Practical Strategies for QKD Key Production[1]

Alan Mink and Anastase Nakassis
Information Technology Laboratory, National Institute of Standards and Technology,
100 Bureau Dr., Gaithersburg, MD 20899
amink@nist.gov, anakassis@nist.gov

Abstract

We present the quantum key distribution (QKD) secure key ratio expression in a form that exposes the parameters that affect the Reconciliation (error correction) stage. Reconciliation is the least well understood in practical terms and is typically described through a model that provides little guidance when it comes to efficient implementation, although it requires significant resources and is required to achieve a performance level commensurate with the other stages of the QKD protocol implementation. We addresses the issue of practical QKD error correction, questions of performance based on our data and data that have been published, and addresses the issue of platforms capable of handling rates of Gb/s.

Keywords: Quantum key distribution, QKD reconciliation, Entropy, LDPC codes, Polar codes

1. Introduction

Over the nearly three decades that followed the introduction of the Quantum Key Distribution (QKD) protocol [2], our understanding of the most efficient attack by Eve on the non-secured quantum channel has evolved and the Quantum Bit Error Rate (QBER) parameter remains as a significant measureable parameter. It has been proven [16] that one can design QKD systems such that the asymptotic ratio of the secret bits, **K**, over sifted bits, **N**, is:

$$[S(X|E) - H(X|Y)]$$

where **S(X|E)** and **H(X|Y)** represent, respectively, Eve's ignorance and Bob's Entropy for the typical qbit value **X**, Eve's measurement **E** and Bob's measurement **Y**.

Real QKD systems work with strings of finite length for post processing (error correction and privacy amplification). That is, Bob will start with a string **y** of **N** sifted bits, and will attempt with Alice's help to correct the errors in **y**. Successful correction of a long y will provide a reasonable estimate of the QBER which can be communicated to Alice either in the clear (with appropriate adjustments in **S**) or protected with secret bits that Alice and Bob already share. Lower bound corrections of **S(X|E)** for finite length data have been derived [17], but a more practical version based on [9] is of the following form:

$$K = (N-V) * [S(X|E) - g*h(p_{est})] - C$$

where **K** is the length of the remaining secrecy of the **N-V** sifted bits after error correction, **h** is the Shannon entropy, $\mathbf{p_{est}}$ is the estimate of the QBER that is obtained through sampling (and discarding) **V** sifted bit values, **g** is an efficiency factor (**g**>1) such that $\mathbf{g*h(p_{est})}$ equals the amount of information exchanged over the classical channel to accomplish

[1] The identification of any commercial product or trade name does not imply endorsement or recommendation by the National Institute of Standards and Technology. Any software code/algorithm is expressly provided "AS IS." NIST makes no warranty of any kind, express, implied, in fact or arising by operation of law, including, without limitation, the implied warranty of merchantability, fitness for a particular purpose, non-infringement and data accuracy.

Quantum Information and Computation XI, edited by Eric Donkor, Andrew R. Pirich, Howard E. Brandt,
Proc. of SPIE Vol. 8749, 874908 · © 2013 SPIE · CCC code: 0277-786X/13/$18
doi: 10.1117/12.2015965

error correction and **C** is a fixed number of bits used for hash signatures to verify error correction and to report the number of errors corrected so that a better estimate of the QBER can be obtained. Because we concentrate on the BB84 protocol $\mathbf{S(X|E) = 1 - h(QBER) \geq 1 - h(p_{sft})}$ with great likelihood, where $\mathbf{p_{sft}}$ is an upper likelihood bound of the QBER over the **N-V** bits. Privacy amplification over the **N-V** bits is possible when **K** is positive and exceeds any constants further deducted during privacy amplification. If **K**>0 but doesn't exceed the privacy amplification constants, the **N-V** bits can be combined with other batches so that the total secrecy of the combined string will allow for privacy amplification.

The typical description of the BB84 protocol assumes that some bits, **V**, are sampled so as to determine an estimate for the QBER, but in general neglects to also mention error correction failures. As a result, the following will measure the average bit-rate efficiency of the error correction scheme:

$$K_{avg} = (1-f) * \{(N-V) * [S(X|E) - g*h(p_{est})] - C\} \quad (1)$$

where **f** is the failure rate (≤ 1.0) of the error correction scheme.

Once the system is designed and implemented, other than calibration of the photonics, little can be done operationally to optimize the amount of secure key production due to **S(X|E)** of the quantum channel. Error correction exposes information to an eavesdropper and impacts the amount of secure key production. But error correction is one area that allows some strategic decisions that can influence the resulting amount of secure key production. Decisions on how error estimation is conducted will influence the parameters in Eq(1). The choice of an error correction scheme is among those choices. **H(X|Y)** represents the minimum entropy required to correct Bob's sifted bits, given the QBER, but codes with that level of entropy may not exist and one has to choose from those that are available. We shall therefore outline, in what follows, modifications to the typical QKD exposition which can improve the performance of the overall system. As research pursues the next QKD level of Gb/s secure key rate [4], highly optimized and parallel implementations of these schemes will be required. We also discuss a range of candidate platforms for such parallel implementations.

2. Practical Consideration for QKD Post Processing

The typical treatment of the error correction phase relies on the following assumptions:

1) That the QBER will be estimated by sampling a subset, **V**, of the data to be error-corrected,
2) That a batch of **N-V** data will be error corrected and the same size batch will be subsequently privacy amplified
3) That error correction will always succeed, and
4) The bounds on **S(X|E)** will be estimated through the QBER obtained in step 1.

None of these assumptions need to be true. Because:

- Real Systems may operate in environments such that the QBER does not change much from batch to batch. While operating in such a mode, it may be preferable to estimate, for error correction purposes, the current QBER from the observed errors in previous batches.
- The error correction scheme does not necessarily work with strings of arbitrary length and it may be more efficient to operate on small segments of the data and then concatenating several error-corrected bit strings into longer bit strings to be privacy-amplified.
- One shot error correction may not succeed with high probability unless the choice of the error-correcting scheme is particularly conservative; as a result the average key performance may suffer. In particular, for high QBER values conservative error-correction may fail to produce a secret.
- Once a batch of data has been corrected, Bob knows exactly how many bits were in error. Rather than performing elaborate evaluations based on the estimate in step 1) above, it is much simpler to have Bob inform Alice of this number which should be used to evaluate **S(X|E)**. When sending this number it should protected with the existing secret bits that the QKD application already has or we should adjust **C** in Eq(1) appropriately.

We must also observe that for conventional communication, data and the error correcting information are put together prior to transmission and are sent over the same error prone channel. QKD error correction takes place after sifting the

received qbit data and then the error correction information is sent separately over the classical channel which provides data integrity and is considered error-free.

