Verification Codes

Michael Luby[*] Michael Mitzenmacher[†]

Abstract

Most work in low-density parity-check codes focuses on bit-level errors. In this paper, we introduce and analyze *verification codes*, which are simple low-density parity-check codes specifically designed to manipulate data in packet-sized units. Verification codes require only linear time for both encoding and decoding and succeed with high probability under random errors. We describe how to utilize *code scrambling* to extend our results to channels with errors controlled by an oblivious adversary. Although verification codes over n packets require $\Omega(\log n)$ bits per packet, in practice they function well for packet sizes as small as thirty-two bits.

1 Introduction

Work on low-density parity-check codes has focused on the scenario where a bitstream is transmitted over a channel that introduces bit-level errors, such as the binary symmetric error channel with a fixed error probability or under Gaussian white noise. (See, e.g., [3, 11, 12, 13, 15, 16]). For this scenario, encoding and decoding schemes normally perform computational operations on and maintain data structures for individual bits. For example, techniques based on belief propagation [10, 16] use a probability for each bit to represent the current belief that the transmitted bit was a zero or one, and perform computations to update these probabilities. In many practical situations, however, the basic unit of transmission might not be single bits, but blocks of bits organized as packets. Here we use the term packets to broadly refer to a collection of bits. For example, packets could be thousands of bits, as is the case with Internet packets, or a packet could simply represent a thirty-two bit integer. To achieve high speeds on many current systems, it is natural to consider coding schemes that perform computational operations at the packet level instead of the bit level. Indeed, the recent burst of work in low-density parity-check codes arose from analysis designed for handling packet-level erasures on the Internet; the resulting codes have found many applications, particularly in reliable multicast [1, 2, 10].

A further motivation for studying packet-level schemes is that they can be used in concatenated codes [5]. An inner code that works at the bit level can be used on individual packets. An outer code designed to deal with errors at the packet level could then be used to correct failures from the inner code.

In this paper, we introduce and analyze *verification codes*, which are simple low-density parity-check codes designed especially for large alphabets. Verification codes are designed to deal with data in packet-sized units of thirty-two bits or more. More specifically, if the code is over n packets, then the packets should have $\Omega(\log n)$ bits, so that the total alphabet size is suitably large compared to the number of packets. We use the term verification codes because an important aspect of our codes is that packet values are verified as well as corrected through a simple message passing algorithm.

We first describe verification codes for the case of the symmetric q-ary channel, for large values of q. In the symmetric q-ary channel, the symbols are numbers in the range $[0, q-1]$, and when an error occurs the resulting symbol is assumed to take on a value uniformly at random from the set of $q-1$ possible incorrect values. We then describe how verification codes for the symmetric q-ary channel can be applied in more general settings in practice by using *code scrambling* techniques [7, 8].

[*]Digital Fountain, Inc. `luby@digitalfountain.com`

[†]Harvard University, Division of Engineering and Applied Sciences, 33 Oxford St., Cambridge, MA 02138. `michaelm@eecs.harvard.edu` Supported in part by an Alfred P. Sloan Research Fellowship and NSF grants CCR-9983832, CCR-0118701, and CCR-0121154. Part of this work was done while visiting Digital Fountain, Inc.

1.1 Previous Work

Gallager considered the question of low-density parity-check codes over large alphabets in his seminal work on the subject [6]; however, Gallager did not recognize the full potential of working over the symmetric q-ary channel. By making better use of properties of the underlying channel, we achieve significantly better results. Davey and MacKay [4] develop low-density parity-check codes for small finite fields, using belief propagation techniques. Belief propagation is not scalable to the very large alphabets we consider here.

One common approach for handling packet errors is to use a checksum, such as a 16 or 32 bit cyclic redundancy check (CRC). An erroneous packet is detected when the packet data fails to match the packet checksum. If an error is found, the packet is discarded; an erasure code could then be used to cope with lost packets. The probability of a false match decreases with the number of bits in the checksum, but the per packet overhead increases with the number of bits. We suspect our approach will prove superior for settings with small or intermediate packet lengths.

