes after the first
                                if (pass > 0) doCascade(pass, error, errorsFound);
                        }
                }
                blockSize *= 2; // double the block size for the next pass
        }

        return;
}

// Input:
//      currentPass - the current pass number
//      error - the location of the error found in the current pass
//      errors[5] - the set of errors found so far in all passes
// Purpose:
//      if an error was found in a pass after the first, go back and do binary
//      on the error block containing the error in earlier passes, since there
//      is potentially a matching error
void Cascade::doCascade(int currentPass, uint error,
                            bitset<messageSize> (&errors)[5])
{
        // loop through each pass up to the current pass
        for(int pass = 0; pass < currentPass; pass++)
        {
                // get the block from the previous pass that contained that error. The
                // second argument to getBlock here is just the blockSize for that pass
                uint block = getBlock(error, startBlockSize * (1 << pass), pass);

                // get the start and finish indices from that block
                uint start  = block * startBlockSize * (1 << pass);
                uint finish = start + startBlockSize * (1 << pass) - 1;

                // set finish correctly for the last (possibly partial) block
                if (finish >= messageSize) finish = messageSize - 1;

                // get Alice's parity for that block
                uint aliceParity = alice->getParity(start, finish, pass);
                // We must leak one bit to verify there is an error. The alternative is
                // to assume there is an error, which will definitely leak Log2(n)
                // bits. If there is no error, we save Log2(n)-1 bits. If there is an
                // error, we leak Log2(n)+1 bits.
                maxBitsLeaked++;
                minBitsLeaked++;

                // if alice's parity matches, don't waste more bits performing a
                // binary search, since no error would be found
                if(getParity(start, finish, pass) == aliceParity) continue;

                // otherwise there is a matching error. Do binary on the block from
                // the pass that contained the error
                uint newError = getError(getBlock(error, startBlockSize * (1 << pass),
                                             pass), pass, startBlockSize * (1 << pass));

                message.flip(newError);  // fix the new error
                errors[pass].set(newError, 1);  // update the error tracking array

                // Since we corrected a bit, adjust the parity of that block so we
                // don't try to correct it twice
                parityString.flip(getBlock(newError, startBlockSize *
                                         (1 << currentPass), currentPass));

                // if Cascade found an error, it may be a new error (not a matching
                // error) and it could have a matching error, so recurse.
                doCascade(pass, newError, errors);
        }
}

// Input:
//      index - an index into the message
//      block size - the block size for the pass we are examining
//      pass - the pass we want information on
// Return value:
//      the block number containing the given index
// Purpose:
//      given an index and a pass, returns the block that index was contained in
uint Cascade::getBlock(uint index, uint blockSize, uint pass)
{
        uint origIndex = getReverseIndex(index, pass);

        return (uint)floor((double)origIndex / blockSize);
}

// Return value:
//      the corrected message
// Purpose:
//      retrieves the corrected message from Bob
bitset<messageSize>& Cascade::getMessage()
{
        return message;
}

// Input:
//      block - the number of the block thought to contain an error
//      pass - the current pass number
//      blockSize - the current block size
// Return value:
//      the index of the bit in error
// Purpose:
//      This function performs a binary search in order to locate a single bit
//      error in the given block. This function is only called if there is a
//      mismatched parity, therefore it will always find a single bit error.
uint Cascade::getError(uint block, uint pass, uint blockSize)
{
        uint start  = block * blockSize; // initialize start to start of the block
        uint finish = 0;

        uint aliceParity = 0;

        // if the message size is an even multiple of the block size, or the block
        // in question is not the last block of the origMessage, then the finish
        // index is just the end of the block. Otherwise, set the finish index to
        // the end of the message
        if ((messageSize % blockSize == 0) || (block != messageSize / blockSize))
                finish = block * blockSize + blockSize - 1;
        else if (block == messageSize / blockSize) finish = messageSize - 1;

        // Increment the minimum and maximum bits leaked counters
        maxBitsLeaked += (uint)ceil(Log2(finish - start + 1));
        minBitsLeaked += (uint)floor(Log2(finish - start + 1));

        while(true)
        {
                // if we're down to one bit, we have our error, return it
                if (finish - start == 0) return Permutation::getIndex(finish, pass);

                // if we're down to two bits, check the first one. If it is not in
                // error, then the error must be in the second bit, so return it.
                if (finish - start == 1)
                {
                        aliceParity = alice->getParity(start, start, pass);

                        if (getParity(start, start, pass) != aliceParity)
                                return(Permutation::getIndex(start, pass));
                        else return(Permutation::getIndex(finish, pass));
                }

                // otherwise there are more than two bits left. Ask alice for the
                // parity of her first half
                aliceParity = alice->getParity(start, start + (finish - start) / 2, pass);

                // check the parities of the first half of the message, if they
                // don't match alice's then the error is in the first half, so reset
                // the finish index
                if (getParity(start, start + (finish - start) / 2, pass) != aliceParity)
                        finish = start + (finish - start) / 2;

                // otherwise the error is in the last half, so reset the start index
                else
                {
                        // Adjustment for odd block sizes
                        if ((finish - start) % 2 != 0)
                                start = finish - (finish - start) / 2;
                        else start = finish - (finish - start) / 2 + 1;

                }
        }
}

// Input:
//      n - an integer value
// Return value:
//      The Log base 2 of n
double Log2(uint n)
{
    // Returns the Log base 2 of a given number
    return log((double)n) / log((double)2);
}

#endif

// Input:
```

```
//      errorRate - the estimated error rate of the key string
// Purpose:
//      sets the starting block size of the protocol. Usually accepted to be
//      .73/p, however these block sizes were empirically determined to be
//      good values for this implementation.
void Cascade::setStartSize(double errorRate)
{
#ifdef zero_percent_tolerance
        #if messageSize == 1000
                if (errorRate <= .005) startBlockSize = 25;
                else if (errorRate <= .010) startBlockSize = 23;
                else if (errorRate <= .015) startBlockSize = 19;
                else if (errorRate <= .020) startBlockSize = 16;
                else if (errorRate <= .025) startBlockSize = 12;
                else if (errorRate <= .030) startBlockSize = 11;
                else if (errorRate <= .035) startBlockSize = 11;
                else if (errorRate <= .040) startBlockSize = 12;
                else if (errorRate <= .045) startBlockSize = 11;
                else if (errorRate <= .050) startBlockSize = 9;
                else if (errorRate <= .055) startBlockSize = 10;
                else if (errorRate <= .060) startBlockSize = 6;
                else if (errorRate <= .065) startBlockSize = 7;
                else if (errorRate <= .070) startBlockSize = 9;
                else if (errorRate <= .075) startBlockSize = 7;
                else if (errorRate <= .080) startBlockSize = 7;
                else if (errorRate <= .085) startBlockSize = 8;
                else if (errorRate <= .090) startBlockSize = 6;
                else if (errorRate <= .095) startBlockSize = 5;
                else if (errorRate <= .100) startBlockSize = 5;
                else if (errorRate <= .105) startBlockSize = 4;
                else if (errorRate <= .110) startBlockSize = 4;
                else if (errorRate <= .115) startBlockSize = 6;
                else if (errorRate <= .120) startBlockSize = 4;
                else if (errorRate <= .125) startBlockSize = 5;
                else if (errorRate <= .130) startBlockSize = 4;
                else if (errorRate <= .135) startBlockSize = 5;
                else if (errorRate <= .140) startBlockSize = 4;
                else if (errorRate <= .145) startBlockSize = 4;
                else if (errorRate <= .150) startBlockSize = 4;
        #endif
        #if messageSize == 10000
                if (errorRate <= .005) startBlockSize = 76;
                else if (errorRate <= .010) startBlockSize = 46;
                else if (errorRate <= .015) startBlockSize = 35;
                else if (errorRate <= .020) startBlockSize = 27;
                else if (errorRate <= .025) startBlockSize = 27;
                else if (errorRate <= .030) startBlockSize = 22;
                else if (errorRate <= .035) startBlockSize = 16;
                else if (errorRate <= .040) startBlockSize = 17;
                else if (errorRate <= .045) startBlockSize = 15;
                else if (errorRate <= .050) startBlockSize = 14;
                else if (errorRate <= .055) startBlockSize = 11;
                else if (errorRate <= .060) startBlockSize = 11;
                else if (errorRate <= .065) startBlockSize = 10;
                else if (errorRate <= .070) startBlockSize = 9;
                else if (errorRate <= .075) startBlockSize = 9;
                else if (errorRate <= .080) startBlockSize = 8;
                else if (errorRate <= .085) startBlockSize = 9;
                else if (errorRate <= .090) startBlockSize = 7;
                else if (errorRate <= .095) startBlockSize = 8;
                else if (errorRate <= .100) startBlockSize = 5;
                else if (errorRate <= .105) startBlockSize = 6;
                else if (errorRate <= .110) startBlockSize = 6;
                else if (errorRate <= .115) startBlockSize = 6;
                else if (errorRate <= .120) startBlockSize = 6;
                else if (errorRate <= .125) startBlockSize = 5;
                else if (errorRate <= .130) startBlockSize = 5;
                else if (errorRate <= .135) startBlockSize = 4;
                else if (errorRate <= .140) startBlockSize = 5;
                else if (errorRate <= .145) startBlockSize = 5;
                else if (errorRate <= .150) startBlockSize = 5;
        #endif
        #if messageSize == 100000
                if (errorRate <= .005) startBlockSize = 81;
                else if (errorRate <= .010) startBlockSize = 57;
                else if (errorRate <= .015) startBlockSize = 45;
                else if (errorRate <= .020) startBlockSize = 32;
                else if (errorRate <= .025) startBlockSize = 27;
                else if (errorRate <= .030) startBlockSize = 24;
                else if (errorRate <= .035) startBlockSize = 22;
                else if (errorRate <= .040) startBlockSize = 19;
                else if (errorRate <= .045) startBlockSize = 13;
                else if (errorRate <= .050) startBlockSize = 14;
                else if (errorRate <= .055) startBlockSize = 12;
                else if (errorRate <= .060) startBlockSize = 11;
                else if (errorRate <= .065) startBlockSize = 11;
                else if (errorRate <= .070) startBlockSize = 10;
                else if (errorRate <= .075) startBlockSize = 10;
                else if (errorRate <= .080) startBlockSize = 9;
                else if (errorRate <= .085) startBlockSize = 9;
                else if (errorRate <= .090) startBlockSize = 10;
                else if (errorRate <= .095) startBlockSize = 8;
                else if (errorRate <= .100) startBlockSize = 7;
                else if (errorRate <= .105) startBlockSize = 7;
                else if (errorRate <= .110) startBlockSize = 6;
                else if (errorRate <= .115) startBlockSize = 6;
                else if (errorRate <= .120) startBlockSize = 5;
                else if (errorRate <= .125) startBlockSize = 6;
                else if (errorRate <= .130) startBlockSize = 6;
                else if (errorRate <= .135) startBlockSize = 5;
                else if (errorRate <= .140) startBlockSize = 6;
                else if (errorRate <= .145) startBlockSize = 6;
                else if (errorRate <= .150) startBlockSize = 6;
        #endif
#endif