Typically, the QBER is estimated by sampling a subset, of size **V**, of the **N** bits to be error-corrected. The question is how many bits must one sample to get a good estimate. In the QKD reconciliation context, good is relative to error correction ranges. If the bounds on the estimate can span two correction regions, then the wrong choice of region will adversely affect key production. If we sample sufficiently many points (typically more than 120) the t-distribution and the normal distribution differ little; moreover, the t-distribution is heavy-tailed and assigns greater probabilities to values above or below a cut-off value. If **p*** is the percentage of errors in a randomly chosen subset of **V** bits, the actual mean, QBER, will be approximately normally distributed with mean **p*** with standard deviation **σ=sqrt(p*(1-p*)/(V-1))**. As a result, for small **V** values, **σ** tends to dominate QBER. The 95% confidence level falls within **p***±1.96**σ** and the with 99% confidence level falls within **p***±2.58**σ**. As an example if **p***=3% and **V**=901, **σ**=0.0057; then the expected 99% range for QBER is between 1.6% and 4.4%. A QBER below 2%, could cause a higher amount of error correction information to be sent thus yielding a lower secure key ratio. Alternatively a QBER above 4% may cause an insufficient amount of error correction information to be sent, thus resulting in no secure key ratio for that batch. If we desire a ±0.5% range on our estimates, then that would require a **V**=7,719 and a ±0.25% range would require **V**=30,873.

It is common for a QKD system to operate at a relatively constant QBER, drifting slowly as the photonics may slide out of calibration or from diurnal temperature variations. In these cases when the observed **p***-value appears to change little from batch to batch, it may be preferable to estimate **p*** from past data. For example, if **p*** varies by ±0.25% from batch to batch, then it is reasonable to assume steady state behavior and obtain an initial estimate for the QBER and use the number of errors corrected in the previous batch to update the current QBER estimate. If one estimates the QBER based on the previous batch of data, this would reduce **V**, in Eq(1), to zero, creating an immediate gain in **K**. If there were a large increase in the QBER, then most likely the error correction would fail and an appropriate correction will be applied to the next batch.

As a rule, it is not a good idea to apply the QBER estimate used for error correction in order to obtain a high likelihood QBER upper bound that will be used to evaluate **S(X|E)** during privacy amplification. Once a batch of sifted bits has been successfully error-corrected, Bob knows the exact number of errors within the batch he corrected and it will cost him at most $\mathbf{\log_2 (N/8) = \log_2 (N)-3}$ bits, given the acceptable QBER range. Therefore, better and tighter QBER estimates for **S(X|E)** become possible rather than the scheme described by Hayashi [9].

The privacy amplification mechanism does not really care how exactly the error correction was actually carried out. So one can error correct in multiple small batches and concatenate those batches into a larger one for privacy amplification. The caveat here is that the statistics must be kept for each error correction batch. The batch statistics needed are the number of errors corrected and the amount of correction information, **N*g*h(p) + C** from Eq(1). These statistics need to be properly combined for the resultant string to be privacy amplified.

3. Error Correction Schemes

The first QKD error correction scheme, Cascade [3], was interactive (multiple round trips that refine information on the data). Interactive schemes do not necessarily need an a priori QBER estimate and the NIST implementation of Cascade [14] did not. For a number of reasons there has been a movement away from Cascade. This may be due to the fact that the data driven nature of Cascade may be leaking information and that multiple trips incur additional latency.

The current replacement for Cascade seems to be the low density parity check (LDPC) scheme [8,13] that can asymptotically approach the Shannon limit. Nevertheless, for real applications, the publicly available LDPC matrices defined for conventional communication that can be adopted for QKD do not appear to be anywhere near the Shannon limit for QKD error rates. These matrices restrict **N** to specific sizes and although they can be scaled (our limited experiments indicate that scaling doesn't change their error correction properties), **N** is still restricted to specific values. A disadvantage of LDPC is that as the error rate increases, different matrices are needed to cover different parts of that range. Thus custom QKD matrices may need to be specified for each discrete QBER value range (e.g., 1%, 2%, … , 11%). Matrices are referenced in terms of their code rate, which is **N/(|CS|+N)**, where **N** is the length of the sifted bits to be corrected and **|CS|** is the length of the correction information exchanged between Alice and Bob.

The latest candidate for error correction is Arikan's polar codes [1]. These codes have been proven to approach the Shannon limit as **N** tends to infinity, provided that they are well designed. There are guidelines for this, but not definitive construction methods, especially for QKD. They operate on **N** values that are powers of two and tend to have relatively high rates of failed error correction.

For discussion here, we focus on LDPC codes and the appendix summarizes the LDPC algorithm along with the pseudo code for two implementations. One implementation uses floating point multiply and divide that is a good fit for computer software. The other implementation uses logarithm table lookups with integer addition and subtraction, and is a good fit for hardware such as field programmable gate arrays (FPGAs) and applications specific integrated circuits (ASICs).

Table 1. QKD LDPC Evaluation for ETSI DVB Matrices.

ETSI DVB Matrices			Failures/Avg iterations/Mb/s - per 1000 samples (Max 31 iterations for Convergence)								
Calculation Mode	1%	2%	3%	4%	5%	6%	7%	8%	9%	10%	11%
ETSI DVB, Rate 5/6 (54000 bits, 10800 Chksums)											
Product (Server)	2/7.8/0.56	51/2.2/0.18	1000/31.0/0.13								
4k table (Server)	1/7.5/0.96	28/21.1/0.31	1000/31.0/0.20								
Product (Laptop)	2/7.8/0.40	51/22.0/0.13	1000/31.0/0.10								
4k table (Laptop)	1/7.5/0.70	28/21.1/0.23	1000/31.0/0.15								
ETSI DVB, Rate 3/4 (48600 bits, 16200 Chksums)											
Product (Server)	0/4.8/0.96	0/6.7/0.64	0/9.4/0.44	0/15.0/0.26	997/31.0/0.12						
4k table (Server)	0/4.7/1.59	0/6.6/1.06	0/9.3/0.71	0/14.9/0.42	996/31.0/0.19						
Product (Laptop)	0/4.8/0.71	0/6.7/0.47	0/9.4/0.32	0/15.0/0.19	997/31.0/0.09						
4k table (Laptop)	0/4.7/1.18	0/6.6/0.78	0/9.3/0.52	0/14.9/0.31	996/31.0/0.14						
ETSI DVB, Rate 2/3 (43200 bits, 21600 Chksums)											
Product (Server)	0/4.0/1.17	0/5.0/0.89	0/5.8/0.73	0/7.0/0.60	0/8.3/0.49	0/10.1/0.39	0/13.1/0.29	2/19.4/0.19	994/30.9/0.12		
4k table (Server)	0/4.0/1.90	0/4.9/1.46	0/5.8/1.18	0/6.9/0.95	0/8.2/0.77	0/10.1/0.61	0/13.0/0.46	0/19.3/0.30	994/30.9/0.18		
Product (Laptop)	0/4.0/0.86	0/4.9/0.66	0/5.8/0.55	0/7.0/0.44	0/8.3/0.36	0/10.1/0.29	0/13.1/0.22	2/19.4/0.14	994/31.0/0.09		
4k table (Laptop)	0/4.0/1.41	0/4.9/1.08	0/5.8/0.88	0/6.9/0.70	0/8.3/0.57	0/10.1/0.45	0/13.0/0.34	0/19.2/0.22	994/31.0/0.14		
ETSI DVB, Rate 3/5 (38880 bits, 25920 Chksums)											
Product (Server)	0/3.4/10.04	0/4.1/0.79	0/4.6/0.68	0/5.1/0.59	0/5.9/0.50	0/6.5/0.44	0/7.5/0.37	1/9.0/0.30	0/11.7/0.23	83/21.5/0.12	1000/31.0/0.0
4k table (Server)	0/3.4/1.61	0/4.0/1.28	0/4.6/1.07	0/5.1/0.93	0/5.9/0.78	0/6.5/0.68	0/7.5/0.57	0/9.0/0.46	0/11.6/0.35	61/21.2/0.18	1000/31.0/0.1
Product (Laptop)	0/3.4/0.75	0/4.0/0.59	0/4.6/0.50	0/5.1/0.44	0/5.9/0.37	0/6.5/0.33	0/7.5/0.28	1/9.0/0.23	0/11.6/0.17	83/21.4/0.09	1000/31.0/0.0
4k table (Laptop)	0/3.4/1.20	0/4.0/0.95	0/4.6/0.80	0/5.1/0.96	0/5.9/0.58	0/6.5/0.51	0/7.5/0.43	0/9.0/0.35	0/11.6/0.26	61/21.2/0.14	1000/31/0.10

