Foundations and Trends® in
Communications and Information Theory
Vol. 6, Nos. 3–4 (2009) 213–322

DOI: 10.1561/0100000060

now
the essence of knowledge

Raptor Codes

By Amin Shokrollahi and Michael Luby

Contents

Foundations and Trends® in
Communications and Information Theory
Vol. 6, Nos. 3–4 (2009) 213–322

DOI: 10.1561/0100000060

now
the essence of knowledge

Raptor Codes

Amin Shokrollahi[1] and Michael Luby[2]

[1] *EPFL, Station 14, Lausanne 1015, Switzerland, amin.shokrollahi@epfl.ch*
[2] *Qualcomm, Inc., 3195 Kifer Road, Santa Clara, CA 95051, USA, luby@qualcomm.com*

Abstract

This monograph describes the theory behind Raptor codes, and elucidates elements of the processes behind the design of two of the most prominent members of this class of codes: R10 and RaptorQ (RQ). R10 has already been adopted by a number of standards' bodies, and RQ is in the process of entering various standards at the time of writing of this monograph.

The monograph starts with the description of some of the transmission problems, which inspired the invention of Fountain codes. Thereafter, Luby transform codes (LT codes) and Raptor codes are introduced and insights are provided into their design. These codes are currently the most efficient realizations of Fountain codes. Different algorithms are introduced for encoding and decoding various versions of these codes, including their systematic versions. Moreover, a hybrid decoding algorithm called "inactivation decoding" is introduced, which is an integral part of all modern implementations of Raptor codes.

The R10 and RQ codes have been continued and will continue to be adopted into a number of standards and thus there are publicly available specifications that describe exactly how to implement these

codes. However, the standards' specifications provide no insight into the rationale for the design choices made. One of the primary purposes of this document is to provide this design rationale.

We provide results of extensive simulations of R10 and RQ codes to show the behavior of these codes in many different scenarios.

1

Introduction

In theory, there is no difference between theory and practice, but in practice there is.[1]

This monograph describes the theory, design, and practical realization of Raptor codes. Section 2 describes conceptual design and analysis tools that provide provably good Raptor code designs. Section 3 describes more detailed design and heuristic analysis tools that provide constructions of practical Raptor codes. Based on their excellent recovery properties and computational complexity, these practical constructions have been standardized, implemented, and deployed.

In general and as seems to be universally typical, the performance of the practical constructions far exceed what can be rigorously proven. Although the theoretical tools and designs provide the big ideas and insights, it is the more detailed and precise heuristic tools and methods that have been developed and honed over the years to become precision instruments that were used to craft the design details of the

[1] Written on the interior glass wall of the EPFL cafeteria in the Computer Science Building BC, just near *Place Alan Turing*. Wikipedia attributed to Johannes L. A. van de Snepscheut and also to Yogi Berra.

practical Raptor codes, the R10 code and the RaptorQ (RQ) code. For the R10 and RQ codes, their provable properties are even more real than a theoretical proof: highly optimized software implementing the R10 and the RQ codes has been developed, tested, and deployed in mission critical applications. The R10 and RQ codes decoding properties have been verified again and again via tens of billions of simulations; their computational complexity has been verified again and again on different platforms with all kinds of parameter settings; their real-world practicality has been demonstrated by their adoption in a variety of commercial standards and by their deployment in commercial and government/defense markets. Thus, the R10 and RQ codes have achieved something much more valuable than just a theoretical existence proof: they have proven to be powerful, efficient, and flexible codes that provide a lot of practical value to a large variety of data communication applications.

1.1 Data Transmission

Digital media have become an integral part of modern lives. Whether surfing the web, making a wireless phone call, watching satellite TV, or listening to digital music, a large part of our professional and leisure time is filled with all things digital.

The replacement of analog media by their digital brethren and the explosion of the Internet use has had a perhaps unintended consequence. Whereas analog media were previously replaced by digital media to preserve quality, the existence of high-speed computer networks makes digital media available to potentially anyone, anywhere, and at any time. This possibility is the basis for modern scientific and economic developments centered around the distribution of digital data to a worldwide audience. The success of web sites such as Apple's iTunes store or YouTube is rooted in the marriage of digital data and the Internet.

Reliable transport of digital media to heterogeneous clients becomes thus a central and at time critical issue. Receivers can be anywhere and they may be connected to networks with widely differing fidelities.

1.2 The Transmission Control Protocol

How are data commonly transported on a network such as the Internet? The basic transmission protocol used by any Internet transmission is the Internet Protocol, commonly known as IP [4]. The data to be transmitted are subdivided into packets; these packets are given headers with information pertaining to their origin and their destination; pretty much like sending a regular letter, where we put the addresses of the receiver and that of the sender on the envelope. Routers that take the role of mail stations inspect these headers and forward the packets to another router closer to the destination. To do this, they consult regularly updated routing tables, through which they can determine the shortest path between them and the destination. Eventually, following the path from one router to another, packets may be delivered to their destinations.

In theory, this protocol is sufficient for data delivery; however, the reality looks different. Routers tend to get overwhelmed at times by incoming traffic, leading them to drop some of the incoming packets. These dropped packets will never reach their destination. To overcome this problem, researchers proposed already in the early days of the Internet the "Transmission Control Protocol," commonly known as TCP [5]. TCP has withstood the test of time, as it remains the most widely used transmission protocol on the Internet. For example, http (the protocol ubiquitously used for surfing the web), ssh (used for establishing a secure connection to a host), sftp (the secure file transfer protocol), and many other transmission protocols used today utilize TCP as a basis.

How does TCP work? We give here a very simplified description which has the advantage of clarifying the main mechanisms behind the real TCP. In effect, for every packet sent, an acknowledgment is expected from the receiver. If the acknowledgment is not received after a prescribed period of time, the packet is considered lost and counter mechanisms are initiated, with the most basic of these counter mechanisms consisting of resending the missing packet. The other integral part of these countermeasures is the reduction of the transmission rate, which is done in the following way: the real TCP does not await

acknowledgments of individual packets before sending the next one, but instead has at any time a number of packets in transit. The acknowledgment of a packet is only expected after all the packets sent previous to the packet have been acknowledged. When a packet is lost, detected at the server by received acknowledgments of packets sent after the lost packet, the number of packets allowed to be in transit is reduced, which effectively reduces the rate at which the packets are sent to the receiver and provides a rate control mechanism. The reason for this reduction in rate is the implicit assumption by TCP that losses have occurred because intermediate routers have been overwhelmed. The reduction of the sending rate is designed to reduce traffic on the routers and to alleviate the burden on the network.

1.3 The User Datagram Protocol

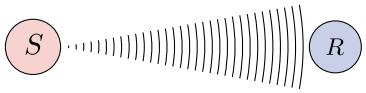
Another transmission protocol of interest to us is the "User Datagram Protocol" (UDP) [18]. Originally, this protocol was envisioned for short messages without strict reliability requirements. Essentially, it is analogous to sending mail through the postal service: each UDP packet contains a source address and a destination address, and the packet is routed through the network toward its destination without any guarantees that it will arrive; the packet may, e.g., be lost enroute due to a router buffer overflow, or to a wireless transmission error, etc. Furthermore, each UDP packet is sent independently of all other UDP packets, and a source may send a sequence of UDP packets at an arbitrarily high rate that can easily overload the network. Thus, UDP lacks TCP's higher-level reliability and rate control mechanisms. Because of this, delivery protocols that use UDP are sometimes blocked by firewalls from entering corporate networks.

Nevertheless, in some situations using UDP can be quite useful. For example, the destination address of a UDP packet can be a group address instead of an individual receiver address. Thus, any receiver that is part of the group can receive UDP packets sent to that group, thus enabling UDP to be used effectively in conjunction with a broadcast/multicast enabled network in a scalable way. For example, broadcast file delivery and broadcast streaming applications often use

UDP because the sent packets can be delivered concurrently to many receivers in a scalable way. In these types of applications, the packet sending rate is fixed at the source according to the available capacity of the network and/or the requirements of the application, e.g., real-time delivery of a 4 mbps video stream of packets. In these types of applications, adding a reliability protocol on top of UDP can be quite valuable, and providing this reliability is one of the main applications of Raptor codes.

1.4 Point-to-point Transmission

The simplest transmission scenario is point-to-point transmission. Here, a sender transmits data to one receiver, as described in the following figure in which a sender S is transmitting data to a receiver R:

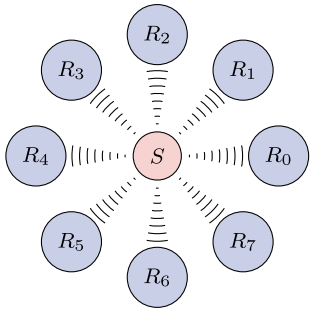


If the distance between the sender and the receiver is not too large, then TCP is a perfect transmission protocol. However, if the distance is large, then TCP exhibits inefficiencies: during the time in which acknowledgments are awaited, transmission is in an idle mode and hence the real capacity of the network may not be achieved. The situation is compounded when there is loss on the network, i.e., the TCP transmission rate slows down even more.

1.5 Point-to-multipoint Transmission

The second transmission scenario is the point-to-multipoint transmission. The situation is described in the figure below, in which a sender S is transmitting data to receivers $R_0, \ldots, R_7$. A typical example is distributing live video over the Internet. Unless the number of receivers is small, TCP turns out to have some scaling issues in this setting. The reason is that the sender needs to keep track of the reception of every individual receiver, and furthermore that each receiver needs to be sent a separate stream of data. Therefore, the server load and the network load increase with the number of receivers, and reliable transmission

becomes more challenging. Ironically, the more popular the content, the more difficult it becomes to deliver it to all the receivers. This phenomenon that is typically referred to as the "curse of popularity" makes it difficult to provide a scalable broadcast service on the Internet. However, in recent years such services have started to be deployed, based on already deployed http caching server infrastructure.

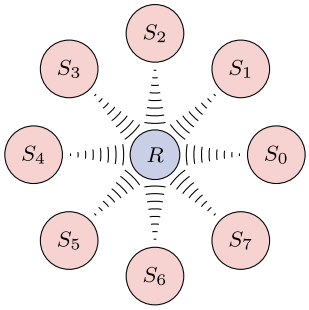


What are sometimes used for point-to-multipoint are protocols based on UDP, using the multicast/broadcast capabilities of UDP to handle the delivery scalability issue when the network is multicast/broadcast enabled, since all receivers in the destination group can attempt to receive a UDP packet sent to that destination group. However, as mentioned previously, UDP does not guarantee delivery of packets; it is a best effort protocol where sent packets may be lost for a variety of reasons, including wireless transmission noise that corrupts packets beyond repair, or because of packet overflows within the routers of the network due to intermittent congestion caused by other sources. Raptor codes can be used to provide reliability in a scalable way to UDP-based protocols.

1.6 Multipoint-to-point Transmission

Another scenario is multipoint-to-point transmission. Here, a group of senders, each possessing a copy of the same data, wants to transmit this copy to one receiver. The following figure shows an example in which senders $S_0, \ldots, S_7$ are transmitting to a common receiver R. In addition to problems discussed in the case of point-to-point transmission based on TCP, multipoint-to-point solutions based on TCP leads to enormous

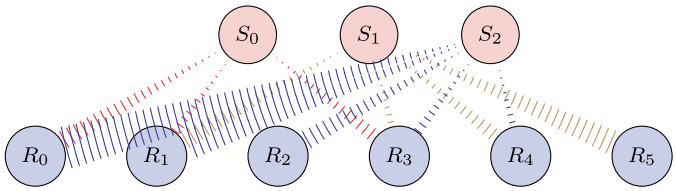
inefficiencies: the packets received from the various senders may not be different if the senders are not coordinated. The reception of duplicate packets is one of the principal sources of inefficient network usage in this case.



The fountain code properties (described in Section 1.8) make Raptor codes ideally suited for basing efficient solutions to the multipoint-to-point transmission protocols, where either UDP or TCP may be the underlying protocol onto which the usage of Raptor codes is layered.

1.7 Multipoint-to-multipoint Transmission

Another scenario is multipoint-to-multipoint transmission, depicted in the following figure in which we have a group of senders denoted S_0, S_1, S_2, each possessing a piece of data, and a group of receivers $R_0, \ldots, R_5$ each of which connects to a subset of the senders and receives the data:



A good example of this transmission scenario is a peer-to-peer network. All the problems discussed for the previous three transmission scenarios are also valid here. These problems are compounded if senders and receivers are transient, as is the case in a large peer-to-peer network. The fountain code properties (described in Section 1.8)

make Raptor codes ideally suited for usage in multipoint-to-multipoint scenarios.

1.8 Fountain Code Overview

At the very core of our solution lies the concept of a *fountain code*. We introduce the general use case for a fountain code, describe ideal abstract properties of a fountain code, describe its application to the scenarios described in the previous sections, and outline a randomized approach for constructing codes that lead toward the realization of practical fountain codes.

Suppose we have a block of data, hereafter called a *source block*, that are to be reliably transmitted over a packet network. The source block is typically partitioned into equal sized portions of data, hereafter called *source symbols*, that are typically sized to fit into a packet. In the following, we let k be the number of source symbols in the source block.

An effective approach to reliably transmitting the source block is to use an *encoder* at senders to generate *encoded symbols* from the source symbols of the source block and then to use *decoders* at receivers to decode the source symbols of the source block from received encoded symbols, where typically each encoded symbol is the same size as a source symbol. The basic idea is for senders to send encoded symbols in packets, and then a receiver can use the encoded symbols received in packets to try and decode the original source block even if some of the packets are not received. There are a variety of reasons a receiver may not receive some of the packets, examples of which include transmission over a wireless network that intermittently experiences enough interference or noise to cause unrecoverable errors in packets that are then discarded at the receiver, packet losses due to intermittent congestion that causes packet buffer overflows in routers, the receiver being only intermittently subscribed to the sessions in which packets are transmitted, and packets that arrive too late at the receiver to be useful when there are time constraints on consumption of the data in the source block.

Fountain codes in particular are especially effective codes that can be used to provide reliable transmission. The ideal abstract properties

of a fountain code are as follows [3]:

(1) A sender should be able to use a fountain encoder to generate as many encoded symbols as required from a source block.
(2) A receiver that receives any subset of k encoded symbols should (in most of the cases[2]) be able to use a fountain decoder to decode an exact copy of the original source block, independent of which subset of the generated encoded symbols is received and independent of whether the subset was generated by one sender or generated by more than one sender from the original block of source data.
(3) The computation time for encoding and decoding should be linear, i.e., the time to generate each encoded symbol should be linearly proportional to its size, and similarly the time to decode an original source block from encoded symbols should be linearly proportional to the original source block size.

These properties bring to mind a "fountain": When filling a bucket from a fountain of water, which particular drops fill the bucket doesn't matter, only the amount of water in the bucket matters. Similarly, with a fountain code, which particular encoded symbols are received doesn't matter, only the number of encoded symbols received matters.

From this description, it should be clear that fountain codes with these properties are very effective at providing reliable transmission over packet networks for any of the scenarios described in previous section. We now describe how fountain codes can be applied to these scenarios in a bit more detail.

In the point-to-point scenario, the sender can generate encoded symbols using a fountain encoder rom the source block and place the encoded symbols into packets, which are transmitted via the UDP protocol, for example. In a real-time application, the packets can be sent at any rate that is below the rate at which encoded symbols can be

[2] It is easily seen that a fountain code over a finite alphabet will not allow decoding from k received symbols in *all* the cases.

generated by the fountain encoder. Provided that this rate is very high, there will essentially be no limit on the transmission speed. Reliability of this transmission method is provided by the fountain property: as soon as the receiver collects k encoded symbols, it can decode the source symbols of the original source block. As k encoded symbols are the absolute bare minimum the receiver needs to collect to be able to decode the k source symbols, the transmission is optimal from an information point of view. The question of rate control remains, and in some cases it can be elegantly solved exploiting the fountain property [10, 11].

In the case of point-to-multipoint transmission, the sender generates encoded symbols and places them into packets and transmits the packets via, for example, broadcast or multicast. The fundamental properties of the fountain code guarantee that each receiver is capable of decoding the original data from reception of the minimal amount of data possible. Thus, one sender is capable of efficiently and reliably delivering to a potentially limitless number of receivers.

In the case of multipoint-to-point transmission, the various senders use fountain encoders applied to the common copy of the source block they each possess. The receiver collects encoded symbols from the various senders; by the properties of the fountain code, from the point of view of the receiver the mix of senders from which it receives encoded symbols does not matter. As soon as the receiver has collected k encoded symbols from the combined set of encoded symbols from the various senders, the original source block can be decoded.

The case of multipoint-to-multipoint transmission is solved in similar fashion and we will not elaborate further.

1.9 Fountain Code Construction Outline

Now that we know that fountain codes provide an elegant solution to various reliable transmission problem, we need to understand how to construct them. We now outline an approach that eventually leads to realizing almost ideal fountain codes. For a given vector $(x_1, \ldots, x_k)$ of source symbols, a fountain encoder produces a potentially limitless stream of encoded symbols $y_1, y_2, \ldots$. Here, a symbol refers to a bit or a sequence of bits. In many applications, symbols are of the same size as

the payload of the transmitted packets, though this is not necessarily the case. In general, the size of the symbols is often dictated by the underlying application and requirements.

The fountain codes that we initially describe operate on 1-bit symbols. Note that codes for larger symbols can be obtained using simple parallel concatenation, i.e., to generate a code that operates on t-bit symbols simply perform the same operations as would be performed on 1-bit symbols to each of the t positions of t-bit symbols in parallel.

These fountain codes are governed by a *probability distribution* $\mathcal{D}$ on the vector space $\mathbb{F}_2^k$. The encoding procedure for generating encoded symbol y_j is as follows:

(1) Sample $\mathcal{D}$ to obtain a vector $(a_1, \ldots, a_k) \in \mathbb{F}_2^k$.
(2) Calculate $y_j = \sum_i a_i x_i$.

The samplings of the fountain encoder are independent from encoded symbol to encoded symbol; this is extremely important as it induces a uniformity property on the encoded symbols generated and ensures that the code has the fountain properties. Note that when the encoded symbols are placed into packets for transmission, typically (but not always) an identifier is also placed in the header of each packet, called an ESI (encoded symbol identifier), that uniquely identifies the encoded symbols contained in that packet. The ESI is used by the decoder to determine the vector $(a_1, \ldots, a_k)$ corresponding to each encoded symbol in the received packet.

The average computational cost for generating an encoded symbol is simply the average weight of the vector $(a_1, \ldots, a_k) \in \mathbb{F}_2^k$ when sampled from $\mathcal{D}$ multiplied by the computational cost of adding two symbols together. Thus, it will be important to keep the average weight as small as possible.

Decoding algorithms will be described later; however, for now it is important to note that the following decoder performance metrics are key, and in particular the design of the probability distribution $\mathcal{D}$ has a large influence on these decoder performance metrics.

An important property we require of a fountain code is that it should be possible to decode the source symbols with little reception overhead with high probability. We say that the *overhead* is o if $k + o$ encoded

symbols are used when decoding is attempted, and if the overhead is written as a percent, i.e., $x\%$, then it is the overhead as a percent of the number of source symbols, i.e., $x = 100 \cdot o/k$.

The fountain code constructions we provide all have the property that encoded symbols are generated independently of one another. In addition, we will assume that the set of received encoded symbols is independent of the values of the encoded symbols in that set, an assumption that is often true in practice. These assumptions imply that for a given value of k, the probability of decoding failure is independent of the pattern of which encoded symbols are received and only depends on how many encoded symbols are received, i.e., the probability of decoding failure depends only on the overhead. Thus, we define the *failure probability* $f(o)$ to be the probability that decoding fails at a specified overhead o, i.e., the failure probability is a function of the overhead, and typically the failure probability should decrease quickly with increasing overhead. We call the set of pairs $\{(o, f(o)) : o = 0, 1, \ldots\}$ the *overhead-failure curve*.

Often we analyze the failure probability at a small overhead of interest as a function of k, i.e., for an overhead of $\varepsilon(k)$ we provide upper bounds on $f(\varepsilon(k))$, where both $\varepsilon(k)/k$ and $f(\varepsilon(k))$ go to zero as k goes to infinity. For example, it might be the case that $\varepsilon(k) = \varepsilon \cdot k$ for some constant ε, $0 < \varepsilon \ll 1$, and $f(\varepsilon \cdot k) = 1/k^c$ for some positive constant c, where preferably $c > 1$. In practice, what is important is that the failure probability decreases as quickly as possible as a function of increasing overhead, i.e., the overhead-failure curve is steep.

Equally important, the decoder should be computationally very efficient.

1.10 The Random Binary Fountain

A natural fountain code to consider is the "random binary fountain code," where the distribution $\mathcal{D}$ is the uniform distribution on $\mathbb{F}_2^k$, where k is the number of 1-bit (binary) source symbols in the source block. As described previously, this can be extended to symbols of arbitrary size.

Let us give a qualitative analysis of a random binary fountain code. The receiver collects $N = k + o$ encoded symbols $y_1, y_2, \ldots, y_N$. Each

of these symbols is a uniform random linear combination of the source symbols $x_1, \ldots, x_k$. The relationship between the source and the collected encoded symbols is described by a matrix, $A \in \mathbb{F}_2^{N \times k}$, as

$$A \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_k \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix}.$$

This matrix A is chosen uniformly at random from the set of binary $N \times k$ matrices.

Recovery of the source symbols is possible iff the rank of A is k. A simple analysis [20, Proposition 2] shows the following result:

Proposition 1.1. For a random binary fountain code operating on a source block with k source symbols, the overhead-failure curve is pointwise majorized by $\{(o, 2^{-o}) : o = 0, 1, \ldots\}$ with respect to the maximum-likelihood decoder.

For example, at an overhead of $c \cdot \log_2(k)$, the failure probability is $1/k^c$. In fact, one can show that for not too small values of o, $f(o)$ is roughly 2^{-o}.

As the above proposition describes, a random binary fountain code has a quickly decreasing failure probability as a function of overhead, i.e., the failure probability decreases by almost exactly a factor of two for each increase by one in the overhead. For example, a failure probability of 10^{-10} can be achieved with an overhead of around 30 symbols, regardless of k. For moderate values of k, say in the low thousands, this overhead relative to k is smaller than 0.3%.

However, random binary fountain codes suffer from a large encoding and decoding computational complexity. To assess this complexity, we will distinguish between "symbol operations" and "bit operations." The former corresponds to XORs of symbols, whereas the latter corresponds to XORs of bits. When the symbol size is large, a symbol size operation may be significantly more computationally expensive than a bit operation.

On average, every encoded symbol will be the XOR of around half the source symbols; hence, take around $k/2$ symbol operations to be created.[3]

The decoding takes $O(k^3)$ bit operations and $O(k^2)$ symbol operations. To prove this, we proceed as follows: first, we determine a $k \times k$-submatrix B of A, which is invertible over $\mathbb{F}_2$, and we determine its inverse B^{-1}. This can be done using the Gaussian elimination, and requires $O(k^3)$ bit operations.[4] The matrix B is determined by k rows of A, say rows $1, \ldots, k$. Next, we multiply B^{-1} with the vector consisting of $y_1, \ldots, y_k$. As B^{-1} has $O(k^2)$ entries equal to 1, the number of symbol operations is $O(k^2)$.

Summarizing, the random binary fountain code achieves a good overhead-failure curve; however, both encoding and decoding are computationally complex. What we would like instead is a fountain code that achieves similar or even an improved overhead-failure curve and has computationally efficient encoding and decoding algorithms.