2 Framework for Low-Density Parity-Check Codes

We briefly summarize the now standard framework for low-density parity-check (LDPC) codes, following [11]. LDPC codes are easily represented by bipartite graphs. One set of nodes, the *variable nodes*, correspond to symbols in the codeword. We assume henceforth that there are n variable nodes. The other set of nodes, the *check nodes* correspond to constraints on the adjacent variable nodes. We assume henceforth that there are m check nodes. The design rate R is given by $R = \frac{n-m}{n}$. [1] In the case of general alphabets with q symbols, the symbols are interpreted as numbers modulo q and the constraints represent constraints on the sum of the variable nodes modulo q. In the case where the alphabet consists of the $q = 2^b$ strings of b bits, the constraints can be taken as parity check constraints, so the sum operation is a bitwise exclusive-or.

A family of codes can be determined by assigning degree distributions to the variable and check nodes. In regular LDPC codes all variable nodes have the same degree, and all check nodes have the same degree. More flexibility can be gained by using *irregular* codes, where the degrees of each set of nodes can vary. The idea of using irregular codes was introduced in [10, 11]. We associate with each degree distribution a vector. Let $(\lambda_2, \ldots, \lambda_{d_v})$ be the vector such that the fraction of edges connected to variable nodes of degree i is λ_i. (We assume a minimum degree of two throughout.) Here d_v is the maximum degree of a variable node. Similarly, let $(\rho_2, \ldots, \rho_{d_c})$ be such that the fraction of edges connected to check nodes of degree i is ρ_i, and d_c is the maximum degree of a check node. Based on these degree sequences, we define the polynomials $\lambda(x) := \sum_{i=2}^{d_v} \lambda_i x^{i-1}$ and $\rho(x) := \sum_{i=2}^{d_c} \rho_i x^{i-1}$, which prove useful in subsequent analysis. The λ_i and ρ_i variables must satisfy a constraint so that the number of edges is the same on both sides of the bipartite graph. This constraint is easily specified in terms of the design rate by the equation $R = 1 - \frac{\int_0^1 \rho(x)dx}{\int_0^1 \lambda(x)dx}$.

Once degrees have been chosen for each node (so that the total degree of the check nodes and the variable nodes are equal), a specific random code can be chosen by mapping the edge connections of the variable nodes to the edge connections of the check nodes. That is, to select a code at random, a random permutation π of $\{1, \ldots, E\}$ is chosen, where E is the number of edges. For all $i \in \{1, \ldots, E\}$, the edge with index i out of the left side is identified with the edge with index π_i out of the right side.

Without loss of generality we assume henceforth that the constraints are such that the sum of the symbols associated with the variable nodes adjacent to each check node is 0. In some circumstances it may be better to design a layered code, as described in [11]; this does not affect the analysis. For the layered version, the encoding time is proportional to the number of edges in the graph. Linear time encoding schemes also exist for the single layer scheme we analyze here [17].

[1] The actual rate R tends to be slightly higher than the design rate R in practice, because the check nodes are not necessarily all linearly independent. This causes at most a vanishingly small difference as n gets large, so we ignore this distinction henceforth.

We consider message passing algorithms described below. To determine the asymptotic performance of such codes, it suffices to consider the case where the neighborhood of each node is a tree for some number of levels. That is, there are no cycles in the neighborhood around each node. Analysis in this case is greatly simplified since random variables that correspond to messages in our message passing algorithms can be treated as independent. The martingale arguments relating the idealized tree model and actual graphs was first applied to coding in [9] and is now standard; see for example [9, 10, 11, 16].

3 Symmetric q-ary channels

3.1 A simple decoding algorithm

To begin our analysis, it is useful to consider the idealized case of a symmetric q-ary channel. Recall that in a symmetric q-ary channel the probability that a symbol is received in error is p, and when an error occurs the received symbol is equally likely to be any of the remaining $q-1$ symbols. Also recall that in our LDPC framework the sum of the variable nodes in a codeword adjacent to a check node must sum to 0.

We begin by describing a simple algorithm NodeVerify that we improve subsequently. With each variable node v there corresponds a true value t_v, a received value r_v, and a current value c_v. Throughout the algorithm NodeVerify, each variable node is in one of two possible states: *unverified* and *verified*. When a node is unverified, the algorithm has not yet fixed the final value for that node. Hence the decoding algorithm begins with all nodes being unverified. When a node is verified, its current c_v becomes fixed. Hence the algorithm NodeVerify should, with high proability, never assign a verified node a value c_v such that $c_v \neq t_v$. In the algorithm that follows, the current value c_v is always equal to r_v when the node is unverified.