#ifdef five_percent_tolerance
        #if messageSize == 1000
                if (errorRate <= .005) startBlockSize = 101;
                else if (errorRate <= .010) startBlockSize = 71;
                else if (errorRate <= .015) startBlockSize = 55;
                else if (errorRate <= .020) startBlockSize = 44;
                else if (errorRate <= .025) startBlockSize = 38;
                else if (errorRate <= .030) startBlockSize = 31;
                else if (errorRate <= .035) startBlockSize = 29;
                else if (errorRate <= .040) startBlockSize = 24;
                else if (errorRate <= .045) startBlockSize = 23;
                else if (errorRate <= .050) startBlockSize = 20;
                else if (errorRate <= .055) startBlockSize = 18;
                else if (errorRate <= .060) startBlockSize = 16;
                else if (errorRate <= .065) startBlockSize = 16;
                else if (errorRate <= .070) startBlockSize = 15;
                else if (errorRate <= .075) startBlockSize = 14;
                else if (errorRate <= .080) startBlockSize = 13;
                else if (errorRate <= .085) startBlockSize = 12;
                else if (errorRate <= .090) startBlockSize = 11;
                else if (errorRate <= .095) startBlockSize = 11;
                else if (errorRate <= .100) startBlockSize = 10;
                else if (errorRate <= .105) startBlockSize = 10;
                else if (errorRate <= .110) startBlockSize = 10;
                else if (errorRate <= .115) startBlockSize = 9;
                else if (errorRate <= .120) startBlockSize = 8;
                else if (errorRate <= .125) startBlockSize = 8;
                else if (errorRate <= .130) startBlockSize = 8;
                else if (errorRate <= .135) startBlockSize = 8;
                else if (errorRate <= .140) startBlockSize = 7;
                else if (errorRate <= .145) startBlockSize = 7;
                else if (errorRate <= .150) startBlockSize = 7;
        #endif
        #if messageSize == 10000
                if (errorRate <= .005) startBlockSize = 199;
                else if (errorRate <= .010) startBlockSize = 104;
                else if (errorRate <= .015) startBlockSize = 69;
                else if (errorRate <= .020) startBlockSize = 53;
                else if (errorRate <= .025) startBlockSize = 42;
                else if (errorRate <= .030) startBlockSize = 35;
                else if (errorRate <= .035) startBlockSize = 31;
                else if (errorRate <= .040) startBlockSize = 26;
                else if (errorRate <= .045) startBlockSize = 23;
                else if (errorRate <= .050) startBlockSize = 21;
                else if (errorRate <= .055) startBlockSize = 19;
                else if (errorRate <= .060) startBlockSize = 17;
                else if (errorRate <= .065) startBlockSize = 16;
                else if (errorRate <= .070) startBlockSize = 15;
                else if (errorRate <= .075) startBlockSize = 14;
                else if (errorRate <= .080) startBlockSize = 13;
                else if (errorRate <= .085) startBlockSize = 12;
                else if (errorRate <= .090) startBlockSize = 11;
                else if (errorRate <= .095) startBlockSize = 11;
                else if (errorRate <= .100) startBlockSize = 10;
                else if (errorRate <= .105) startBlockSize = 10;
                else if (errorRate <= .110) startBlockSize = 9;
                else if (errorRate <= .115) startBlockSize = 9;
                else if (errorRate <= .120) startBlockSize = 9;
                else if (errorRate <= .125) startBlockSize = 8;
                else if (errorRate <= .130) startBlockSize = 8;
                else if (errorRate <= .135) startBlockSize = 8;
                else if (errorRate <= .140) startBlockSize = 7;
                else if (errorRate <= .145) startBlockSize = 7;
                else if (errorRate <= .150) startBlockSize = 7;
        #endif
        #if messageSize == 100000
                if (errorRate <= .005) startBlockSize = 210;
                else if (errorRate <= .010) startBlockSize = 106;
                else if (errorRate <= .015) startBlockSize = 69;
                else if (errorRate <= .020) startBlockSize = 52;
                else if (errorRate <= .025) startBlockSize = 41;
                else if (errorRate <= .030) startBlockSize = 34;
                else if (errorRate <= .035) startBlockSize = 29;
                else if (errorRate <= .040) startBlockSize = 25;
                else if (errorRate <= .045) startBlockSize = 22;
                else if (errorRate <= .050) startBlockSize = 20;
                else if (errorRate <= .055) startBlockSize = 18;
                else if (errorRate <= .060) startBlockSize = 17;
                else if (errorRate <= .065) startBlockSize = 15;
                else if (errorRate <= .070) startBlockSize = 14;
                else if (errorRate <= .075) startBlockSize = 13;
                else if (errorRate <= .080) startBlockSize = 12;
                else if (errorRate <= .085) startBlockSize = 12;
                else if (errorRate <= .090) startBlockSize = 11;
                else if (errorRate <= .095) startBlockSize = 10;
                else if (errorRate <= .100) startBlockSize = 10;
                else if (errorRate <= .105) startBlockSize = 9;
                else if (errorRate <= .110) startBlockSize = 9;
                else if (errorRate <= .115) startBlockSize = 8;
                else if (errorRate <= .120) startBlockSize = 8;
                else if (errorRate <= .125) startBlockSize = 8;
                else if (errorRate <= .130) startBlockSize = 7;
                else if (errorRate <= .135) startBlockSize = 7;
                else if (errorRate <= .140) startBlockSize = 7;
                else if (errorRate <= .145) startBlockSize = 7;
```

```
                else if (errorRate <= .150) startBlockSize = 6;
        #endif
#endif

#ifdef ten_percent_tolerance
        #if messageSize == 1000
                if (errorRate <= .005) startBlockSize = 123;
                else if (errorRate <= .010) startBlockSize = 87;
                else if (errorRate <= .015) startBlockSize = 61;
                else if (errorRate <= .020) startBlockSize = 52;
                else if (errorRate <= .025) startBlockSize = 44;
                else if (errorRate <= .030) startBlockSize = 35;
                else if (errorRate <= .035) startBlockSize = 32;
                else if (errorRate <= .040) startBlockSize = 28;
                else if (errorRate <= .045) startBlockSize = 25;
                else if (errorRate <= .050) startBlockSize = 22;
                else if (errorRate <= .055) startBlockSize = 21;
                else if (errorRate <= .060) startBlockSize = 19;
                else if (errorRate <= .065) startBlockSize = 18;
                else if (errorRate <= .070) startBlockSize = 16;
                else if (errorRate <= .075) startBlockSize = 15;
                else if (errorRate <= .080) startBlockSize = 14;
                else if (errorRate <= .085) startBlockSize = 13;
                else if (errorRate <= .090) startBlockSize = 12;
                else if (errorRate <= .095) startBlockSize = 12;
                else if (errorRate <= .100) startBlockSize = 11;
                else if (errorRate <= .105) startBlockSize = 11;
                else if (errorRate <= .110) startBlockSize = 10;
                else if (errorRate <= .115) startBlockSize = 10;
                else if (errorRate <= .120) startBlockSize = 9;
                else if (errorRate <= .125) startBlockSize = 9;
                else if (errorRate <= .130) startBlockSize = 9;
                else if (errorRate <= .135) startBlockSize = 8;
                else if (errorRate <= .140) startBlockSize = 8;
                else if (errorRate <= .145) startBlockSize = 8;
                else if (errorRate <= .150) startBlockSize = 7;
        #endif
        #if messageSize == 10000
                if (errorRate <= .005) startBlockSize = 228;
                else if (errorRate <= .010) startBlockSize = 117;
                else if (errorRate <= .015) startBlockSize = 76;
                else if (errorRate <= .020) startBlockSize = 57;
                else if (errorRate <= .025) startBlockSize = 45;
                else if (errorRate <= .030) startBlockSize = 38;
                else if (errorRate <= .035) startBlockSize = 32;
                else if (errorRate <= .040) startBlockSize = 28;
                else if (errorRate <= .045) startBlockSize = 25;
                else if (errorRate <= .050) startBlockSize = 22;
                else if (errorRate <= .055) startBlockSize = 20;
                else if (errorRate <= .060) startBlockSize = 18;
                else if (errorRate <= .065) startBlockSize = 17;
                else if (errorRate <= .070) startBlockSize = 16;
                else if (errorRate <= .075) startBlockSize = 15;
                else if (errorRate <= .080) startBlockSize = 14;
                else if (errorRate <= .085) startBlockSize = 13;
                else if (errorRate <= .090) startBlockSize = 12;
                else if (errorRate <= .095) startBlockSize = 11;
                else if (errorRate <= .100) startBlockSize = 11;
                else if (errorRate <= .105) startBlockSize = 10;
                else if (errorRate <= .110) startBlockSize = 10;
                else if (errorRate <= .115) startBlockSize = 9;
                else if (errorRate <= .120) startBlockSize = 9;
                else if (errorRate <= .125) startBlockSize = 9;
                else if (errorRate <= .130) startBlockSize = 8;
                else if (errorRate <= .135) startBlockSize = 8;
                else if (errorRate <= .140) startBlockSize = 8;
                else if (errorRate <= .145) startBlockSize = 7;
                else if (errorRate <= .150) startBlockSize = 7;
        #endif
        #if messageSize == 100000
                if (errorRate <= .005) startBlockSize = 225;
                else if (errorRate <= .010) startBlockSize = 111;
                else if (errorRate <= .015) startBlockSize = 73;
                else if (errorRate <= .020) startBlockSize = 54;
                else if (errorRate <= .025) startBlockSize = 43;
                else if (errorRate <= .030) startBlockSize = 36;
                else if (errorRate <= .035) startBlockSize = 30;
                else if (errorRate <= .040) startBlockSize = 27;
                else if (errorRate <= .045) startBlockSize = 23;
                else if (errorRate <= .050) startBlockSize = 21;
                else if (errorRate <= .055) startBlockSize = 19;
                else if (errorRate <= .060) startBlockSize = 17;
                else if (errorRate <= .065) startBlockSize = 16;
                else if (errorRate <= .070) startBlockSize = 15;
                else if (errorRate <= .075) startBlockSize = 14;
                else if (errorRate <= .080) startBlockSize = 13;
                else if (errorRate <= .085) startBlockSize = 12;
                else if (errorRate <= .090) startBlockSize = 11;
                else if (errorRate <= .095) startBlockSize = 11;
                else if (errorRate <= .100) startBlockSize = 10;
                else if (errorRate <= .105) startBlockSize = 9;
                else if (errorRate <= .110) startBlockSize = 9;
                else if (errorRate <= .115) startBlockSize = 9;
                else if (errorRate <= .120) startBlockSize = 8;
                else if (errorRate <= .125) startBlockSize = 8;
                else if (errorRate <= .130) startBlockSize = 8;
                else if (errorRate <= .135) startBlockSize = 7;
                else if (errorRate <= .140) startBlockSize = 7;
                else if (errorRate <= .145) startBlockSize = 7;
                else if (errorRate <= .150) startBlockSize = 7;
        #endif
#endif

                if (startBlockSize == 0) startBlockSize = 4;
}

//----------------- UNCLASSIFIED//FOR OFFICIAL USE ONLY ---------------------//
```

Winnow

```
//----------------- UNCLASSIFIED//FOR OFFICIAL USE ONLY ---------------------//
// Winnow.h
//
// Lt Jim Johnson
// Air Force Institute of Technology
//
// Created on: December, 2011
// Last Updated: January, 2012
//
// Description: An implementation of the Winnow error reconciliation protocol
// written in C++. Derived from an earlier implementation of the Winnow class
// developed by Kevin Lustic.
//---------------------------------------------------------------------------//

#ifndef WINNOW_H_
#define WINNOW_H_

#include "Common.h"
#define Bob_instance
#define permute_bits_between_passes
```

```
#define permute_bits_before_first_pass

class Winnow
{
        public:
                Winnow* counterpart; // a pointer to an Alice or Bob object

                uint badBlocks;          // the number of blocks with a parity mismatch
                uint bitsExposed;        // the actual number of bits exposed
                uint syndromeArray[messageSize/8]; // the array used to store syndromes

                // the array used to store the locations of bad blocks
                uint badBlockArray[messageSize/8];

                // Bob's block parities used by Alice
                bitset<messageSize/8> bobParities;
                bitset<messageSize/8> parityBuffer; // our block parities

                Winnow();                // default constructor
                virtual ~Winnow();       // default destructor

                // the only constructor that should be called
                Winnow(bitset<messageSize> *message, uint seed, double rate,
                           Winnow* partner = NULL);

                uint fixErrors(); // performs the Winnow protocol to fix all errors

                void nextPass(); // readies the next pass of the protocol
                void buildSyndromeString(); // builds the syndrome string

                bitset<messageSize>& getKeystring();

        private:
                MTRand RNG; // a random number generator

                uint blockSize;                  // the current block size
                uint schedule[8];                // the schedule of block sizes
                uint numOfBlocks;                // the current number of blocks
                uint netBitsExposed;             // the exposed bits not yet removed
                uint syndromeLength;             // the current syndrome length