For future use in Section 3.3.1, we mention that the concept of a random fountain is not limited to the field $\mathbb{F}_2$. More generally, we talk about a "q-ary random fountain code," or a "fountain code over $\mathbb{F}_q$" if the distribution $\mathcal{D}$ is chosen to be uniform over $\mathbb{F}_q^k$.

[3] This is not full proof; it is conceivable that a faster algorithm is available than simply XORing all the corresponding source symbols. Because of the random structure of the random binary fountain code, this seems highly unlikely though.

[4] There are faster algorithms based on fast matrix multiplication; however, they are not practically relevant.

2

Foundations

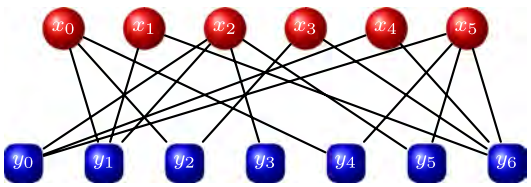
The goal of this section is to lay down some of the theoretical foundations needed for the design and analysis of Raptor codes. We are going to describe Luby transform code (LT codes), the first class of efficient fountain codes, and their encoding and decoding algorithms. Raptor codes that form an extension of LT codes and can possess linear time encoding and decoding algorithms are discussed next, and some of the theoretical tools for their asymptotic design and analysis are explained. A systematic version of these codes is of great practical interest and will be discussed in Section 2.3, along with several theoretical insights as to why trivial methods to obtain systematic Raptor codes fail. To achieve very good decoding performance for small numbers of source symbols, a different decoding algorithm is needed, and this is described in Section 2.4. This method, which we call "inactivation decoding," combines the efficiency of belief-propagation decoding with the error performance of maximum-likelihood decoding to yield optimal, yet efficient, decoding algorithms.

2.1 LT Codes

In order to create computationally efficient fountain codes, we start from an efficient algorithm that may or may not succeed. Later, we

will design the codes around the algorithm, i.e., design them in such a way that the algorithm succeeds with high probability. This decoding algorithm, which is known under the names of "belief-propagation decoder," "peeling decoder," or "greedy decoder," has been rediscovered many times [7, 8, 13, 14, 25]. In the case of the erasure channel, the belief-propagation decoder takes on a combinatorial form. This algorithm is best described in terms of the "decoding graph" corresponding to the relationship between the source symbols and the collected encoded symbols. This is a bipartite graph between k source symbols and N encoded symbols, where N is the number of collected encoded symbols. Encoded symbol y_i is connected to source symbols $x_{j_1}, \ldots, x_{j_\ell}$ iff encoded symbol y_i is the XOR of the source symbols $x_{j_1}, \ldots, x_{j_\ell}$. Here is an example:

$$
\begin{aligned}
y_0 &= x_2 + x_4 + x_5 \\
y_1 &= x_0 + x_1 + x_2 \\
y_2 &= x_0 + x_3 \\
y_3 &= x_2 \\
y_4 &= x_0 + x_5 \\
y_5 &= x_2 + x_5 \\
y_6 &= x_1 + x_3 + x_4 + x_5
\end{aligned}
$$



The belief-propagation decoder repeats the following until failure occurs in Step (1), or the decoder stops successfully in Step (4):

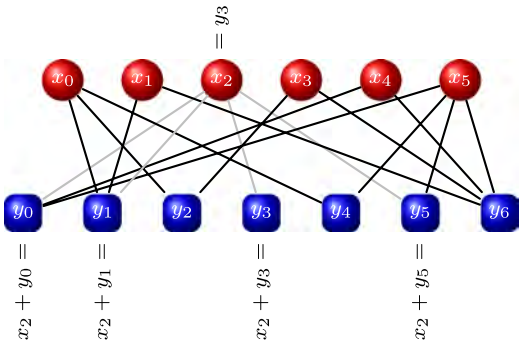
(1) Find an encoded symbol, say with index i, of degree 1; let j be the index of its unique neighbor among the source symbols. If there is no such degree 1 encoded symbol, then decoding fails.
(2) Decode $x_j = y_i$.
(3) Let $i_1, \ldots, i_\ell$ denote the indices of encoded symbols connected to source symbol j; set $y_{i_s} = y_{i_s} + x_j$ for $s = 1, \ldots, \ell$, and remove source symbol j and all edges emanating from it from the graph.
(4) If there are unrecovered source symbols, then goto Step (1). Else STOP.

Note that if this is repeated k times without a failure in Step (1) then decoding completes successfully with all k source symbols recovered, since each repetition recovers one of the k source symbols.

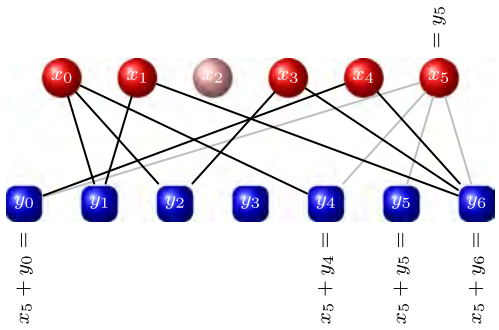
The complexity of belief-propagation decoding is essentially same as the complexity of the encoding algorithm, in the sense that there is exactly one symbol operation performed for each edge in the bipartite graph between the source symbols and the encoded symbols during both encoding and belief-propagation decoding. This is perhaps the main attraction of belief-propagation decoding, as it is typically decoding that is hard to make efficient. Thus, the computational complexity of belief-propagation decoding is linear in the average degree of the degree distribution multiplied by the size of the source block.

The challenge is to design the code in such a way that the average number of XOR operations is small, and so that belief propagation successfully decodes with high probability using approximately k encoded symbols to decode the k source symbols. We address this challenge after providing some more examples of how a belief-propagation decoder works.

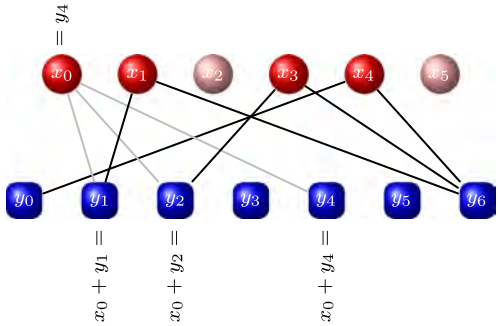
Let us examine how this algorithm works for the example above. First, we find an encoded symbol of degree 1. The only such encoded symbol has index 3 and corresponds to y_3. We decode the value of its unique neighbor, with index 2, to $x_2 = y_3$. Next, we add x_2 to the values of the neighbors of source symbol x_2 excluding y_3, namely the values y_0, y_1, and y_5. In the following figure, the edges that will be deleted from the graph after the computational operations are colored light gray.



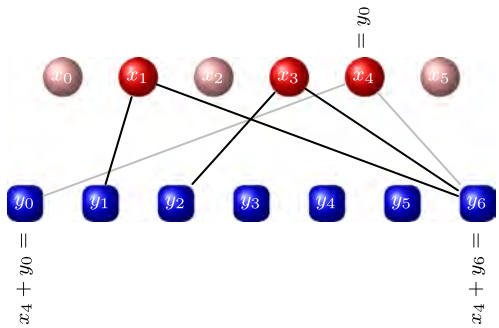
After removing the gray edges, we need to find an encoded symbol of degree one. The only such encoded symbol is the one corresponding to y_5, which will recover the value of x_5.



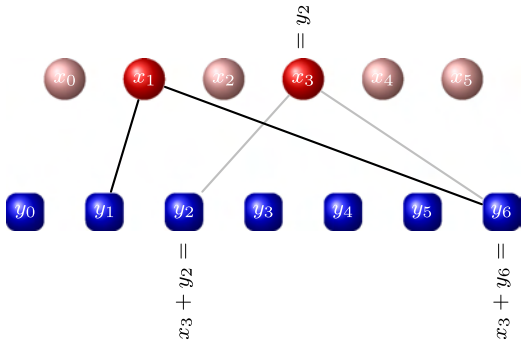
Continuing, we find that now the encoded symbol corresponding to y_4 has degree one.



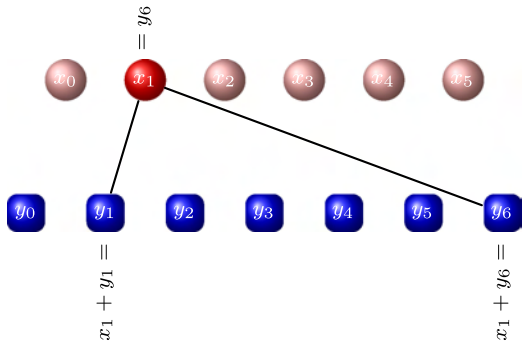
At this point, we have three choices for an encoded symbol of degree one, namely the encoded symbols corresponding to y_0, y_1, and y_2. We choose y_0 that recovers x_4.



Next, we choose y_2 that recovers x_3.



Finally, we choose y_6 to recover x_1.



A much simpler way of describing the decoding process is by means of a "schedule." This is a table with 2 rows and k columns. The columns correspond to the steps of the decoding process, each step corresponding to the recovery of one source symbol. The top row stores indices of encoded symbols used for decoding, and the bottom row stores the indices of source symbols recovered at the corresponding step. More precisely, the value of the first row at the ith column is the index of the encoded symbol, which is taken for the recovery of the source symbol recovered at step i, and the bottom row gives the index of this source symbol. The schedule for the above example is as follows:

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

The two problems that might occur when using this decoder are as follows:

- There may not be any encoded symbols of degree one at some intermediate step of the decoding (leading to decoding failure).
- There may be too many encoded symbols of degree one at some intermediate step of the decoding (leading to many redundant encoded symbols and thus high overhead).

For example, for the random binary fountain code, at the start of decoding with high probability there will be no encoded symbols of degree one, and thus decoding cannot even get started. Indeed, the probability that the random binary fountain encoder produces an encoded symbol of degree one is $k/2^k$, so that $k + \varepsilon(k)$ received encoded symbols are of degree larger than one with probability $(1 - k/2^k)^{k+\varepsilon(k)}$, which is roughly $\mathrm{e}^{-(k^2+k\cdot\varepsilon(k))/2^k}$. For meaningful values of $\varepsilon(k)$ (say proportional to k), this probability converges to 1 exponentially fast as k goes to infinity. Clearly, the distribution $\mathcal{D}$ needs to be chosen carefully if we are to avoid both of these problems.

Such a distribution was given by Luby [9], leading to the class of LT codes. In this distribution, elements in $\mathbb{F}_2^k$ of the same weight are assigned the same probability. More precisely, fix a probability distribution Ω on the integers $\{1, 2, \ldots, k\}$, assigning probability Ω_i to the integer i. Ω induces a probability distribution $\mathcal{D}_\Omega$ on $\mathbb{F}_2^k$, which assigns probability $\Omega_d / \binom{k}{d}$ to a vector of Hamming weight d. To sample from $\mathcal{D}_\Omega$, we first sample from Ω to obtain an integer d, and then we sample uniformly at random a vector $x \in \mathbb{F}_2^k$ of weight d. We call the pair (k, Ω) the *parameters* of the LT code, and call Ω the corresponding *degree distribution*.

How should Ω be chosen? Clearly, Ω_1 should be larger than zero, or else decoding cannot start. On the other hand, if we want a small overhead, then Ω_1 should not be too large. In fact, we have the following result.

Proposition 2.1. Suppose that we have a sequence of LT codes with parameters $(k, \Omega^{(k)})$, such that the maximum-likelihood decoding algorithm succeeds with high probability for a overhead $\varepsilon(k)$ such

that $\varepsilon(k)/k$ converges to 0 as $k \to \infty$. Then, $\Omega_1^{(k)}$ has to converge to zero.

---

proof (Sketch). Suppose that $\Omega_1^{(k)} > \tau$ for some constant $\tau > 0$, and consider the bipartite graph between encoded symbols of degree one and the source symbols. In this graph, the expected number of source symbols connected to at least two encoded symbols is a constant. In fact, if k is large, for each source symbol the probability that its degree is d is well approximated by $\mathrm{e}^{-\alpha}\alpha^d/d!$, where $\alpha = \Omega_1^{(k)}(1 + \varepsilon(k)/k)$. Therefore, the probability that a source symbol is of degree 2 or larger is $\mathrm{e}^{-\alpha}\sum_{d\geq 2}\alpha^d/d! = 1 - \mathrm{e}^{-\alpha}(1+\alpha)$, which is a constant.

Each such source symbol of degree 2 or larger as just described leads to at least one "redundant" encoded symbol, i.e., an encoded symbol that cannot be used to decode a source symbol. It follows that the expected number of redundant encoded symbols of degree one is a constant fraction of k, say $\xi \cdot k$. Therefore, the number of collected encoded symbols has to be at least $\xi \cdot k$ larger than the minimum number k, which implies that the overhead is at least $\xi \cdot k$. This contradicts successful decoding with high probability at an overhead that converges to zero as $k \to \infty$. □

A heuristic way to design a good degree distribution is to use an expectation analysis. For the sake of simplicity, we assume that every encoded symbol chooses its neighbors among the source symbols randomly and with replacement.

As described above, the belief-propagation decoder proceeds in steps and recovers one source symbol at each step. Step 0 is before decoding starts, the ith source symbol is recovered in step i, and thus step k is the last step in a successful decoding. Following the notation in [9], we call the set of encoded symbols of reduced degree one after step i the *ripple* at step i. We say that an encoded symbol is *released* at step i if its degree is larger than one before step i, and it is equal to one after step i, so that recovery of the source symbol at step i reduces the degree of the encoded symbol to one. The probability that an encoded symbol of initial degree d releases at step i can be easily calculated as follows: this is the probability that the encoded symbol has exactly one neighbor

among the $k - i$ source symbols that are not yet recovered after step i, and that not all the remaining $d - 1$ neighbors are among the $i - 1$ already recovered source symbols before step i. The probability that the encoded symbol has exactly one neighbor among the unrecovered source symbols and that all its other neighbors are within a set of size s contained in the set of remaining source symbols is $d(1 - \frac{i}{k})(\frac{s}{k})^{d-1}$, since we are assuming that the encoded symbol chooses its neighbors with replacement. Therefore, the probability that the encoded symbol is released at step i given that its original degree is d equals

$$d\left(1 - \frac{i}{k}\right)\left(\left(\frac{i}{k}\right)^{d-1} - \left(\frac{i-1}{k}\right)^{d-1}\right).$$

Multiplying the term with the probability Ω_d that the degree of the symbol is d, summing over all d, and setting

$$\Omega(x) = \sum_d \Omega_d x^d,$$

we obtain the following expression for the probability that an encoded symbol is released at step i:

$$\left(1 - \frac{i}{k}\right)\left(\Omega'\left(\frac{i}{k}\right) - \Omega'\left(\frac{i-1}{k}\right)\right).$$

Note that

$$\Omega'\left(\frac{i}{k}\right) - \Omega'\left(\frac{i-1}{k}\right) \sim \frac{1}{k}\Omega''\left(\frac{i}{k}\right) + O\left(\frac{1}{k^2}\right).$$

Suppose that the decoder collects $N = k + \varepsilon(k)$ encoded symbols. Then, the expected number of encoded symbols releasing at step i is N times the probability that an encoded symbol releases at step i, which, by the above, is approximately equal to

$$\frac{N}{k}\left(1 - \frac{i}{k}\right)\Omega''\left(\frac{i}{k}\right) + O\left(\frac{N}{k^2}\right).$$

In order to construct asymptotically optimal codes, i.e., codes that can recover the k source symbols from any N encoded symbols for values of N arbitrarily close to k, we require that this expectation be equal to 1 when $N = k$ and $k \to \infty$. Setting $x = i/k$, this means that

$$(1 - x)\Omega''(x) = 1$$

for $0 < x < 1$. Solving this equation and keeping in mind that $\Omega(1) = 1$, this shows that

$$\Omega(x) = \sum_{d \geq 2} \frac{x^d}{d \cdot (d-1)}.$$

This distribution is only valid in the limit, and for this reason we call it the *limit degree distribution*. Note that the limit degree distribution does not work at all using only a belief-propagation decoder, since it lacks any encoded symbols of degree one. A more detailed analysis appears in [9], where an expectation analysis is provided for finite values of k and for the case where the neighbors of an encoded symbol are chosen without replacement. The distribution derived there is given by

$$\Omega(x) = \frac{x}{k} + \sum_{k \geq d \geq 2} \frac{x^d}{d \cdot (d-1)},$$

and is called the *Soliton distribution*. The Soliton distribution above and the limit degree distribution are very similar, and they are identical in the limit of $k \to \infty$.

As was mentioned above, the limit degree distribution does not work well using only a belief-propagation decoder in the sense that for any given value of k the variance in the decoding process will cause it to fail. A more robust degree distribution is given by Luby [9], where degree distributions are exhibited, which, for a target error probability of δ, have an overhead of $O(\log^2(k/\delta) \cdot \sqrt{k})$, and each encoded symbol has an average degree of $O(\log(k/\delta))$. Thus, the average number of symbol operations per encoded symbol generated is $O(\log(k/\delta))$ and the average number of symbol operations to decode the k source symbols is $k \cdot O(\log(k/\delta))$

The Soliton distribution and any of its robust variants have one feature in common: the average degree of an encoded symbol under any of these distributions is $O(\log(k))$. This means that on average every encoded symbol needs $O(\log(k))$ symbol operations for its generation, and that the decoding algorithm needs $O(k \log(k))$ symbol operations. Is it possible to reduce the former running time to a constant, and the latter one to $O(k)$, perhaps by changing the degree distribution, or the decoding algorithm? It turns out that the answer to this question is

NO. A very simple argument, laid out in [9, 20, Proposition 1] based on a simple coupon collector analysis, shows that even if maximum-likelihood 'decoding is used the average degree of an encoded symbol has to be at least of the order of $\log(k)$ to guarantee a failure probability of the order of $1/k^c$ for constant $c > 0$.

2.2 Raptor Codes

Initially motivated by the objective of improving the encoding and decoding complexity, Raptor codes were invented in late 2000, and a patent was filed in 2001 [23]. An extension of LT codes, Raptor codes are a class of fountain codes that have extremely fast encoding and decoding algorithms, i.e, a small constant average number of symbol operations per encoded symbol generated, and a similar small constant number of symbol operations per source symbol recovered. Thus, overall Raptor codes achieve an optimal encoding and decoding complexity to within a constant factor.

The key to the invention of Raptor codes is the insight alluded to at the end of the present section explaining why it is difficult to design LT codes for which the average degree of the output symbols is constant: in this case there is, with high probability, a constant fraction of the source symbols that do not contribute to the values of any of the collected encoded symbols. These source symbols can therefore never be recovered, no matter what algorithm is used. The basic idea behind Raptor codes is to use a (normally high-rate) code to precode the source symbols. These symbols will be called intermediate symbols. Next, a suitable LT-code with constant average degree is applied to the intermediate symbols to produce the encoded symbols. Once the LT decoder finishes its operation, a small fraction of the intermediate symbols will still be unrecovered. If the precode is chosen appropriately, then this set can be recovered using an erasure decoding algorithm for the precode.

One example of a Raptor code construction described and analyzed in [20] has the following properties: For any constant $\varepsilon > 0$ one can construct a Raptor code such that for source blocks with k source symbols the average number of symbol operations per generated encoded symbol is $O(\log(1/\varepsilon))$, the number of symbol operations to decode the

source block is $O(k \cdot \log(1/\varepsilon))$, and for overhead $\varepsilon \cdot k$ the failure probability is $1/k^c$ for a constant $c > 1$ that is independent of ε.

All versions of Raptor codes outperform LT codes in terms of computational complexity. More advanced Raptor codes have better overhead-failure curves than LT codes in practice. For example, the Raptor code described in [15] exhibits an overhead-failure curve that essentially is that of a random binary fountain code. The most advanced Raptor codes, the RQ code described in [16], have an even much better overhead-failure curve.

Raptor codes achieve linear time encoding and decoding performance based on a simple idea: an appropriate binary block code $\mathcal{C}$ is used to encode the vector $(x_1, \ldots, x_k)$ of source symbols to generate *redundant symbols* $(z_1, \ldots, z_{n-k})$, where $n - k$ is a small fraction of k. The concatenation $(x_1, \ldots, x_k, z_1, \ldots, z_{n-k})$ of the source symbols and the redundant symbols is called the *intermediate block*, and we also refer to the n symbols in the intermediate block as the *intermediate symbols*. There are $n - k$ constraints that define the relationship between the source symbols and the redundant symbols of the intermediate block, and these constraints can be viewed as symbols, hereafter called *constraint symbols*. The value of each constraint symbol is zero, i.e., the constraint symbol constrains the sum of its neighboring intermediate symbols to be equal to zero.

Why is precoding a potentially good strategy? An appropriate LT code is applied to the intermediate block to obtain the encoded symbols. The intuitive advantage of precoding is that the redundancy amongst the intermediate symbols allows recovery of all the intermediate symbols using the decoder for $\mathcal{C}$ if most of the intermediate symbols are known. Suppose, for example, that the precode $\mathcal{C}$ is capable of correcting up to a δ-fraction of erasures among the intermediate symbols. Then, the LT decoder only needs to recover a $(1 - \delta)$-fraction of the intermediate symbols from the received encoded symbols. From the description of the LT code provided earlier, this intuitively implies that the average degree used by the LT code can be $O(\log(1/\delta))$.

Both the received encoded symbols and the constraint symbols are used for decoding the intermediate block, where the constraint symbol values are known at the decoder to be zeros. Thus, if N encoded

symbols are received then $N + (n - k)$ symbols can be used to decode the n intermediate symbols. Note that decoding can occur from k received encoded symbols, i.e., with overhead 0, if the k received encoded symbols and $n - k$ constraint symbols are linearly independent (though the decoder may not be the belief-propagation decoder of Section 2.1, see Section 2.4).

A toy example is given in Figure 2.1, where the precode $\mathcal{C}$ generates two redundant symbols from six source symbols. The values of the two redundant symbols z_1 and z_2 for this code can be derived from the relationship between the constraint symbols and the intermediate symbols as shown in the lower part of the left box.