The decoding algorithm simply applies the following rules in any order as much as possible:

1. If the sum of the current values of all the neighbors of a check node equals 0, all currently unverified neighbors become verified and the final values are fixed to the current values.

2. If all but one of the neighbors of a check node is verified, the remaining neighbor becomes verified, with its final value being set so that the sum of all neighbors of the check node equals 0.

In order to understand and analyze this algorithm, we refine our description of the node state. An unverified node is *correct* if its value was received correctly, and *incorrect* if it was received in error. This refinement is used only in the analysis, and not in the decoding algorithm, which does not know if an unverified node is correct or incorrect. Also, recall that we assume that an incorrect packet takes on a random incorrect value.

In this decoding process the check nodes play two roles. First, they may *verify* that all of their neighbors are correct, according to Rule 1. This verification rule applies because of the following fact: if the sum of the current values of all neighbors of a check node is 0, then with high probability all the neighboring variable nodes must be correct.

Lemma 1 *At any step where a check node attempts to verify all of its neighbors, the probability of an error is at most $1/(q-1)$. Over the entire decoding algorithm, if C verification steps are attempted, the probability of an error is at most $C/(q-1)$.*

Proof: Under the assumption that errors are random and check nodes are correct, for an erroneous verification to occur, two or more neighbors of a check node c must be in error. Consider an arbitrary neighbor in error, v. Given the values of the other neighbors of c, there is at most one possible erroneous value for v that would lead to a false verification. Under the assumption that errors take on an incorrect value that is uniform over all $q-1$ possibilities, the probability that v takes on this precise value is at most $1/(q-1)$. Hence at each step where a check node attempts to verify all of its neighbors, the probability of an error is at most $1/(q-1)$. The second statement of the lemma then follows by a simple union bound. $\square$

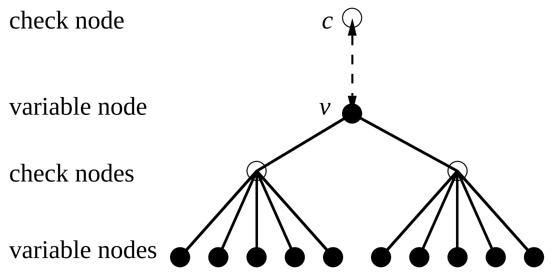

Figure 1: The neighborhood around (v, c).

To see the value of the above lemma, consider the case where symbols consist of b bits, so $q = 2^b$. The probability of a failure from a false verification is exponentially small in b at each step. It is straightforward to design a decoding process so that the number of verification steps attempted is linear in the size of the graph. Initially, all check nodes may gather the received values of their neighboring variable nodes, and see if a verification is possible. A check node can also take action whenever the state of one of its neighboring nodes changes. The total work done is then proportional to the number of edges in the graph. For bounded degree graphs, the number of edges will be $O(n)$. Hence, the packet size b needs to be only $\Omega(\log n)$ bits in order that the probability of failure due to a false verification be polynomially small in n.

The other role of a check node is to correct a neighboring variable node that was received incorrectly, according to Rule 2. A check node can correct a neighbor after all other neighbors have been verified and therefore are known (with high probability) to have the correct value. In this case, the value of the unverified neighbor is obtained by determining the value that results in a 0 sum at the check node.

3.2 A message-passing decoding algorithm

We now develop a message-passing version of this decoding process to aid our analysis. The goal is to determine the asymptotic error threshold p^*, which is the limiting fraction of errors tolerable under our decoding process as n grows large. To picture the decoding process, we focus on an individual edge (v, c) between a variable node v and a check node c, and an associated tree describing the neighborhood of v. Recall that we assume that the neighborhood of v is accurately described by a tree for some fixed number of rounds. The tree is rooted at v, and the tree branches out from the check nodes of v excluding c, as shown in Figure 1.

Given that our graph is chosen at random, we can specify how this tree branches in a natural way. This specification is the approximation obtained by thinking of the tree growing from v as a branching process, which is correct in the limit as the number of nodes grows to infinity. As the probability that edge (v, c) has degree j is λ_j, with probability λ_j there are $j - 1$ other check node neighbors of v. Similarly, every such neighbor c' has $j - 1$ other variable node neighbors with probability ρ_j, and so on down the tree.