                // the current message size. Since this implementation uses bitsets,
                // which must have their size determined at compile time, the current
                // message size must be maintained as an upper bound.
                uint newMessageSize;

                bitset<messageSize> keyString; // the key string to be reconciled

                // the parity check matrix. In this implementation, it has a maximum
                // size of 10x1023.
                bool parityCheckMatrix[10][1023];

                uint getSyndrome(uint);                  // retrieves a block syndrome
                uint getNumRemainingPasses();            // retrieves passes remaining
                uint getParity(uint start, uint end);    // retrieves a block parity

                void firstPass();                        // prepares for the first pass
                void getParities();                      // retrieves the block parities
                void createMatrix();                     // creates the parity matrix
                void fixWithSyndrome();                  // fixes errors in a key string
                void discardParityBits();                // removes leaked parity bits
                void discardSyndromeBits();              // removes leaked syndrome bits
                void permuteBuffer(uint seed);           // permutes the key string
                void disagreeingBlockParities();         // determines mismatched blocks

                // sets the block size schedule based on the error rate
                void setBlockSchedule(double errorRate);

};

#endif /* WINNOW_H_ */

//----------------- UNCLASSIFIED//FOR OFFICIAL USE ONLY ---------------------//
```

```
//----------------- UNCLASSIFIED//FOR OFFICIAL USE ONLY ---------------------//
// Winnow.cpp
//
// Lt Jim Johnson
// Air Force Institute of Technology
//
// Created on: December, 2011
// Last Updated: January, 2012
//
// Description: An implementation of the Winnow error reconciliation protocol
// written in C++. Derived from an earlier implementation of the Winnow class
// developed by Kevin Lustic.
//---------------------------------------------------------------------------//

#include "Winnow.h"

Winnow::Winnow()
{
        printf("Default constructor for Winnow... use other constructor.\n");
}

Winnow::~Winnow()
{
}

// Input:
//      message - the key string to be corrected
//      seed - seed for the permutation functions
//      rate - the estimated error rate of the key string
// Purpose:
//      initializes global variables before initiating the first pass of the
//      protocol
Winnow::Winnow(bitset<messageSize> *message, uint seed, double rate,
                            Winnow* partner) : keyString(*message), RNG(seed)
{
        if (partner != NULL) counterpart = partner;

        badBlocks      = 0;
        blockSize      = 0;
        bitsExposed    = 0;
        numOfBlocks    = 0;
        netBitsExposed = 0;
        syndromeLength = 0;
        newMessageSize = 0;

        bobParities.reset();
        parityBuffer.reset();

        newMessageSize  = messageSize;

        for (int i = 0; i < messageSize/8; i++)
        {
                syndromeArray[i] = 0;
                badBlockArray[i] = 0;
        }
        for (int i = 0; i < 7; i++) schedule[i] = 0;

        for (int i = 0; i < 10; i++)
                for (int j = 0; j < 1023; j++)
                        parityCheckMatrix[i][j] = false;

        firstPass();
        setBlockSchedule(rate);
}

// Purpose:
//      the first pass of the protocol. Initializes the block size schedule,
//      determinines the initial blocksize, syndrome length, and number of
//      blocks, and generates the parity-check matrix.
void Winnow::firstPass()
{
        // for the first pass, we will always choose a block size of 8
        blockSize      = 8;
        syndromeLength = 3;

        // note that this number does not include an incomplete final block
        numOfBlocks = newMessageSize / blockSize;

        // generates of the parity-check matrix
        createMatrix();

        // permute the bits in the key buffer if desired
        #ifdef permute_bits_before_first_pass
                permuteBuffer(RNG.randInt());
        #endif

        // gets the block parites
        getParities();

        // discards one parity bit for each bit exposed
        discardParityBits();

        return;
}

//
// Purpose:
//      the next pass of the protocol. This function adjusts the blocksize
//      according to the schedule, creates the new parity check matrix and
//      adjusts any other values related to the block size.
void Winnow::nextPass()
{
        // iterates through the block size schedule
        for (uint i = 0; i < 8; i++)
        {
                if (schedule[i] > 0)
                {
                        syndromeLength = (uint)(i + 3);  // set the syndrome length

                        // the block size = 2^syndromeLength (Power not XOR)
                        blockSize = 1 << syndromeLength;

                        schedule[i]--;
                        break;
                }
                else if (i >= 7)
                {
                        // We reached end of schedule with no block choices left
                        cout << "Time to terminate.\n";
                        system("Pause");
                }
        }
        // set the number of blocks for the current pass
        numOfBlocks = newMessageSize / blockSize;

        // generate the parity check matrix
        createMatrix();

        // Permute the bits in the key buffer if desired
        #ifdef permute_bits_between_passes
                permuteBuffer(RNG.randInt());
        #endif

        // get the block parities
        getParities();

        // discard one parity bit for each bit exposed
        discardParityBits();

        return;
}

// Purpose:
//      gets the parities for each of the blocks and stores them in the global
//      parityBuffer bitset.
void Winnow::getParities()
{
        uint end    = 0;
        uint start  = 0;
        uint parity = 0;

        parityBuffer.reset();

        // iterate through the blocks. Note we ignore the last block if it is not
        // a full block.
        for(uint i = 0; i < numOfBlocks; i++)
        {
                start  = i * blockSize;
                end    = start + blockSize - 1;

                // parity of bits from start to end, inclusive
                parity = getParity(start, end);

                // each parity bit retrieved is counted as a bit exposed
                bitsExposed++;
                netBitsExposed++;

                // Update the parity buffer
                if(parity == 1) parityBuffer.set(i);
                else parityBuffer.reset(i);
        }
        return;
}

// Input:
//      start - a beginning index
//      end - an ending index
// Return value:
//      the parity of the given range of the keystring
uint Winnow::getParity(uint start, uint end)
{
        uint count = 0; // stores the number of 1's found

        for (uint i = start; i <= end; i++)      if (keyString.at(i) == 1) count++;

        return count % 2; // 0 if even parity, 1 if odd
}
```

```
// Purpose:
//      determines the block numbers of blocks where Alice and Bob's parities
//      do not match, and store those block numbers in the global badBlockArray
void Winnow::disagreeingBlockParities()
{
        uint counter = 0;

        badBlocks = 0; // reset the number of bad blocks

        for(uint i = 0; i < numOfBlocks; i++)
        {
                // If Alice and Bob have parities that don't match, add the index
                // (block #) to our list
                if(parityBuffer.at(i) != bobParities.at(i))
                {
                        badBlockArray[badBlocks] = i;
                        badBlocks++;
                }
        }
}

// Purpose:
//      Discards a bit from each block as a part of the Winnow privacy
//      maintenance. This implementation discards the first bit from each
//      block, for ease of coding. Discard is performed by copying the bits of
//      each block which aren't part of the key string backwards.
//      Note: the final block is ignored if incomplete.
void Winnow::discardParityBits()
{
        uint newIndex = 0;
        uint oldIndex = 0;

        bitset<messageSize> temp;

        for(; oldIndex < newMessageSize; oldIndex++, newIndex++)
        {
                // if we are at the first bit in the block
                if(oldIndex % blockSize == 0 && oldIndex != numOfBlocks * blockSize)
                {
                        // we want to remove this bit. Do not copy it.
                        netBitsExposed--;
                        newIndex--;
                }
                // otherwise this isn't the first bit in the block, copy it.
                else
                {
                        if(keyString.at(oldIndex) == 1) temp.set(newIndex);
                        else temp.reset(newIndex);
                }
        }
        // New block size, since we deleted one bit from each
        blockSize--;

        // New length of key, since we delete one bit per block
        newMessageSize = newIndex;

        keyString.reset();
        keyString ^= temp;

        return;
}

// Purpose:
//      function for generating the parity check matrix that Alice uses for
//      computing syndromes, and that Bob uses for correcting errors.
void Winnow::createMatrix()
{
        uint size = (1 << syndromeLength) - 1;

        for(uint i = 0; i < syndromeLength; i++)
        {
                for(uint j = 1; j <= size; j++)
                {
                        parityCheckMatrix[i][j - 1] = j / (1 << i) & 0x1;
                }
        }
        return;
}

void Winnow::buildSyndromeString()
{
        bobParities = counterpart->parityBuffer; // retrieve Bob's block parities

        disagreeingBlockParities();

        for (uint i = 0; i < badBlocks; i++)
        {
                syndromeArray[i] = getSyndrome(badBlockArray[i]);
        }

        discardSyndromeBits();
}

// Input:
//      The block number of the block for which to calculate the syndrome
// Return value:
//      The syndrome, in uint form
// Purpose:
//      The syndrome is returned as an unsigned integer. The blocksize would
//      have to be > 2^32-1 to break this method, which is an unrealistic block
//      size in practice. Unsigned integers for this purpose is less unwieldy
//      than, say, a separate buffer for every syndrome.
//
//      An example is a syndrome of '1 1 0' would be returned as '6' rather
//      than a three-element buffer.
uint Winnow::getSyndrome(uint blockNumber)
{
        // ensure the block number is legal
        if(blockNumber > numOfBlocks)
        {
                printf("Illegal block number. Returning blockSize + 1 for new syndrome.\n");
                return (blockSize + 1);
        }

        uint temp = 0;
        uint newSyndrome = 0;

        // compute the highest order bit of the syndrome first and work down
        for(int i = syndromeLength - 1; i >= 0; i--)
        {
                newSyndrome <<= 1; // Push previous bit up

                // Multiply the block by the (i-1)th row of the parity check matrix
                for(uint j = 0; j < blockSize; j++)
                {
                        temp = (temp + parityCheckMatrix[i][j] * keyString.at(blockNumber *
                                    blockSize + j)) & 0x1;
                }
                newSyndrome += temp; // Add the resulting sum to the syndrome
                temp = 0;
        }

        // Number of bits exposed is equal to the number of bits in the syndrome
        bitsExposed    +=syndromeLength;
        netBitsExposed +=syndromeLength;

        return newSyndrome;
}

// Purpose:
//      The bits at indices of the form 2^j-1 are removed. These correspond to
//      the linearly independent columns of the parity check matrix. Note that
//      this appears to be O(n^2) but the inner for loop is deceiving. This
//      operation is just O(n).
void Winnow::discardSyndromeBits()
{
        uint blockNum           = 0;
        uint newCounter              = 0;
        uint oldCounter         = 0;
        uint errorBlocksCounter = 0;

        bitset<messageSize> temp;

        // Loop through the blocks
        for(blockNum = 0, newCounter = 0; blockNum < numOfBlocks; blockNum++)
        {
                // If we haven't hit all the bad blocks and the counter (blockNum) is
                // at a bad block
                if(errorBlocksCounter < badBlocks &&
                   badBlockArray[errorBlocksCounter] == (uint)blockNum)
                {
                        uint power = 0;

                        errorBlocksCounter++;

                        // Iterate through the block
                        for(uint bitNum = 0; bitNum < blockSize; bitNum++)
                        {
                                // if the current bit location +1 is a power of 2, don't copy
                                if(bitNum + 1 == (uint)(1 << power))
                                {
                                        power++;
                                        oldCounter++;
                                        netBitsExposed--;
                                }
                                else // otherwise copy the bit
                                {
                                        if(keyString.at(oldCounter)) temp.set(newCounter);
                                        else temp.reset(newCounter);

                                        newCounter++;
                                        oldCounter++;
                                }
                        }
                }
                else // otherwise, we're not at a bad block copy the bit
                {
                        if(keyString.at(oldCounter)) temp.set(newCounter);
                        else temp.reset(newCounter);

                        newCounter++;
                        oldCounter++;
                }
        }

        // copy the rest of the bits not copied above
        while (oldCounter < newMessageSize)
        {
                if(keyString.at(oldCounter)) temp.set(newCounter);
                else temp.reset(newCounter);
```

```
		newCounter++;
		oldCounter++;
	}

	// copy the new message over and set associated parameters
	newMessageSize = newCounter;

	keyString.reset();
	keyString ^= temp;

	return;
}

// Input:
//	seed - a seed for the permutation
// Purpose:
//	permutes the key string by randomly shuffling the bits
void Winnow::permuteBuffer(uint seed)
{
	MTRand rand(seed); // random number generator seeded with seed

	uint oldIndex = 0;
	uint newIndex = 0;