Table 2. QKD LDPC Evaluation for IEEE 11n Matrices

IEEE 11n Matrices		Failures/Avg iterations/Mb/s - per 1000 samples (Max 31 iterations for Convergence)								
Calculation Mode	1%	2%	3%	4%	5%	6%	7%	8%	9%	10%	11%
IEEE 11n, Rate 5/6 (1620 bits, 324 Chksums)											
Product (Server)	1/4.6/1.60	421/19.4/0.33	988/30.9/0.22								
4k table (Server)	1/4.6/1.92	419/19.4/0.38	988/30.9/0.25								
Product (Laptop)	1/4.6/0.84	420/19.4/0.17	988/30.9/0.11								
4k table (Laptop)	1/4.6/1.38	419/19.4/0.28	988/30.9/0.18								
8-bit GPU (14x512)	12/5.1/22.04	441/20.1/6.89	989/30.9/4.52								
IEEE 11n, Rate 3/4(1458 bits, 486 Chksums)											
Product (Server)	0/3.2/2.68	0/4.6/1.58	2/7.1/0.92	203/15.9/0.37	814/28.4/0.21						
4k table (Server)	0/3.2/2.53	0/4.6/1.61	3/7.1/0.97	205/15.9/0.40	815/28.4/0.22						
Product (Laptop)	0/3.2/1.20	0/4.6/0.75	3/7.1/0.45	203/15.9/0.19	814/28.4/0.11						
4k table (Laptop)	0/3.2/1.80	0/4.6/1.20	3/7.1/0.72	205/15.9/0.30	815/28.4/0.17						
8-bit GPU (14x512)	0/3.3/31.71	1/4.7/24.36	3/7.2/17.02	202/16.0/8.67	815/28.4/4.96						
IEEE 11n, Rate 2/3 (1296 bits, 648 Chksums)											
Product (Server)	0/2.9/2.30	0/3.5/1.73	0/4.3/1.36	0/5.1/1.08	0/6.5/0.82	10/8.9/0.57	144/15.1/0.33	625/25.4/0.20	944/30.4/0.17		
4k table (Server)	0/2.9/2.68	0/3.5/2.03	0/4.3/1.57	0/5.1/1.24	1/6.5/0.93	10/8.9/0.65	142/15.0/0.37	621/25.3/0.22	944/30.4/0.18		
Product (Laptop)	0/2.9/1.26	0/3.5/0.94	0/4.3/0.76	0/5.1/0.59	0/6.5/0.45	10/8.9/0.32	145/15.1/0.18	625/25.3/0.11	944/30.4/0.09		
4k table (Laptop)	0/2.9/1.98	0/3.5/1.48	0/4.3/1.17	0/5.1/0.92	1/6.5/0.70	10/8.9/0.49	142/15.0/0.27	621/25.3/0.16	944/30.4/0.14		
8-bit GPU (14x512)	0/2.9/28.64	0/3.6/23.83	0/4.3/21.24	0/5.2/17.85	0/6.5/15.04	8/8.9/11.26	140/15.1/7.07	615/25.3/4.28	942/30.3/3.58		

Although these are efficient algorithms, their implementations have not been optimized and there is room for improvement in their execution performance. The results presented here are meant for comparison and demonstration purposes rather than operational performance. The integer table lookup implementation with 12-bits of precision performs error correction as well as the floating point implementation with full float precision. The Laptop and Server implementations are identical and the only difference is the speed of the Server, which enhances the execution performance by about double, but there is no difference in the error correction performance.

We employ a selective set of matrices from standards groups that incorporate LDPC and have published their family of acceptable matrices, such as the IEEE 802.11n Std [10], IEEE 802.16e Std [11] and ETSI DVB Std [6]. We present an evaluation of both the error correction performance and computation performance of these matrices for a QKD environment. Tables 1-3, show the QKD performance for the largest defined LDPC matrices from ETDI DVB, IEEE 802.11n and IEEE 802.16e that were run on a laptop, a Server and a graphics processing unit (GPU). Porting these implementations to the GPU required major revisions in the supporting data structures, which imposed a limit on the matrix size. Thus, there are no entries for the ETSI DVB matrices run on the GPU since the fast local memory for our GPU wasn't large enough to hold the needed mapping tables and placing them in slow global memory would significantly impact execution rates. Each column of a table is the mean QBER of the generated data for which the matrix was being evaluated. Each table entry is the result of 1000 randomly chosen datasets and consists of three values, the number of attempts that failed to converge (i.e., failed to succeed in correction), the average number of iterations to converge and the data processing rate in Mb/s. The failed rate is never zero, there is always some "absorbing" states that the codes would gravitate towards for various error patterns. Thus a zero for failed to converge just means mean a low failure rate and indicates a successful operation point for QKD, not necessarily a good operating point in terms of coding efficiency and secure key ratio. Scanning the tables from left to right, we look for the transition from low failure rate to high failure rate that indicates where a given matrix is no longer effective. This is shown graphically in the plot of Fig. 2 for the ETSI DVB matrices. Both ETSI and IEEE also uses a secondary error correction scheme along with LDPC to correct the (assumed) few remaining errors if LDPC fails. If the remaining errors exceed the few expected, than the secondary code could fail also. ETSI uses a BCH code and IEEE uses a Turbo code. Elkouss, et al [5] have also incorporated a secondary BCH code in their QKD reconciliation for this purpose. These bits would be included in the C factor of Eq(1), but BCH only needs to be executed if LDPC fails.