We think of the decoding process as happening in rounds, with each round having two stages. In the first stage, each variable node passes to each neighboring check node in parallel its current value and state. In the second stage, each check node c' sends to v a flag denoting whether it should change its state to verified; if c' verifies v, it also sends the appropriate value. Based on this information, v changes its value and state appropriately. For convenience in the analysis, we think of each variable node as passing on to the check node c the current value *excluding any information obtained directly from c*. (This avoids the problem of a circular flow of information.) That is, when the variable node v passes information to c regarding its value and state, it only considers changes in its state caused by other nodes.

We provide an analysis based on the tree model. Consider an edge (v, c) in the graph. Let a_j be the probability that in round j the message from v to c contains the true value t_v but v is unverified. Similarly let b_j be the probability that in round j the message from v to c contains an incorrect value for v. We ignore the possibility in the analysis that a false verification occurs, since as we have already argued, for a sufficiently large alphabet this occurs with negligible probability. Hence $1 - a_j - b_j$ is the probability that in round j, v can confirm to c that it has been verified via another check node. Initially a_0 is simply the initial probability a correct word is sent and $b_0 = 1 - a_0$. If $a_j + b_j$ tends to 0, then our decoding algorithm will be successful, since then the probability that an edge (and therefore its corresponding node) remains unverified falls to 0.

The evolution of the process from round to round, assuming that the neighborhood of v is given by a tree, is given by:

$$a_{j+1} = a_0\lambda(1 - \rho(1 - b_j)), \tag{1}$$
$$b_{j+1} = b_0\lambda(1 - \rho(1 - a_j - b_j)). \tag{2}$$

We explain the derivation of equation (2) by considering the decoding from the point of view of the edge (v, c). For an incorrect value to be passed in the $(j+1)$st round, the node v must have been received incorrectly; this corresponds to the factor b_0. Also, it cannot be the case that there is some check node c' other than c neighboring v that has all of its children verified after j rounds, or else v could be corrected and verified for the $(j+1)$st round. Now each c' has $k-1$ children below it with probability ρ_k, and each child is verified after j rounds with probability $1 - a_j - b_j$. The probability that v has not been corrected due to a specific check node c' by round j is therefore

$$\sum_i \rho_i (1 - a_j - b_j)^{i-1} = \rho(1 - a_j - b_j).$$

As v has $k-1$ other neighboring check nodes besides c with probability λ_k, the probability that v remains uncorrected when passing to node c in round $j+1$ is

$$\sum_i \lambda_i (1 - \rho(1 - a_j - b_j))^{i-1} = \lambda(1 - \rho(1 - a_j - b_j)).$$

This yields equation (2); equation (1) is derived by similar considerations.

We show how to use this analysis to find codes with good properties. We first modify the above equations as follows:

$$a_{j+1} = a_0\lambda(1 - \rho(1 - b_j)), \tag{3}$$
$$b_{j+1} = b_0\lambda(1 - \rho(1 - a_{j+1} - b_j)). \tag{4}$$

Here equation (4) differs from equation (2) in that we have replaced a_j with a_{j+1}. This change does not change the final performance of the decoding; intuitively, this change is equivalent to changing our processing at each round by splitting it into two subrounds. In the first subround, variable nodes that have the correct value have their state updated. In the second subround, variable nodes with an incorrect value are corrected. The split clearly does not affect the final outcome; however, it allows us to replace a_j with a_{j+1} in the defining equation for b_j.

With this change, we find

$$b_{j+1} = b_0\lambda(1 - \rho(1 - (1 - b_0)\lambda(1 - \rho(1 - b_j)) - b_j)). \tag{5}$$

We have reduced the analysis to an equation in a single family of variables b_j. It is clear that if b_j converges to 0, then so does a_j, by the definition of the decoding process. Hence we need only consider the b_j. For $b_j \to 0$ with inital error probability b_0, we require that the sequence of b_j decrease to 0. It therefore suffices to find $\lambda(x)$ and $\rho(x)$ so that $b_{j+1} < b_j$, or equivalently

$$b_0\lambda(1 - \rho(1 - (1 - b_0)\lambda(1 - \rho(1 - x)) - x)) < x$$

for $0 < x \leq b_0$. Based on our discussion, we have the following theorem.