The right interplay between the precode $\mathcal{C}$ and the LT code used to create the encoded symbols is crucial for obtaining codes with good overhead-failure curves. For example, LT codes form a special subclass of Raptor codes: for these codes the precode $\mathcal{C}$ is trivial, i.e., $n = k$, no redundant symbols are added to the intermediate block and thus the intermediate block is the same as the original source block. At the other extreme, there are the *precode only* (PCO) codes [20] for which

$$
\begin{aligned}
y_1 &= x_1 \\
y_2 &= x_1 + x_5 \\
y_3 &= x_1 + x_3 + x_4 \\
y_4 &= x_1 + x_3 + x_4 + x_6 \\
y_5 &= x_4 + z_2 \\
y_6 &= x_5 + z_2 \\
y_7 &= x_4 + x_5 + x_6 + z_1 \\
\hline
0 &= x_1 + x_3 + x_4 + x_6 + z_1 \\
0 &= x_1 + x_2 + x_4 + x_5 + x_6 + z_2
\end{aligned}
$$

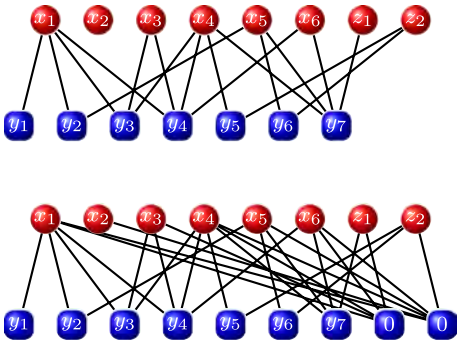


Fig. 2.1 Toy example of a Raptor code. The seven received encoded symbols are shown on the left, together with the relations among the intermediate symbols defined by the two constraint symbols. The intermediate symbols are partitioned into the source symbols $x_1, \ldots, x_6$ and the redundant symbols z_1, z_2. The top graph is the one between the encoded symbols and the intermediate symbols, and as can be seen source symbol x_2 is not a neighbor of any encoded symbol and cannot be directly recovered by LT decoding alone. In the lower graph, the constraint symbols are added to the graph, and the source symbol x_2 is a neighbor of a constraint symbol and can be potentially recovered.

the degree distribution Ω is trivial (it assigns a probability of 1 to degree 1, and zero probability to all other degrees). The paper [20] gives a thorough analysis of these codes. All Raptor codes in use are somewhere between these two extremes: they use a non-trivial (high-rate) precode, and they have a degree distribution that is typically an intricate variation of the limit degree distribution for which the average degree is constant.

The asymptotic design of Raptor codes uses the tree analysis of [12], nowadays also called density evolution. Suppose that the Raptor codes are to be designed so that decoding with overhead $k + \varepsilon \cdot k$ is successful with high probability. The analysis of the LT decoding process, applied to this case, reveals that asymptotically the expected fraction of intermediate symbols connected to encoded symbols of reduced degree one is $1 - x - \mathrm{e}^{-(1+\varepsilon)\Omega'(x)}$ if x is the fraction of intermediate symbols that have already been recovered. Additionally, the analysis of [12] reveals that if x_0 is the smallest root of the equation $1 - x - \mathrm{e}^{-(1+\varepsilon)\Omega'(x)}$ in the interval $[0,1)$, then asymptotically the expected fraction of intermediate symbols still undecoded at the end of the LT decoding process is $1 - x_0$, and for each instantiation of the decoding graph the real fraction is sharply concentrated around this value. (More details can be found in [20, Section VI].) It follows that, asymptotically, the degree distribution Ω has to be designed in such a way as to ensure that

$$\sup\{x \in [0,1) \mid 1 - x - \mathrm{e}^{-(1+\varepsilon)\Omega'(x)} > 0\} \tag{2.1}$$

is maximized. From this, Shokrollahi [20] constructs Raptor codes with an average encoded symbol degree of $O(\log(1/\varepsilon))$, a decoding complexity of $O(k \cdot \log(1/\varepsilon))$, and a failure probability that is inversely polynomially in k for an overhead of $\varepsilon \cdot k$.

Using the heuristic that the changes in the ripple size follow a random walk, the finite length inequality

$$1 - x - \mathrm{e}^{-(1+\varepsilon)\Omega'(x)} \geq \gamma\sqrt{\frac{1-x}{k}}$$

for $x \in [0, 1 - \delta]$ was derived to ensure that the decoding process will continue with high probability until it has recovered all but a δ-fraction of the intermediate symbols. Here, γ is a positive design parameter; the

larger it is, the more probable will it be for the decoder to decode all but a δ-fraction of the intermediate symbols. At the same time, however, the larger γ, the smaller the maximum possible achievable δ will be. Using this approach, good finite length Raptor codes were designed. These codes typically perform with an overhead of a few percent (3–5%) of k with a failure probability of 10^{-14} or less, but only for values of k in the range of 50,000 or higher. An example is furnished by the code with degree distribution

$$\begin{aligned}\Omega(x) = {} & 0.007969 + 0.49357x^2 + 0.1662x^3 + 0.072646x^4 \\ & + 0.082558x^5 + 0.056058x^8 + 0.037229x^9 + 0.05559x^{19} \\ & + 0.025023x^{65} + 0.003135x^{66} \end{aligned} \tag{2.2}$$

for $k = 65536$. The plot of

$$1 - x - \mathrm{e}^{-(1+\varepsilon)\Omega'(x)} - \gamma\sqrt{\frac{1-x}{65536}}$$

for various values of γ and $\varepsilon = 0.038$ is given in Figure 2.2(a). As can be seen, the values $\gamma = 2.5$ and 3 are not reliable, since their corresponding curves intersect the x-axis fairly early on. To see the effect of the overhead ε, Figure 2.2(b) shows plots of the function $1 - x - \mathrm{e}^{-(1+\varepsilon)\Omega'(x)}$ for various values of ε. As can be seen, asymptotically this degree distribution can afford an overhead of less than $0.005k$, but more than $0.002k$.

2.3 Systematic Version

A code is called *systematic* if the encoded symbols include the source symbols, e.g., from a source block of source symbols $(x_1, \ldots, x_k)$ the encoder generates encoded symbols $y_1, y_2, \ldots$, such that $y_i = x_i$ for $i = 1, \ldots, k$. Up until now, the LT codes and Raptor codes that we have introduced have been non-systematic.

Systematic codes are important in a variety of applications. For example, suppose that the deployment of a Raptor code is done in phases during which some receivers are equipped with a decoder, and others are not. Suppose further that a broadcast network is used to send data to the receivers. If a non-systematic Raptor code is used for this

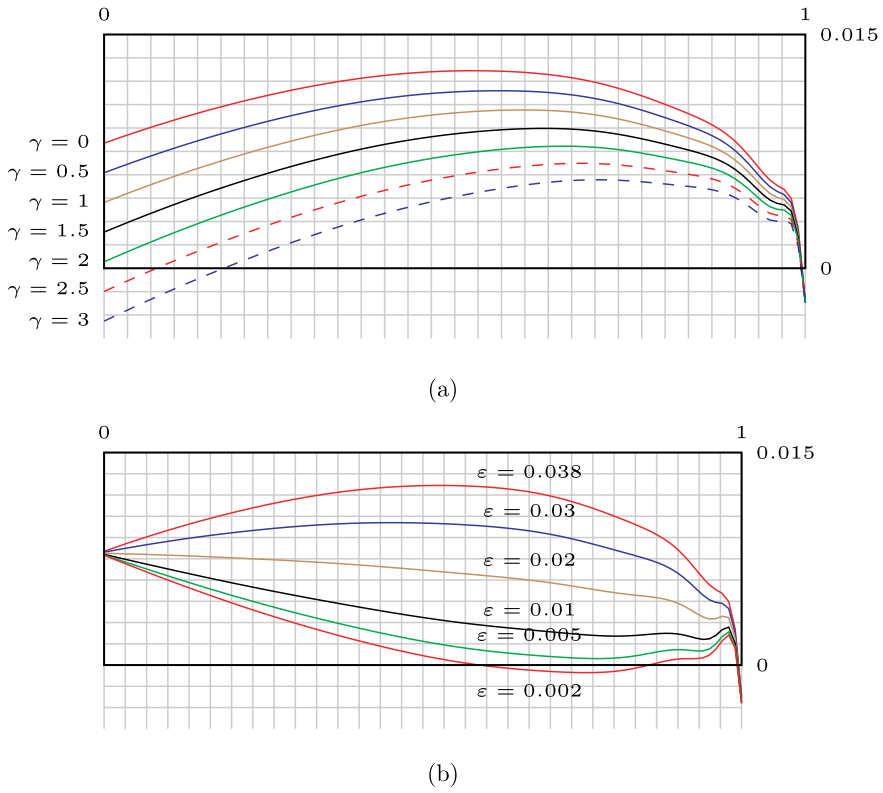


Fig. 2.2 Evolution of the asymptotic and finite length behavior.

application, then the application needs to transmit a stream of source symbols to be used by receivers without a decoder and another stream of encoded symbols to be used by receivers equipped with a decoder. This strategy wastes network resources, i.e., the network resource usage can be essentially double of what it would be if a systematic Raptor code was used instead. There are a variety of other applications for systematic Raptor codes, and thus systematic Raptor codes are preferable to non-systematic Raptor codes.

For systematic Raptor codes, the symbols among the encoded symbols that are not source symbols are called *repair symbols*. A systematic Raptor code designed to successfully decode with overhead $\varepsilon \cdot k$ with high probability should have the following decoding property: For any $m \leq k$,

reception of m source symbols among $x_1, \ldots, x_k$ and $k \cdot (1 + \varepsilon) - m$ repair symbols among $y_{k+1}, y_{k+2}, \ldots$ is sufficient to decode $x_1, \ldots, x_k$ with high probability. In other words, the overhead-failure curve should be independent of the mix of received source symbols and repair symbols. This decoding property is analogous to the decoding property for non-systematic Raptor codes and has the feature that the mix between the received number of source symbols and repair symbols has no impact on the decodability of the source block; it is only the total number of encoded symbols received that determines decodability.

One possible trivial construction of a systematic Raptor code is to simply use the encoded symbols generated from a non-systematic Raptor code as the repair symbols and then just designate the source symbols of the source block to also be encoded symbols. This trivial construction works very poorly with respect to the systematic decoding property described above, i.e., the overhead-failure curve depends strongly on the mix of received source symbols and repair symbols, and is particularly bad when among the received encoded symbols a small fraction are source symbols and a large fraction are repair symbols. Details are provided in [21].

An entirely different approach is thus needed to design systematic Raptor codes. Such an approach is outlined in [20, 24]. The main idea behind the method is the following: Suppose we want to construct a systematic Raptor code S to encode a source block with k source symbols $(x_1, \ldots, x_k)$ to generate repair symbols $(y_{k+1}, y_{k+2}, \ldots)$, and thus the encoded symbols are $(y_1, \ldots, y_k, y_{k+1}, y_{k+2}, \ldots)$, where $(y_1, \ldots, y_k) = (x_1, \ldots, x_k)$. We use a non-systematic Raptor code R in an unusual way. Suppose that from a source block of k source symbols $(z_1, \ldots, z_k)$, R can generate encoded symbols $(w_1, w_2, \ldots)$. Furthermore, suppose R has the property that the first k encoded symbols $(w_1, \ldots, w_k)$ can be used to decode the source block. The encoding of repair symbols for S proceeds as follows using the encoder and decoder for R.

- Set $(w_1, \ldots, w_k) = (x_1, \ldots, x_k)$.
- Use the decoder for R to decode the values of its source block $(z_1, \ldots, z_k)$ from the encoded symbols $(w_1, \ldots, w_k)$=$(x_1, \ldots, x_k)$.

- Use the encoder for R to generate w_{k+i} from source block $(z_1, \ldots, z_k)$, and then set $y_{k+i} = w_{k+i}$, to generate repair symbol y_{k+i}.

Thus, the systematic Raptor code S for encoding uses a non-systematic Raptor code R to first decode and then to encode.

Suppose that a receiver has received encoded symbols $(y_{i_1}, \ldots, y_{i_n})$ generated according to the encoding algorithm for S as described above, i.e., there are $n \geq k$ encoded symbols received in total. Suppose further that m of the encoded symbols $(y_{i_1}, \ldots, y_{i_m})$ are original source symbols of S, i.e., $(x_{i_1}, \ldots, x_{i_m}) = (y_{i_1}, \ldots, y_{i_m})$, and that the remaining $n - m$ encoded symbols $y_{i_{m+1}}, \ldots, y_{i_n}$ are repair symbols generated using the encoder for S described above. Decoding of the source block for S proceeds as follows using the encoder and decoder for R.

- For $\ell = 1, \ldots, n$, set $w_{i_\ell} = y_{i_\ell}$.
- Use the decoder for R to decode the values of its source block $(z_1, \ldots, z_k)$ from the encoded symbols $(w_{i_1}, \ldots, w_{i_n})$.
- Use the encoder for R to generate $(w_1, \ldots, w_k)$ from source block $(z_1, \ldots, z_k)$, and then set $(x_1, \ldots, x_k) = (w_1, \ldots, w_k)$ to recover the original source block.

Thus, the systematic Raptor code S for decoding also uses the non-systematic Raptor code R to first decode and then to encode. There are many optimizations possible, including not encoding to recover already received source symbols in the last step.

An example of a systematic Raptor code together with its encoding procedure is provided in Figure 2.3.

The basic idea behind this approach is that the source symbols of the systematic Raptor code S can be viewed as randomly generated encoded symbols of the non-systematic Raptor code R, i.e., they are indistinguishable in terms of how they were generated from the other randomly generated encoded symbols of R that are used as the repair symbols for S. In terms of decodability of the intermediate block of R, it does not matter what mix of source and repair symbols are received; they are all just randomly generated encoded symbols. Once the intermediate block of R has been decoded, any missing source symbols of

$$
\begin{aligned}
z_1 &= x_1 \\
z_5 &= x_2 + z_1 \\
z_8 &= x_6 + z_5 \\
z_4 &= x_5 + z_8 \\
z_3 &= x_3 + z_1 + z_4 \\
z_6 &= x_4 + z_1 + z_3 + z_4 \\
z_2 &= s_2 + z_1 + z_4 + z_5 + z_6 + z_8 \\
z_7 &= s_1 + z_3 + z_4 + z_6 \\
\hline
y_1 &= z_2 + z_4 + z_5 + z_7 \\
y_2 &= z_2 + z_3 + z_6 \\
y_3 &= z_3 + z_5 + z_8
\end{aligned}
$$

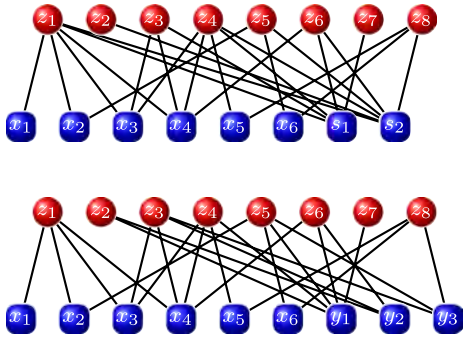


Fig. 2.3 Toy example of a systematic Raptor code S constructed from a non-systematic Raptor code R. The top diagram shows the six source symbols $x_1, \ldots, x_6$ of S placed at the first six encoded symbol values of R, and the constraint symbols s_1, s_2 describe the relations dictated by the precode $\mathcal{C}$ of R, and their values are 0. In a first step, the intermediate symbols $z_1, \ldots, z_8$ of R are obtained from the source symbols of S by applying the decoder for R. The sequence of operations leading to the z_i is given on the left. The bottom diagram shows how the repair symbols of S are generated from these intermediate symbols of R. In this example, three repair symbols y_1, y_2, y_3 are generated. Note that by construction the x_i are also XOR's of those z_i to which they are connected.

S can be recovered using the encoder for R applied to this block. This is intuitively why the systematic Raptor code has the decoding property described previously, i.e., the systematic Raptor code S essentially inherits the overhead-failure curve of the non-systematic Raptor code R upon which it is based, but with the additional feature that the source symbols of S are among the encoded symbols that R generates.

Note that the trivial construction described previously did not have the property that the source symbols were generated from the same distribution as the repair symbols, and this is at the heart of why it performs so poorly in terms its overhead-failure curve.

One very important assumption in the design of the systematic Raptor code construction described above is that the first k encoded symbols generated by the non-systematic Raptor code R should be able to decode the source block of k source symbols. We call such a code construction a *systematic code construction*. We describe methods for generating

a systematic code construction for k from a value $J(k)$, which we call the *systematic index*. The quality of systematic code constructions can differ in terms of the goodness of the overhead-failure curve and in terms of computational complexity of the encoder and decoder. For each relevant value of k, finding a systematic index $J(k)$ that generates a good systematic construction can be done offline. Then, the systematic indices that have been found can be incorporated into the specification of the systematic Raptor code S to provide a full specification. For example, for the standardized Raptor code R10 [1, Annex B] discussed in Section 3.1, there is a 16-bit systematic index $J(k)$ specified for each k between 1 and 8,192. The specific designs of Raptor codes provided in Section 3 describe systematic indices in more detail.

2.4 Inactivation Decoding

2.4.1 Outline and an Example

As discussed earlier, using belief-propagation decoding could require a large overhead for small values of k to achieve a reasonably small failure probability. To remedy this situation, a different decoding algorithm has been devised [22]. Called *inactivation decoder*, this decoder combines the optimality of the Gaussian elimination with the efficiency of the belief-propagation algorithm. It has been inspired by the algorithm in [8, 17] and has some similarities to the algorithms in [19].

It is best to describe inactivation decoding using matrix notation. We refer to the received encoded symbols and the constraint symbols as *row symbols*, and we refer to the intermediate symbols, which are a combination of the source symbols and redundant symbols, as *column symbols*. From the decoding graph of the code, we can obtain a matrix representation in a straightforward manner: the rows of the matrix correspond to the row symbols and the columns of the matrix correspond to the column symbols. There is a one at position (i, j) of the matrix if and only if column symbol j contributes to the value of row symbol i, and otherwise there is a zero. The decoding then corresponds to solving a system of linear equations: the goal is to use the row symbols to solve for the column symbols, where there are at least as many row symbols as column symbols.

Inactivation decoding is useful in conjunction with the scheduling process alluded to in Section 2.1 and outlined in [1, Annex C], and belief propagation is used in a modified form. Initially, all column symbols are active and all row symbols are unpaired. At each belief-propagation step, the degree of an unpaired row symbol is the number of active column symbols upon which it depends, and thus initially the degree of each row symbol is its original degree. Belief propagation is used to find an unpaired row symbol of degree one, at which point this unpaired row symbol can be paired with the one remaining active column symbol upon which it depends. Then, the paired row symbol is subtracted from the values of all other unpaired row symbols that depend on that active column symbol, thereby removing their dependence on that active column symbol and thus reducing their degree by one.

Belief propagation repeats this process until either all active column symbols are paired, or until there is no unpaired row symbol of degree one. In the latter case, when there is no unpaired row symbol of degree one but there are still active column symbols that are not paired with a row symbol, an unmodified belief propagation would be stuck and decoding would fail, even if is mathematically possible to decode all of the column symbols. Instead, the modified belief propagation used for inactivation decoding continues decoding as follows: if all unpaired row symbols are of degree zero then it is not mathematically possible to decode all of the column symbols and the process fails; if instead there is at least one unpaired row symbol of degree two then one of the active column symbols upon which this row symbol depends is declared *inactivated*, and thus the degree of the unpaired row symbol is reduced from two to one and belief propagation can continue to pair remaining active column symbols with unpaired row symbols as described above. (One may have to inactivate more than one column symbol because the smallest degree of an unpaired row symbol may be greater than two; this happens rarely for a Raptor code with a good degree distribution.) Belief propagation finishes successfully if all of the active column symbols are paired with row symbols by this process.

Unlike when belief propagation is used without inactivation, the final value at the end of this process of each active column symbol is not necessarily the value of the row symbol with which it is paired, as the

row symbol may still depend on some inactivated column symbols, but it is the case that the row symbol does not depend on any active column symbols other than the one with which it is paired. Thus, the paired row symbols can be used to remove the dependency of the unpaired row symbols on the active column symbols: for each unpaired row symbol and for each active column symbol upon which it depends, subtract from the unpaired row symbol the value of the row symbol paired with that active column symbol.

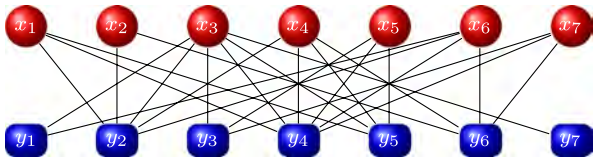
At this point the unpaired row symbols depend only on the inactivated column symbols, and this defines a linear system of equations that can be used to solve for the values of the inactivated column symbols, for example, using the Gaussian elimination. If this system of equations cannot be used to solve for the inactivated column symbols then it is not mathematically possible to decode all of the column symbols.

Finally, the values for the active column symbols can be solved using another round of belief propagation based on the original values of the paired row symbols and the values of the inactivated column symbols. This process is guaranteed to recover all of the active column symbols if the previous belief propagation and Gaussian-elimination processes were successful.

The main motivation behind inactivation decoding is to provide a much more powerful decoding algorithm, i.e., decode whenever mathematically possible, but at the same time employ efficient belief propagation decoding as much as possible. The key to the design of the linear system of equations is to ensure that the matrix is full rank with high probability (this guarantees successful decoding), while at the same time minimizing the total number of decoding symbol operations. For example, a design is good if the average degree of a row symbol is constant and if the number of inactivated column symbols is proportional to the square root of the total number of column symbols, as this means that the total number of symbol operations for inactivation decoding is linearly proportional to the number of column symbols. This is because the number of symbol operations to solve for the inactivated column symbols using the Gaussian elimination is bounded by the square of the number of inactivated column symbols, and the number of symbol operations for the belief propagation decoding steps is linear in the number of non-zero entries in the original matrix.

One can view the Gaussian elimination as a special case of inactivation decoding in which inactivation is done at every step. Successful decoding using only the belief-propagation decoding algorithm is also a special case: here the number of inactivations is zero.

For example, the decoder for the graph shown below



corresponds to finding the solution x of the system of equations

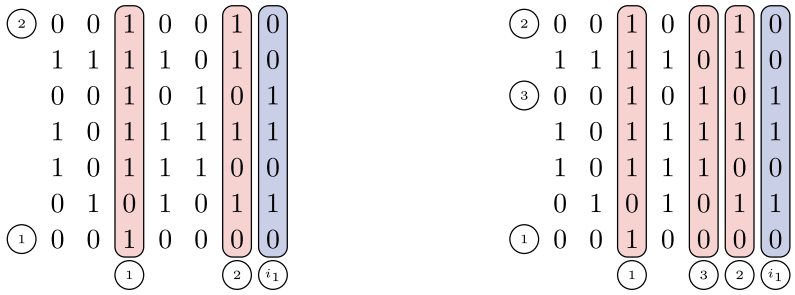
$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix}.$$

Our first goal is to bring this matrix into almost triangular form using a series of row and column permutations, following the belief-propagation decoding algorithm described in Section 2.1. To begin with, we identify a row of degree one, remove it from the set of rows, and continue with identifying another row of degree one, etc. In this particular case, this process can be repeated twice:

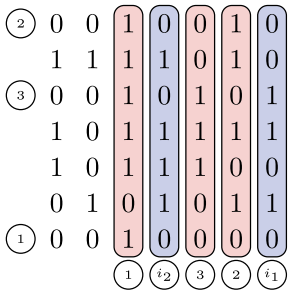
$$\begin{array}{cccccccc} & 0 & 0 & \boxed{1} & 0 & 0 & 1 & 0 \\ & 1 & 1 & \boxed{1} & 1 & 0 & 1 & 0 \\ & 0 & 0 & \boxed{1} & 0 & 1 & 0 & 1 \\ & 1 & 0 & \boxed{1} & 1 & 1 & 1 & 1 \\ & 1 & 0 & \boxed{1} & 1 & 1 & 0 & 0 \\ & 0 & 1 & \boxed{0} & 1 & 0 & 1 & 1 \\ \textcircled{1} & 0 & 0 & \boxed{1} & 0 & 0 & 0 & 0 \\ & & & \textcircled{1} & & & & \end{array} \qquad \begin{array}{cccccccc} \textcircled{2} & 0 & 0 & \boxed{1} & 0 & 0 & \boxed{1} & 0 \\ & 1 & 1 & \boxed{1} & 1 & 0 & 1 & 0 \\ & 0 & 0 & \boxed{1} & 0 & 1 & 0 & 1 \\ & 1 & 0 & \boxed{1} & 1 & 1 & 1 & 1 \\ & 1 & 0 & \boxed{1} & 1 & 1 & 0 & 0 \\ & 0 & 1 & \boxed{0} & 1 & 0 & 1 & 1 \\ \textcircled{1} & 0 & 0 & \boxed{1} & 0 & 0 & \boxed{0} & 0 \\ & & & \textcircled{1} & & & \textcircled{2} & \end{array}$$

The circled numbers in the rows show the order in which the rows are chosen, whereas the circled numbers under the columns show the

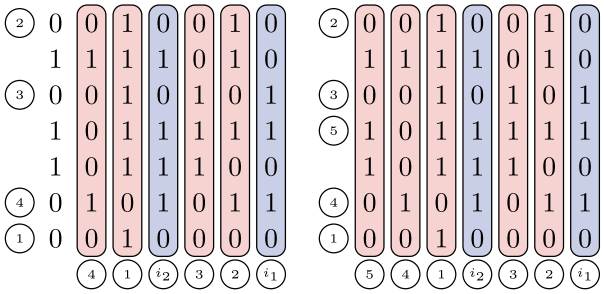
order in which the x_i are recovered. A column is colored pink if it has been recovered. After the second step, the process seems to terminate since there are no rows that have exactly one 1 outside the pink columns. This is where the inactivation kicks in. We inactivate one of the columns, for example, the last, and color it light blue. We call the light blue columns *inactive*. In contrast, we call the pink columns *active*. This has the effect that a row of degree one outside the pink and the blue columns is created.