	// loop through the message, and at each point swap the value at the
	// current index with the value at a randomly generated index.
	for (uint i = 0; i < newMessageSize; i++)
	{
		oldIndex = i;
		newIndex = rand.randInt(newMessageSize-1);

		bool value = keyString[oldIndex];

		keyString[oldIndex] = keyString[newIndex];
		keyString[newIndex] = value;
	}

	return;
}

#ifdef Bob_instance

// Return value:
//	number of passes remaining
// Purpose:
//	returns the number of passes left according to block schedule. This
//	function is responsible for the termination of the algorithm
uint Winnow::getNumRemainingPasses()
{
	uint count = 0;

	for(uint i = 0; i < 8; i++) count += schedule[i];

	return count;
}

// Purpose:
//	This is used by Bob. Bob computes the syndrome of his block, and xor's
//	it with the syndrome from Alice. The result is an offset pointing to
//	the exact location of the erroneous bit, if the first bit of the block
//	is considered to be at position '1'.
void Winnow::fixWithSyndrome()
{
	badBlocks = counterpart->badBlocks; // get Alice's bad blocks

	// retrieve and copy Alice's bad block array and syndrome array
	uint *temp1 = counterpart->badBlockArray;
```

```
        uint *temp2 = counterpart->syndromeArray;

        for (uint i = 0; i < badBlocks; i++)
        {
                badBlockArray[i] = temp1[i];
                syndromeArray[i] = temp2[i];
        }

        for (uint i = 0; i < badBlocks; i++)
        {
                // get the syndrome of the corresponding block
                uint mySyndrome = getSyndrome(badBlockArray[i]);

                // store in syndrome the result of xor-ing the two syndromes. This will
                // give an offset representing the location of the error to fix
                syndromeArray[i] ^= mySyndrome;

                if(syndromeArray[i] == 0)
                        { /* The error was discarded in the parity cleanup! */ }
                else keyString.flip(badBlockArray[i] * blockSize +
                                        (syndromeArray[i] - 1));
        }

        // discard one syndrome bit for each bit exposed
        discardSyndromeBits();

        return;
}


// Return value: the key string
bitset<messageSize>& Winnow::getKeystring() { return keyString; }

// Return value:
//      the number of errors remaining in Alice and Bob's versions of the key
//      string.
//Purpose:
//      used for convenience while running experiments. This function gives a
//      good example of the overall protocol. Counterpart in this case should
//      point to an Alice Winnow object, since Bob should lead the process.
uint Winnow::fixErrors()
{
        int passLimit = 1;

        while (passLimit > 0)
        {
                counterpart->buildSyndromeString();

                fixWithSyndrome();

                nextPass();

                counterpart->nextPass();

                passLimit = getNumRemainingPasses();
        }
        return (counterpart->getKeystring()^getKeystring()).count();
}

#endif

// Input:
//      errorRate - the estimated error rate of the key string
//Puprose:
//      sets the block schedule of the protocol based on the key string length
//      and the estimated error rate
void Winnow::setBlockSchedule(double errorRate)
```

```
{
#ifdef zero_percent_tolerance
        #if messageSize == 1000
                if(errorRate <= 0.0105)

                {
                        schedule[0] = 1; schedule[1] = 0; schedule[2] = 1; schedule[3] = 1;
                        schedule[4] = 1; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.0205)
                {
                        schedule[0] = 1; schedule[1] = 1; schedule[2] = 2; schedule[3] = 0;
                        schedule[4] = 0; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.0305)
                {
                        schedule[0] = 1; schedule[1] = 3; schedule[2] = 0; schedule[3] = 0;
                        schedule[4] = 0; schedule[5] = 1; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.0405)
                {
                        schedule[0] = 2; schedule[1] = 1; schedule[2] = 1; schedule[3] = 1;
                        schedule[4] = 0; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.0505)
                {
                        schedule[0] = 2; schedule[1] = 1; schedule[2] = 1; schedule[3] = 0;
                        schedule[4] = 0; schedule[5] = 1; schedule[6] = 1; schedule[7] = 1;
                }
                else if(errorRate <= 0.0605)
                {
                        schedule[0] = 2; schedule[1] = 1; schedule[2] = 2; schedule[3] = 1;
                        schedule[4] = 0; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.0705)
                {
                        schedule[0] = 2; schedule[1] = 1; schedule[2] = 2; schedule[3] = 2;
                        schedule[4] = 1; schedule[5] = 0; schedule[6] = 0; schedule[7] = 0;
                }
                else if(errorRate <= 0.0805)
                {
                        schedule[0] = 2; schedule[1] = 2; schedule[2] = 1; schedule[3] = 2;
                        schedule[4] = 0; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.0905)
                {
                        schedule[0] = 2; schedule[1] = 2; schedule[2] = 1; schedule[3] = 2;
                        schedule[4] = 1; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.1005)
                {
                        schedule[0] = 3; schedule[1] = 1; schedule[2] = 2; schedule[3] = 1;
                        schedule[4] = 0; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.1105)
                {
                        schedule[0] = 3; schedule[1] = 2; schedule[2] = 1; schedule[3] = 1;
                        schedule[4] = 0; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.1205)
                {
                        schedule[0] = 3; schedule[1] = 2; schedule[2] = 3; schedule[3] = 0;
                        schedule[4] = 0; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.1305)
                {
                        schedule[0] = 4; schedule[1] = 2; schedule[2] = 2; schedule[3] = 1;
```

```
		schedule[4] = 0; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
	}
	else if(errorRate <= 0.1405)
	{
		schedule[0] = 4; schedule[1] = 2; schedule[2] = 2; schedule[3] = 0;
		schedule[4] = 0; schedule[5] = 0; schedule[6] = 0; schedule[7] = 2;
	}
	else if(errorRate <= 0.1505)
	{
		schedule[0] = 4; schedule[1] = 3; schedule[2] = 2; schedule[3] = 0;
		schedule[4] = 0; schedule[5] = 0; schedule[6] = 1; schedule[7] = 0;
	}
#elif messageSize == 10000
	if(errorRate <= 0.0105)

	{
		schedule[0] = 1; schedule[1] = 0; schedule[2] = 1; schedule[3] = 0;
		schedule[4] = 0; schedule[5] = 2; schedule[6] = 0; schedule[7] = 1;
	}
	else if(errorRate <= 0.0205)
	{
		schedule[0] = 1; schedule[1] = 0; schedule[2] = 2; schedule[3] = 1;
		schedule[4] = 0; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
	}
	else if(errorRate <= 0.0305)
	{
		schedule[0] = 1; schedule[1] = 1; schedule[2] = 0; schedule[3] = 2;
		schedule[4] = 0; schedule[5] = 1; schedule[6] = 0; schedule[7] = 1;
	}
	else if(errorRate <= 0.0405)
	{
		schedule[0] = 1; schedule[1] = 1; schedule[2] = 1; schedule[3] = 1;
		schedule[4] = 1; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
	}
	else if(errorRate <= 0.0505)
	{
		schedule[0] = 1; schedule[1] = 1; schedule[2] = 1; schedule[3] = 2;
		schedule[4] = 0; schedule[5] = 1; schedule[6] = 0; schedule[7] = 1;
	}
	else if(errorRate <= 0.0605)
	{
		schedule[0] = 1; schedule[1] = 1; schedule[2] = 2; schedule[3] = 2;
		schedule[4] = 0; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
	}
	else if(errorRate <= 0.0705)
	{
		schedule[0] = 1; schedule[1] = 1; schedule[2] = 2; schedule[3] = 3;
		schedule[4] = 1; schedule[5] = 0; schedule[6] = 0; schedule[7] = 0;
	}
	else if(errorRate <= 0.0805)
	{
		schedule[0] = 2; schedule[1] = 1; schedule[2] = 1; schedule[3] = 2;
		schedule[4] = 0; schedule[5] = 0; schedule[6] = 1; schedule[7] = 1;
	}
	else if(errorRate <= 0.0905)
	{
		schedule[0] = 2; schedule[1] = 1; schedule[2] = 1; schedule[3] = 1;
		schedule[4] = 1; schedule[5] = 1; schedule[6] = 0; schedule[7] = 2;
	}
	else if(errorRate <= 0.1005)
	{
		schedule[0] = 2; schedule[1] = 1; schedule[2] = 1; schedule[3] = 3;
		schedule[4] = 0; schedule[5] = 0; schedule[6] = 0; schedule[7] = 2;
	}
	else if(errorRate <= 0.1105)
	{
		schedule[0] = 2; schedule[1] = 1; schedule[2] = 3; schedule[3] = 1;
```

```
                schedule[4] = 0; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
        }
        else if(errorRate <= 0.1205)
        {
                schedule[0] = 2; schedule[1] = 3; schedule[2] = 1; schedule[3] = 1;
                schedule[4] = 0; schedule[5] = 1; schedule[6] = 0; schedule[7] = 1;
        }
        else if(errorRate <= 0.1305)
        {
                schedule[0] = 3; schedule[1] = 1; schedule[2] = 2; schedule[3] = 1;
                schedule[4] = 1; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
        }
        else if(errorRate <= 0.1405)
        {
                schedule[0] = 3; schedule[1] = 2; schedule[2] = 1; schedule[3] = 1;
                schedule[4] = 1; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
        }
        else if(errorRate <= 0.1505)
        {
                schedule[0] = 3; schedule[1] = 2; schedule[2] = 2; schedule[3] = 1;
                schedule[4] = 1; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
        }
#elif messageSize == 100000
        if(errorRate <= 0.0105)

        {
                schedule[0] = 1; schedule[1] = 0; schedule[2] = 0; schedule[3] = 1;
                schedule[4] = 0; schedule[5] = 1; schedule[6] = 1; schedule[7] = 1;
        }
        else if(errorRate <= 0.0205)
        {
                schedule[0] = 1; schedule[1] = 0; schedule[2] = 1; schedule[3] = 0;
                schedule[4] = 0; schedule[5] = 2; schedule[6] = 1; schedule[7] = 1;
        }
        else if(errorRate <= 0.0305)
        {
                schedule[0] = 1; schedule[1] = 0; schedule[2] = 1; schedule[3] = 2;
                schedule[4] = 1; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
        }
        else if(errorRate <= 0.0405)
        {
                schedule[0] = 1; schedule[1] = 0; schedule[2] = 2; schedule[3] = 2;
                schedule[4] = 0; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
        }
        else if(errorRate <= 0.0505)
        {
                schedule[0] = 1; schedule[1] = 1; schedule[2] = 0; schedule[3] = 2;
                schedule[4] = 2; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
        }
        else if(errorRate <= 0.0605)
        {
                schedule[0] = 1; schedule[1] = 1; schedule[2] = 2; schedule[3] = 2;
                schedule[4] = 0; schedule[5] = 0; schedule[6] = 1; schedule[7] = 1;
        }
        else if(errorRate <= 0.0705)
        {
                schedule[0] = 1; schedule[1] = 1; schedule[2] = 2; schedule[3] = 2;
                schedule[4] = 1; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
        }
        else if(errorRate <= 0.0805)
        {
                schedule[0] = 2; schedule[1] = 1; schedule[2] = 1; schedule[3] = 1;
                schedule[4] = 0; schedule[5] = 0; schedule[6] = 0; schedule[7] = 2;
        }
        else if(errorRate <= 0.0905)
        {
                schedule[0] = 2; schedule[1] = 1; schedule[2] = 1; schedule[3] = 1;
                schedule[4] = 1; schedule[5] = 0; schedule[6] = 1; schedule[7] = 1;
        }
        else if(errorRate <= 0.1005)
        {
                schedule[0] = 2; schedule[1] = 1; schedule[2] = 1; schedule[3] = 1;
                schedule[4] = 0; schedule[5] = 1; schedule[6] = 1; schedule[7] = 1;
        }
        else if(errorRate <= 0.1105)
        {
                schedule[0] = 2; schedule[1] = 2; schedule[2] = 1; schedule[3] = 1;
                schedule[4] = 0; schedule[5] = 0; schedule[6] = 0; schedule[7] = 2;
        }
        else if(errorRate <= 0.1205)
        {
                schedule[0] = 2; schedule[1] = 2; schedule[2] = 3; schedule[3] = 0;
                schedule[4] = 0; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
        }
        else if(errorRate <= 0.1305)
        {
                schedule[0] = 2; schedule[1] = 3; schedule[2] = 1; schedule[3] = 0;
                schedule[4] = 0; schedule[5] = 2; schedule[6] = 0; schedule[7] = 1;
        }
        else if(errorRate <= 0.1405)
        {
                schedule[0] = 3; schedule[1] = 1; schedule[2] = 2; schedule[3] = 1;
                schedule[4] = 0; schedule[5] = 0; schedule[6] = 1; schedule[7] = 1;
        }
        else if(errorRate <= 0.1505)
        {
                schedule[0] = 3; schedule[1] = 2; schedule[2] = 2; schedule[3] = 0;
                schedule[4] = 0; schedule[5] = 1; schedule[6] = 0; schedule[7] = 1;
        }
#elif messageSize == 1000000
        if(errorRate <= 0.01)