Theorem 1 *Given a design rate R and an error probability b_0, then if there are λ and ρ vectors satisfying the rate constraint $R = 1 - \frac{\int_0^1 \rho(x)dx}{\int_0^1 \lambda(x)dx}$ and the code constraint $b_0\lambda(1-\rho(1-(1-b_0)\lambda(1-\rho(1-x))-x)) < x$ for $0 < x \leq b_0$, then for any $\epsilon > 0$ there are verification codes of rate R that can correct errors on a symmetric q-ary channel with error probability $b_0 - \epsilon$ with high probability using the decoding scheme described above.*

We emphasize that similar theorems using various decoding schemes are implicit throughout the rest of the paper. Equation (5) therefore provides us a tool for determining good sequences $\vec{\lambda}$ and $\vec{\rho}$. This is a nonlinear equation in the coefficients λ_j and ρ_j, yielding a nonlinear optimization problem for which standard numerical techniques can be applied.

Unfortunately solving the nonlinear optimization problem directly for a specific code rate does not shed a great deal of insight into what can be said for general code rates. We can, however, demonstrate a provable bound for this family of codes based on the family of codes determined in [11] for erasures.

Lemma 2 (from [11]) *For any $0 < R < 1$ and $\epsilon > 0$, there exist sequences $\vec{\lambda}$ and $\vec{\rho}$ corresponding to a family of erasure codes that correct a $(1-R)(1-\epsilon)$ fraction of errors with high probability such that*

$$\lambda(1-\rho(1-x)) < \frac{x}{(1-R)(1+\epsilon)}\ ,\ 0 < x \leq 1. \tag{6}$$

This lemma allows the following theorem.

Theorem 2 *For any $0 < R < 1$ and $\epsilon > 0$, there exists a family of verification codes of rate R that correct a $1 - \frac{R}{2} - \frac{\sqrt{4R-3R^2}}{2} - \epsilon$ fraction of errors with high probability.*

Proof: We use the λ and ρ sequences defined by the erasure codes of Lemma 2, and apply the corresponding inequality to find the asymptotic fraction of errors we can correct using the corresponding verification codes for these degree sequences. Let $\gamma = (1-R)(1+\epsilon)$. We seek the maximum value of b_0 for which

$$b_0\lambda(1-\rho(1-(1-b_0)\lambda(1-\rho(1-x))-x)) < x.$$

By repeatedly using Lemma 2, we find

$$\begin{aligned} b_0\lambda(1-\rho(1-(1-b_0)\lambda(1-\rho(1-x))-x)) &< \frac{b_0((1-b_0)\lambda(1-\rho(1-x))+x)}{\gamma} \\ &< \frac{\frac{b_0(1-b_0)x}{\gamma} + b_0 x}{\gamma} \\ &= x\frac{b_0(1-b_0)+b_0\gamma}{\gamma^2}. \end{aligned}$$

Here the first inequality follows by applying Lemma 2 to the outer $\lambda(1-\rho(1-z))$ expression, and the second inequality similarly follows by then applying it to the inner expression. The final right hand size is less than or equal to x whenever we choose $\frac{b_0(1-b_0)+b_0\gamma}{\gamma^2} \leq 1$, and solving this quadratic we find we may choose

$$b_0 \leq \frac{1+\gamma-\sqrt{1+2\gamma-3\gamma^2}}{2}. \tag{7}$$

As γ can be arbitrarily close to $1-R$, asymptotically there exist verification codes that can correct anything less than a fraction $1 - \frac{R}{2} - \frac{\sqrt{4R-3R^2}}{2}$ of errors. $\square$

4 Improvements

4.1 Additional verification

We may improve our verification-based decoding by allowing further means of verification. We describe the changes to the message-passing algorithm. In the first stage, each variable node passes to each neighboring check node its current value and state. In the second stage, each neighboring check node c' sends to v a flag denoting whether c' can verify v directly, using one of the two rules given previously. If so, c' again sends the verified value as before. If not, c' also sends to v a *proposed value*, which is the value that v should take if all of the other neighbors of c' have sent the correct value. Now suppose v receives two proposed values that are the same. In this case, v should change its value to the proposed value and label itself as verified for the next round.