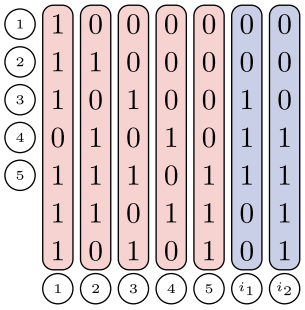
Again, we are in a situation where no row has exactly one 1 outside the pink and the blue columns. We inactivate another column, say column number 4.



Now we can find a row of degree one for two consecutive iterations.



Reordering rows and columns of this matrix according to the circled numbers (and assuming that the non-numbered rows get the numbers 6 and 7, and that the columns i_1 and i_2 are the last two columns of the matrix), we obtain the following matrix:



Now, the key is that the matrix given by the active columns (the pink part) is lower triangular and *sparse*. We start eliminating the rows of the entire matrix by following the columns of the active part of the matrix.

$$
\begin{array}{ccccccc}
\downarrow & & & & & & \\
1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 & 1 & 1 \\
0 & 1 & 1 & 0 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 & 1 & 0 & 1
\end{array}
\rightsquigarrow
\begin{array}{ccccccc}
 & \downarrow & & & & & \\
1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 & 1 & 0 & 1
\end{array}
\rightsquigarrow
$$

$$
\begin{array}{ccccccc}
 & & \downarrow & & & & \\
1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 1
\end{array}
\rightsquigarrow
\begin{array}{ccccccc}
 & & & \downarrow & & & \\
1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 1
\end{array}
\rightsquigarrow
$$

$$\begin{array}{ccccccc} & & & & \downarrow & & \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{array}$$

The final matrix has a very simple form: the active columns contain only one 1, and the last 2 rows of the active part are zero. Performing the permutations of columns on the vector x, and the permutation of rows as well as the elimination steps on the vector y to obtain the vector $\hat{y}$, we arrive at the following simple system of linear equations:

$$\left(\begin{array}{ccccc|cc} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ \hline 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{array}\right) \cdot \begin{pmatrix} x_3 \\ x_6 \\ x_5 \\ x_2 \\ x_1 \\ x_7 \\ x_4 \end{pmatrix} = \begin{pmatrix} \hat{y}_7 \\ \hat{y}_1 \\ \hat{y}_3 \\ \hat{y}_6 \\ \hat{y}_4 \\ \hat{y}_2 \\ \hat{y}_5 \end{pmatrix}. \tag{2.3}$$

From this, we can recover x_7 and x_4 using the lower right 2×2-submatrix of the matrix above:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{pmatrix} x_7 \\ x_4 \end{pmatrix} = \begin{pmatrix} \hat{y}_2 \\ \hat{y}_5 \end{pmatrix}.$$

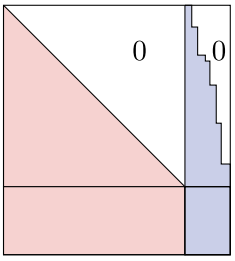
We obtain $x_7 = \hat{y}_5$, and $x_4 = \hat{y}_2 + \hat{y}_5$. Going back to (2.3), we obtain further

$$x_3 = \hat{y}_7, \quad x_6 = \hat{y}_1, \quad x_5 = \hat{y}_3 + x_7, \quad x_2 = \hat{y}_6 + x_4 + x_7,$$
$$x_1 = \hat{y}_4 + x_4.$$

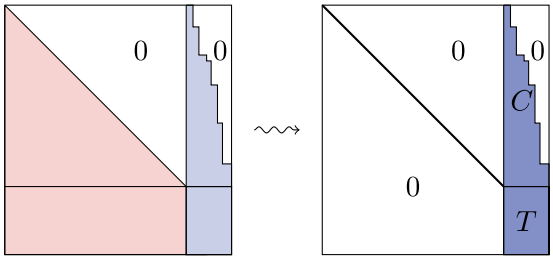
2.4.2 The General Case

In general, the goal of the inactivation decoding is to use row and column permutations to transform the decoding matrix to the

following form:



Then, we eliminate the entries below the diagonal of the left part of the matrix to obtain another one:



The darker tone of blue on the right figure is to indicate that the corresponding matrices have become denser during the elimination process. In practice, we only need to keep track of the matrices on the right since we do know the final form of the matrices on the left. This leads to a reduction in memory. This elimination step is accompanied by XORing the row symbols y_i, which can potentially be costly. However, as the matrices on the left are sparse, the cost of this step in terms of XOR's of row symbols is linear in the number of row symbols.

The decoding process boils down to solving a system of equations of the form

$$T \cdot u = v, \tag{2.4}$$

wherein u is a subvector of x, and v is a subvector of $\hat{y}$, and $\hat{y}$ is the vector obtained from y after permuting the rows and applying the elimination process. If T is full-rank, this system can be solved using the Gaussian elimination. Once it is solved, the entries of u are used

in conjunction with the matrix C to obtain the values of the other entries of the unknown vector x. In practice, the solution of the system of Equation (2.4) is not costly, if T has a number of columns that is at most of the order of square root of the number k of source symbols, and the symbol size is not too small. To do so, an invertible square submatrix of T is chosen whose number of columns, g, equals that of T, its inverse is calculated and multiplied with a suitable subvector of v to obtain u. This multiplication requires $O(g^2)$ XOR's of symbols; hence, if g is of the order of $\sqrt{k}$, then this cost is linear in k.

For a detailed description of other techniques and inactivation strategies, we refer the reader to [22].

3

Standardized Raptor Codes

This section discusses the design and implementation of two of the most successful commercial versions of Raptor codes: the R10 code and the RQ (RaptorQ) code. They are codes of different natures: the R10 code was designed for range of applications with relatively modest requirements, e.g., mobile broadcast applications where fast encoding and decoding, reasonable overhead-failure curves, and support for medium-sized source blocks are sufficient. The RQ code, on the other hand, was designed for a much larger range of applications with more stringent requirements, e.g., high-end streaming applications where fast encoding and decoding and exceptional overhead-failure curves are crucial, or large mobile data delivery applications where support for large source blocks is mandatory.

3.1 Standardization

The R10 code has already been adopted into a number of different standards:

(1) 3GPP Multimedia Broadcast Multicast Service (3GPP TS 26.346), where the R10 code together with associated file

delivery and streaming technology was adopted for broadcast/multicast file delivery and streaming applications.

(2) IETF RFC 5053, where the R10 code together with associated file delivery technology was adopted for broadcast/multicast file delivery applications.

(3) OMA Mobile Broadcast Services V1.0 — Broadcast Distribution System, where the R10 code together with associated file delivery and streaming technology was adopted for broadcast/multicast file delivery and streaming applications.

(4) BMCO Forum Recommendation for Implementation Profile: OMA BCAST, where the R10 code together with associated file delivery and streaming technology was adopted for broadcast/multicast file delivery and streaming applications.

(5) IP Datacast (DVB-IPDC) (ETSI TS 102 472 v1.2.1) for DVB-H and DVB-SH, where the R10 code together with associated file delivery technology was adopted for broadcast/multicast file delivery applications.

(6) IPTV (DVB-IPTV) (ETSI TS 102 034 v 1.3.1) Streaming, where the R10 code together with associated streaming technology was adopted for broadcast/multicast streaming applications.

(7) MPE IFEC Satellite Handheld (DVB-SH) (DVB Bluebook A131), where the R10 code together with associated link layer technology was adopted for broadcast/multicast data delivery applications.

(8) DVB Bluebook A054r4, "Interaction channel for Satellite Distribution Systems (draft EN 301 790 V1.5.1 — DVB-RCS+M), where the R10 code together with associated link layer technology was adopted for broadcast/multicast data delivery applications.

(9) IPTV (DVB-IPTV) (DVB A086r7, draft ETSI TS 102 034 v 1.4.1) Content Download, where the R10 code together with associated file delivery technology was adopted for broadcast/multicast file delivery applications.

(10) ATIS IIF Media Formats and Protocols specification (WT 18), where the R10 code together with associated streaming technology was adopted for broadcast/multicast streaming applications.
(11) Recommendation ITU-T H.701 (2009), Content delivery error recovery for IPTV services, and Recommendation ITU-T H.721 IPTV terminal devices: Basic model, where the R10 code together with associated streaming technology was adopted for broadcast/multicast streaming applications.

Some other standard bodies are adopting the R10 code and the RQ code as well, among them ATSC NRT, 3GPP2 BCMCS, and the IETF FECFRAME working group.

We present the basic design principles behind R10 and RQ, and discuss some of the properties of these codes. We also make explicit references to the specifications of these codes used for their standardization.

3.2 The R10 Code (Raptor 10)

3.2.1 Theoretical Background

In this section, we introduce the first Raptor code that has been adopted into a number of different standards, called either the Raptor 10 code or the R10 code. The first standard to adopt the R10 code was 3GPP MBMS [1]. Thereafter, other standards such as IETF [15], DVB [6], and others followed.

The R10 code is designed to work on source blocks of up to 8,192 source symbols, and supports up to 65,536 encoded symbols. The R10 code overhead-failure curve is designed so that it drops off quickly to achieve a failure probability of around 10^{-6} with an overhead of a few symbols, for the entire range of supported number of source symbols per source block.

Perhaps the first question to ask is what could be expected in the best case for the overhead-failure curve of the decoder and the number of overhead symbols. As received encoded symbols are generated by

a random process that is independent between encoded symbols, it is natural not to expect a better failure probability than that of a random binary fountain code. One of the ideas that one might have would be to actually use a random binary fountain code, at least when the number of source symbols is not large. However, this only shifts the problem because there will be an intermediate range where the encoding/decoding complexity of the random binary fountain code is prohibitive, and the number of source symbols is still not large enough for the random processes involved to concentrate tightly around their expectations. Moreover, it is also unclear how to best switch from a random binary fountain code to whatever other solution we may find for larger numbers of source symbols.

For these reasons, it is better to have a uniform description of the code for the entire range of the number of source symbols.

Inactivation decoding described in Section 2.4 is key to the design of the R10 code. It not only offers improved decoding, but it also shifts the analysis from belief-propagation decoding to that of the rank deficiency of the decoding matrix. The latter may in some cases be easier to analyze, at least heuristically. On the other hand, the design of short fountain codes (or LDPC codes, for that matter) that can decode at very low overheads with a small failure probability using the belief-propagation algorithm alone remains a tough, and to a large extent unsolved, problem.

How can we go about designing a Raptor code that behaves like a random binary fountain code, but is much more efficient? The key to this solution is a simple observation: *to obtain the decoding failure probability of a random binary fountain code it suffices to choose the neighbors of only some of the constraint symbols like in a random binary fountain.* In the following, we are going to justify (and also qualify) this assertion.

To begin with, suppose that we have a Raptor code that uses a sparse LT code and a sparse precode represented by an $N \times n$ decode matrix A, where N is the total number of encoded symbols and constraint symbols represented by A and n is the number of intermediate symbols. Suppose A does not represent all the encoded and constraint symbols, i.e., $N = n - H$, and we add another matrix B with H rows

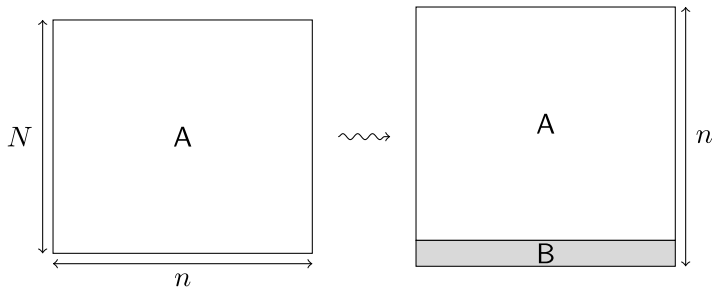


Fig. 3.1 Adding dense rows to mimic the failure probability of a random binary fountain code.

to the matrix A representing encoded or constraint symbols chosen uniformly and independently from $\mathbb{F}_2$, so that we have n encoded and constraint symbols in total. The situation is depicted in Figure 3.1.

What is the rank of this new combined matrix? This problem has been discussed in detail in Appendix A. Suppose that t is the probability that the rank of the matrix A is N, i.e., that A is full rank. Then, the probability that the combined matrix is full rank n is $t \cdot \prod_{i=1}^{H}(1 - \frac{1}{2^i})$. Note that $\prod_{i=1}^{H}(1 - \frac{1}{2^i})$ is the probability that an $H \times H$-matrix with uniform and independent elements from $\mathbb{F}_2$ is invertible. Hence, if $t \approx 1$, then the probability that the combined matrix is invertible (and hence of rank n) is almost the same as the probability that a random square $\mathbb{F}_2$-matrix is invertible.

A first idea would be to modify the LT distribution used for the Raptor code to allow for the generation of completely uniformly generated random linear combinations every once in a while, so that in the combined matrix there are H such rows with high probability. The idea is straightforward, but not great. First, the number of "heavy symbols" (those corresponding to matrix B) is not deterministic and has a variance. This leads to an uncertainty of the rank of the combined matrix. Second, the symbols generated by the fountain code may not have a particular structure, and their generation (and later elimination from the matrix) may not be efficient.

3.2.2 Dense Rows, and the Design

A much better idea is to have these heavy symbols as constraint symbols. More precisely, we choose the constraint matrix of the precode

of our Raptor code to have the matrix B shown in Figure 3.1 as a submatrix. (We call the code used to generate the redundant symbols of the intermediate symbols the precode.) In this manner, we are sure that B is always a part of the decoding matrix. A similar idea had already been proposed for one of the first versions of Raptor codes, e.g., the one appearing in [20]. There, B was chosen to be the check matrix of a Hamming code.

For reasons of computational efficiency, it is important that B is not really uniformly random. In fact, what is important is that B behaves as if it were uniformly random in terms of its rank properties, and that there is a fast algorithm for multiplying B with a vector of length n. To see why this is, we remind the reader of inactivation decoding in Section 2.4. After a suitable rearrangement of the columns of the decoding matrix, we arrive at a system of linear equations of the following type:

$$\begin{array}{|c|c|}\hline \multirow{2}{*}{L} & \multirow{2}{*}{A_1} \\ & \\ \hline \mathsf{C}_1 & \mathsf{C}_2 \\ \hline \multicolumn{2}{|c|}{\mathsf{P}\cdot\mathsf{B}\cdot\mathsf{Q}} \\ \hline \end{array} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix}.$$

Here, L is a lower triangular matrix, and P and Q are permutation matrices. The task is to eliminate all the entries below L. This is done by multiplying the above system from the left with an appropriate matrix:

$$\left(\begin{array}{c|c} \mathsf{L}^{-1} & 0 \\ \hline \left(\dfrac{\mathsf{C}_1}{\mathsf{P}\cdot\mathsf{B}\cdot\mathsf{Q}_1}\right)\cdot\mathsf{L}^{-1} & -\mathsf{I} \end{array}\right) \cdot \left(\begin{array}{c|c} \mathsf{L} & \mathsf{A}_1 \\ \hline \mathsf{C}_1 & \mathsf{C}_2 \\ \hline \mathsf{P}\cdot\mathsf{B}\cdot\mathsf{Q}_1 & \mathsf{P}\cdot\mathsf{B}\cdot\mathsf{Q}_2 \end{array}\right) \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

$$= \left(\begin{array}{c|c} \mathsf{L}^{-1} & 0 \\ \hline \left(\dfrac{\mathsf{C}_1}{\mathsf{P}\cdot\mathsf{B}\cdot\mathsf{Q}_1}\right)\mathsf{L}^{-1} & -\mathsf{I} \end{array}\right) \cdot \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix}.$$

Here Q_1 is the matrix consisting of the first ℓ columns of Q where ℓ is the size of L, and Q_2 is the matrix consisting of the last $n - \ell$ columns of Q. In practice, what is important is how fast we can calculate the right-hand side of this equation. There are three quantities to calculate

$$z = \mathsf{L}^{-1} \cdot \begin{pmatrix} y_1 \\ \vdots \\ y_\ell \end{pmatrix}, \quad \mathsf{C}_1 \cdot z, \quad \mathsf{P} \cdot \mathsf{B} \cdot \mathsf{Q}_1 \cdot z.$$

The first one, z, is easy to calculate, as L is sparse and lower triangular (essentially, we will use the belief-propagation algorithm). The second one is also easy to calculate as C_1 is sparse. It is the third quantity for which we need an efficient algorithm for the multiplication of B with a generic vector: as P is a permutation matrix, and Q_1 is also almost a permutation matrix (it consists of the first ℓ columns of a permutation matrix), the algorithm for B can be used to efficiently calculate the third quantity as well.

To design a matrix B that "looks" random and has an efficient multiplication algorithm, we proceed as follows. We set

$$\hat{\mathsf{B}} = (T_1 \mid T_2 \mid \cdots \mid T_{n-H}) \cdot \begin{pmatrix} 1 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 1 & \cdots & 0 & 0 \\ 0 & 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 1 \\ 0 & 0 & 0 & \cdots & 0 & 1 \end{pmatrix}^{-1}, \quad \mathsf{B} = (\hat{\mathsf{B}} \mid I_H), \tag{3.1}$$

where $T_1, \ldots, T_n$ are (mostly) sparse. A judicious, yet efficient, choice of the T_i will ensure that the columns of the matrix $\hat{\mathsf{B}}$ have a Hamming weight that is roughly half of the length of the columns. To do this, a binary reflected Gray code is used. Details are found in Section 3.2.3. The algorithm for multiplying B with a given vector is essentially the same as the one given on Equation (3.3).

For the above ideas to work, it is imperative that the matrix A of Figure 3.1 is such that it has rank $n - H$ with high probability. In fact, to obtain the same performance as a random binary fountain code also

in terms of the decay of the failure probability as a function of the overhead, it is important that the first few terms in the rank profile[1] of the matrix A behave as the first few terms of a random binary matrix of the same size. (It is not possible to get the same performance as that of a random binary matrix for a sparse matrix over the entire length of the rank profile; however, it is possible to do so for an initial segment of it, and this is mostly sufficient in practice.) To achieve this, a combination of a good LT code and a good precode is needed.

As the dense part of the constraint matrix of the precode is already fixed, we need to specify the sparse part of the constraint matrix. A lot of designs are possible for this part. What is important is to make sure that all the intermediate symbols are covered more or less the same number of times, and that this number is not too small (in practice, 3 works quite well, and this is what the R10 code does).

The design of the LT code is the most challenging task. We do not expect to use a degree distribution that is very much like the ideal distribution, because such a distribution often leads to various deficiencies in terms of the rank of the corresponding matrix. On the other hand, the distribution should not be too far from the ideal distribution either, because otherwise the decoding will not be very efficient and we would need a lot of inactivations. The actual design follows a number of heuristics and proceeds to some extent using trial and error. The final outcome of the design is given in Section 3.2.3.

3.2.3 Construction

In this section, we give the details of the construction of the R10 code. In doing so, we make references to the specification of this code [15] and use the same terminology. The R10 code is systematic fountain code designed to support a number of source symbols K between 4 and 8,192.

Associated with each encoded symbol of the R10 code is an ESI, which is a 16-bit value that identifies the encoded symbols of a source block. As the R10 code is a systematic code, the source symbols are among the possible encoded symbols: the source symbols are

[1] We refer the reader to Appendix A for a definition of the rank profile of a matrix

$C'_0, \ldots, C'_{K-1}$ with corresponding ESIs $0, \ldots, K-1$, and the repair symbols are $C'_K, C'_{K+1}, \ldots$ with corresponding ESIs $K, K+1, \ldots$.

The encoding and decoding for the R10 code is defined by two types of relationships: *constraint relationships* among the intermediate symbols and *LT relationships* between the intermediate symbols and the encoded symbols. Encoding proceeds by determining the intermediate symbol values based on (1) the source symbol values, (2) LT relationships between the source symbols and the intermediate symbols, and (3) the constraint relationships among the intermediate symbols. The values of repair symbols can be generated from the intermediate symbols based on LT relationships between the intermediate symbols and the repair symbols.

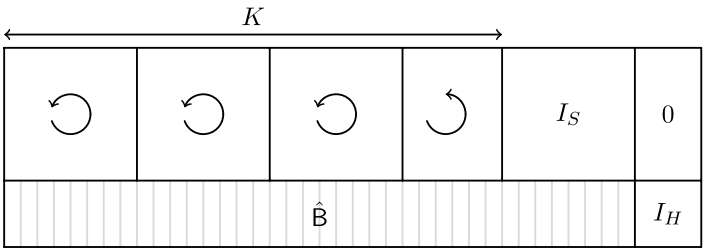
Similarly, decoding proceeds by determining the intermediate symbol values based on (1) the received encoded symbol values, (2) LT relationships between the received encoded symbols and the intermediate symbols, and (3) the constraint relationships among the intermediate symbols. The values of missing source symbols can be generated from the intermediate symbols based on LT relationships between the intermediate symbols and the missing source symbols. Thus, encoding and decoding are essentially symmetric procedures.

The R10 encoder produces $L = K + S + H$ intermediate symbols $C_0, \ldots, C_{L-1}$ from the K source symbols $C'_0, \ldots, C'_{K-1}$, wherein S is the number of LDPC redundant symbols and H is the number of HDPC redundant symbols. The term HDPC stands for "High Density Parity Check" and refers to the fact that these symbols depend on a large number of source symbols. As described in [15, Section 5.4.2.3], the number of LDPC symbols S is the smallest prime number greater than or equal to $\lceil 0.01 \cdot K \rceil + X$ where X is the smallest positive integer such that $X \cdot (X-1) \geq 2K$; the number of HDPC symbols H is the smallest integer such that

$$\binom{H}{\lceil H/2 \rceil} \geq K + S.$$

H is chosen like this so that the columns of the matrix B in Figure 3.2 are all distinct.