        {
                schedule[0] = 1; schedule[1] = 0; schedule[2] = 0; schedule[3] = 1;
                schedule[4] = 0; schedule[5] = 0; schedule[6] = 1; schedule[7] = 2;
        }
        else if(errorRate <= 0.02)
        {
                schedule[0] = 1; schedule[1] = 0; schedule[2] = 1; schedule[3] = 0;
                schedule[4] = 0; schedule[5] = 1; schedule[6] = 2; schedule[7] = 1;
        }
        else if(errorRate <= 0.03)
        {
                schedule[0] = 1; schedule[1] = 1; schedule[2] = 0; schedule[3] = 0;
                schedule[4] = 1; schedule[5] = 0; schedule[6] = 2; schedule[7] = 1;
        }
        else if(errorRate <= 0.04)
        {
                schedule[0] = 1; schedule[1] = 1; schedule[2] = 0; schedule[3] = 0;
                schedule[4] = 1; schedule[5] = 0; schedule[6] = 2; schedule[7] = 1;
        }
        else if(errorRate <= 0.05)
        {
                schedule[0] = 1; schedule[1] = 1; schedule[2] = 0; schedule[3] = 1;
                schedule[4] = 0; schedule[5] = 1; schedule[6] = 2; schedule[7] = 1;
        }
        else if(errorRate <= 0.06)
        {
                schedule[0] = 1; schedule[1] = 1; schedule[2] = 1; schedule[3] = 0;
                schedule[4] = 1; schedule[5] = 0; schedule[6] = 2; schedule[7] = 2;
        }
        else if(errorRate <= 0.07)
        {
                schedule[0] = 2; schedule[1] = 0; schedule[2] = 1; schedule[3] = 0;
```

```
                        schedule[4] = 1; schedule[5] = 0; schedule[6] = 2; schedule[7] = 1;
                }
                else if(errorRate <= 0.08)
                {
                        schedule[0] = 2; schedule[1] = 1; schedule[2] = 0; schedule[3] = 0;
                        schedule[4] = 1; schedule[5] = 0; schedule[6] = 2; schedule[7] = 1;
                }
                else if(errorRate <= 0.09)
                {
                        schedule[0] = 2; schedule[1] = 1; schedule[2] = 0; schedule[3] = 1;
                        schedule[4] = 0; schedule[5] = 1; schedule[6] = 2; schedule[7] = 1;
                }
                else if(errorRate <= 0.10)
                {
                        schedule[0] = 2; schedule[1] = 1; schedule[2] = 1; schedule[3] = 0;
                        schedule[4] = 1; schedule[5] = 0; schedule[6] = 2; schedule[7] = 1;
                }
                else if(errorRate <= 0.11)
                {
                        schedule[0] = 3; schedule[1] = 0; schedule[2] = 1; schedule[3] = 0;
                        schedule[4] = 1; schedule[5] = 0; schedule[6] = 2; schedule[7] = 1;
                }
                else if(errorRate <= 0.12)
                {
                        schedule[0] = 3; schedule[1] = 1; schedule[2] = 0; schedule[3] = 0;
                        schedule[4] = 1; schedule[5] = 0; schedule[6] = 2; schedule[7] = 1;
                }
                else if(errorRate <= 0.13)
                {
                        schedule[0] = 3; schedule[1] = 1; schedule[2] = 0; schedule[3] = 1;
                        schedule[4] = 0; schedule[5] = 1; schedule[6] = 2; schedule[7] = 1;
                }
                else if(errorRate <= 0.14)
                {
                        schedule[0] = 3; schedule[1] = 1; schedule[2] = 1; schedule[3] = 0;
                        schedule[4] = 1; schedule[5] = 0; schedule[6] = 2; schedule[7] = 1;
                }
                else if(errorRate <= 0.15)
                {
                        schedule[0] = 4; schedule[1] = 0; schedule[2] = 1; schedule[3] = 0;
                        schedule[4] = 1; schedule[5] = 0; schedule[6] = 2; schedule[7] = 1;
                }
        #endif
#endif

#ifdef five_percent_tolerance
        #if messageSize == 1000
                if(errorRate <= 0.0105)

                {
                        schedule[0] = 1; schedule[1] = 0; schedule[2] = 0; schedule[3] = 1;
                        schedule[4] = 0; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.0205)
                {
                        schedule[0] = 1; schedule[1] = 0; schedule[2] = 1; schedule[3] = 0;
                        schedule[4] = 0; schedule[5] = 1; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.0305)
                {
                        schedule[0] = 1; schedule[1] = 1; schedule[2] = 0; schedule[3] = 1;
                        schedule[4] = 0; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.0405)
                {
                        schedule[0] = 1; schedule[1] = 1; schedule[2] = 1; schedule[3] = 0;
                        schedule[4] = 0; schedule[5] = 1; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.0505)
                {
                        schedule[0] = 1; schedule[1] = 1; schedule[2] = 1; schedule[3] = 1;
                        schedule[4] = 0; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.0605)
                {
                        schedule[0] = 1; schedule[1] = 1; schedule[2] = 2; schedule[3] = 0;
                        schedule[4] = 0; schedule[5] = 1; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.0705)
                {
                        schedule[0] = 1; schedule[1] = 2; schedule[2] = 1; schedule[3] = 0;
                        schedule[4] = 0; schedule[5] = 1; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.0805)
                {
                        schedule[0] = 1; schedule[1] = 2; schedule[2] = 2; schedule[3] = 0;
                        schedule[4] = 0; schedule[5] = 1; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.0905)
                {
                        schedule[0] = 1; schedule[1] = 2; schedule[2] = 3; schedule[3] = 0;
                        schedule[4] = 0; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.1005)
                {
                        schedule[0] = 1; schedule[1] = 3; schedule[2] = 2; schedule[3] = 0;
                        schedule[4] = 0; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.1105)
                {
                        schedule[0] = 2; schedule[1] = 2; schedule[2] = 1; schedule[3] = 0;
                        schedule[4] = 1; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.1205)
                {
                        schedule[0] = 2; schedule[1] = 2; schedule[2] = 2; schedule[3] = 0;
                        schedule[4] = 0; schedule[5] = 1; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.1305)
                {
                        schedule[0] = 2; schedule[1] = 3; schedule[2] = 2; schedule[3] = 0;
                        schedule[4] = 0; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.1405)
                {
                        schedule[0] = 3; schedule[1] = 2; schedule[2] = 0; schedule[3] = 1;
                        schedule[4] = 1; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.1505)
                {
                        schedule[0] = 3; schedule[1] = 2; schedule[2] = 1; schedule[3] = 1;
                        schedule[4] = 0; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
                }
        #elif messageSize == 10000
                if(errorRate <= 0.0105)

                {
                        schedule[0] = 1; schedule[1] = 0; schedule[2] = 0; schedule[3] = 0;
                        schedule[4] = 1; schedule[5] = 0; schedule[6] = 1; schedule[7] = 1;
                }
                else if(errorRate <= 0.0205)
                {
                        schedule[0] = 1; schedule[1] = 0; schedule[2] = 1; schedule[3] = 1;
                        schedule[4] = 0; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.0305)
                {
                        schedule[0] = 1; schedule[1] = 0; schedule[2] = 1; schedule[3] = 1;
                        schedule[4] = 0; schedule[5] = 0; schedule[6] = 1; schedule[7] = 1;
                }
                else if(errorRate <= 0.0405)
                {
                        schedule[0] = 1; schedule[1] = 0; schedule[2] = 2; schedule[3] = 1;
                        schedule[4] = 0; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.0505)
                {
                        schedule[0] = 1; schedule[1] = 1; schedule[2] = 1; schedule[3] = 0;
                        schedule[4] = 1; schedule[5] = 0; schedule[6] = 0; schedule[7] = 2;
                }
                else if(errorRate <= 0.0605)
                {
                        schedule[0] = 1; schedule[1] = 2; schedule[2] = 0; schedule[3] = 0;
                        schedule[4] = 1; schedule[5] = 0; schedule[6] = 1; schedule[7] = 1;
                }
                else if(errorRate <= 0.0705)
                {
                        schedule[0] = 1; schedule[1] = 2; schedule[2] = 0; schedule[3] = 0;
                        schedule[4] = 2; schedule[5] = 0; schedule[6] = 1; schedule[7] = 1;
                }
                else if(errorRate <= 0.0805)
                {
                        schedule[0] = 1; schedule[1] = 2; schedule[2] = 1; schedule[3] = 0;
                        schedule[4] = 1; schedule[5] = 1; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.0905)
                {
                        schedule[0] = 1; schedule[1] = 2; schedule[2] = 1; schedule[3] = 1;
                        schedule[4] = 0; schedule[5] = 1; schedule[6] = 1; schedule[7] = 1;
                }
                else if(errorRate <= 0.1005)
                {
                        schedule[0] = 2; schedule[1] = 0; schedule[2] = 2; schedule[3] = 1;
                        schedule[4] = 1; schedule[5] = 1; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.1105)
                {
                        schedule[0] = 2; schedule[1] = 1; schedule[2] = 2; schedule[3] = 0;
                        schedule[4] = 0; schedule[5] = 1; schedule[6] = 1; schedule[7] = 1;
                }
                else if(errorRate <= 0.1205)
                {
                        schedule[0] = 2; schedule[1] = 1; schedule[2] = 2; schedule[3] = 1;
                        schedule[4] = 1; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.1305)
                {
                        schedule[0] = 2; schedule[1] = 2; schedule[2] = 1; schedule[3] = 2;
                        schedule[4] = 0; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.1405)
                {
                        schedule[0] = 3; schedule[1] = 2; schedule[2] = 0; schedule[3] = 0;
                        schedule[4] = 0; schedule[5] = 1; schedule[6] = 1; schedule[7] = 2;
                }
                else if(errorRate <= 0.1505)
                {
                        schedule[0] = 3; schedule[1] = 2; schedule[2] = 0; schedule[3] = 1;
                        schedule[4] = 0; schedule[5] = 1; schedule[6] = 1; schedule[7] = 1;
                }
        #elif messageSize == 100000
```

```
if(errorRate <= 0.0105)

{
        schedule[0] = 1; schedule[1] = 0; schedule[2] = 0; schedule[3] = 0;
        schedule[4] = 2; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
}
else if(errorRate <= 0.0205)
{
        schedule[0] = 1; schedule[1] = 0; schedule[2] = 0; schedule[3] = 0;
        schedule[4] = 1; schedule[5] = 1; schedule[6] = 1; schedule[7] = 2;
}
else if(errorRate <= 0.0305)
{
        schedule[0] = 1; schedule[1] = 0; schedule[2] = 1; schedule[3] = 0;
        schedule[4] = 1; schedule[5] = 0; schedule[6] = 0; schedule[7] = 3;
}
else if(errorRate <= 0.0405)
{
        schedule[0] = 1; schedule[1] = 0; schedule[2] = 1; schedule[3] = 1;
        schedule[4] = 1; schedule[5] = 0; schedule[6] = 1; schedule[7] = 1;
}
else if(errorRate <= 0.0505)
{
        schedule[0] = 1; schedule[1] = 1; schedule[2] = 0; schedule[3] = 0;
        schedule[4] = 1; schedule[5] = 3; schedule[6] = 0; schedule[7] = 1;
}
else if(errorRate <= 0.0605)
{
        schedule[0] = 1; schedule[1] = 1; schedule[2] = 1; schedule[3] = 0;
        schedule[4] = 1; schedule[5] = 0; schedule[6] = 1; schedule[7] = 2;
}
else if(errorRate <= 0.0705)
{
        schedule[0] = 1; schedule[1] = 2; schedule[2] = 0; schedule[3] = 0;
        schedule[4] = 1; schedule[5] = 1; schedule[6] = 1; schedule[7] = 1;
}
else if(errorRate <= 0.0805)
{
        schedule[0] = 1; schedule[1] = 2; schedule[2] = 0; schedule[3] = 2;
        schedule[4] = 1; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
}
else if(errorRate <= 0.0905)
{
        schedule[0] = 1; schedule[1] = 2; schedule[2] = 1; schedule[3] = 0;
        schedule[4] = 2; schedule[5] = 1; schedule[6] = 0; schedule[7] = 1;
}
else if(errorRate <= 0.1005)
{
        schedule[0] = 2; schedule[1] = 1; schedule[2] = 1; schedule[3] = 0;
        schedule[4] = 0; schedule[5] = 1; schedule[6] = 1; schedule[7] = 2;
}
else if(errorRate <= 0.1105)
{
        schedule[0] = 2; schedule[1] = 1; schedule[2] = 1; schedule[3] = 1;
        schedule[4] = 0; schedule[5] = 1; schedule[6] = 1; schedule[7] = 1;
}
else if(errorRate <= 0.1205)
{
        schedule[0] = 2; schedule[1] = 1; schedule[2] = 2; schedule[3] = 2;
        schedule[4] = 0; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
}
else if(errorRate <= 0.1305)
{
        schedule[0] = 2; schedule[1] = 2; schedule[2] = 1; schedule[3] = 0;
        schedule[4] = 1; schedule[5] = 2; schedule[6] = 0; schedule[7] = 1;
}
else if(errorRate <= 0.1405)
```