Table 3. QKD LDPC Evaluation for IEEE 16e Matrices

IEEE 16e Matrices	Failures/Avg iterations/XEQ time (sec) - per 1000 samples (Max 31 iterations for Convergence)										
Calculation Mode	1%	2%	3%	4%	5%	6%	7%	8%	9%	10%	11%
IEEE 16e, Rate 5/6 (1920 bits, 384 Chksums)											
Product (Server)	1/4.7/1.56	418/19.9/0.31	989/30.9/0.22								
4k table (Server)	1/4.6/1.83	417/2.0/0.37	989/30.9/0.24								
Product (Laptop)	1/4.6/0.84	418/19.9/0.17	989/30.9/0.11								
4k table (Laptop)	1/4.6/1.35	417/19.8/0.27	989/30.9/0.18								
8-bit GPU (14x512)	4/5.0/25.05	443/20.7/7.27	990/30.9/4.93								
IEEE 16e, Rate 3/4 A (1728 bits, 576 Chksums)											
Product (Server)	0/3.2/2.21	0/4.6/1.29	9/7.8/0.76	367/19.4/0.28	941/30.2/0.20						
4k table (Server)	0/3.2/2.61	0/4.6/1.60	9/7.8/0.86	365/19.4/0.33	941/30.2/0.21						
Product (Laptop)	0/3.2/1.23	0/4.6/0.75	9/7.7/0.41	367/19.4/0.15	941/30.2/0.10						
4k table (Laptop)	0/3.2/1.91	0/4.6/1.19	9/7.8/0.64	365/19.4/0.24	941/30.2/0.16						
8-bit GPU (14x512)	0/3.2/28.85	0/4.6/21.10	8/7.8/14.03	364/19.5/6.05	941/30.2/3.99						
IEEE 16e, Rate 3/4 B (1728 bits, 576 Chksums)											
Product (Server)	0/3.3/2.06	0/4.7/1.32	1/7.1/0.80	208/16.3/0.33	851/29.0/0.19						
4k table (Server)	0/3.3/2.37	0/4.7/1.52	2/7.1/0.93	208/16.2/0.38	851/29.0/0.21						
Product (Laptop)	0/3.3/1.13	0/4.7/0.72	1/7.1/0.44	208/16.3/0.18	851/29.0/0.10						
4k table (Laptop)	0/3.3/1.79	0/4.7/1.14	2/7.1/0.69	208/16.2/0.28	851/29.0/0.16						
8-bit GPU (14x512)	0/3.4/26.39	0/4.7/20.42	2/7.2/14.01	212/16.4/6.78	851/29.1/3.90						
IEEE 16e, Rate 2/3 A (1536 bits, 768 Chksums)											
Product (Server)	0/2.9/2.48	0/3.6/1.83	0/43.3/1.47	0/5.2/1.15	0/6.5/0.88	3/8.9/0.63	101/14.7/0.36	543/24.5/0.22	917/30.2/0.18		
4k table (Server)	0/2.8/2.93	0/3.6/2.17	0/4.3/1.71	0/5.2/1.34	0/6.5/1.02	3/8.8/0.72	102/14.6/0.42	543/24.4/0.25	917/30.2/0.20		
Product (Laptop)	0/2.9/1.41	0/3.6/1.01	0/4.3/0.81	0/5.2/0.64	0/6.5/0.49	3/8.8/0.35	101/14.6/0.20	543/24.4/0.12	917/30.2/0.10		
4k table (Laptop)	0/2.8/2.14	0/3.6/1.59	0/4.3/1.26	0/5.2/0.99	0/6.5/0.75	3/8.8/0.53	102/14.6/0.31	543/24.4/0.18	917/30.2/0.15		
8-bit GPU (14x512)	0/2.9/32.7	0/3.6/27.68	0/4.4/23.25	0/5.3/20.51	0/6.6/16.70	3/8.9/12.95	101/14.6/8.37	538/24.5/5.09	915/30.2/4.13		
IEEE 16e, Rate 2/3 B (1536 bits, 768 Chksums)											
Product (Server)	0/2.8/2.57	0/3.4/1.88	0/4.3/1.44	0/5.5/1.06	11/8.0/0.70	159/14.3/0.38	564/24.0/0.23	923/30.0/0.19	997/31.0/0.18		
4k table (Server)	0/2.8/3.08	0/3.4/2.24	0/4.3/1.70	0/5.5/1.25	12/8.0/0.80	159/14.3/0.43	566/24.0/0.25	921/30.0/0.20	997/31.0/0.20		
Product (Laptop)	0/2.8/1.45	0/3.4/1.07	0/4.3/0.80	0/5.5/0.59	11/8.0/0.39	159/14.3/0.21	564/24.0/0.12	923/30.0/0.10	997/31.0/0.10		
4k table (Laptop)	0/2.8/2.19	0/3.4/1.67	0/4.3/1.25	0/5.5/0.92	12/8.0/0.60	159/14.3/0.32	566/24.0/0.19	921/30.0/0.15	997/31.0/0.15		
8-bit GPU (14x512)	0/2.8/32.31	0/3.4/29.02	0/4.3/23.64	0/5.5/19.66	8/8.0/14.86	154/14.2/8.79	558/23.9/5.46	919/30.0/4.32	997/31.0/4.12		

Choosing the proper error correction parameters (such as the LDPC matrix) is based on good QBER estimation as mentioned earlier. The consequences of such choices when LDPC correction is used are graphically depicted by Elkouss, et al [5] in an efficiency plot of the error-correction scheme, as a function of the QBER. This plot resembles a saw tooth shape with sharp jumps when passing from one LDPC matrix to the next. Fig 1 shows this type of efficiency plot for four of the ETSI DVB Matrices, where the label M_a/b represents the curve for the matrix of rate a/b. For

example M_5/6 is the curve for the 5/6 rate matrix that can cover up to a ~2% error rate. Elkouss uses eight matrices to span the QBER range (i.e., 1% - 11%). Additional information is needed to go along with an efficiency plot, a failure rate plot as shown in Fig 2. The failure plot indicates how effective the correction is for each matrix as a function of the QBER and shows the failure transition, where the matrix no longer corrects errors. There is a small QBER range, of 1% or less, in which a given matrix begins to fail but still error corrects most of the time. For example, in Table 2 for the 802.11n matrices at a 4% QBER, one can choose the 3/4 rate matrix that exposes less information but fails about 20% of the time or the 2/3 rate matrix that exposes more information but has a low failure rate.

Elkouss also applies an idea proposed by Renner [16]. When the estimated QBER is near the bottom of the range that the current parameters are able to correct, it is preferable that Alice and Bob do not converge on Alice's string **x**. Instead, Alice constructs **x´** by inserting random errors in **x** and error correction is targeted to recreate **x´**. This will not impact Bob's error correction but Eve will know less about **x´** than she knew for **x**. For example, if we were using the IEEE 802.11n 3/4 rate matrix from Table 2 over a 2% to 4% QBER range and our estimated QBER is 2.5%. We could randomly add an additional 1% of errors to Alice's sifted bits, **x**, thus increasing the QBER of this new string, **x´**, to ~3.5%. Alice would then consider this new string, **x´**, to be her sifted bits and send Bob the associated error correction information based on **x´**. Bob would be able to correct his sifted bits to **x´**, without changing his LDPC matrix and without exposing any additional error correction information that would reduce secure key production, while Eve would be faced with an additional ~1% of errors.

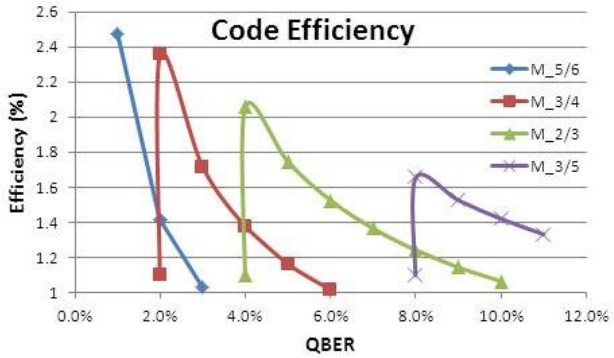

Fig 1. Error Correction Efficiency for ETSI DVB Matrices.