Another important parameter for the code is the systematic index $J(K)$, This list of systematic indices for the supported values of K is



i	1	2	3	4	10	11	40
Ω_i	0.00971	0.4580	0.2100	0.1130	0.1110	0.0797	0.0156

Fig. 3.2 The constraint matrix of the precode and the LT degree distribution for the R10 code.

provided in the table shown in [15, Section 5.7]. The systematic index $J(K)$ is used to define the LT relationships that guarantee that the R10 code is a good systematic code for a source block with K source symbols, i.e., that the L intermediate symbols can be decoded from the K source symbols, and that the overhead-failure curve is good.

The constraint matrix described below defines the constraint relationships among the L intermediate symbols $C_0, \ldots, C_{L-1}$. The constraint matrix overall structure is given in Figure 3.2. It consists of two submatrices: the top one, consisting of S rows, describing the LDPC relations, and the second one, consisting of H rows, describing the HDPC relations.

We now describe the two submatrices of the constraint matrix in Figure 3.2 in more detail. In [15], the structure of this matrix is described implicitly in Section 5.4.2.3 as the "For" loop describing the first part of the precoding relations. The first $K + S$ columns of the upper part of the constraint matrix consist of circulant matrices and an $S \times S$ identity matrix. The circulant matrices are composed of K of the columns, and each (except possibly the last) has S columns. The number of these circulant matrices is $\lceil K/S \rceil$. The columns in these circulant matrices all have degree 3. The first column of the ith circulant matrix has ones at positions 0, $(i + 1)$mod-S, and $(2 \cdot i + 1)$mod-S and zeros elsewhere. The other columns are cyclic shifts of the first, wherein by a shift we mean a cyclic down-shift.

The lower H rows of the matrix in Figure 3.2 mimic the behavior of a matrix in which entries are chosen uniformly and independently from $\mathbb{F}_2$. In the specification [15], the symbols obtained by multiplying the non-identity part of the matrix with intermediate symbols are called "Half-Symbols," mostly because the columns of this matrix have a relative weight of $1/2$ (when H is even, and very close to $1/2$ when H is odd).

As was mentioned above, for efficiency reasons, the matrix $\hat{\mathsf{B}}$ is chosen to have a form described in Equation (3.1). Because we want the columns of $\hat{\mathsf{B}}$ to have weight equal to $\lceil H/2 \rceil$, we proceed as follows: we enumerate vectors of this weight and length H using a binary reflected Gray Code in such a way that the Hamming distance between consecutive columns is exactly 2. Specification [15] describes at the end of Section 5.4.2.3 a method of how to obtain these columns using the Gray enumeration of binary vectors of length H. A more efficient algorithm is given in [2]. Denote the columns of $\hat{\mathsf{B}}$ by $G_1, G_2, \ldots, G_{K+S}$. For $i \geq 2$, if T_i denotes the vector of weight 2, which contains ones in positions in which G_i and G_{i-1} differ, and if T_1 is the vector consisting of $\lceil H/2 \rceil$ ones followed by consecutive zeros, then Equation (3.1) is indeed satisfied for $\hat{\mathsf{B}}$, since

$$\begin{pmatrix} 1 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 1 & \cdots & 0 & 0 \\ 0 & 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 1 \\ 0 & 0 & 0 & \cdots & 0 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 & 1 \\ 0 & 1 & 1 & \cdots & 1 & 1 \\ 0 & 0 & 1 & \cdots & 1 & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 1 \\ 0 & 0 & 0 & \cdots & 0 & 1 \end{pmatrix}.$$

We give an example of the constraint matrix of the precode for $K = 10$. In this case, $H = 6$, $S = 7$, and thus there are $L = K + S + H = 23$ intermediate symbols. The specific constraint matrix for the R10 code for $K = 10$ is shown in Figure 3.3, which is a specific instance of the general form of the R10 code constraint matrix shown in Figure 3.2.

We now describe the LT relationship between the encoded symbols $C'_0, \ldots, C'_{K-1}, C'_K, C'_{K+1}, \ldots$, and the intermediate symbols $C_0, \ldots, C_{L-1}$. The LT relationship is defined by the `Triple Generator` described in [15, Section 5.4.4.4], which depends on the systematic

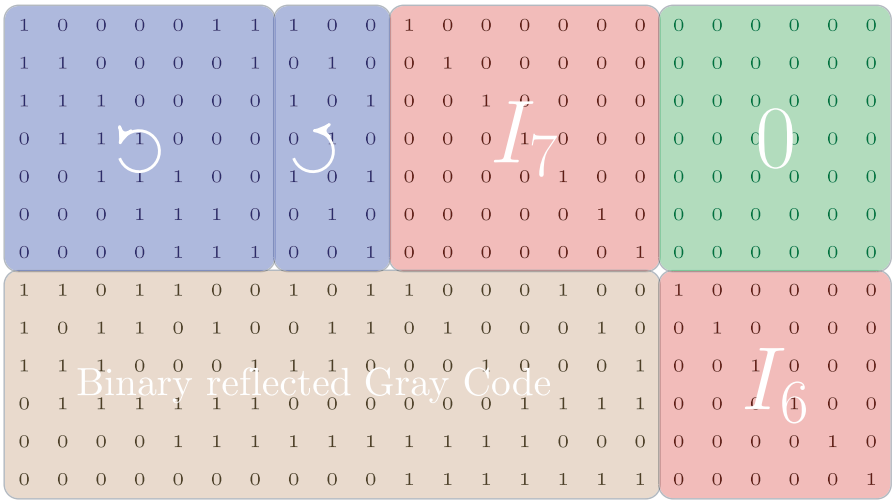


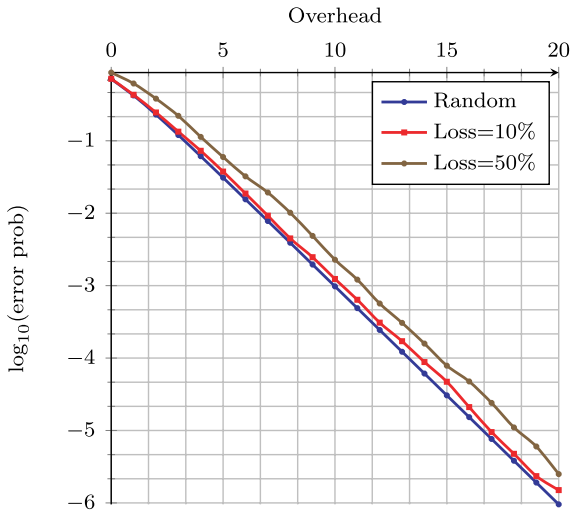
Fig. 3.3 Constraint matrix of the precode of the R10 code for $K = 10$.

index $J(K)$. An ESI X is mapped to a non-negative integer $Y = (B + X \cdot A) \text{mod-}Q$, where $Q = 65,521$, $A = (53,591 + 997 \cdot J(K)) \bmod Q$ and $B = 10,267 \cdot (J(K) + 1) \text{mod-}Q$. Then, the value of Y is mapped to a triple (d, a, b) using the pseudo-random function `Rand` defined in [15, Section 5.4.4.1]. The input to `Rand` is Y, a non-negative integer $i < 256$, and a positive integer m, and it produces an integer between 0 and $m - 1$. The value of i varies in different calls to `Rand` to ensure that the outputs from calls to `Rand` with the same value of y as input are largely independent of one another. To generate the LT degree d, `Rand` is called to generate a random integer v between 0 and $2^{20} - 1$, and then the function `Deg` is called with input v to choose a degree d using a table lookup based on v and Table 1 shown in [15, Section 5.4.4.2], where the degree distribution that this generates is as shown in Figure 3.2. The values of a and b are also generated using `Rand`.

The number d is the LT degree of the encoded symbol and a and b are used to select the subset of size d from the intermediate symbols. More specifically, d is an integer in the set $\{1, 2, 3, 4, 10, 11, 40\}$ and a and b are integers between 0 and $L' - 1$ (and a is non-zero), where L' is the smallest prime number greater than or equal to L.

The triple (d, a, b) is used to define the LT relationships between encoded symbols and intermediate symbols: the value of the encoded symbol C'_X satisfies the following: We generate the numbers $r_0 = a, r_1 = (a + b) \text{mod-} L', r_2 = (2 \cdot a + b) \text{mod-} L', \ldots$ until d of these numbers fall in the interval $[0, L' - 1]$. If these numbers are denoted $\ell_1, \ldots, \ell_d$, then C'_X is the XOR of $C_{\ell_1}, C_{\ell_2}, \ldots, C_{\ell_d}$. An efficient method of computing the value of encoded symbol C'_X based on the triple (d, a, b) is described in [15, Section 5.4.4.3].

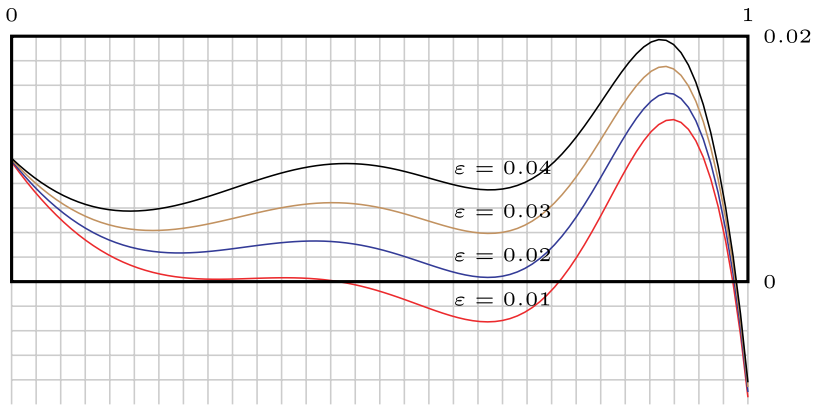
3.2.4 Some Properties



One of the most amazing properties of the R10 code is that its performance is close to that of a random binary fountain code, in terms of its overhead-failure curve for a wide range of values of K and for a wide range of loss probabilities. The following figure provides a plot of the overhead-failure curve for the R10 code for $K = 1{,}000$ and loss rates of 10% and 50%, as well as for the random binary fountain. As the R10 code is systematic, its performance is somewhat sensitive to the loss probability; this is not surprising, since for a loss probability of zero all of the source symbols are received and thus decoding is always successful. As can be seen, in the given range the overhead-failure curve decreases exponentially fast, i.e., the failure probability decreases by

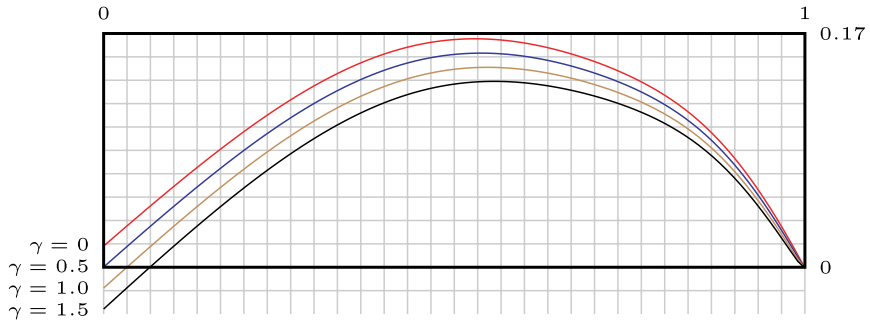
almost a factor of two for each increase of one in the overhead. For values of K up to around 2,000, the overhead-failure curve is essentially the same as that of a random binary fountain code. For values of K larger than 2,000 the overhead-failure curve starts to deviate from that of a random binary fountain code; however, it is still the case that the % overhead-failure curve for these larger values of K is at least as good as the % overhead-failure curve for $K = 2{,}000$, and thus overall in a practical sense the overhead-failure curve is quite good.

The R10 code uses a well-designed structured approach that allows fast encoding and decoding, i.e., orders of magnitude faster encoding and decoding than is possible using a random binary fountain code.



The degree distribution of the R10 code is specifically designed to work well with inactivation decoding. To see this, let us look at the condition in (2.1), i.e., at the smallest root of $1 - x - \mathrm{e}^{-(1+\varepsilon)\Omega'(x)}$ in the interval $[0,1)$. On the right is a plot of this function for various values of ε. It shows that asymptotically an overhead of about 2% is needed for the code to function properly (at least in order to recover around 98% of the intermediate symbols). If we take $K = 1{,}000$, for example, then the number of intermediate symbols of the LT code equals $1{,}072$ (the value of X from the caption of Figure 3.2 is 47, the value of S is 59, and the value of H is 13) and if we use the heuristic described at the end of Section 2.2 and look at the plot of $1 - x - \mathrm{e}^{-(1+\varepsilon)\Omega'(x)} - \gamma\sqrt{(1-x)/1072}$ for $\varepsilon = 60\%$, say, and various

values of γ, we obtain the figure below:



We see that even with such a huge overhead it is unlikely that the belief-propagation decoding succeeds if we choose $K = 1{,}000$. The plots also tell us where the process is worst: at the start. This is because the fraction of encoded symbols of degree one is very small, and there is a good chance that there are very few such encoded symbols received.

Figure 3.4 shows the distribution of the number of inactivations of the R10 code for the case $K = 1,000$ for varying overheads. As can be seen, for small overheads the distribution on the number of inactivations

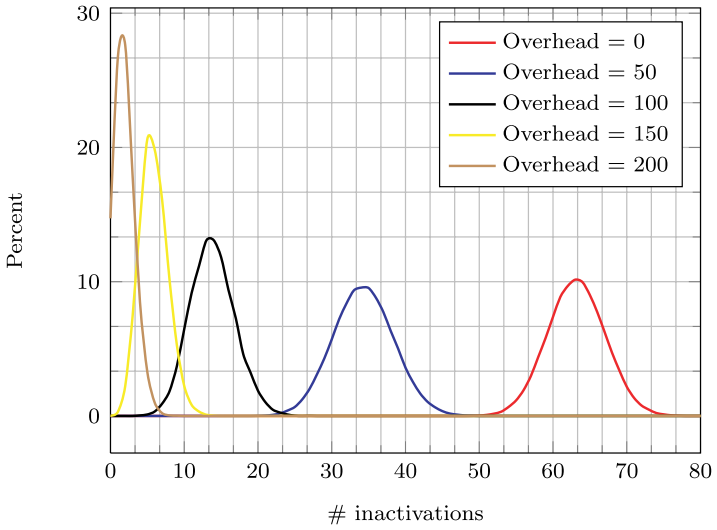


Fig. 3.4 Distribution of the number of inactivations of the R10 code for $K = 1{,}000$ and various overheads o.

is close to normal. For an overhead of 0, we expect (and see) the largest mean which is at approximately 63 inactivations. The average number of inactivations is considerably smaller when the overhead is 50: it is roughly 35. As we increase the overhead, we expect the number of inactivations to decrease, and this can also be seen in Figure 3.4.

3.3 The RQ code

In this section, we introduce the most advanced Raptor code that is available commercially, called either the RaptorQ code or the RQ code. The RQ code is improved compared with the R10 code in many ways. The RQ code supports source blocks of up to 56,403 source symbols, and supports up to 16,777,216 encoded symbols. These limits on supported values for the RQ code were set due to practical considerations based on perceived application requirements, and not due to limitations of the RQ code design.

The RQ code has an overhead-failure curve that essentially mimics that of a random fountain code over the field $\mathbb{F}_{256}$, for the entire range of supported number of source symbols per source block and for any loss probability. The RQ code is a fountain code, i.e., as many encoded symbols as needed can be generated by the encoder on-the-fly from the source symbols of a source block. The RQ code is a systematic code, meaning that all the source symbols are among the encoded symbols that can be generated, and thus encoded symbols can be considered to be a combination of the original source symbols and repair symbols generated by the encoder.

A full specification of the RQ code is provided in [16].

Before describing the RQ code, we describe some properties of the R10 code and the practical implications of these properties that motivated the design and development of the RQ code. Although R10 is a very good systematic fountain code, there are some improvements that would increase its practical application. Two potential improvements of importance are a steeper overhead-failure curve and a larger number of supported source symbols per source block. As we have seen, the R10 overhead-failure curve is essentially that of a random binary fountain code (up to $K = 2,000$). In some applications, the overhead-failure

curve of a random binary fountain code is not ideal. For example, in a streaming application, the range of the number of source symbols in a source block can be wide, e.g., $K = 40, 200, 1000$, and 10,000. To provide a good streaming experience, the failure probability is required to be low, e.g., a failure probability of 10^{-5} or 10^{-6}. As bandwidth is often at a premium for streaming applications, the percentage of repair symbols sent as a fraction of the source symbols should be minimized. Suppose, for example, that the network over which the stream is sent should be protected against up to 10% packet loss when using source blocks with $K = 200$, and the failure probability is required to be at most 10^{-6}. The R10 code requires an overhead of 24 to achieve a failure probability of 10^{-6}, i.e., the receiver needs 224 encoded symbols. A total of 249 encoded symbols need to be sent for each source block to meet the requirements, as $224/(1 - 0.1)$ rounded up is 249. Thus, the repair symbols add an extra 24.5% to the bandwidth requirements for the stream. If instead the RQ codes were used, an overhead of 2 ensures a failure probability of less than 10^{-6}. Thus, then only a total of 225 encoded symbols need to be sent for each source block to meet the requirements, since $202/(1 - 0.1)$ rounded up is 225. In this case, the repair symbols add an extra 12.5% to the bandwidth requirements for the stream, i.e., essentially a factor of two smaller bandwidth overhead than required by the R10 code. Thus, the RQ code overhead-failure curve that is an improvement over the R10 overhead-failure has some very positive practical consequences.

R10 supports up to $8,192$ source symbols per source block. However, there are applications where support for more source symbols per source block is desirable. For example, in a mobile file broadcast application, it is advantageous from a network efficiency point of view to encode the file as a single source block or, more generally, to partition the file into as few source blocks as is practical. Suppose, for example, that a file of 50 million bytes is to be broadcast, and that the available size within each packet for carrying an encoded symbol is 1,000 bytes. To encode the file as a single source block requires $K = 50{,}000$ be supported. (Note that there are sub-blocking techniques, for example, as described in [15] which allow decoding using substantially less memory than the source block size.)

There are a few reasons that the number of source symbols is limited to 8,192 for R10. One reason is that with the R10 degree distribution the failure probability at zero overhead increases to almost 1 as K increases beyond 8,192, making it harder to find systematic indices that yield a good systematic code construction. In contrast, because the overhead-failure curve for the the RQ code design is so steep for all values of K, it is easily possible to find good systematic indices and thus to support much larger values of K.

The RQ code combines two major new ideas to achieve its performance: the first is the use of symbols over larger alphabets, and the second is the use of a technique called "permanent inactivation." In this section, we explain these two concepts, elaborate on the design of the RQ code, and highlight some of the properties of this code.

3.3.1 Non-binary Alphabets

To obtain a better overhead-failure curve, one can use a random fountain code over the field $\mathbb{F}_q$ for larger q. The following simple argument gives an upper bound on the failure probability of such a code as a function of the overhead.

Theorem 3.1. Suppose that $m \geq n$, and that A is an $m \times n$ matrix in which each entry is independently and uniformly chosen from the field $\mathbb{F}_q$. Then, we have

$$\Pr[\mathrm{rk}(\mathsf{A}) < n] \leq \frac{1}{(q-1)q^{m-n}}.$$

Proof. We define an equivalence relation $\sim$ on $\mathbb{F}_q^n \setminus \{0\}$ defined as $x \sim y$ iff there exists $0 \neq \alpha \in \mathbb{F}_q$ such that $\alpha x = y$, and we let V denote a set of representatives of $\sim$-classes. By the union bound, we have

$$\begin{aligned}
\Pr[\mathrm{rk}(\mathsf{A}) < n] &= \Pr[\exists 0 \neq x \in \mathbb{F}_q^n \colon \mathsf{A} \cdot x = 0], \\
&= \Pr[\exists x \in V \colon \mathsf{A} \cdot x = 0], \\
&\leq \sum_{x \in V} \Pr[\mathsf{A} \cdot x = 0].
\end{aligned}$$

Since $\sim$-classes contain $q-1$ elements, and for $x \sim y$ the conditions $\mathsf{A} \cdot x = 0$ and $\mathsf{A} \cdot y = 0$ are equivalent, we have

$$\sum_{x \in V} \Pr[\mathsf{A} \cdot x = 0] = (q-1) \sum_{0 \neq x \in \mathbb{F}_q^n} \Pr[\mathsf{A} \cdot x = 0].$$

Suppose that x is a fixed non-zero element of $\mathbb{F}_q^n$, and that v is a vector chosen uniformly at random from $\mathbb{F}_q^n$. Then, $\Pr[\langle v \mid x \rangle = 0] = 1/q$, where $\langle \cdot \mid \cdot \rangle$ denotes the standard scalar multiplication of vectors. Hence, we have for any fixed $0 \neq x \in \mathbb{F}_q^n$:

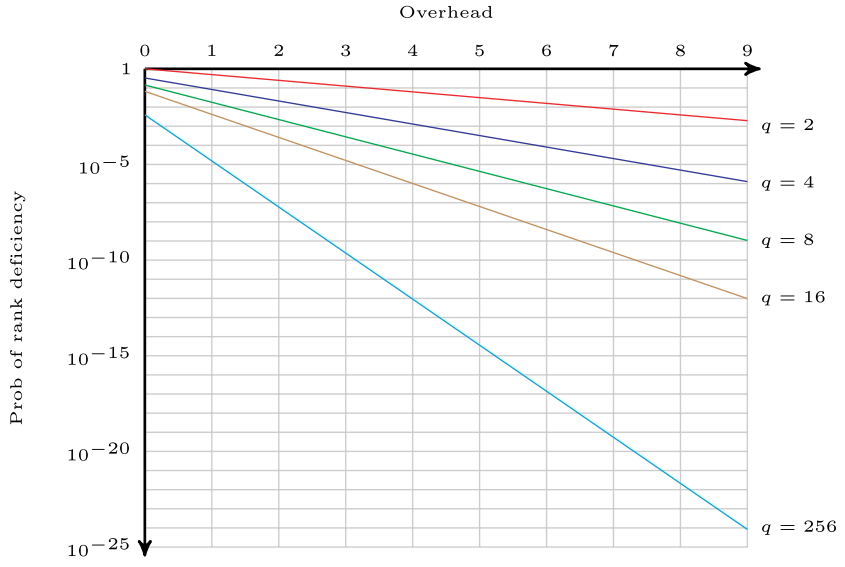
$$\Pr[\mathsf{A} \cdot x = 0] = \frac{1}{q^m}.$$

It follows that

$$\Pr[\mathrm{rk}(\mathsf{A}) < n] \leq \frac{1}{q-1} \sum_{0 \neq x \in \mathbb{F}_q^n} \Pr[\mathsf{A} \cdot x = 0] = \frac{q^n - 1}{(q-1)q^m} < \frac{1}{(q-1)q^{m-n}},$$

which proves the assertion. □

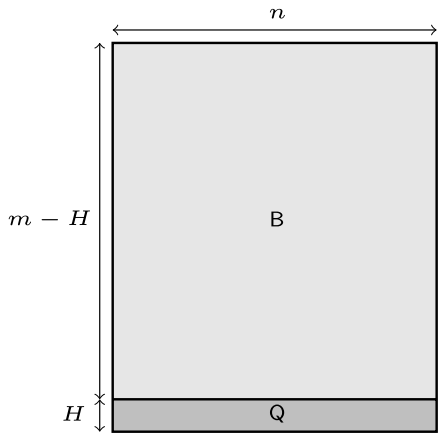
Even though this theorem only gives an upper bound on the probability that the matrix A is rank deficient, this bound is astonishingly close to the exact result. The following is a plot of $\Pr[\mathrm{rk}(\mathsf{A}) < n]$ for a random $m \times n$ matrix A over $\mathbb{F}_q$ as a function of the overhead $m - n$ for various values of q.