The reasoning behind this improvement is similar to the original argument for verification. If all neighbors besides v for a check node c' are in fact correct, the proposed value will be the correct value for v. If not, the proposed value will be random over all incorrect possibilities, and hence the probability of a match is small. We must adopt the additional restriction that there are no cycles of length four in the bipartite graph that represents the code, however. Otherwise, two check node neighbors of v could be neighbors with the same variable node v_2; if v_2 is in error, the proposed values of the check nodes would match, inducing a subsequent incorrect verification at v.

This improvement does increase the probability of a false verification, since there are now many additional ways a false verification could occur. Assuming a constant maximum degree, $b = \Omega(\log n)$ bits per packet still ensures that no false verification occurs with high probability.

The resulting equations describing the asymptotic behavior in this situation are somewhat more difficult. Again we have

$$a_{j+1} = a_0\lambda(1-\rho(1-b_j)),$$

by the same reasoning as before. To determine an equation for b_{j+1}, note that for v to continue to hold an incorrect value to pass to c, one of the following events must occur:

- all neighbors of v other than c received an incorrect value from some other neighbor in the previous round, or
- all but one neighbor of v other than c received an incorrect value from some other neighbor in the previous round, and the one neighbor that received all correct values did not have all of these values verified.

The probability that the first case occurs is just $\lambda(1-\rho(1-b_j))$. For the second case, when v has $i-1$ other neighbors, the probability that a specific set of $i-2$ neighbors other than c receive at least one incorrect value during round j is $(1-\rho(1-b_j))^{i-2}$. Ignoring (temporarily) the probability that the last neighbor sends a correct value, since there are $i-1$ possible ways of choosing the correct neighbor, we have the term

$$\sum_i \lambda_i(i-1)(1-\rho(1-b_j))^{i-2} = \lambda'(1-\rho(1-b_j)),$$

where λ' is the derivative of λ. We multiply this term by the probability that the last check node sends the correct but unverified proposed value, which is $(\rho(1-b_j)-\rho(1-a_j-b_j))$. Putting this all together yields

$$b_{j+1} = b_0\left[\lambda(1-\rho(1-b_j)) + \lambda'(1-\rho(1-b_j))(\rho(1-b_j)-\rho(1-a_j-b_j))\right]. \tag{8}$$

Again, we can modify the equation for b_j into one that does not involve a_j by replacing a_j by a_{j+1} and substituting the equation for a_{j+1} in the above.

Finding a code then again corresponds to finding $\lambda(x)$ and $\rho(x)$ so that $b_{j+1} < b_j$. A greater fraction of errors can be tolerated with this decoding scheme, at the cost of a higher probability of a false verification.

4.2 Using Reed-Solomon codes with Verification Codes

We may vary our codes by allowing the check nodes to hold information other than the sum of the variable nodes. A natural approach is to associate several check nodes with the same neighbors, and to use the check nodes as redundant representations of the variable node symbols via a Reed-Solomon code. For example, let us consider the case where check nodes are associated in pairs. If the check nodes are associated with k variable nodes, then the check node values are calculating using a $(k+2,k,1)$ Reed-Solomon code. The check nodes then have the property that they can correct any single error among the k neighboring variable nodes. As we explain below, the check nodes have further correction properties; however, it is instructive to consider a decoding scheme which *just* uses the above property.

Consider our message-passing decoding scheme in this setting. For any pair of check nodes and the associated set of variable nodes, if there is a single error among the variable nodes, it can be corrected and all the associated variable nodes verified. Again, we have used here our assumption that an error replaces a value with a value taken uniformly at random. Because of this assumption, with high probability throughout the execution of the decoding no set of variable nodes containing multiple errors will be falsely verified with high probability.