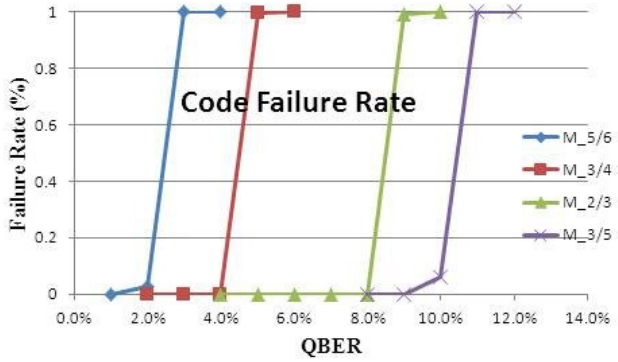

Fig 2. Code Failure Rates for ETSI DVB Matrices.

Both Elkouss and Jouguet, et al [12] provide data concerning the performance of their error correction implementations, LDPC and Polar codes respectively. Jouguet proposes as a measure of coding efficiency, the achievable ratio, which he expresses as

$$K/N \approx (1-f) * [\beta * I(x:y) - S(x:E)]$$

where **I(x:y)** and **S(x:E)** measure Bob's and Eve's information concerning **x**, respectively, **I(x:y) =1-h(p)** and for large **N**, **β≈ (1-g*h(p)/(1-h(p))** where **f**, **g** and **h(p)** are the same as in Eq(1). Note that **g*h(p)** is the amount of error correction information sent to Bob. Jouguet provides an example of LDPC performance operating over $\mathbf{N}=2^{17}$ bits for a 2% QBER, LDPC convergence probability of 0.99, and a **β** value equal to 0.929. Following Elkouss for BB84, **H(X|Y)=1-S(X|E)=h(p)**, which for a 2% QBER is h(0.02)~0.14 and plugging these values into the above equation yields the expected secret key ratio as

$$K/N \approx 0.99 * [0.929 * 0.86 - 0.14] = 0.652.$$

For a Polar code using $\mathbf{N}=2^{16}$ bits employed by Jouguet, using the same 2% QBER, convergence was attained with probability 0.91 and a **β** =0.935. Again, substituting in the above equation yields the expected secret key ratio of

$$K/N \approx 0.91*[.935*0.86 - 0.14] = 0.604$$

On the surface, at least, and for **N**=64 Kbits or 128 Kbits the Polar code does not appear to outperform the corresponding LDPC code in producing more secure key. Moreover, using the parameters Jouguet reports for dataset sizes **N**=2^{20} and 2^{24} bits, **β** values of 0.963 and 0.98 and error correction failure rates of 11% and 8%, respectively. Substituting these values in the above equation, yields secret output ratios equal to 0.617 and 0.646, respectively, which still lag Jouguet's LDPC data. It also lags the LDPC results of **K/N**~.7, reported by Elkouss. Using LDPC and the ETSI DVB 5/6 rate matrix with 54,000 information bits and 10,800 checksums, that converges with probability ~97% for a 2% QBER and a corresponding **β** of 0.932. Substituting these values in the above equation, yields secret key ratios equal to 0.642. Hence, LDPC matrices designed for environments radically different than those of QKD, are **β**-wise competitive at the 2% QBER. The benefit of Jouguet's Polar code implementation appears to be a higher execution performance on a computer, about an order of magnitude faster than the comparable LDPC computer performance, but about the same as an LDPC GPU implementation.

It is of course regrettable that the data about the polar code were limited to the 2% QBER. At high error rates, when **h(p)** is close to 0.5, the **β** value will determine if any secret can be extracted. It would therefore be interesting to know how the Polar codes behave for QBERs above 8%. For example, from Table 1, the ETSI DVB 3/5 rate matrix exposes **g*h(p)= |CS|/N**=25920/38880~.67 bits of information. At a 9% QBER, **β**=0.59 and **S(X|E)=1-h(9%)**~0.56. Thus there are no secure keys produced since **S(X|E) - g*h(p)** = 0.56 – 0.67= -0.11 < 0.

4. Platform Strategies

As research pursues the next QKD level of Gb/s secure key range [4] highly optimized and parallel implementations will be required to handle QKD post processing. Since the one-way post processing algorithms tend to be coarse grained computations in that a given data set does not need to communicate until a solution is obtained such methods are amenable to parallel implementation. Each data set can be assigned to a separate computation engine, which would operate independently in parallel, and each result could be collected sequentially at completion to maintain their order. Maintaining synchronization between Alice and Bob's bits is required throughout these operations. We discuss a number of options for such platforms.

Parallel processing, using multiple computers, is the easiest and most flexible platform for implementation. Because we are dealing with software and large amounts of memory, it's easy to change parameters and algorithms. Scaling with multiple PCs becomes bulky and difficult to control. A small supercomputer can be sized to attain Gb/s rates, but can be expensive.

The next level of complexity for implementing parallel processing is a programmable attached processor. Examples of these are a graphic processing unit (GPU) or field programmable gate array (FPGA) both of these are integrated with associated circuitry on a printed circuit board (PCB). These devices are connected to a computer via a bus interface or a communication link, along with the necessary access protocol. GPUs consist of a number of multiprocessors each with a number of processing elements. Each multiprocessor is of a single instruction multiple data (SIMD) architecture. This means that each multiprocessor executes a single instruction (such as a multiply) and each processing element would execute that instruction on a different data element (such as different elements of a vector). Current GPUs contain up to a few 10s of multiprocessors each with less than 100 SIMD processing elements. A GPU usually connects to a computer by plugging into one of its buses (e.g., PCIe) and transfers data and instructions between computer memory and GPU memory. Although the GPU memory is quite large, its access time by the GPU multiprocessors is long (e.g., 100s of GPU clock cycles). High speed memory is limited to small amounts of memory dedicated to individual GPU multiprocessors that can limit performance. Programming can be accomplished though variations of the C programming language. Up to 100 Mb/s of LDPC error correction performance [7] has been reported for a GPU implementation. Thus 5 computers with 2 GPUs each could error correct at a Gb/s rate.

FPGAs provide a different type of platform. Instead of processors and instruction, they provide basic logic constructs (building blocks) from which unique processing elements can be assembled that can operate in parallel. Programming is accomplished in a high level language that incorporates the concept of time to distinguish between sequential and parallel logical operations. FPGAs have limited memory resources, less than 10 Mbytes. LDPC requires about an order of magnitude more memory than Cascade does. We have implemented both Cascade and LDPC algorithms in our FPGA environment. Our single FPGA chip can support 4 threads of cascade, but only 2 threads of LDPC. For our

implementation of Cascade we have measured performance up to 5 Mb/s per thread of sifted bits and about half of that for each LDPC Thread. Manufacturers offer FPGA based evaluation PCBs for about the same cost of a GPU PCB, a few thousand dollars. Scalability to Gb/s for evaluation PCBs is not feasible, but they are sufficient to prove the feasibility of fast implementations. Potential FPGA designs could attain performance up to 30 Mb/s per thread. Thus, developing a custom PCB with multiple large FPGAs may be able to attain Gb/s performance. Such custom designs become expensive, one or two orders of magnitude more expensive than evaluation PCBs or GPUs.

Finally, we consider application specific integrated circuits (ASIC), which are custom chips. These devices are similar to custom FPGAs, but are more expensive in that they require a custom PCB design in addition to the design of the chip itself. Similarly a bus or communication link is necessary to interface to a computer. Such chips are not programmable, but can support algorithm variations through parameter specifications that are passed to the ASIC. They can provide better performance than FPGAs since they can be designed to operate at higher clock rates and provide optimized storage and computation resources. LDPC performance of 47 Gb/s [18] has been reported for 10 Gb/s Ethernet. It's not clear whether this chip could operate with appropriate QKD matrices, but that chip indicates that such designs are possible.