As can be seen, the failure probability, i.e., the probability of rank deficiency, as a function of the overhead can be substantially improved by passing to larger alphabets.

How can this observation be used? A naive way would be to use a random fountain code over $\mathbb{F}_q$. However, this leads to a huge penalty in terms of the computational efficiency of the code: on the one hand, as was the case for random binary fountain codes, the random structure of the matrices involved will make it difficult to encode and decode efficiently. On the other hand, the larger finite fields will lead to a further deterioration of the speed of encoding and decoding because of the complications involved in performing arithmetic operations in such fields.

One of the main ideas for the design of the RQ code is to use as part of the precoding a code over $\mathbb{F}_q$, which looks like a random code, but is computationally efficient. To begin with, consider a matrix of the following type:



Herein, B is a binary $(m-h)\times n$ matrix, which is sampled from some probability distribution $\mathcal{D}$ on $\mathbb{F}_2^{(m-h)\times n}$, and Q is a q-ary $h\times n$ matrix in which each entry is sampled independently and uniformly from $\mathbb{F}_q$. What is the probability that this combined matrix has rank less than n?

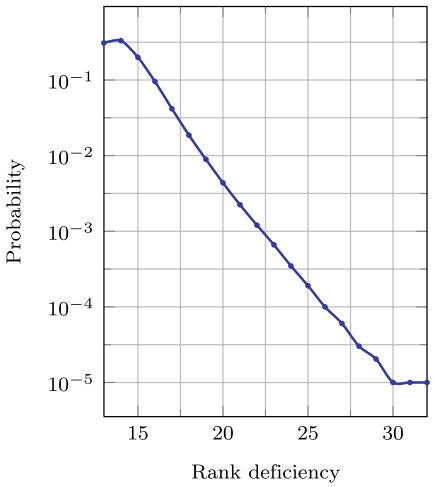
This problem is studied in Appendix A, and solved in Theorem A.2. Suppose that $(p_n, p_{n-1}, \ldots, p_0)$ is the rank profile of the matrix B, i.e., p_i is the probability that the rank of B is i. As the rank of a matrix does not change when operations are allowed over a larger field than the field

from which the elements of the matrix are chosen, p_i is the probability that the rank of B is i, even if we view B as a matrix over $\mathbb{F}_q$. Denote by p'_i the probability that the rank of the combined matrix is i. Then, we have by Lemma A.1 in Section A:

$$\begin{pmatrix} p'_n \\ p'_{n-1} \\ \vdots \\ p'_1 \\ p'_0 \end{pmatrix} = M_n(q)^H \cdot \begin{pmatrix} p_n \\ p_{n-1} \\ \vdots \\ p_1 \\ p_0 \end{pmatrix}. \tag{3.2}$$

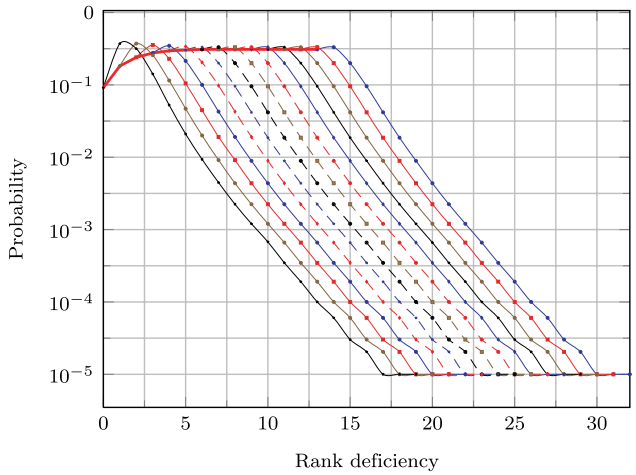
In particular, the probability that the combined matrix above has rank less than n is $1 - p'_n$. How does the rank profile of the combined matrix develop as random rows are added to Q starting with zero rows in Q? Let us look at an example.

Example 3.1. Let $K = 1{,}000$, $S = 59$, $H = 13$, $n = K + S + H = 1{,}072$, and $m = n$. Suppose that B is a matrix of the following form: the first $K = 1{,}000$ rows are created according to the degree distribution Ω for the R10 code (see Figure 3.2 and [15]). The next S rows constitute the upper matrix, i.e., the upper S rows, of the R10 precode shown in Figure 3.2. The resulting matrix B has $1{,}072 - 13 = 1{,}059$ rows and $n = 1{,}072$ columns. After extensive simulations, we obtain the following graph for the rank profile of this B matrix:



In this plot, the horizontal axis is the index ℓ, and the y-axis is $p_{n-\ell}$. Values ℓ for which there is no corresponding point correspond to $p_{n-\ell} = 0$. Obviously, $p_{n-\ell} > 0$ implies $\ell \geq 13$, since B has 13 more columns than rows. Note that this is a qualitative graph only. In particular, it is possible that a larger number of simulations reveal a non-zero probability for even larger values of ℓ.

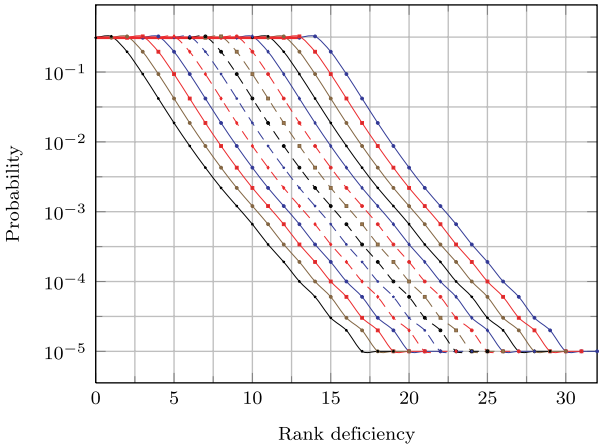
If the matrix Q is binary, then addition of every row of Q to the combined matrix changes the rank profile in the following way:



As it moves from right to left, the solid red curve tracks the evolution of $p_{n-13+\ell}$ for $\ell = 0, 1, \ldots, 13$, where ℓ is the number of random binary rows in $\mathcal{B}$. In particular, the leftmost point of the solid red curve is at p_n when the number of rows in B is 13, i.e., it gives the probability of successful decoding (or alternatively, the probability that the matrix is full rank) at zero reception overhead. Note that the value of p_n at this leftmost point is around 0.1, which means that the failure probability of the R10 code for $K = 1{,}000$ at zero overhead is almost 0.9 if the redundant symbols are generated completely randomly from $\mathbb{F}_2$.

What is interesting about this result is that $p_{n-13+\ell}$ decreases significantly as ℓ increases from 0 to 13 (which at the index 13 will be the probability of successful decoding at zero reception overhead).

Now suppose that $q = 256$. In this case, the previous plot looks very different:



In contrast to the case in which the entries of B are chosen from $\mathbb{F}_2$, when the entries of B are chosen from $\mathbb{F}_{256}$, the initial curve is almost replicated from one round to the next as rows are added to B, and in particular, the solid red curve that tracks the evolution of $p_{n-13+\ell}$ for $\ell = 0, 1, \ldots, 13$, where ℓ is the number of random binary rows in B, is almost perfectly level, i.e., it basically stays at the same value as ℓ increases from 0 to 13.

Thus, for $q = 256$, the rank profile of the combined matrix almost has the property that $p'_n \approx p_{n-h}$.

What we saw in the previous example is not an isolated event. If q is large, then the matrix in (3.2) is the upper shift matrix, and left multiplication by this matrix has the effect of shifting the elements of the vector to the left by one position almost perfect, i.e., with almost no deterioration from the previous vector.

Motivated by this result, the RQ code uses a precode that is similar to that of the R10 code in which the HDPC symbols are replaced by pseudo-random symbols over $\mathbb{F}_{256}$. The field $\mathbb{F}_{256}$ was chosen because it offers excellent trade-offs with respect to the failure probability and computational efficiency: its members can be represented by one byte, its multiplication table is not too large, and, as can be seen in the previous example, it is large enough to lead to a shift of the rank

profile. An exact description of the design of the RQ code is provided in Section 3.3.3.

The idea of using symbols over a larger alphabet is a powerful one. However, it has its limitations. In the following, we will describe some of these limitations and exemplify them using a particular instantiation of a Raptor code. This code is identical to R10, except that the part of the constraint matrix of the precode is replaced by a uniform random matrix over $\mathbb{F}_{256}$. For the purposes of this section, we call this code the R10-256 code.

Failure probability at low overheads. For cases when the overhead is small, but still a significant fraction of the number of source symbols, the failure probability of the R10-256 code can be an issue for some applications. The plots in Figure 3.5 illustrate this. Each of these is a

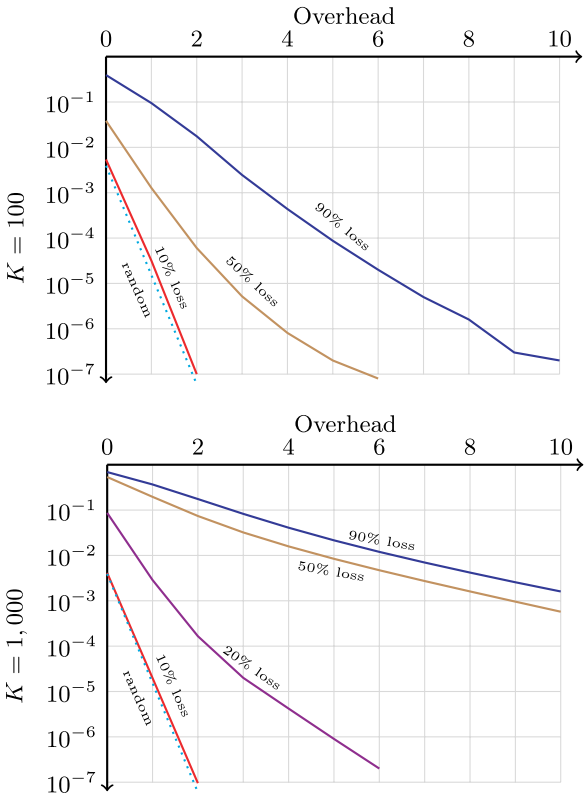
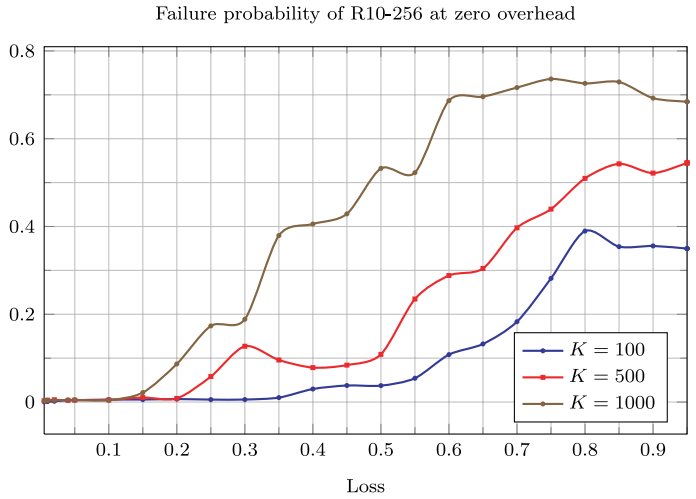


Fig. 3.5 Plot of failure probability *versus* loss for R10-256 and two different values of K.

plot of the failure probability *versus* the number of overhead symbols, at various loss rates. The left plot depicts the case $K = 100$ whereas the right one does so for $K = 1,000$. As can be seen, only at low loss probabilities do we see a failure probability behavior that is close to that of a random fountain over $\mathbb{F}_{256}$ (dotted line). When the loss rate grows, the failure probability behavior changes dramatically. This behavior becomes worse as the number of source symbols grows.

Failure probability is a function of the loss rate. In general, in any systematic fountain code the failure probability of the decoder will be a function of the loss rate. After all, if the loss rate is zero, the decoder never produces a failure. What one would like is a graceful degradation of the failure probability as the loss rate grows, terminating at the failure probability that is equal to that of the non-systematic version of the code. The following plot depicts this problem. It shows the failure probability of the code as a function of the loss rate when the overhead is zero. One would like these zero overhead-failure probabilities to remain basically level as the loss rate increases from 0 to 1, and this is not the case in the shown plot. Similar plots can be produced when the overhead is any small number, not necessarily zero.



Systematic indices. For construction of the systematic version of the R10-256 code, we need to find for each value of K a set of K encoded symbols that can be used to decode the K source symbols. The above

plot shows that a random set of K encoded symbols is unlikely to be able to decode the K source symbols, particularly when K is large (this corresponds to decoding at zero overhead and loss rate 1). Therefore, for large K, a systematic code construction will be far from the average construction. This will inevitably lead to discrepancies in the behavior of the code when different loss rates are applied, as can be seen in the above plots.

In summary, we would like to construct a code that is very efficient in terms of its encoding and decoding speeds, is predictable in terms of its failure probability as a function of overhead, and for which the dependency of the failure probability on the loss rate is minimal, and which behaves like a random fountain over $\mathbb{F}_{256}$ for a large range of overheads.

3.3.2 Permanent Inactivation

The second main ingredient of the RQ code is the use of, what we call, *permanent inactivation*. This technique is able to overcome many of the shortcomings that the R10-256 exhibits.

Recall the definition of inactivation given in Section 2.4, which we hereafter call *dynamic inactivation* to distinguish from the concept of permanent inactivation introduced in this section. Recall that the basic idea behind dynamic inactivation is to designate an intermediate symbol in the decoding matrix as dynamically inactive whenever belief-propagation decoding is stalled because there is no encoded symbol or constraint symbol with remaining degree 1. To restart the belief-propagation decoding process again, one or more intermediate symbols are designated to be considered as "decoded" for the remainder of belief-propagation decoding. The intermediate symbol(s) that is designated to be "decoded" is chosen in such a way that, without them, there is an encoded symbol of remaining degree 1 that allows belief propagation to continue. The designated intermediate symbol(s) is declared to be *dynamically inactive*. The Gaussian elimination at the end of the process is used to recover the values of the dynamically inactive symbols, and these in turn determine the values of the other intermediate symbols.

For permanent inactivation, we designate a part of the intermediate symbols as inactive already before decoding starts. We call these intermediate symbols *permanently inactive* (PI) symbols. To distinguish them from the PI symbols, we call the remaining intermediate symbols *LT symbols*. The terminology choice will become obvious below.

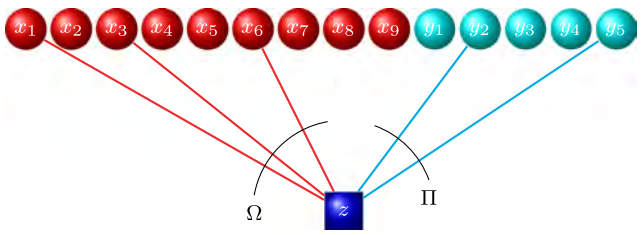
In addition, we also change the way encoded symbols are generated from the intermediate symbols, differentiating between the set of LT symbols and the set of PI symbols. In the most general form, we would do the following to generate an encoded symbol. Let W denote the number of LT symbols, $x_1, \ldots, x_W$ denote the LT symbols, P denote the number of PI symbols, and $y_1, \ldots, y_P$ denote the PI symbols. As usual, $\Omega(x)$ will denote the encoded degree distribution of the corresponding LT code; it is a probability distribution on $\{1, \ldots, W\}$. In addition, we also use a probability distribution $\Pi(x)$ on $\{1, \ldots, P\}$.

(1) Sample from Ω to obtain a degree d_1 in $\{1, \ldots, W\}$.
(2) Choose a subset S of size d_1 from $\{1, \ldots, W\}$ uniformly at random.
(3) Use d_1 and Π to sample a number d_2 in $\{1, \ldots, P\}$.
(4) Choose a subset T of size d_2 from $\{1, \ldots, P\}$ uniformly at random.
(5) Calculate the value of the encoded symbol as

$$z = \sum_{i \in S} x_i + \sum_{j \in T} y_j,$$

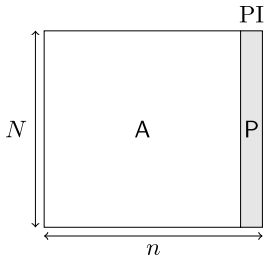
i.e., encoded symbol z is of degree $d = d_1 + d_2$.

The following is a pictorial description of the encoding:



Why is this method beneficial? To answer this question, we first discuss the case where $\Pi(x) = (1 + x)^P/2^P$, which defines the uniform

distribution on $\mathbb{F}_2^P$. Let us look at a typical decoding matrix. It has the following form:



The last P columns of this matrix, i.e., the submatrix P, correspond to the PI symbols, and each column of P is chosen uniformly at random from $\mathbb{F}_2^P$. Suppose that $p_n, p_{n-1}, \ldots, p_0$ is the rank profile of the matrix A. As W is the number of columns of A (so $n = W + P$), $p_n = \cdots = p_{W+1} = 0$.

We are interested in analyzing p_W when $N = n - H = P + W - H$ for some non-negative integer H that is small relative to P and n. Recall that A is the decoding matrix of an LT code on the LT symbols, and thus the overhead on the LT symbols defined by the A matrix would be $n - H - W = P - H$ symbols. We want to have P large enough so that A has a good chance of being full rank W, i.e., p_W is close to 1. For moderate P and small H, this is quite a large overhead for an LT code, and thus with a good LT code design it will be the case that A has rank W with overwhelming probability, i.e., p_W is close to 1. For example, setting $P = c \cdot \sqrt{n}$ for a constant $c > 0$ assures that $p_W \approx 1$ if the degree distribution is chosen properly, and if n is not too large.

What is the rank profile of the combined matrix? Using Equation (A.1), this is equal to

$$M_N(2)^P \cdot (0, \ldots, 0, p_W, p_{W-1}, \ldots, p_0)^\top.$$

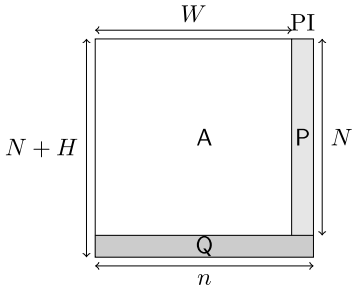
It follows that p_N, the probability that the rank of the combined matrix is N is equal to

$$p_N = p_W \cdot \prod_{i=0}^{P-H-1} \left(1 - 2^{i-P}\right).$$

Hence, p_N is approximately equal to $p_W \cdot (1 - 2^{-H})$, and thus p_N is almost equal to p_W for values of H that are relatively small, e.g., if $H = \log_2(n)$ then p_N is approximately $p_W \cdot (1 - 1/n)$. Note that $H = 0$ corresponds to $N = n$, i.e., decoding with zero overhead, in which case the failure probability is approximately the same as that of a random binary fountain code as long as p_W is close to 1.

The usage of permanent inactivations as just described provides some benefits. For example, the overhead-failure probability curve of the resulting code constructed using permanent inactivation is similar to that of a random binary fountain code, whereas the constructed decoder matrix potentially only has a small number of dense columns, the PI columns, compared with a random binary fountain code where all of the decoder matrix columns are dense.

The usage of permanent inactivations becomes even more compelling when we combine it with HDPC rows defined over $\mathbb{F}_q$ for $q > 2$, for example, $\mathbb{F}_{256}$. To appreciate this approach, we give an example that will serve as a roadmap for the design of the RQ code in the following section. Consider the following diagram:



Here, Q is a matrix with H rows in which every entry is independently and uniformly sampled from $\mathbb{F}_q$. Suppose that $N = n - H$; hence, the matrix corresponds to decoding at zero overhead. In this case, the combined matrix $(\mathsf{A}|\mathsf{P})$ has $n - H$ rows, and thus from the discussion above, the matrix $(\mathsf{A}|\mathsf{P})$ has rank $n - H$ with probability t, where $t \approx p_W \cdot (1 - 2^{-H})$. As an example, if H is set to $\log_2(q)$ (and $p_W \approx 1$ as described above), then $t \approx 1 - 1/q$.

Thus, the combined matrix $((\mathsf{A}|\mathsf{P})/\mathsf{Q})$ has rank n with probability

$$t \cdot \prod_{i=1}^{H} \left(1 - \frac{1}{q^i}\right).$$

If t is close to 1 then this quantity is very close to the probability that a random square matrix with entries in $\mathbb{F}_q$ has full rank: $\prod_{i=1}^{H}(1-1/q^i) \approx \exp(1/(q-1)) \approx (q-2)/(q-1)$. By Lemma 3.1, the latter number is essentially equal to the probability that a random square matrix with entries in $\mathbb{F}_q$ has full rank. For example, setting $q = 256$, then the failure probability at zero overhead is proportional to $1/255$.

This reasoning is also valid when $N > n - H$ (in fact, it is even more valid, as the matrix A will have an even higher probability of being of rank W in this case). We have thus created a matrix that is largely sparse and consists almost entirely of symbols over $\mathbb{F}_2$ (with only a small number of symbols that are over a larger field $\mathbb{F}_q$), and yet exhibits almost the failure probability of a random matrix over $\mathbb{F}_q$.

In practice, we cannot choose the matrix P to be completely random (binary), because this would lead to inefficiencies in encoding and decoding. However, for the arguments above to be valid, we only need that each column of P has (in absolute numbers) many non-zero entries. This can be achieved by assigning as little as 2–3 non-zero positions to every row of P. The fraction of ones in every columns of P would then be $2/P$ or $3/P$, which is often sufficient to simulate the random behavior of the columns. For example, if $P = c \cdot \sqrt{n}$ and there are n rows, then on average there are $O(\sqrt{n})$ ones in each column.

The decoding of this code is very similar to the decoding described for the general inactivation decoding. We just consider the PI symbols as additional inactive symbols.

3.3.3 Construction

In this section, we give the details of the construction of the RQ code. In doing so, we make references to the specification of this code [16]. The RQ code is a systematic fountain code designed to support a number of source symbols K between 1 and 56,403.

The RQ code limits the number of source block sizes that are supported within this range as follows. Given a source block with K source symbols to be encoded or decoded, a K' value is selected based on the table shown in [16, Section 5.6]. The first column in the table lists the possible values for K'. The value of K' selected is the smallest

value among the possibilities such that $K \leq K'$. The K source symbols $C'_0, \ldots, C'_{K-1}$ are padded with $K' - K$ symbols $C'_K, \ldots, C'_{K'-1}$ with values set to zeros to produce a source block consisting of K' source symbols $C'_0, \ldots, C'_{K'-1}$, and then encoding and decoding are performed on this padded source block.

The above approach has the benefit of reducing the number of systematic indices that need to be supported. There is no disadvantage in terms of the overhead-failure curve for K, as it is the same as the overhead-failure curve for the selected K': Given the value of K, the decoder can compute the value of K' and set the values of $C'_K, \ldots, C'_{K'-1}$ to zeros, and thus it only has to decode the remaining K of the K' source symbols of the source block. The only potential disadvantages are that slightly more memory or computational resources might be needed for encoding and decoding with slightly more source symbols. However, the spacing between consecutive values of K' is roughly 1% for larger values of K', and thus the potential disadvantage is negligible.