```
                {
                        schedule[0] = 3; schedule[1] = 2; schedule[2] = 0; schedule[3] = 0;
                        schedule[4] = 0; schedule[5] = 0; schedule[6] = 1; schedule[7] = 3;
                }
                else if(errorRate <= 0.1505)
                {
                        schedule[0] = 3; schedule[1] = 2; schedule[2] = 0; schedule[3] = 0;
                        schedule[4] = 1; schedule[5] = 1; schedule[6] = 1; schedule[7] = 1;
                }
        #endif
#endif

#ifdef ten_percent_tolerance
        #if messageSize == 1000
                if(errorRate <= 0.0105)

                {
                        schedule[0] = 1; schedule[1] = 0; schedule[2] = 0; schedule[3] = 0;
                        schedule[4] = 1; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.0205)
                {
                        schedule[0] = 1; schedule[1] = 0; schedule[2] = 0; schedule[3] = 1;
                        schedule[4] = 0; schedule[5] = 1; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.0305)
                {
                        schedule[0] = 1; schedule[1] = 1; schedule[2] = 0; schedule[3] = 0;
                        schedule[4] = 1; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.0405)
                {
                        schedule[0] = 1; schedule[1] = 1; schedule[2] = 0; schedule[3] = 1;
                        schedule[4] = 0; schedule[5] = 0; schedule[6] = 1; schedule[7] = 1;
                }
                else if(errorRate <= 0.0505)
                {
                        schedule[0] = 1; schedule[1] = 1; schedule[2] = 1; schedule[3] = 0;
                        schedule[4] = 1; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.0605)
                {
                        schedule[0] = 1; schedule[1] = 1; schedule[2] = 1; schedule[3] = 1;
                        schedule[4] = 0; schedule[5] = 0; schedule[6] = 1; schedule[7] = 1;
                }
                else if(errorRate <= 0.0705)
                {
                        schedule[0] = 1; schedule[1] = 1; schedule[2] = 2; schedule[3] = 0;
                        schedule[4] = 1; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.0805)
                {
                        schedule[0] = 1; schedule[1] = 2; schedule[2] = 1; schedule[3] = 1;
                        schedule[4] = 0; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.0905)
                {
                        schedule[0] = 1; schedule[1] = 2; schedule[2] = 2; schedule[3] = 0;
                        schedule[4] = 1; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.1005)
                {
                        schedule[0] = 2; schedule[1] = 2; schedule[2] = 0; schedule[3] = 0;
                        schedule[4] = 1; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.1105)
                {
```

```
                schedule[0] = 2; schedule[1] = 2; schedule[2] = 0; schedule[3] = 1;
                schedule[4] = 1; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
        }
        else if(errorRate <= 0.1205)
        {
                schedule[0] = 2; schedule[1] = 2; schedule[2] = 1; schedule[3] = 1;
                schedule[4] = 0; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
        }
        else if(errorRate <= 0.1305)
        {
                schedule[0] = 2; schedule[1] = 3; schedule[2] = 1; schedule[3] = 0;
                schedule[4] = 0; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
        }
        else if(errorRate <= 0.1405)
        {
                schedule[0] = 2; schedule[1] = 3; schedule[2] = 1; schedule[3] = 1;
                schedule[4] = 0; schedule[5] = 1; schedule[6] = 0; schedule[7] = 1;
        }
        else if(errorRate <= 0.1505)
        {
                schedule[0] = 3; schedule[1] = 2; schedule[2] = 1; schedule[3] = 0;
                schedule[4] = 0; schedule[5] = 1; schedule[6] = 0; schedule[7] = 1;
        }
#elif messageSize == 10000
        if(errorRate <= 0.0105)

        {
                schedule[0] = 1; schedule[1] = 0; schedule[2] = 0; schedule[3] = 0;
                schedule[4] = 0; schedule[5] = 1; schedule[6] = 1; schedule[7] = 1;
        }
        else if(errorRate <= 0.0205)
        {
                schedule[0] = 1; schedule[1] = 0; schedule[2] = 0; schedule[3] = 2;
                schedule[4] = 0; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
        }
        else if(errorRate <= 0.0305)
        {
                schedule[0] = 1; schedule[1] = 0; schedule[2] = 1; schedule[3] = 1;
                schedule[4] = 0; schedule[5] = 0; schedule[6] = 0; schedule[7] = 2;
        }
        else if(errorRate <= 0.0405)
        {
                schedule[0] = 1; schedule[1] = 0; schedule[2] = 2; schedule[3] = 0;
                schedule[4] = 0; schedule[5] = 1; schedule[6] = 0; schedule[7] = 1;
        }
        else if(errorRate <= 0.0505)
        {
                schedule[0] = 1; schedule[1] = 1; schedule[2] = 1; schedule[3] = 0;
                schedule[4] = 0; schedule[5] = 1; schedule[6] = 0; schedule[7] = 2;
        }
        else if(errorRate <= 0.0605)
        {
                schedule[0] = 1; schedule[1] = 1; schedule[2] = 1; schedule[3] = 1;
                schedule[4] = 0; schedule[5] = 1; schedule[6] = 0; schedule[7] = 1;
        }
        else if(errorRate <= 0.0705)
        {
                schedule[0] = 1; schedule[1] = 2; schedule[2] = 0; schedule[3] = 0;
                schedule[4] = 1; schedule[5] = 1; schedule[6] = 1; schedule[7] = 1;
        }
        else if(errorRate <= 0.0805)
        {
                schedule[0] = 1; schedule[1] = 2; schedule[2] = 1; schedule[3] = 0;
                schedule[4] = 1; schedule[5] = 0; schedule[6] = 0; schedule[7] = 2;
        }
        else if(errorRate <= 0.0905)
        {
                schedule[0] = 1; schedule[1] = 2; schedule[2] = 1; schedule[3] = 0;
                schedule[4] = 2; schedule[5] = 0; schedule[6] = 1; schedule[7] = 1;
        }
        else if(errorRate <= 0.1005)
        {
                schedule[0] = 2; schedule[1] = 0; schedule[2] = 2; schedule[3] = 0;
                schedule[4] = 3; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
        }
        else if(errorRate <= 0.1105)
        {
                schedule[0] = 2; schedule[1] = 1; schedule[2] = 1; schedule[3] = 1;
                schedule[4] = 0; schedule[5] = 1; schedule[6] = 1; schedule[7] = 1;
        }
        else if(errorRate <= 0.1205)
        {
                schedule[0] = 2; schedule[1] = 1; schedule[2] = 2; schedule[3] = 1;
                schedule[4] = 0; schedule[5] = 0; schedule[6] = 1; schedule[7] = 1;
        }
        else if(errorRate <= 0.1305)
        {
                schedule[0] = 2; schedule[1] = 2; schedule[2] = 1; schedule[3] = 1;
                schedule[4] = 1; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
        }
        else if(errorRate <= 0.1405)
        {
                schedule[0] = 3; schedule[1] = 2; schedule[2] = 0; schedule[3] = 0;
                schedule[4] = 0; schedule[5] = 0; schedule[6] = 2; schedule[7] = 2;
        }
        else if(errorRate <= 0.1505)
        {
                schedule[0] = 3; schedule[1] = 2; schedule[2] = 0; schedule[3] = 0;
                schedule[4] = 1; schedule[5] = 1; schedule[6] = 1; schedule[7] = 1;
        }
#elif messageSize == 100000
        if(errorRate <= 0.0105)

        {
                schedule[0] = 1; schedule[1] = 0; schedule[2] = 0; schedule[3] = 0;
                schedule[4] = 1; schedule[5] = 0; schedule[6] = 1; schedule[7] = 1;
        }
        else if(errorRate <= 0.0205)
        {
                schedule[0] = 1; schedule[1] = 0; schedule[2] = 0; schedule[3] = 0;
                schedule[4] = 2; schedule[5] = 1; schedule[6] = 0; schedule[7] = 1;
        }
        else if(errorRate <= 0.0305)
        {
                schedule[0] = 1; schedule[1] = 0; schedule[2] = 1; schedule[3] = 0;
                schedule[4] = 2; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
        }
        else if(errorRate <= 0.0405)
        {
                schedule[0] = 1; schedule[1] = 0; schedule[2] = 1; schedule[3] = 1;
                schedule[4] = 1; schedule[5] = 0; schedule[6] = 0; schedule[7] = 2;
        }
        else if(errorRate <= 0.0505)
        {
                schedule[0] = 1; schedule[1] = 0; schedule[2] = 2; schedule[3] = 1;
                schedule[4] = 1; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
        }
        else if(errorRate <= 0.0605)
        {
                schedule[0] = 1; schedule[1] = 1; schedule[2] = 1; schedule[3] = 1;
                schedule[4] = 0; schedule[5] = 1; schedule[6] = 0; schedule[7] = 1;
        }
        else if(errorRate <= 0.0705)
        {
```

```
                        schedule[0] = 1; schedule[1] = 1; schedule[2] = 1; schedule[3] = 1;
                        schedule[4] = 1; schedule[5] = 1; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.0805)
                {
                        schedule[0] = 1; schedule[1] = 1; schedule[2] = 2; schedule[3] = 2;
                        schedule[4] = 0; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.0905)
                {
                        schedule[0] = 1; schedule[1] = 2; schedule[2] = 1; schedule[3] = 1;
                        schedule[4] = 0; schedule[5] = 0; schedule[6] = 2; schedule[7] = 1;
                }
                else if(errorRate <= 0.1005)
                {
                        schedule[0] = 2; schedule[1] = 1; schedule[2] = 0; schedule[3] = 1;
                        schedule[4] = 1; schedule[5] = 0; schedule[6] = 2; schedule[7] = 1;
                }
                else if(errorRate <= 0.1105)
                {
                        schedule[0] = 2; schedule[1] = 1; schedule[2] = 1; schedule[3] = 1;
                        schedule[4] = 0; schedule[5] = 1; schedule[6] = 0; schedule[7] = 2;
                }
                else if(errorRate <= 0.1205)
                {
                        schedule[0] = 2; schedule[1] = 1; schedule[2] = 2; schedule[3] = 1;
                        schedule[4] = 0; schedule[5] = 1; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.1305)
                {
                        schedule[0] = 2; schedule[1] = 2; schedule[2] = 1; schedule[3] = 2;
                        schedule[4] = 0; schedule[5] = 0; schedule[6] = 0; schedule[7] = 1;
                }
                else if(errorRate <= 0.1405)
                {
                        schedule[0] = 2; schedule[1] = 3; schedule[2] = 1; schedule[3] = 0;
                        schedule[4] = 0; schedule[5] = 1; schedule[6] = 0; schedule[7] = 2;
                }
                else if(errorRate <= 0.1505)
                {
                        schedule[0] = 3; schedule[1] = 2; schedule[2] = 0; schedule[3] = 0;
                        schedule[4] = 1; schedule[5] = 0; schedule[6] = 2; schedule[7] = 1;
                }
        #endif
#endif

        schedule[0]--;
}

//----------------- UNCLASSIFIED//FOR OFFICIAL USE ONLY ---------------------//
```

```
//----------------- UNCLASSIFIED//FOR OFFICIAL USE ONLY ---------------------//
// LDPC.h
//
// Lt Jim Johnson
// Air Force Institute of Technology
//
// Created on: December, 2011
// Last Updated: January, 2012
//
// Description: An implementation of the Sum Product decoding algorithm for
// Low Density Parity Check Codes.
//---------------------------------------------------------------------------//
```