In the message-passing algorithm, a variable node v will transmit an incorrect value to a neighboring check node pair c after $j+1$ rounds if and only if it was received incorrectly and every other neighboring check node pair had another incorrect message node neighbor after round j. This yields the following recurrence equation to describe b_j:

$$b_{j+1} = b_0\lambda(1-\rho(1-b_j)). \tag{9}$$

This is exactly the same recurrence for the erasure codes developed in [10]. By using Reed-Solomon codes, we are making errors "equivalent" to erasures in the low-density parity-check code setting. As described in Lemma 2, in [10] vectors $\vec{\lambda}$ and $\vec{\rho}$ were determined that are essentially optimal: for codes of rate R we can come arbitrarily close to the optimal tolerable loss probability $1-R$. We can apply these distributions to obtain the following theorem:

Theorem 3 *For any $0<R<1$ and $\epsilon>0$, there exist sequences $\vec{\lambda}$ and $\vec{\rho}$ corresponding to a family of verification codes using Reed-Solomon codes as described above that corrects a $(1-R)/2-\epsilon$ fraction of errors with hihg probability.*

Proof: This follows immediately from Lemma 2; note here that we must divide the $(1-R)$ loss probability acceptable for erasure codes by two to get $(1-R)/2$ since we use two Reed-Solomon values per check pair. $\square$

We find that under this simple decoding rule we can correct almost the same fraction of errors as Reed-Solomon codes. Again, we emphasize the caveat that our results hold asymptotically with high probability assuming random errors according to the symmetric channel model, while Reed-Solomon codes are not so restricted.

The decoding scheme above can be improved slightly. We are not taking advantage of the following additional power of the check nodes: whenever a check node pair has only two unverified neighbors, the two neighbors can be corrected, as the k correct values suffice to reconstruct the remaining two. A decoding scheme that takes advantage of this fact is not substantially more complex, although the equations that describe it are. Consider again a recursive description of the b_j. It is now not enough that every neighboring check node pair of a variable node v has another incorrect variable node neighbor; now it must also have at least one other unverified neighbor.

Consider the probability that after round j a check node pair c' neighboring v has one other incorrect variable node neighbor but no other unverified neighbors. For a pair with k neighbors, we must choose the one other incorrect neighboring message node, and all other nodes are verified. Hence this probability is $\sum_{k\geq 2}\rho_k(k-1)b_j(1-a_j-b_j)^{k-2}$, which equals $b_j\rho'(1-a_j-b_j)$, where ρ' is the derivative of the ρ. This yields the recurrence

$$b_{j+1} = b_0\lambda(1-\rho(1-b_j)-b_j\rho'(1-a_j-b_j)). \tag{10}$$

A similar formula for the a_j values can be obtained. Here, a variable node with the correct value will become verified if there is a neighboring check node pair c' such that all other variable node neighbors of c' have the correct value, or if there is a neighboring check node pair c' with only one erroneous neighbor. This yields the recurrence:

$$a_{j+1} = a_0 \lambda(1 - \rho(1 - b_j) - b_j \rho'(1 - b_j)). \tag{11}$$

Similar equations can be developed for Reed-Solomon codes that can correct more errors. More details will appear in the full version.

5 Code Scrambling

Up to this point, we have assumed that our codes function on a symmetric q-ary channel. We can avoid this assumption by using the techniques of *code scrambling* [8, 7].[2] The idea of code scrambling is as follows. Suppose the sender and receiver have a source of shared random bits. We model the errors introduced by the channel as being governed by an oblivious adversary, who is unaware of the random bits shared by the sender and receiver. In this setting, the sender and receiver can use the random bits to ensure that regardless of the strategy of the adversary, the errors introduced appear equivalent to those introduced by a symmetric q-ary channel, in the following sense: if the adversary modifies d transmitted symbols, the effect is as though d randomly selected transmitted symbols take on erroneous values taking on random values from $GF(q)$.

The sender and receiver use their shared random bits as follows. When sending values $x_1, \ldots, x_n$, the random bits are used to determine values $a_1, \ldots, a_n$ chosen independently and uniformly at random from the non-zero elements $GF(q)^*$ and values $b_1, \ldots, b_n$ chosen independently and uniformly at random from $GF(q)$. Instead of sending the symbol x_i, the sender sends the symbol $a_i x_i + b_i$. Further, before sending the modified symbols, a permutation π on $\{1, \ldots, n\}$ is chosen uniformly at random and the symbols are sent in the order given by the permutation.