Because of the padding of the source block from K to K', the identifier for encoded symbols $C'_0, C'_1, \ldots$ within the RQ code is called the *Internal Symbol Identifier* (ISI), where $C'_0, \ldots, C'_{K'-1}$ are the source symbols and $C'_{K'}, C'_{K'+1}, \ldots$ are the repair symbols. External applications employing the encoder and decoder use an ESI that ranges from 0 to $K - 1$ to identify the original source symbols $C'_0, \ldots, C'_{K-1}$ and that continues $K, K + 1, \ldots$ to identify repair symbols $C'_{K'}, C'_{K'+1}, \ldots$. Thus, a repair symbol C'_X identified with ISI X within the RQ code is identified externally with an ESI $X - (K' - K)$. This is described in more detail in [16, Section 5.3.1].

The encoding and decoding for the RQ codes are defined by two types of relationships: *constraint relationships* among the intermediate symbols and *LT–PI relationships* between the intermediate symbols and the encoded symbols. Encoding proceeds by determining the intermediate symbol values based on (1) the source symbol values, (2) LT–PI relationships between the source symbols and the intermediate symbols, and (3) the constraint relationships among the intermediate symbols. The values of repair symbols can be generated from the intermediate symbols based on LT–PI relationships between the intermediate symbols and the repair symbols.

Similarly, decoding proceeds by determining the intermediate symbol values based on (1) the received encoded symbol values, (2) LT–PI relationships between the received encoded symbols and the intermediate symbols, and (3) the constraint relationships among the intermediate symbols. The values of missing source symbols can be generated from the intermediate symbols based on LT–PI relationships between the intermediate symbols and the missing source symbols. Thus, encoding and decoding are essentially symmetric procedures.

We now describe how some of the key parameters are derived and used. In the specification [16], the values of J, S, H, and W are determined based on the table shown in [16, Section 5.6]: J, S, H, and W are determined by the entries in the row of this table associated with K'.

The value J is the systematic index to use to define the code construction with respect to K'. In the specification, J is the second entry in the row associated with K'. For each value of K', the value of J has been precomputed so that the values of the intermediate symbols are uniquely determined by the values of the source symbols $C'_0, \ldots, C'_{K'-1}$, the constraint relationships among the intermediate symbols, and the LT–PI relationships between the intermediate symbols and the source symbols.

The number S is the number of LDPC symbols, and it is a prime number that is approximately $0.01 \cdot K' + \sqrt{2 \cdot K'}$. In the specification, S is the third entry in the row associated with K'.

The number H is the number of HDPC symbols, and it is also the width of the $\mathbb{F}_{256}$-part of the constraint matrix of the precode. The value of H is between 10 and 16, and H grows very slowly as a function of K'. In the specification, H is the fourth entry in the row associated with K'.

The number of intermediate symbols is $L = K' + S + H$.

The intermediate symbols are partitioned into LT symbols and PI symbols. The number W of LT symbols of the intermediate block is chosen so that the number P of PI symbols is proportional to $\sqrt{K'}$, and so that W is a prime number so as to simplify the tuple generation from the LT symbols of the intermediate block. In the specification, W is the fifth entry in the row associated with K' and then the number of PI symbols is $P = L - W$.

All of the LDPC symbols are classified as LT symbols, and all of the HDPC symbols are classified as PI symbols. The number of LT symbols that are not LDPC symbols is $B = W - S$, and the number of PI symbols that are not HDPC symbols is $U = P - H$.

Overall, the structure of the intermediate symbols $C_0, \ldots, C_{L-1}$ is that the first B symbols are the LT symbols that are not LDPC symbols, the next S symbols are the LDPC symbols, the next U symbols are the PI symbols that are not HDPC symbols, and the last H symbols are the HDPC symbols. Thus, the first $W = B + S$ symbols are LT symbols, and the remaining $P = U + H$ are PI symbols.

The calculation of the other parameters is described at the beginning of [16, Section 5.3.3.3].

The constraint matrix described below defines the constraint relationships among the L intermediate symbols $C_0, \ldots, C_{L-1}$. The constraint matrix overall structure is given in Figure 3.6. It consists of two submatrices: the top one, consisting of S rows, contains binary entries, and the second one, consisting of H rows, contains a mixture of entries from $\mathbb{F}_{256}$ (matrix Q) and binary entries (identity matrix I_H).

We now describe the two submatrices of the constraint matrix in Figure 3.6 in more detail. The upper submatrix consisting of the first S rows has two parts: the submatrix consisting of the last P columns of this matrix (the part that is in pink) corresponds to the PI part of this matrix. It consists of two consecutive (modulo P) ones in each row. In [16], the structure of this matrix is described implicitly in

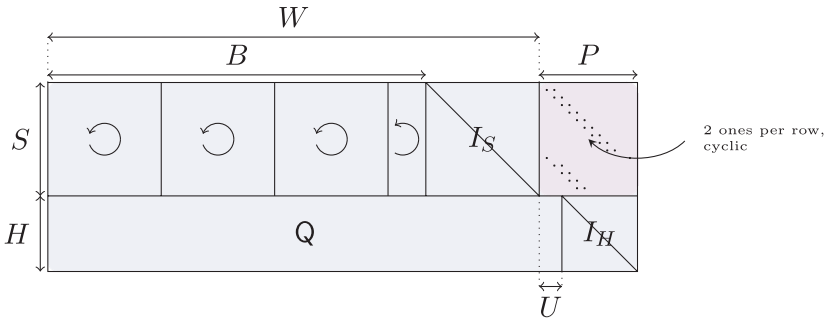


Fig. 3.6 The constraint matrix of the precode of the RQ code.

Section 5.3.3.3 as the second "For" loop describing the first part of the preceding relations. The first W columns of the upper part of the constraint matrix consist of circulant matrices and an $S \times S$ identity matrix. The circulant matrices are composed of B of the columns, and each (except possibly the last) has S columns. The number of these circulant matrices is $\lceil B/S \rceil$. The columns in these circulant matrices all have degree 3. The first column of the ith circulant matrix has ones at positions 0, $(i+1)$mod-S, and $(2 \cdot i + 1)$mod-S and zeros elsewhere. The other columns are cyclic shifts of the first, wherein by a shift we mean a cyclic down-shift.

The lower H rows mimic the behavior of a matrix in which entries are chosen uniformly and independently from $\mathbb{F}_{256}$. For efficiency reasons to be described later, the matrix Q (which is the most important part) is constructed in such a way that there is an efficient algorithm for the multiplication $\mathsf{Q} \cdot z$ for a generic input vector z. More explicitly, the matrix Q is given as

$$\begin{aligned} \mathsf{Q} &= (\Delta_1 \mid \Delta_2 \mid \cdots \mid \Delta_{K'+S-1} \mid Y) \cdot \Gamma, \\ \Gamma &= \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & 0 \\ \alpha & 1 & 0 & \cdots & 0 & 0 \\ 0 & \alpha & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & 0 & \cdots & \alpha & 1 \end{pmatrix}^{-1}, \\ Y &= \begin{pmatrix} \alpha^0 \\ \alpha^1 \\ \vdots \\ \alpha^{H-2} \\ \alpha^{H-1} \end{pmatrix}, \end{aligned} \tag{3.3}$$

where α is an element of $\mathbb{F}_{256}$ with minimal polynomial $x^8 + x^4 + x^3 + x^2 + 1$ over $\mathbb{F}_2$. $\Delta_1, \ldots, \Delta_{K'+S-1}$ are pseudo-random columns of degree 2. This matrix is described toward the end of [16, Section 5.3.3.3]. The matrix Γ is called GAMMA there.

Because of the structure of the matrix Q, multiplication of this matrix with a given vector z of length W is quite efficient. To describe

it, suppose that for $i = 1, \ldots, K' + S - 1$ the 2 non-zero positions of Δ_i are denoted by $p(i)$ and $q(i)$, and suppose that

$$(u_1, \ldots, u_H)^\top = \mathsf{Q} \cdot (z_1, \ldots, z_{K'+S})^\top.$$

In a first step, we compute $(v_1, \ldots, v_{K'+S})^\top = \mathsf{\Gamma} \cdot (z_1, \ldots, z_{K'+S})^\top$. This can be done via the following simple algorithm:

(1) $v_1 = z_1$.
(2) For $i = 2, \ldots, K' + S$, set $v_i = \alpha^{-1}(v_{i-1} + z_i)$.

Next, we initialize the values $u_1 = \alpha^0 v_{K'+S}, \ldots, u_H = \alpha^{H-1} v_{K'+S}$. Thereafter, we proceed as follows:

(1) For $i = 1, \ldots, K' + S - 1$ do
  (1.1) $u_{p(i)} = u_{p(i)} + v_i$,
  (1.2) $u_{q(i)} = u_{q(i)} + v_i$.

We give an example of the constraint matrix of the precode for $K' = 10$. In this case, $H = 10$, $S = 7$, and thus there are $L = K' + S + H = 27$ intermediate symbols. The number of LT symbols is $W = 17$ of which $B = W - S = 10$ are not LDPC symbols. The number of PI symbols is $P = 10$ of which $U = P - H = 0$ are not HDPC symbols (and thus in this example all of the PI symbols are HDPC symbols). The specific constraint matrix for the RQ code for $K' = 10$ is shown in Figure 3.7, which is a specific instance of the general form of the RQ constraint matrix shown in Figure 3.6.

1	0	0	0	0	1	1	1	0	0	1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0
1	1	0	0	0	0	1	0	1	0	0	1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
1	1	1	0	0	0	0	1	0	1	0	0	1	0	0	0	0	0	0	1	1	0	0	0	0	0	0
0	1	1	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	1	1	0	0	0	0	0
0	0	1	1	1	0	0	1	0	1	0	0	0	0	1	0	0	0	0	0	0	1	1	0	0	0	0
0	0	0	1	1	1	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1	0	0	0
0	0	0	0	1	1	1	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1	0	0
AF	3F	7F	5F	4F	4F	4F	A7	D3	09	84	42	21	90	40	20	10	1	0	0	0	0	0	0	0	0	0
79	5C	CE	67	5B	4D	A6	53	A1	D0	88	44	22	11	80	40	20	0	1	0	0	0	0	0	0	0	0
81	C0	60	30	F8	9C	AE	57	4B	A5	D2	89	C4	62	31	90	40	0	0	1	0	0	0	0	0	0	0
21	90	A8	54	CA	65	5A	CD	0E	07	83	C1	08	04	02	01	80	0	0	0	1	0	0	0	0	0	0
B1	38	FC	76	DB	E5	F2	99	2C	FE	77	B3	D1	08	04	02	01	0	0	0	0	1	0	0	0	0	0
C2	61	58	CC	8E	47	A3	39	7C	DE	8F	C7	E3	F1	18	04	02	0	0	0	0	0	1	0	0	0	0
EE	77	5B	A5	D2	89	C4	62	31	78	DC	8E	47	A3	D1	08	04	0	0	0	0	0	0	1	0	0	0
03	81	C0	60	30	F8	9C	46	23	91	28	FC	9E	47	A3	D1	08	0	0	0	0	0	0	0	1	0	0
8A	45	A2	51	48	24	12	E9	F4	72	31	78	DC	8E	47	A3	D1	0	0	0	0	0	0	0	0	1	0
BE	57	A3	D1	08	04	02	01	80	40	20	10	E8	9C	AE	57	A3	0	0	0	0	0	0	0	0	0	1

Fig. 3.7 Constraint matrix of the precode of the RQ code for $K' = 10$.

The matrix Q that simulates a random matrix over $\mathbb{F}_{256}$ is given by Equation (3.3). The elements of $\mathbb{F}_{256}$ are represented by octets, as described in [16, Section 5.7], and these octets are represented by two hexadecimal integers in Figure 3.7. More precisely, the finite field $\mathbb{F}_{256}$ is represented using the irreducible polynomial $f(x) = x^8 + x^4 + x^3 + x^2 + 1$ over $\mathbb{F}_2$. The polynomial $a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7$ is mapped to the integer $a_0 + 2a_1 + \cdots + 2^7a_7$, and this integer is represented by two hexadecimal digits, wherein the first digit is the less significant one. Hence, for example, the hexadecimal number AF corresponds to the integer $10 + 16 \cdot 15 = 250$, which has the binary expansion $2 + 2^3 + 2^4 + 2^5 + 2^6 + 2^7$, and hence corresponds to the polynomial $x + x^3 + x^4 + x^5 + x^6 + x^7$.

We now describe the LT–PI relationship between the encoded symbols $C'_0, \ldots, C'_{K'-1}, C'_{K'}, \ldots$ and the intermediate symbols $C_0, \ldots, C_{L-1}$. The symbols $C'_0, \ldots, C'_{K'-1}$ are composed of the source symbols whereas $C'_{K'}, C'_{K'+1}, \ldots$ are composed of the repair symbols. The LT–PI relationship is defined by the `TupleGenerator` described in [16, Section 5.3.5.4]. The relationship between an encoded symbol and ISI X depends on the systematic index J. The systematic index J can be used to generate a non-negative integer y from X and K' as described in [16, Section 5.3.5.4]. The value of y is then used to derive a tuple $(d, a, b, d1, a1, b1)$. The number d is the LT degree of the encoded symbol, and a and b are used to select the subset of size d from the LT symbols. More specifically, d is an integer between 1 and 30, and a and b are integers between 0 and $W - 1$ (and a is non-zero). Similarly, the number $d1$ is the PI degree of the encoded symbol, and $a1$ and $b1$ are used to select the subset of size $d1$ from the PI symbols. More specifically, $d1$ is an integer that is either 2 or 3, and $a1$ and $b1$ are integers between 0 and $P1 - 1$, where $P1$ is the smallest prime greater than or equal to P (and $a1$ is non-zero).

A pseudo-random generator, called `Rand` and defined in [16, Section 5.3.5.1], is used to generate the tuple. The input to `Rand` is y, a non-negative integer $i < 256$, and a positive integer m, and it produces an integer between 0 and $m - 1$. The value of i varies in different calls to `Rand` to ensure that the outputs from calls to `Rand` with the same value of y as input are largely independent of one another. To generate

i	Ω_i	i	Ω_i	i	Ω_i	i	Ω_i	i	Ω_i	i	Ω_i
1	0.0050	6	0.0333	11	0.0091	16	0.0042	21	0.0024	26	0.0015
2	0.5000	7	0.0238	12	0.0076	17	0.0037	22	0.0022	27	0.0014
3	0.1666	8	0.0179	13	0.0064	18	0.0033	23	0.0020	28	0.0013
4	0.0833	9	0.0139	14	0.0055	19	0.0029	24	0.0018	29	0.0012
5	0.0500	10	0.0111	15	0.0048	20	0.0026	25	0.0017	30	0.0295

Fig. 3.8 The degree distribution used for the RQ code.

the LT degree d, `Rand` is called to generate a random integer v between 0 and $2^{20} - 1$, and then as described in [16, Section 5.3.5.2] the function `Deg` is called with input v to choose a degree d using a table lookup based on v and Table 1 shown in [16, Section 5.3.5.2], where the degree distribution that this generates is as shown in Figure 3.8.

The output of the tuple generator is used to define the relationship between the encoded symbol and the intermediate symbols: If $C_0, C_1, \ldots, C_{W-1}, C_W, \ldots, C_{W+P-1}$ are the intermediate symbols, then the value of the encoded symbol C'_X is the XOR of A and B, where

- A is the XOR of $C_a, C_{(b+a) \bmod W}, \ldots, C_{(b+(d-1)\cdot a) \bmod W}$.
- We generate the numbers $r_0 = a1, r_1 = (a1 + b1) \bmod P1, r_2 = (2 \cdot a1 + b1) \bmod P1, \ldots$ until $d1$ of these numbers fall in the interval $[0, P - 1]$. If these numbers are denoted $\ell_1, \ldots, \ell_{d1}$, then B is the XOR of $C_{W+\ell_1}, C_{W+\ell_2}, \ldots, C_{W+\ell_{d1}}$.

An efficient method of computing the value of encoded symbol C'_X from the tuple $(d, a, b, d1, a1, b1)$ is described in [16, Section 5.3.5.3].

Figure 3.9 shows a schematic of a decoding matrix for the RQ code. The PI part corresponding to the LT encoded symbols consists of 2 or 3 ones per row. This means that every encoded symbol contains an XOR of 2 or 3 PI symbols. In the specific design, the exact number depends on the degree of the LT part of the encoded symbol: if it is 2 or 3, then the degree of the PI part is chosen to be 2 or 3 with probability 1/2. If the degree of the LT part is 4 or larger, then the degree of the PI part is always 2. More information and the exact design details can be found in [16].

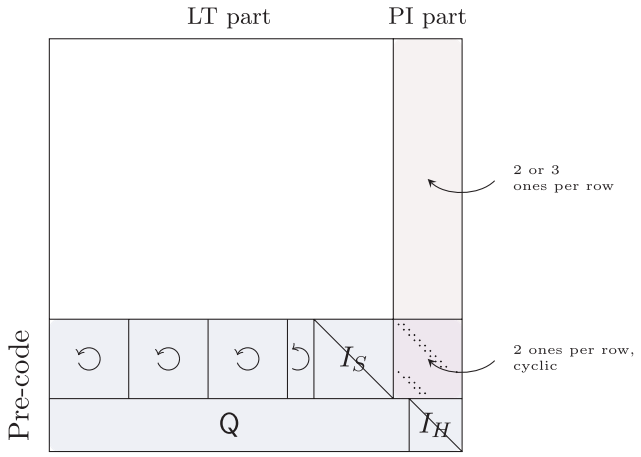


Fig. 3.9 The matrix structure of the RQ code. The top part of the matrix corresponds to the LT part, i.e., each row in the upper part corresponds to an encoded symbol.

3.3.4 Duplicates

It is interesting to note why we chose the degree of the PI part of RQ to be 2 or 3 if the degree of the LT part is 2 or 3. The reason is the avoidance of what we call *duplicates* and which turns out to be one of the main reasons for the failure probability at low overheads.

We call two encoded symbols' duplicates if they have the same neighbors among the PI symbols and LT symbols. If the overhead is zero, then duplicates will account for part of the failure probability. If this part is a meaningful fraction of the overall failure probability, then duplicates should be avoided.

What is the probability that two encoded symbols are duplicates? Let us turn the question around. What is the probability that among all the received encoded symbols there are no duplicates?

Let D denote the PI degree of the encoded symbols. We will analyze the probability that the encoded symbols have duplicates for different values of D in order to show why we chose D the way we did.

Obviously, if two encoded symbols have different LT degrees, they cannot be duplicates. Hence, it makes sense to calculate the probability of having duplicates among encoded symbols of the same LT degree only. Suppose that this degree is ℓ. We are going to calculate

the probability $\Pi_{K,m}(\ell, D)$ that there are at least two duplicate encoded symbols among m received encoding symbols of LT degree ℓ. Note that we choose the LT neighbors and the PI neighbors of an encoded symbol according to the tuple generator. Hence, two encoded symbols with the same values for $(a, b, a1, b1)$ have the same neighbors (though they could also have the same neighbors if the tuples are different). Moreover, if the LT degrees of the encoded symbols are one, then it suffices for them to have the same $(b, a1, b1)$ triples if $D \geq 2$, and it suffices for them to have the same $(b, b1)$ pair if $D = 1$.

Even though in the actual neighbor generation process the degrees and the quadruples $(a, b, a1, b1)$ are correlated, we will disregard this correlation in our analysis and assume that the LT degrees are chosen independently using a perfect source of randomness. If the LT degree is 2, then the quadruples $(a, b, a1, b1)$ and $((a + b) \bmod W, -a \bmod W, a1, b1)$ generate the same subset of LT symbols. We will make another simplifying assumption: for $D = 2$, we will assume that PI neighbors are sampled uniformly among all 2-subsets of the set of PI neighbors. For $D > 2$, we assume that the possible set of PI neighbors is $P(P - 1)$ (i.e., we discount the rare cases when two $(a1, b1)$-pairs give rise to the same set of neighbors). In summary, if $\sigma_K(\ell, D)$ denotes the number of possible pairs of subsets of size ℓ of the LT symbols and of size D of the PI symbols, then we will approximate

$$\sigma_K(\ell, D) \approx \begin{cases} WP, & \text{if } D = \ell = 1, \\ \binom{W}{2} P, & \text{if } D = 2, \ell = 1, \\ W\binom{P}{2}, & \text{if } D = 1, \ell = 2, \\ \binom{W}{2}\binom{P}{2}, & \text{if } D = \ell = 2, \\ W(W-1)P(P-1), & \text{if } D, \ell > 2. \end{cases}$$

Note that K appears indirectly on the right-hand side as well, since W and P depend on K.

The analysis resembles now that of the birthday paradox. We have

$$\Pi_{K,m}(\ell, D) \approx 1 - \prod_{i=0}^{m-1} \left(1 - \frac{i}{\sigma_K(\ell, D)}\right).$$

The probability that there are m encoded symbols of degree ℓ among the K received encoded symbols is $\binom{K}{m} \Omega_\ell^m (1 - \Omega_\ell)^{K-m}$. Hence, the

probability $\Pi_K(\ell, D)$ that there are at least two duplicates among the received encoded symbols of degree ℓ is

$$\Pi_K(\ell, D) = 1 - \sum_{m=0}^{K} \binom{K}{m} \Omega_\ell^m (1 - \Omega_\ell)^{K-m} \Pi_{K,m}(\ell, D). \tag{3.4}$$

The table below lists these numbers for some exemplary values of ℓ and D for $K = 10$:

	$D = 1$	$D = 2$	$D = 3$	$D = 4$
$\ell = 1$	8.7e − 6	1.9e − 6	9.6e − 7	9.6e − 7
$\ell = 2$	9.6e − 3	2.1e − 3	1.1e − 3	1.1e − 3
$\ell = 3$	5.4e − 4	1.2e − 4	6.0e − 5	6.0e − 5
$\ell = 4$	1.3e − 4	3e − 5	1.5e − 5	1.5e − 5

whereas the following table shows the same values for $K = 101$:

	$D = 1$	$D = 2$	$D = 3$	$D = 4$
$\ell = 1$	7.4e − 5	1.1e − 5	5.3e − 6	5.3e − 6
$\ell = 2$	1.3e − 2	1.9e − 3	9.5e − 4	9.5e − 4
$\ell = 3$	7.4e − 4	1.1e − 4	5.3e − 5	5.3e − 5
$\ell = 4$	1.0e − 48	2.6e − 5	1.3e − 5	1.3e − 5

As the failure probability of a random fountain code over $\mathbb{F}_{256}$ at zero overhead is about 0.0039, we want the failure probability caused by duplicates to be smaller than this amount, without paying too much in terms of the number of XOR's we need to perform. This shows that we need to have D at least 2, and even better, $D = 3$ if $\ell = 2$. However, the choice of $D = 3$ for $\ell = 2$ results in about 0.5 more XOR's per source symbol compared with when $D = 2$ (because half of the symbols are of LT weight 2). Therefore, we choose $D = 2$ or 3 with equal probability when $\ell = 2$ or 3.

3.3.5 Some Properties

In this section, we will report on some of the properties of RQ and show that we have largely solved the design goals laid out in Section 3.3.1. Further properties of RQ can be found in Appendix B.