```
#ifndef LDPC_H
#define LDPC_H

#include <math.h>
#include <vector>
#include "Common.h"

using namespace std;

class LDPC
{
        public:
                // constructor
                LDPC(double errorRate, int arraySize,
                         bitset<codeSize>* syndrome = NULL,
                         bitset<messageSize>* message = NULL);

                // used for experiments. Contains an example of the protocol
                int fixErrors();

                // calculates a syndrome for a given input message
                bitset<codeSize> getSyndrome(bitset<messageSize>& message);

        private:
                int    size;    // smaller dimension of the parity check matrix
                bool   first;   // flag for first iteration
                double p;       // estimated error rate

                // array used to store decisions for the decision step
                double D_Array[messageSize];

                // stores the list of variable nodes each check node is connected to
                vector<int>    C_Array[codeSize];

                // stores the list of check nodes each variable node is connected to
                vector<int>    V_Array[messageSize];

                vector<double> Q_Array[messageSize]; // stores the R-messages
                vector<double> R_Array[codeSize];    // stores the Q-messages

                bitset<codeSize> targetSyndrome;          // the target syndrome
                bitset<messageSize> correctedMessage;     // the key string to correct

                void verticalStep();                      // the vertical step
                void decisionStep();                      // the decision step
                void horizontalStep();                    // the horizontal step
                void getMatrices(string filename);        // retrieves the parity matrix
};

#endif;

//----------------- UNCLASSIFIED//FOR OFFICIAL USE ONLY ---------------------//
```

```
//----------------- UNCLASSIFIED//FOR OFFICIAL USE ONLY ---------------------//
// LDPC.cpp
//
// Lt Jim Johnson
// Air Force Institute of Technology
//
// Created on: December, 2011
// Last Updated: January, 2012
//
// Description: An implementation of the Sum Product decoding algorithm for
// Low Density Parity Check Codes.
//---------------------------------------------------------------------------//
```

```
#include "LDPC.h"

// Input:
//      errorRate - the estimated error rate of the key string
//      arraySize - the smaller dimension of the parity check matrix
//      syndrome (optional) - the target syndrome for Bob to match
//      message (optional) - the (flawed) received message
LDPC::LDPC(double errorRate, int arraySize, bitset<codeSize>* syndrome,
               bitset<messageSize>* message)
{
        string fileName = ""; // file containing the parity check matrix (alist)

        // if optional parameters are supplied, copy them into global counterparts
        if (message != NULL) correctedMessage = *message;
        if (syndrome != NULL) targetSyndrome = *syndrome;

        p     = errorRate; // local error rate variable
        first = true;      // flag for the first pass
        size = arraySize;  // local variable that tracks the parity matrix size

        for(int i = 0; i < messageSize; i++) D_Array[i] = 0;

        // retrieve the appropriate parity check file based on the input array size
        // note: shown for a 100 000 bit key length and must be modified for other
        //       values
        if      (arraySize == 10000) fileName = "Matrices_9.txt";
        else if (arraySize == 15000) fileName = "Matrices_85.txt";
        else if (arraySize == 20000) fileName = "Matrices_8.txt";
        else if (arraySize == 25000) fileName = "Matrices_75.txt";
        else if (arraySize == 30000) fileName = "Matrices_7.txt";
        else if (arraySize == 35000) fileName = "Matrices_65.txt";
        else if (arraySize == 40000) fileName = "Matrices_6.txt";
        else if (arraySize == 45000) fileName = "Matrices_55.txt";
        else if (arraySize == 50000) fileName = "Matrices_5.txt";
        else if (arraySize == 55000) fileName = "Matrices_45.txt";
        else if (arraySize == 60000) fileName = "Matrices_4.txt";
        else if (arraySize == 65000) fileName = "Matrices_35.txt";
        else if (arraySize == 70000) fileName = "Matrices_3.txt";
        else if (arraySize == 75000) fileName = "Matrices_25.txt";
        else if (arraySize == 80000) fileName = "Matrices_2.txt";
        else if (arraySize == 85000) fileName = "Matrices_15.txt";
        else if (arraySize == 90000) fileName = "Matrices_1.txt";
        else system("Pause");

        // read in the matrix
        getMatrices(fileName);
}

// Input:
//      filename - name of the file containing the parity check matrix
// Purpose:
//      reads in the parity check matrix and stores it in two arrays. The
//      C_Array maintains a list of the check nodes and which variable nodes
//      they are connected to. The V_Array maintains a list of the variable
//      nodes and which check nodes they are connected to.
void LDPC::getMatrices(string filename)
{
        char buffer[500];

        fstream file;

        memset(buffer, '\0', 500);

        file.open(filename, fstream::in); // open the file

        for (int i = 0; i < size; i++)
```

```
	{
		file.getline(buffer, 500); // read a line from the file

		for(int j = 0; j < messageSize; j++)
		{
			char num[10];
			memset(num, '\0', 10);

			// copy the first number from the input line
			strncpy_s(num, buffer, strcspn(buffer, " "));

			// remove the number from the current list, and break when the
			// list is empty
			if(strlen(buffer) != 0)
				strncpy_s(buffer, strchr(buffer, ' ') + 1, 500);
			else break;

			// zero values signifiy the end of the line
			if (atoi(num) != 0)
			{
				// The input file contains a list of check nodes, therefore we
				// can add those to the array sequentially. However, for
				// variable nodes we must add the current check node index at
				// the appropriate variable node value, in this case num-1
				// since non-zero numbering is used in the partiy matrix file.
				C_Array[i].push_back(atoi(num)-1);
				V_Array[atoi(num)-1].push_back(i);
			}
		}
	}

	return;
}

// Input:
//	message - the message to calculate a syndrome for
// Return value:
//	the syndrome for the input message calculated using the parity matrix
// Purpose:
//	generates a syndrome based on the input string and the parity matrix
bitset<codeSize> LDPC::getSyndrome(bitset<messageSize>& message)
{
	bitset<codeSize> syndrome;

	// used to iterate through the list of parity checks
	vector<int>::iterator index;

	syndrome.reset();

	// iterate through all the check nodes
	for (int i = 0; i < size; i++)
	{
		// Calculate the parity for each node based on the message
		for (index = C_Array[i].begin(); index != C_Array[i].end(); index++)
		{
			// flip the parity every time we reach a 1. Computationally
			// efficient
			if (message[*index] == 1) syndrome[i].flip();
		}
	}

	return syndrome;
}

// Purpose:
//	After the horizontal and vertical steps, iterate through the current
//	list of likelihoods and determine if the present value of the message
```

```
//      is likely incorrect. If so, flip it.
void LDPC::decisionStep()
{
        int indices[codeSize]; // used to maintain the current index into the array

        vector<int>::iterator index; // used to iterate through the V_Array

        for(int i = 0; i < size; i++) indices[i] = 0;

        // iterate through the message
        for(int i = 0; i < messageSize; i++)
        {
                // Calculate the channel error probability
                if(correctedMessage[i] == 0) D_Array[i] = log((1-p)/p);
                else D_Array[i] = log(p/(1-p));

                // Add the R values to the decision
                for (index = V_Array[i].begin();
                         index != V_Array[i].end(); index++)
                {
                        // indices is used to keep track of our depth in the vector, as to
                        // prevent adding a value twice
                        D_Array[i] += R_Array[*index][indices[*index]];

                        indices[*index]++;
                }

                // the decision threshold for this implementation is set at +-10.
                if (D_Array[i] < -10) correctedMessage[i] = 1;
                else if (D_Array[i] >  10) correctedMessage[i] = 0;
        }
}

// Purpose:
//      This is the R-message update step. The R-messages are updated based on
//      the Q-messages and and error probability.
void LDPC::verticalStep()
{
        int indices[codeSize]; // used to maintain the depth of the vectors

        // used to iterate through the vector values
        vector<int>::iterator V_Node1;
        vector<int>::iterator V_Node2;

        for(int i = 0; i < size; i++) indices[i] = 0;

        // iterate through the message
        for(int i = 0; i < messageSize; i++)
        {
                int    count  = 0;
                double result = 0;

                // calculate the channel error probability and add it to the result
                if(correctedMessage[i] == 0) result = log((1-p)/p);
                else result = log(p/(1-p));

                // if this is not the first iteration, the R-message is the error
                // probability plus the sum of all the Q-messages from all of the
                // variable nodes except the one we're preparing the R-message for.
                // Initially, we add all the Q-messages to the R-message. Later we will
                // subtract off the one we don't need. This substantially improves
                // runtime vs. summing all the various combinations.
                if(!first)
                {
                        for(V_Node1 = V_Array[i].begin();
                                V_Node1 != V_Array[i].end(); V_Node1++)
                        {
                                result += R_Array[*V_Node1].at(indices[*V_Node1]);
                        }
                }

                // iterate through all the variable nodes
                for(V_Node1 = V_Array[i].begin();
                        V_Node1 != V_Array[i].end(); V_Node1++)
                {
                        // if this is the first iteration the R-message is simply the error
                        // probability, so we're done.
                        if (first)
                        {
                                Q_Array[i].push_back(result);
                                continue;
                        }

                        // we need to subtract off the value of the Q-message from the
                        // variable node that we're calculating an R-message for, since it
                        // should not have been included in the result
                        double R_Value = result - R_Array[*V_Node1].at(indices[*V_Node1]);

                        // in the event that the R-message is zero, set it equal to the
                        // LLR of the error probability. (protects against divide-by-zero)
                        if (R_Value == 0) R_Value = result;

                        // save the R-message
                        Q_Array[i].push_back(R_Value);

                        // keep track of our position in the vector of check nodes
                        indices[*V_Node1]++;
                }
        }

        first = false; // done with the first iteration

        // we're done with the Q-messages. Clear them out for the next iteration
        for (int i = 0; i < size; i++) R_Array[i].clear();
}

// Purpose:
//      This is the Q-message update step. The Q-messages are updated based on
//      the R-Messages.
void LDPC::horizontalStep()
{
        // used to iterate through the variable nodes
        vector<int>::iterator C_Node1;

        // used to keep track of our position in the vectors
        int indices[messageSize];

        for(int i = 0; i < messageSize; i++) indices[i] = 0;

        // iterate through all of the check nodes
        for (int i = 0; i < size; i++)
        {
                // used to store the product of all the hyperbolic tangent operations
                double Tanh_Product = 1;

                // iterate through all of the R-messages, and calculate the product of
                // the hyperbolic tangents. Similar to above, we calculate one product
                // up front, then divide out the value we do not want.
                for(C_Node1 = C_Array[i].begin(); C_Node1 != C_Array[i].end();
                        C_Node1++)
                {
                        Tanh_Product *= tanh(Q_Array[*C_Node1][indices[*C_Node1]] / 2);
                }
                // iterate through all the R-messages
                for(C_Node1 = C_Array[i].begin(); C_Node1 != C_Array[i].end();
                        C_Node1++)
                {
                        // divide out the value of the Q-message that we don't need. Even
                        // though divisions are expensive, this method saves cycles
                        double newProduct = Tanh_Product /
                                    tanh(Q_Array[*C_Node1][indices[*C_Node1]] / 2);

                        // bound the value of newProduct at +-(1 - 10^12). Since the
                        // inverse hyperbolic tangent is asymptotic at +-1, this step
                        // prevents infinite results.
                        if (newProduct > (1 - pow(10.0, -12)))
                                newProduct = 1 - pow(10.0, -12);
                        if (newProduct < (pow(10.0, -12)) - 1)
                                newProduct = pow(10.0, -12) - 1;

                        // calculate the final value of the Q-message
                        double R_Value = pow((double)-1, targetSyndrome[i])*2*
                                     ((log(1+newProduct)-log(1-newProduct))/2);