Informally, it is clear that these steps stifle the adversary. Since each symbol is now equally likely to take on any value from $GF(q)$, after the permutation of the symbols to be sent the adversary cannot distinguish the original position of any transmitted symbol. Hence if the adversary modifies d symbols, their locations will appear random to the receiver after the permutation is undone. Further, suppose the adversary adds an error e_i to the transmitted symbol $a_i x_i + b_i$, so that the received value is $a_i x_i + b_i + e_i$. The receiver will reverse the symbol transformation, obtaining the symbol $x_i + (a_i)^{-1} e_i$ in place of x_i. Since a_i is uniform over $GF(q)^*$ and unknown to the adversary, the erroneous symbol appears to take on a random erroneous value over $GF(q)$. A more formal proof appears in [7]. Specified to our situation, we have the following:

Theorem 4 *The probability p that an adversary who introduces d errors into a verification code using code scrambling as described above causes a decoding error is equal to the probability that the verification code makes an error when d errors are introduced in the symmetric q-ary channel.*

In practice, a truly large sequence of random bits would not be necessary. A pseudo-random generator of small complexity would perform adequately, as errors introduced in real channels are not generally adversarial.

6 Conclusion

Verification codes with code scrambling demonstrate that the power of low-density parity-check codes can be exploited for handling errors at the packet level. An important open question is to design provably optimal families of codes designed from the underlying equations. The verification code framework may also apply to other settings; for example, [14] extends verification codes to yield polynomial time codes for packet-based deletion channels.

[2]We developed the techniques for code scrambling on the q-ary channel independently; however, we adopt the framework in [7] for our description.

References

[1] Byers, J. W., Luby, M., and Mitzenmacher, M. Accessing Multiple Mirror Sites in Parallel: Using Tornado Codes to Speed Up Downloads. In *Proceedings of IEEE INFOCOM '99* (March 1999), pp. 275–83.

[2] Byers, J. W., Luby, M., Mitzenmacher, M., and Rege, A. A Digital Fountain Approach to Reliable Distribution of Bulk Data. In *Proceedings of ACM SIGCOMM '98* (Vancouver, September 1998).

[3] S. Chung, G. D. Forney, T. Richardson, and R. Urbanke. On the Design of Low-Density Parity-Check Codes within 0.0045 db of the Shannon Limit. *IEEE Communications Letters*, 5 (2001), pp. 58-60.

[4] M.C. Davey and D. J. C. MacKay. Low Density Parity Check Codes over GF(q). *IEEE Communications Letters*, 2:6 (1998), pp. 165-167.

[5] G. D. Forney, Jr. **Concatenated Codes**. MIT Press, 1966.

[6] R. G. Gallager. **Low-Density Parity-Check Codes**. MIT Press, 1963.

[7] P. Gopalan, R. Lipton, and Y. Z. Ding. Codes, Adversaries, and Information: a Computational Approach. Submitted, 2001.

[8] R. J. Lipton. A New Approach to Information Theory. In *11th Symposium on Theoretical Aspects of Computer Science*, pp. 699-708, 1994.

[9] M. Luby, M. Mitzenmacher, and M. A. Shokrollahi. Analysis of Random Processes via And-Or Tree Evaluation. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 364-373, 1998.

[10] M. Luby, M. Mitzenmacher, M. A. Shokrollahi, and D. Spielman. Efficient Erasure Correcting Codes. *IEEE Transactions on Information Theory*, 47(2), pp. 569-584, 2001.

[11] M. Luby, M. Mitzenmacher, M. A. Shokrollahi, and D. Spielman. Improved Low-Density Parity-Check Codes Using Irregular Graphs. *IEEE Transactions on Information Theory*, 47(2), pp. 585-598, 2001.

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

[13] D. J. C. MacKay and R. M. Neal, Near Shannon Limit Performance of Low Density Parity Check Codes. *Electronic Letters*, 32 (1996), pp.1645-1646.

[14] M. Mitzenmacher. Verification Codes for Deletions. In preparation.

[15] T. Richardson, A. Shokrollahi, and R. Urbanke, Design of Capacity-Approaching Irregular Low-Density Parity-Check Codes. *IEEE Transactions on Information Theory*, 47(2), pp. 619-637, 2001.

[16] T. Richardson and R. Urbanke, The Capacity of Low-Density Parity-Check Codes under Message-Passing Decoding. *IEEE Transactions on Information Theory*, 47(2), pp. 599-618, 2001.

[17] T. Richardson and R. Urbanke, Efficient Encoding of Low-Density Parity-Check Codes. *IEEE Transactions on Information Theory*, 47(2), pp. 638-656, 2001.