5. Conclusion

We presented a set of QKD error-correction options that are typically passed over when a proof-of-concept is offered but are important when real systems are designed. Namely, we described performance enhancing options such as alternatives to the continuous bit-error-estimation and the notion that the error-corrected bit-batches are in one to one correspondence with the bit batches to be privacy amplified. We discuss practical measures of performance; such as throughput per unit of time, speed of execution, and error-correction failure probabilities. We presented and compared performance data for the LDPC and Polar codes in the context of QKD correction. Much of the published data covers low QBER values (2%) and the available data suggest that all schemes, including some of the publicly available LDPC matrices we tested, have similar results for such QBERs. It would have been useful to compare data at higher QBERs, especially values between 8% and 11% where (1) inefficient error correction could eliminate any secure key production (2) the bit-batches to be corrected must be bigger, and (3) the computation time per bit is longer. Finally, we discussed computational platforms for error correction capable of attaining Gb/s QKD rates and concluded that such rates require specially crafted hardware with high levels of parallelism.

References

[1] E. Arıkan, "Channel polarization: A method for constructing capacity achieving codes for symmetric binary-input memoryless channels," IEEE Trans. Inform. Theory, Vol. 55, pp. 3051–3073, 2009.

[2] C. H. Bennet and G. Brassard, "Quantum Cryptography: Public key distribution and coin tossing", Proc of the IEEE Intern'l Conf on Computers, Systems, and Signal Processing, Bangalore, India, 1984.

[3] G. Brassard and L. Salvail, "Secret-Key Reconciliation by Public Discussion", proceedings of Eurocrypt'94, Lecture notes in computer Science, 765, Spriger Verlag, pp 410-423, 1994.

[4] DARPA-BAA-12-42, "Quiness: Macroscopic Quantum Communications", May 15, 2012. <https://www.fbo.gov/index?s=opportunity&mode=form&id=6a3a61d577305f71d9be268925c4b201&tab=core&_cview=0>

[5] D. Elkouss, A. Leverrier, R. Alleaume, and J. Boutros, "Efficient reconciliation protocol for discrete-variable quantum key distribution", Proc 2009 IEEE international Symposium on Information Theory – Vol. 3, ISIT'09, IEEE Press, Piscataway, NJ, USA, pp 1879–1883, 2009.

[6] ETSI EN 302 307 V1.2.1 (2009-08), "Digital Video Broadcasting (DVB); Second generation framing structure, channel coding and modulation systems for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications (DVB-S2)", Sect 5.3 and Annex B, 2009.

[7] G. Falcao, V. Silva, and L. Sousa. "How GPUs can outperform ASICs for fast LDPC decoding", Proc of the 23rd International Conference on Supercomputing, pp 390–399. ACM, 2009.

[8] R. Gallager, "Information Theory and Reliable Communication", New York: Wiley, 1968.

[9] M.Hayashi and T.Tsurumaru, "Concise and tight security analysis of the Bennett-Brassard 1984 protocol with finite key length", New Journal of Physics **14** (2012) 093014 (39pp) <http://iopscience.iop.org/1367-2630/14/9/093014/>
[10] IEEE Std 802.11n 2009, "Wireless LAN Medium Access Control & Physical Layer Specification", Part 11, Amendment 5, Annex R, Oct. 2009.
[11] IEEE Std 802.16e 2005, "Air Interface for Fixed and Mobile Broadband Wireless Access Systems", Part 16, Amendment 2, Annex H, Feb. 2006.
[12] P. Jouguet and S. Kunz-Jacques, "High Performance Error Correction for Quantum Key Distribution using Polar Codes", arXiv:1204.5882v2 [quant-ph], Jan 2013.
[13] D. MacKay, "Good Error-Correcting Codes Based on Very Sparse Matrices", IEEE Trans Information Theory, Vol. 45, N0. 2, pp 399-431, Mar 1999.
[14] A. Nakassis, J. Bienfang, and C. Williams, "Expeditious reconciliation for practical quantum key distribution", Proc. SPIE: Quantum Information and Computation II, #5436, 2004.
[15] A. Nakassis and A. Mink, "LDPC error correction in the context of Quantum Key Distribution", Proc. SPIE: Defense Security & Sensing, Balt., MD, Apr. 2012.
[16] R. Renner, N. Gisin, and B. Kraus, "Information-theoretic security proof for quantum-key-distribution protocols", Phys. Rev. A, Vol. **72**, No. 1, 012332, July 2005. <http://pra.aps.org/pdf/PRA/v72/i1/e012332>
[17] M. Tomamichel, C. C. Lim, N. Gisin and R. Renner, "Tight finite-key analysis for quantum cryptography", Nature Comm, Jan 2012, 3:634. doi: 10.1038/ncomms1631.
<http://www.nature.com/ncomms/journal/v3/n1/suppinfo/ncomms1631_S1.html>
[18] Z. Zhang, V. Anantharam, M. Wainwright and B. Nikolic. "A 47 Gb/s LDPC Decoder with Improved Low Error Rate Performance", Symposium on VLSI Circuits, June, 2009.

APPENDIX

LDPC error correction in QKD starts with Alice's sequence of sifted bits $\mathbf{x}=\{x_0, x_1, \ldots, x_{n-1}\}$ and Bob's corresponding set of sifted bits $\mathbf{y}=\{y_0, y_1, \ldots, y_{n-1}\}$. The overview of LDPC algorithm in QKD is shown Fig. A1. Alice computes the check sum vector $\mathbf{[CS]=[M_1] * [x]^T}$, a non-zero m-bit vector, and sends it to Bob over the reliable classical channel, which is received error free. Bob computes $\mathbf{[DS] = [M_1] x [y]^T}$ and compares the two. If $\mathbf{[CS] \neq [DS]}$, then there are errors in **y**, and the belief propagation part of the LDPC algorithm is iteratively applied to correct **y** until it converges (i.e., the results are $\mathbf{[CS] = [DS]}$) or the maximum number of iterations is reached. A hash signature is further used to verify, with high probability, that **y** has been corrected and the number of errors corrected, $|\mathbf{d}|$, is shared with Alice to update the QBER estimate.

A pseudo code implementation of the LDPC algorithm [15] that uses floating point multiplication and division is shown in Fig A3 that uses the following information:

y[1:n] - a list of Bob's original info bits
y1[1:n] - a list of Bob's corrected info bits
c[1:m] - a list of Alice's checksum values
b[1:m] - a list of Bob's original checksum values
d[1:m] - c XOR b, of Alice & Bob's checksum s
cs_list[1:k] - a list of checksum info (the sparse matrix info) as follows:
  cs_list [i].bit# - the i-th participating bit number
  cs_list [i].LL_ptr - a pointer to the associated LL_reg
LL_reg[1;k] - a list of the belief registers

where n is the number of info bits, m is the number of checksum values and k is the number of non-empty entries in the sparse matrix, $\mathbf{[M_1]}$. In addition to these data lists, we also need mappings showing the start and end of each checksum in the cs_list list as well as a mapping showing the start and end of each bit's belief register group in the LL_reg list, see Fig A2. The cs_list defines each checksum, the participating bit numbers and a pointer to its associated belief register, and is ordered by checksum. The i-th entry of the cs_index list points to the start of the i-th checksum definition. The LL_reg list contains the belief values and is ordered by bit number. The i-th entry of the LL_index list points to the start of the i-th bit belief register group.