                        // save the Q-message
                        R_Array[i].push_back(R_Value);

                        // increment our position in the vector of variable nodes
                        indices[*C_Node1]++;
                }
        }

        // we're done with the R-messages. Clear them out for the next iteration
        for (int i = 0; i < messageSize; i++) Q_Array[i].clear();
}

// Purpose:
//      The steps of the protocol. Used for simulations and is provided as an
//      example for future use.
int LDPC::fixErrors()
{
        int count = 0;

        while (true) // Loop until all errors are corrected or 200 iterations
        {
                verticalStep();         // perform the vertical step
                horizontalStep();       // perform the horizontal step
                decisionStep();         // perform the decision step

                count++; // increment the iteration counter

                // check the current value of the message syndrome against the target.
                // If they match, we're done
                if((getSyndrome(correctedMessage)^targetSyndrome).count() == 0) break;

                if (count % 200 == 0) break;
        }

        return count; // return the iteration count
}

//----------------- UNCLASSIFIED//FOR OFFICIAL USE ONLY ---------------------//
```

VIII. Bibliography

Ardehali, M., Chau, H., & Lo, H.-K. (2005). Efficient Quantum Key Distribution. *Journal of Cryptography, 18*(2), 133-165.

Bellot, P., & Dang, M.-D. (2009). BB84 Implementation and Computer Reality. *International Conference on Computing and Communication Technologies*, (pp. 1-8).

Bennett, C. H., & Brassard, G. (1984). Quantum Cryptography: Public Key Distribution and Coin Tossing. *International Conference on Computers, Systems & Signal Processing.* Bangalore, India.

Bennett, C. H., Bessette, F., Brassard, G., Salvail, L., & Smolin, J. (1991). Experimental Quantum Cryptography. *Eurocrypt 1990*, (pp. 253-265). Aarhus, Denmark.

Boughattas, M. B., Iyed, B. S., & Rezig, H. (2010). Correcting Codes in the quantum keys reconciliation: scenarios of privacy maintenance. *IEEE International Conference on Social Computing*, (pp. 1022-1025). Minneapolis, MN.

Brassard, G. (1993). A Bibliography of Quantum Cryptography. *ACM SIGACT News*, pp. 16-20.

Brassard, G., & Salvail, G. (1994). Secret-key Reconciliation by Public Discussion. *Eurocrypt 1993*, (pp. 410-423). Lofthus, Norway.

Buttler, W. T., Torgerson, J. R., Nickel, G. H., Donahue, C. H., & Peterson, C. G. (2003). Fast, efficient error reconciliation for quantum cryptography. *Physical Review A, 67*(5).

Calver, T. I. (2011). An Empirical Analysis of the Cascade Secret Key Reconciliation Protocol for Quantum Key Distribution. *Masters Thesis.* Air Force Institute of Technology.

Chapman, N. (2012, Jan 4). *Very Sleepy*. Retrieved Jan 15, 2012, from Codernotes.com: http://www.codersnotes.com/sleepy

Chen, J., Dholakia, A., Eleftheriou, E., Fossorier, M. P., & Hu, X.-Y. (2005). Reduced-Complexity Decoding of LDPC Codes. *IEEE Transactions on Communications, 53*(7), 1232.

Chung, S.-Y., Forney, D. G., Richardson, T. J., & Urbanke, R. (2001). On the Design of Low-Density Parity-Check Codes within .0045 dB of the Shannon Limit. *IEEE Communications Letters*, *5*(2), 58-60.

Elkouss, D., Leverrier, A., Alleaume, R., & Boutros, J. J. (2009). Efficient Reconciliation Protocol for Discrete-Variable Quantum Key Distribution. *IEEE International Symposium on Information Theory*, (pp. 1879-1883). Seoul, Korea.

Elkouss, D., Martinex, J., Lancho, D., & Martin, V. (2010). Rate Compatible Protocol for Information Reconciliation: An Application to QKD. *IEEE Information Theory Workshop*, (pp. 1-5).

Gallager, R. (1962). Low-Density Parity-Check Codes. *IRE Transactions on Information Theory, 8*(1), 21-28.

Gudmundsen, M. (2010). Improved Secret Key Rate in Quantum Key Distribution using highly irregular Low-Density Parity-Check Codes. *Master's Thesis.* Norwegian University of Science and Technology.

Hu, X.-Y., Eleftheriou, E., & Arnold, D.-M. (2001). Progressive Edge-Growth Tanner Graphs. *Global Communications Conference*, (pp. 995-1001). San Antonio, TX.

Kasai, K., Matsumoto, R., & Sakaniwa, K. (2010). Information Reconciliation for QKD with Rate-Compatible Non-Binary LDPC Codes. *International Symposium on Information Theory and its Applications*, (pp. 922-927). Taichung, Taiwan.

Lin, Y.-K., Chen, C.-L. L.-C., & Chang, H.-C. (2008). Structured LDPC Codes with Low Error Floor based on PEG Tanner Graphs. *IEEE International Symposium on Circuits and Systems*, (pp. 1846-1849).

Luby, M., Mitzenmacher, M., Shokrollahi, A., & Spielman, D. (1998). Analysis of Low Density Codes and Improved Designs Using Irregular Graphs. *30th Annual ACM Symposium on Theory of Computing*, (pp. 249-258). New York, NY.

Luby, M., Mitzenmacher, M., Shokrollahi, A., Spielman, D., & Stemann, V. (1997). Practical Loss-Resilient Codes. *29th Annual ACM Symposium on Theory of Computing*, (pp. 150-159). El Paso, TX.

Lustic, K. (2011). Performance Analysis and Optimization of the Winnow Secret Key Reconciliation Protocol. *Master's Thesis.* Air Force Institute of Technology.

MacKay, D. J. (1999). Good Error-Correcting Codes Based on Very Sparse Matrices. *IEEE Transactions on Information Theory, 45*(2), 399-431.

MacKay, D. J., Mitchison, G., & McFadden, P. L. (2004). Sparse-Graph Codes for Quantum Error Correction. *IEEE Transactions on Information Theory, 50*(10).

MacKay, D., & Hu, X.-Y. (2011, 10 20). *Source Code for Progressive Edge Growth Parity-Check Matrix Construction*. Retrieved 11 1, 2011, from The Inference Group: http://www.inference.phy.cam.ac.uk/mackay/PEG_ECC.html

Matsumoto, M., & Nishimura, T. (1998). Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Transactions on Modeling and Computer Simulation, 8*(1), 3-30.

Matsumoto, R. (2009, Aug 14). *Problems in application of LDPC Codes to Information Reconciliation in Quantum Key Distribution Protocols.* Retrieved Nov 2011, from Cornell University Library: http://arxiv.org/abs/0908.2042v2

Mesiti, F., Delgado, M., Mondin, M., & Daneshgaran, F. (2010). Sparse-graph Codes for Information Reconciliation in QKD Applications. *Third International Symposium on Applied Sciences in Biomedical and Communication Technologies*, (pp. 1-5). Rome, IT.

Otmani, A., Tillich, J.-P., & Andriyanova, I. (2007). On the Minimum Distance of Generalized LDPC Codes. *IEEE International Symposium on Information Theory*, (pp. 751-755). Nice, France.

Pearl, J. (1982). Reverend Bayes on Inference Engines: A Distributed Hierarchical Approach. *The Second National Conference on Artificial Intelligence.* Pittsburgh, PA.

Rass, S., & Kollmitzer, C. (2009). Adaptive Error Correction with Dynamic Initial Block Size in Quantum Cryptographic Key Distribution Protocols. *Third International Conference on Quantum, Nano and Micro Technologies*, (pp. 90-95). Cancun, Mexico.

Richardson, T. J., & Urbanke, R. L. (2001). The Capacity of Low-Density Parity-Check Codes Under Message-Passing Decoding. *IEEE Transactions on Information Theory, 47*(2), 599-618.

Richter, G. (2005). An improvement of the PEG Algorithm for LDPC Codes in the Waterfall Region. *The International Conference on Computer as a Tool*, (pp. 1044-1047).

Rivest, R. L., Shamir, A., & Adleman, L. M. (1970, Jan 23). *Patent No. US3657476.* USA.

Shannon, C. E. (1948). A Mathematical Theory of Communication. *Bell System Technical Journal, 27,* 379-423, 623-656.

Shor, P. W. (1994). Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum-Computer. *35th Annual Symposium on Foundations of Computer Science*, (pp. 124-134). Santa Fe, NM.

Sugimoto, T., & Yamazaki, K. (2000). A Study on Secret Key Reconciliation Protocol "Cascade". *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, E83-A*(10), 1897-1991.

Tanner, R. (1981). A Recursive Approach to Low Complexity Codes. *IEEE Transactions on Information Theory, 27*(5), 533-547.

Trappe, W., & Washington, L. C. (2005). *Introduction to Cryptography with Coding Theory* (2 ed.). Upper Saddle River, NJ: Prentice Hall.

Wiesner, S. (1983). Conjugate Coding. *ACM SIGACT News - A Special Issue on Cryptography*, 78-80.

Wootters, W. K., & Zurek, W. H. (1982). A single quantum cannot be cloned. *Nature, 299*(5886), 802-803.

Yan, H., Ren, T., Peng, X., Lin, X., Jiang, W., & Liu, T. (2008). Information Reconciliation Protocol in Quantum Key Distribution System. *Fourth International Conference on Natural Computation*, (pp. 637-641). Jinan, China.

REPORT DOCUMENTATION PAGE			*Form Approved* *OMB No. 074-0188*
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of the collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to an penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**			
1. REPORT DATE *(DD-MM-YYYY)* 22-03-2012	**2. REPORT TYPE** Master's Thesis		**3. DATES COVERED** *(From – To)* August 2010 – March 2012
TITLE AND SUBTITLE An Analysis of Error Reconciliation Protocols for use in Quantum Key Distribution			**5a. CONTRACT NUMBER**
			5b. GRANT NUMBER
			5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Johnson, James S., 1st Lieutenant, USAF			**5d. PROJECT NUMBER**
			5e. TASK NUMBER
			5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way, Building 640 WPAFB OH 45433-8865			**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT/GCE/ENG/12-06
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Dr. Gerald Baumgartner Laboratory for Telecommunications Sciences 8080 Greenmead Drive College Park, MD 20740 (240) 373-2743			**10. SPONSOR/MONITOR'S ACRONYM(S)** LTS
			11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.			
13. SUPPLEMENTARY NOTES **This material is declared work of the U.S. Government and is not subject to copyright protection in the United States.**			

14. ABSTRACT

Quantum Key Distribution (QKD) is a method for transmitting a cryptographic key between a sender and receiver in a theoretically unconditionally secure way. Unfortunately, the present state of technology prohibits the flawless quantum transmission required to make QKD a reality. For this reason, error reconciliation protocols have been developed which preserve security while allowing a sender and receiver to reconcile the errors in their respective keys. The most famous of these protocols is Brassard and Salvail's Cascade, which is effective, but suffers from a high communication complexity and therefore results in low throughput. Another popular option is Buttler's Winnow protocol, which reduces the communication complexity over Cascade, but has the added detriment of introducing errors, and has been shown to be less effective than Cascade. Finally, Gallager's Low Density Parity Check (LDPC) codes have recently been shown to reconcile errors at rates higher than those of Cascade and Winnow with a large reduction in communication, but with greater computational complexity. This research seeks to evaluate the effectiveness of these LDPC codes in a QKD setting, while comparing real-world parameters such as runtime, throughput and communication complexity empirically with the well-known Cascade and Winnow algorithms. Additionally, the effects of inaccurate error estimation, non-uniform error distribution and varying key length on all three protocols are evaluated for identical input key strings. Analyses are performed on the results in order to characterize the performance of all three protocols and determine the strengths and weaknesses of each.

15. SUBJECT TERMS
Quantum Key Distribution, QKD, Error Reconciliation, Cascade, Winnow, LDPC, Low Density Parity Check Codes, Information Reconciliation, Secret Key

16. SECURITY CLASSIFICATION OF: UNCLASSIFIED			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Jeffrey Humphries, Lt Col, USAF
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER** *(Include area code)* (937) 255-6565 x7253 Jeffrey.Humphries@afit.edu
U	U	U	UU	178	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18