cs_index[j] - ptr to the start of the j-th chksum (j=1, … , m+1), where cs_index[m+1]=k+1
LL_index[i] - ptr to the start of the LL_reg group for the i-th info bit (i=1, … , n+1), where LL_index [n+1]=k+1

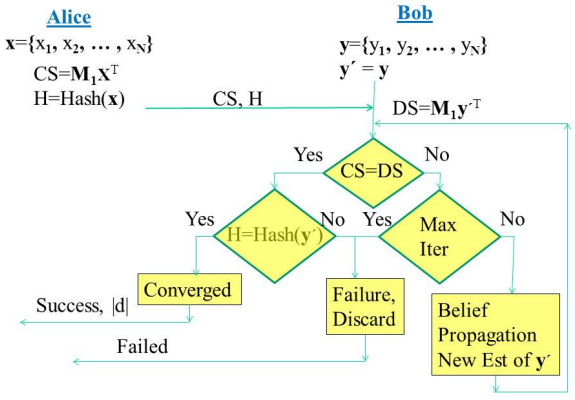

Figure A1. LDCP Error Correction Algorithm Flowchart.

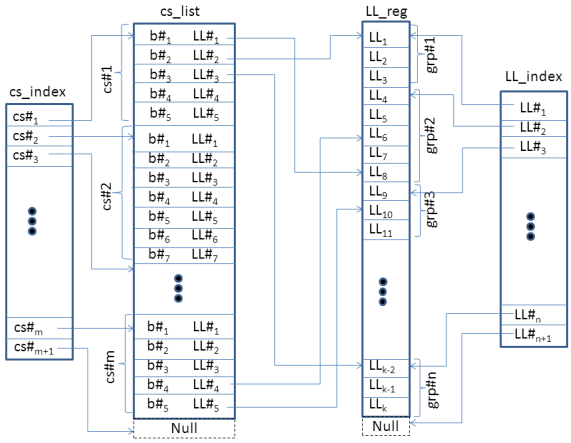

Figure A2. A diagram of the LDPC data mapping lists

An alternative pseudo code implementation that uses lookup tables with integer addition and subtraction is listed in Fig A4. It can have advantages for hardware implementation where floating point multiplication and division would incur high overhead operations (in time and resources). The following pseudo code will populate two 4K-entry lookup tables using the natural log function and the values Na=555, Ma=3893, Nf=556, Mf=3896.

```
for (i=1; i<=Ma; i++)
{   a = exp(-i/Na);
    f = (1+a)/(1-a);
    z = ln(f);
    j = (int) (Nf*z+0.5);
    if (j<=0) j=1;
    if (j>Mf) j=Mf;
   a2f[i] = j;
}
```

(a) a-to-f table

```
for (i=1; i<=Mf; i++)
{   f = exp(i/Nf);      // power of e
    a = (f-1)/(f+1);
    z = -ln(a);         // natural logarithm
    j = (int) (Na*z+0.5); // integer map
    if (j<=0) j=1;      // keep in limits
    if (j>Ma) j=Ma;
    f2a[i] =  j;        // load table value
}
```

(b) f-to-a table

The one missing element needed to apply LDPC is constructing the sparse matrix, $[\mathbf{M_1}]$. Each row defines a single checksum, denoted by a non-zero entry in the participating bit columns. Each non-zero entry in a column indicates in which checksums that bit participates. But constructing matrices with low failure rate correction near the Shannon limit is difficult even with the guidelines in the literature.

A GPU is a single instruction multiple data (SIMD) multiprocessor (MP), each of the MPs has a number of processing elements (PEs), see Fig A5. SIMD means that each multiprocessor executes a single instruction (such as a multiply) and each processing element would execute that instruction, in parallel and in lock-step, on a different data element (such as different elements of a vector). A GPU has a common large, slow memory and each MP has a local fast, small memory shared between its PEs. The table lookup with integer addition and subtraction implementation would not be an efficient GPU port since the lookup tables would use most of the fast local GPU memory. The floating-point multiply and divide implementation is a good match for porting to a GPU, but the data structures require significant revision.

By using a two dimensional array to hold the belief register (LL_Reg) values, we are able to eliminate all the lookup tables in Fig A2 except for the CS_index table, see Fig A6. The LL_Regs are organized so that each row contained all of the LL_Regs for a given bit and each column represented a checksum occurrence of that bit. So an LL_Reg bit group can be located by its bit number, but a lookup table is needed to find the start of each checksum LL_Reg and each LL_Reg contains a pointer to the next LL_Reg for that checksum. A NULL pointer terminates each checksum chain. This lookup table resides in the fast local GPU memory, while the LL_Regs reside in the large, but slow, common

memory. Because we tried to limit the slow common memory accesses and pack pointers into the LL_Reg while trying to maintain its size at 32-bits, this left only 8 bits for belief register values.

Porting the floating-point multiply and divide implementation to a GPU also required coding changes that would adapt it to the parallel processing of a GPU, beyond the syntax changes required for a co-processor's programming environment. One was assigning separate code words (data sets) to each GPU MP. Another was to assign each checksum and bit group to separate PEs of that MP.

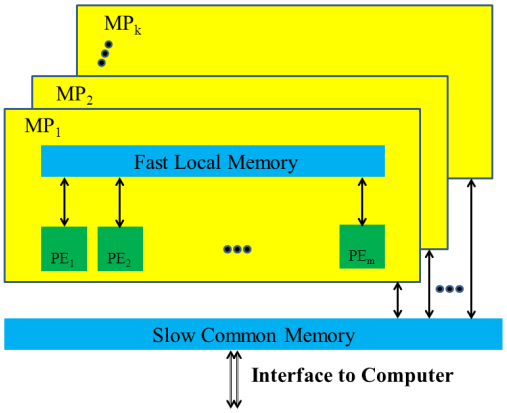

Figure A5. Diagram of a GPU Architecture.

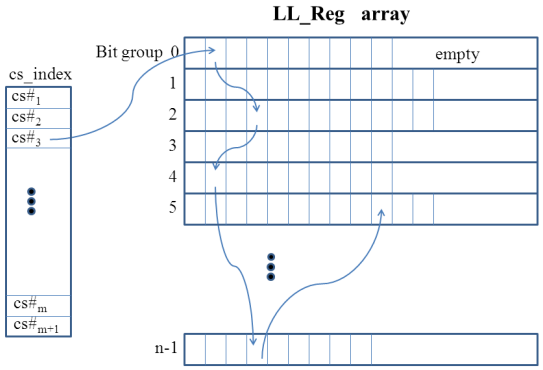

Figure A6. Diagram of our GPU LDPC data mapping structures.

```
  // top level LDPC floating point algorithm
ldpc_algo() {
    LL_init();                      // initialization
    success = 0;   i = 0;
    for  i < max_loops
    {    cs_msgs_2_bits(); // compute & update belief by chksums
         bit_msgs_2_cs();  // compute & update belief by bits & Bob's bits
         success = converged(); // recomputed chksums & test for convergence
         i++;
         if (success = 1) { print("LDPC converged in %d loops\n", i);  break; }
    }
    if (success = 0)    print("LDPC failed\n");
    return (success);
} // end prog
-------------------------------------------------------------------------
  // initialize log belief with p=QBER
LL_init() {
    f_init =  (1-p)/p;      //  initial f - belief
   a_init = (1-2*p);        //  initial a - belief
   for (i=1; i<=k; i++)  // initial all belief regs to channel belief
         LL_reg[i] = a_init;
} //  end fct
-------------------------------------------------------------------------
     // compute new belief ratios by checksum groups & convert to "f"
cs_msgs_2_bits() {
     for (i=1; i<=m; i++)  // for each chksum
     {    if (d[i] == 1) big_alpha=-1;
          else big_alpha= 1;
          j1=cs_index[i]; j2=cs_index[i+1];    // get chksum indices
          for  (j=j1; j<j2; j++)         // compute Belief for i-th chksum
          {         a1 = LL_reg[cs_list[j].LL_ptr];
                    big_alpha=big_alpha*a1;
          }
          if  (big_alpha<0 && big_alpha>-.0001) big_alpha= -.0001;
          if  (big_alpha>=0 && big_alpha<.0001) big_alpha=.0001;
          for  (j=j1; j<j2; j++)       // update each belief contribution
          {       a1 = LL_reg[cs_list[j].LL_ptr];
       f = (a1+ big_alpha) / (a1 - big_alpha);
                  if  (f < .001)    f = .001
                  if  (f > .999 && f<1.0)    f = .999
                  if  (f >=1.0 && f<1.0001)    f = 1.001
                  if  (f > 1000)   f = 1000
                  LL_reg[cs_list[j].LL_ptr] = f;
          }
     }  // end loop on checksums
} //  end fct
-------------------------------------------------------------------------
          // compute new belief values by bit groups
bit_msgs_2_cs() {
     for (i=0; i<n; i++)                          // for each info bit
     {    j1= LL_index[i];  j2= LL_index[i+1]; // indices for i-th bit LL_reg group
          f_tot = f_init;                         // init belief value for i-th bit
          for  (j=j1; j<j2; j++)                  // sum individual beliefs
          {    f= LL_reg[j];
               f_tot = f_tot * f;
          }
          for  (j=j1; j<j2; j++)          // update individual beliefs & convert to "a"
          {    f = f_tot – LL_reg[j];
               a = (f_tot – f)/(f_tot + f);
               if (a<0)
               {       if  (a > -.001) a = -.001;
                       if  (a < -.999) a = -.999;
               } else
               {       if  (a < .001) a = .001;
                       if  (a > .999) a = .999;
               }
               LL_reg[j] = a;
          }
          if (f_tot < 1.0)  y1[i] = 1 – y[i];     // update Bob's corrected bits
          else              y1[i] = y[i];
     }  // end loop on info bit
} //  end fct
-------------------------------------------------------------------------
          // check if LDPC belief propagation algorithm has converged
converged() {
     success = 1; // init to success
     for (i=1; i<=m; i++) // for each chksum
     {       sum = 0;
             j1=cs_index[i]; j2=cs_index[i+1];     // get chksum indices
             for  (j=j1; j<j2; j++)                // compute revised chksum
             {       sum = sum XOR y1[cs_list[j].bit#]
             }
             if (sum ≠ c[i]) )
                     success = 0;  // set to failure
     }
     return (success);
} //  end fct
```

Figure A3. Pseudo code for the LDPC floating point multiple/divide algorithm.

```
        // top level LDPC lookup table & integer algorithm
ldpc_algo() {
      LL_init();                  // initialization
      success = 0;   i = 0;
      for  i < max_loops
      {   cs_msgs_2_bits(); // compute & update belief by chksums
          bit_msgs_2_cs();  // compute & update belief by bits & Bob's bits
          success = converged(); // recomputed chksums & test for convergence
          i++;
          if (success = 1) { print("LDPC converged in %d loops\n", i);  break; }
      }
      if (success = 0)    print("LDPC failed\n");
      return (success);
} // end prog
-------------------------------------------------------------------------
            // initialize log belief with p=QBER
LL_init() {
      f_init = f_init_list[QBER]; //  Nf*ln( (1-p)/p ),   initial f - belief
      a_init = a_init_list[QBER]; // Na*ln( (1-2*p) ),  initial a - belief
      for (i=1; i<=k; i++)  // initial all belief regs to channel belief
            LL_reg[i] = a_init; //
} //  end fct
-------------------------------------------------------------------------
            // compute new belief ratios by checksum groups
cs_msgs_2_bits() {
      for (i=1; i<=m; i++)  // for each chksum
      {         sign=d[i]; big_alpha=0;
                j1=cs_index[i]; j2=cs_index[i+1];    // get chksum indices
                for  (j=j1; j<j2; j++)              // compute Belief for i-th chksum
                {         a1 = LL_reg[cs_list[j].LL_ptr];
                          if (a1<0) { sign=1-sign; big_alpha=big_alpha-a1;}
                          else big_alpha=big_alpha+a1;
                }
                for  (j=j1; j<j2; j++)       // update each belief contribution
                {         a1 = LL_reg[cs_list[j].LL_ptr];
                          if (a1 < 0) { p_sign=1-sign; p_alpha=big_alpha+a1;}
                          else { p_sign=sign; p_alpha=big_alpha-a1;}
                          if (p_alpha<=0)   p_alpha=1;
                          if (p_alpha>Ma) p_alpha=Ma;
                          if (p_sign==0)    LL_reg[cs_list[j].LL_ptr] = p_alpha;
                          else                    LL_reg[cs_list[j].LL_ptr] = -p_alpha;
                }
      }   // end loop on checksums
} //  end fct
```

```
            // compute new belief values by bit groups
bit_msgs_2_cs() {
      for (i=0; i<n; i++)  // for each info bit
      {     j1= LL_index[i];  j2= LL_index[i+1]; // indices for i-th bit LL_reg group
            f_tot = f_init;                          // init belief value for i-th bit
            for  (j=j1; j<j2; j++)                   // sum individual beliefs
            {     u= LL_reg[j];
                  if (u>0) u = a2f[u];               // convert a-to-f
                  else       u = -a2f[-u];
                  LL_reg[j] = u;
                  f_tot = f_tot + u;
            }
            for  (j=j1; j<j2; j++)                   // update individual beliefs
            {      k = f_tot – LL_reg[j];
                   if (k<0) {p_sign=1; k=-k;}
                   else       p_sign=0;
                   if (k<1)  k=1; if (k>Mf) k=Mf;
                   if (p_sign == 1)  LL_reg[j] = -f2a[k]; // convert f-to-a
                   else  LL_reg[j] = f2a[k];
            }
            if (f_tot < 0)  y1[i] = 1 – y[i];        // update Bob's corrected bits
            else              y1[i] =y[i];
      }  // end loop on info bit
} //  end fct
-------------------------------------------------------------------------
            // check if LDPC belief propagation algorithm has converged
converged() {
      success = 1; // init to success
      for (i=1; i<=m; i++) // for each chksum
      {       sum = 0;
              j1=cs_index[i]; j2=cs_index[i+1];     // get chksum indices
              for  (j=j1; j<j2; j++)                // compute revised chksum
              {         sum = sum XOR y1[cs_list[j].bit#]
              }
              if (sum ≠ c[i]) )
                        success = 0;  // set to failure
      }
      return (success);
} //  end fct
```

Figure A4. Pseudo code for the LDPC lookup table & integer add/subtract algorithm.