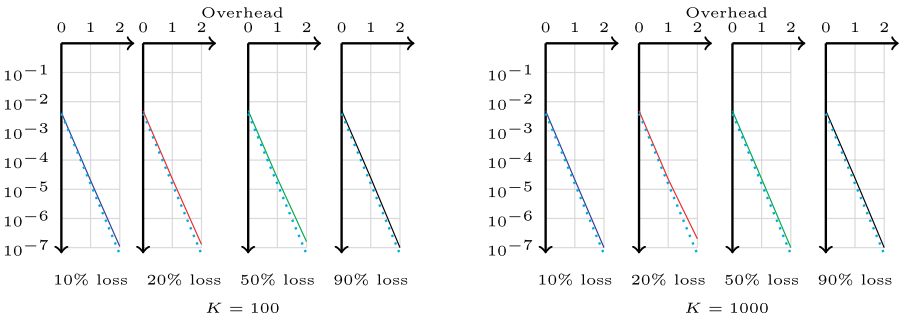
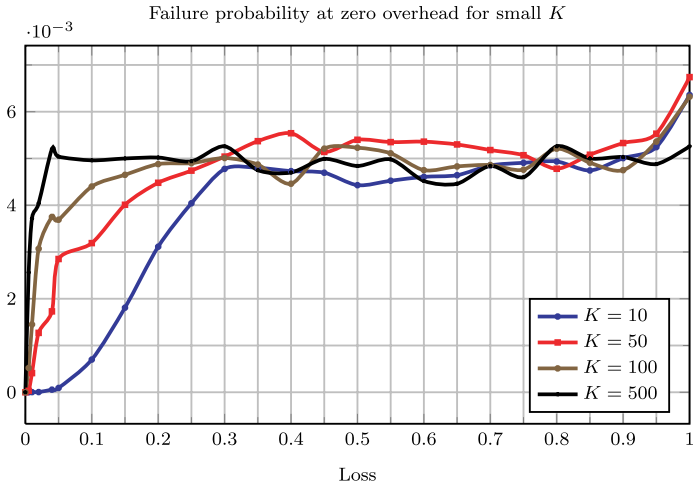


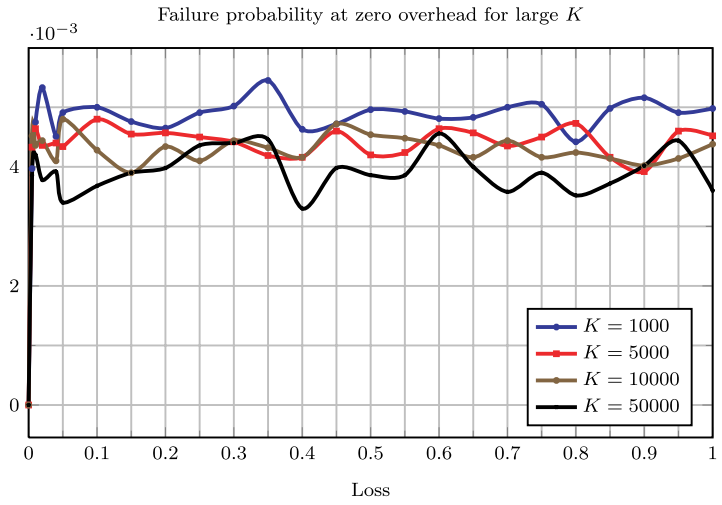
Fig. 3.10 Plot of failure probability *versus* loss for RQ and two different values of K.

We start with studying the failure probability of RQ as a function of the number of overhead symbols, analogous to what we did for the code R10-256 in Figure 3.5. The results are summarized in Figure 3.10. These results are for 10^8 runs for each K and each loss probability. The dotted line is the behavior of the random fountain over $\mathbb{F}_{256}$. As can be seen, the performance of RQ is essentially the same as that of the random fountain over $\mathbb{F}_{256}$.

The failure probability of RQ for 0 overhead as a function of the loss rate for various values of K is given in the next two plots. The following plot depicts the case of small K values.

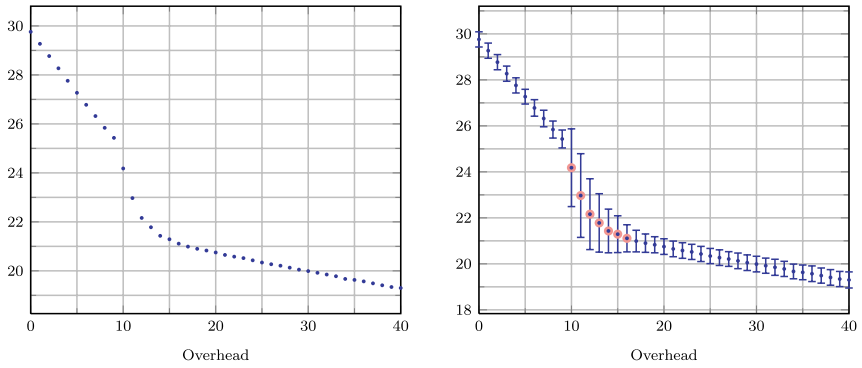


As can be seen, the failure probability is well concentrated around the failure probability of the random fountain over $\mathbb{F}_{256}$, which is about 0.0039. This is in stark contrast to R10-256 for which the failure probability depends heavily on the loss rate. Interestingly, the same observation holds also for larger K values.



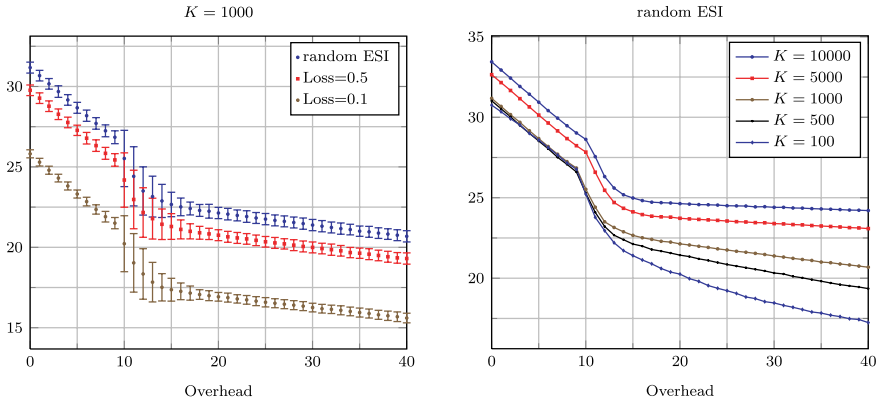
A figure of merit for any Raptor code is the number of XOR's the decoder performs for a given number K of source symbols. We choose to normalize this number by K, and look at the number of XOR's per source symbol. We call this number the *decoding cost* of the code. The decoding cost depends on many parameters: typically, the larger the overhead, the smaller the decoding cost. For a systematic Raptor code, the decoding cost also depends on the loss rate: the smaller the loss rate, the smaller the decoding cost, as fewer source symbols are lost and need to be recovered in the last step of decoding.

RQ has an interesting behavior with respect to the number of XOR's. The following two plots provide a representative description of this behavior. The left plot depicts the average decoding cost in 1000 runs as a function of the number of overhead symbols for $K = 1000$ and loss rate 0.8. The right plot overlays the standard deviation of the decoding cost.

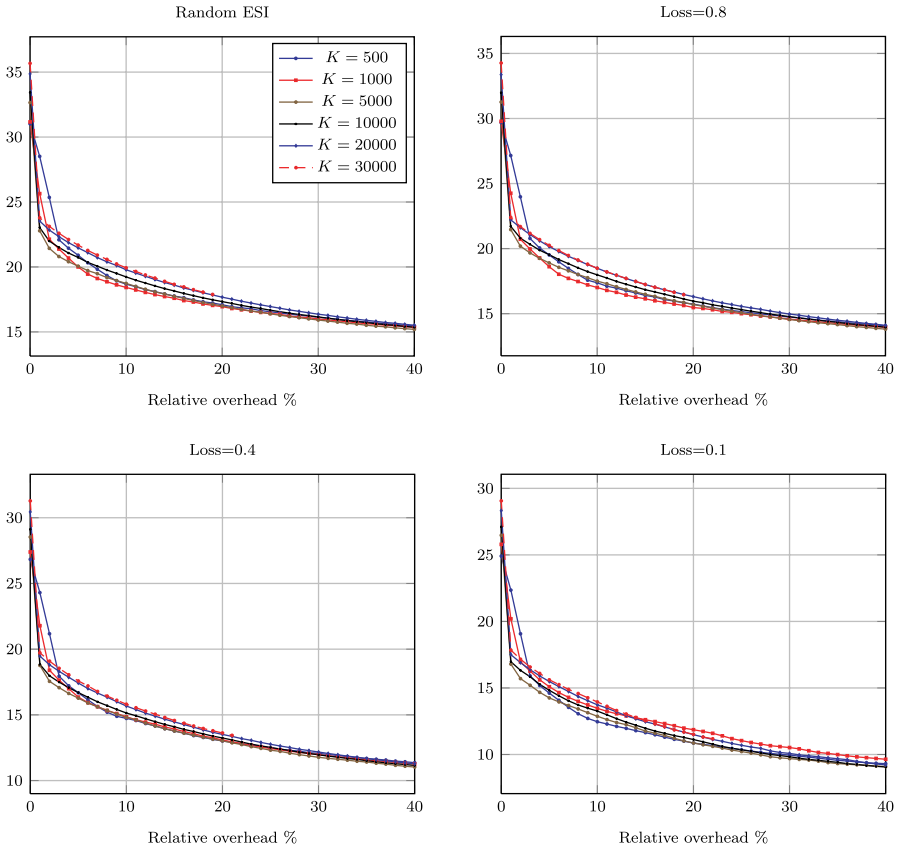


As we can see, the average decoding cost starts high, and then at around 10 overhead symbols it drops sharply, and then continues decreasing but at a slower rate than at the beginning. The drop occurs at a value of 10. This is the number of HDPC symbols of RQ, and there is a reason that the decoding cost decreases at around this value: once 10 or more overhead symbols are received, the HDPC symbols are likely not to be used for decoding. The plot on the right provides a more accurate view: the standard deviation of the decoding cost is much larger between 10 and 16 overhead symbols than for any other number of overhead symbols. The points for which the standard deviation is unusually high are marked with pink circles.

The following plots show that the situation depicted in the last plots are typical and qualitative results depend neither on the number of source symbols, nor on the loss rate:

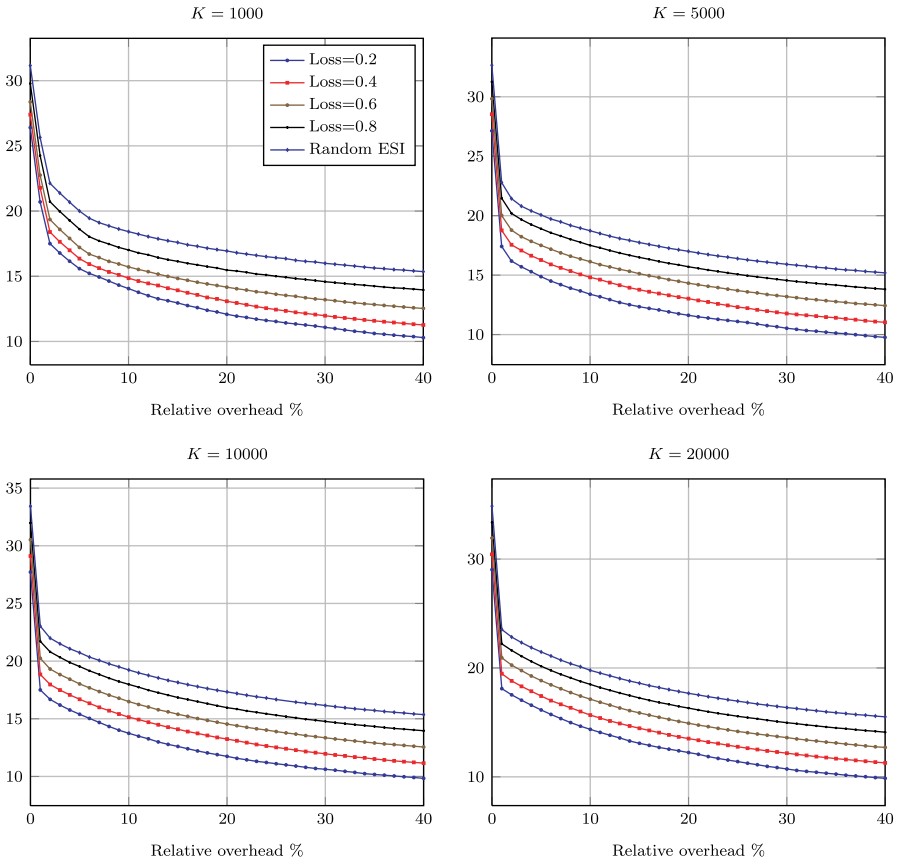


Next, we concentrate on the average cost of the decoder when the overhead is a fixed fraction of K, the number of source symbols. We expect that for larger overheads, this cost is largely independent of K. The following plots suggest that this is indeed the case:



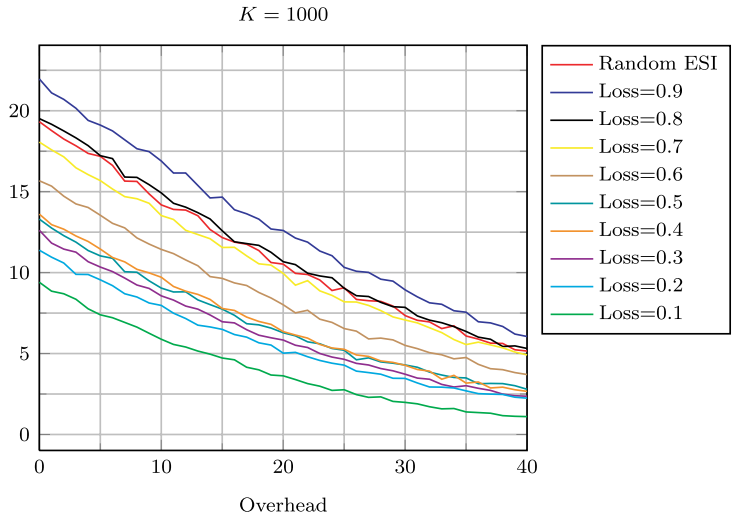
The average number of XOR's is also a function of the loss rate. This is because of the particular way the systematic decoder works: first, a decoding algorithm for the non-systematic version is applied to the received symbols in order to determine the intermediate symbols. Thereafter, for every source symbol that has not been received, the symbols in the intermediate block are used to encode those source symbols. The difference in the number of XOR's stems from the last step: the more source symbols need to be recovered, the more XOR operations need to be performed. On average, the difference in decoding cost

between the case where p-fraction of the source symbols need to be recovered and the case where a q-fraction needs to be recovered should be $(p-q)\alpha$ in expectation, where α is the average encoding cost of a repair symbol. For RQ, α is the average degree of the LT-part of a symbol, plus the average degree of the PI-part. The former is roughly 4.8173, whereas the latter is $2 + 0.333 = 2.333$. Hence, we expect the difference in the decoding cost between the two cases outlined above to be $(p-q) \times 7.1506$. The situation is exemplified in the following plots.

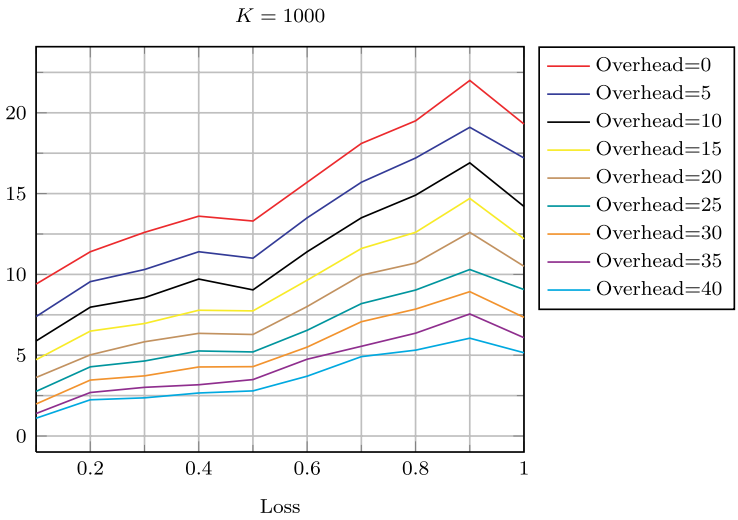


The number of dynamic inactivations is another important and interesting parameter for the performance of the RQ-code. The average number of dynamic inactivations is generally a decreasing function of the overhead. Its dependency on the loss rate is, however, slightly more

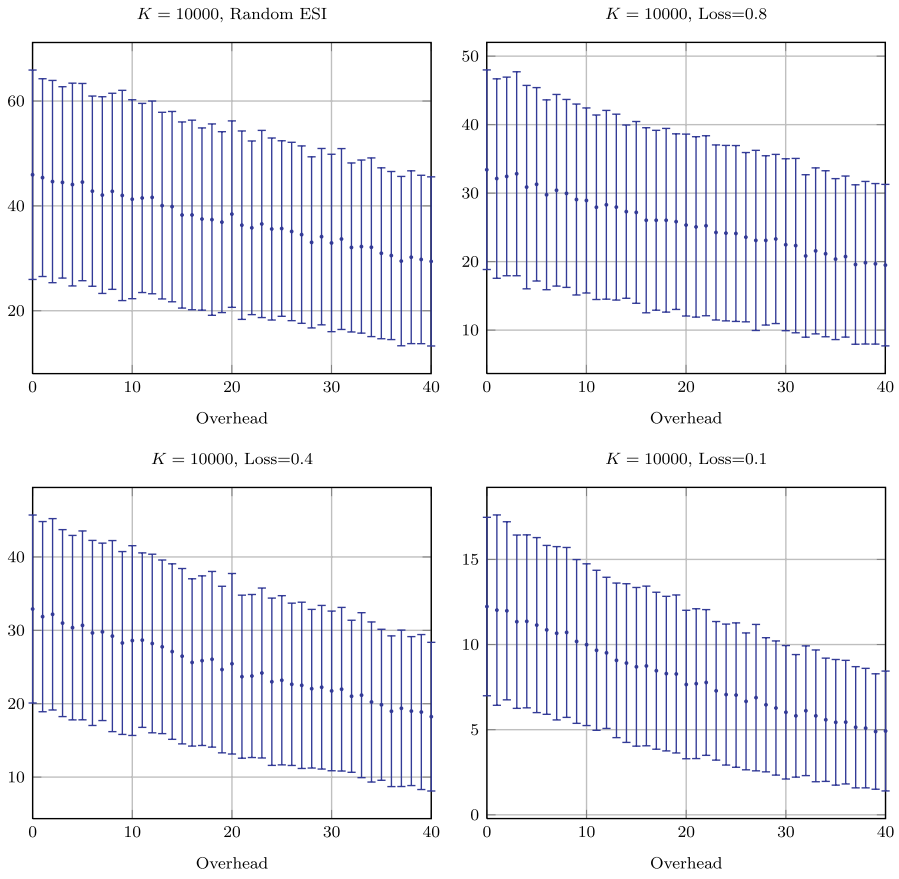
complicated and depends to a large extent on the systematic indices chosen. The graph below shows on the vertical axis the average number of dynamic inactivations, as a function of the overhead, for various loss rates.



The plot below essentially shows the same information, except that now the horizontal axis is the loss rate, and each plot belongs to one specific overhead.

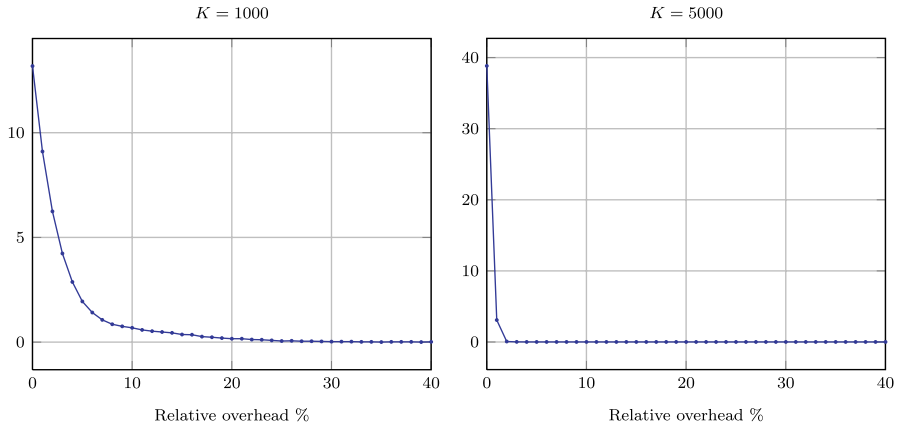


Unlike the number of XORs, the number of dynamic inactivations is subject to a large variance. To a large extent, the variance does not depend on the loss rate; however, it does depend on the overhead. The following plots show the average number of dynamic inactivations and their variance as a function of the number of overhead symbols for various loss rates and for the example of $K = 10{,}000$. Other values for K produce very similar plots.



One of the advantages of working with the degree distribution given in Figure 3.8 is that the average number of dynamic inactivations goes quickly toward zero as the relative overhead grows. This is particularly true for large values of K. This situation is depicted in the following

figures that show the average number of dynamic inactivations as a function of the relative overhead for different values of K.



More information on properties of RQ is given in Appendix B.

A

Rank of Random Matrices

In the following, we assume that m, n are fixed positive integers, $m \geq n$, and that $\mathcal{D}$ is a probability distribution on $\mathbb{F}_q^{m \times n}$.

Definition A.1. The *rank profile* of the distribution $\mathcal{D}$ is the vector $(p_n, p_{n-1}, \ldots, p_0)$, where

$$p_i = \Pr[\mathrm{rk}(A) = i],$$

A being a random matrix sampled from the distribution $\mathcal{D}$. We sometimes refer to the vector $(p_n, \ldots, p_0)$ also as the *rank profile* of the matrix A.

We are interested in the rank profile of uniform random matrices over $\mathbb{F}_q$. These are matrices for which every entry is chosen independently and uniformly over the field $\mathbb{F}_q$.

The rank profile is such matrices can be computed using a simple dynamic programming approach. The procedure is as follows: we start with the unifform distribution on $\mathbb{F}_q^{0 \times n}$, which has rank profile $(0, 0, \ldots, 0, 1)$. Thereafter, we increase the number m of rows and go from the uniform distribution on $\mathbb{F}_q^{(m-1) \times n}$ to the uniform distribution

on $\mathbb{F}_q^{m\times n}$. To calculate the evolution of the rank profile, we will need the matrix $M_n(q)$ defined as

$$M_n(q) := \begin{pmatrix} 1 & 1-\frac{1}{q} & 0 & \cdots & 0 & 0 \\ 0 & \frac{1}{q} & 1-\frac{1}{q^2} & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & \frac{1}{q^{n-1}} & 1-\frac{1}{q^n} \\ 0 & 0 & 0 & \cdots & 0 & \frac{1}{q^n} \end{pmatrix}. \tag{A.1}$$

The following result is well-known. We include its proof for completeness.

Lemma A.1. Let $\mathcal{D}_{m,n}$ be the uniform distribution on $\mathbb{F}_q^{m\times n}$. Then, the rank profile of $\mathcal{D}$ is given by the last column of the matrix $M_n(q)^m$, i.e., by the vector

$$M_n(q)^m \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}.$$

Proof. The proof uses induction on m. For $m = 0$, the rank profile of $\mathcal{D}_{0,n}$ is obviously $(0, 0, \ldots, 0, 1)$. Suppose now that the rank profile of $\mathcal{D}_{m,n}$ is $(p_n, p_{n-1}, \ldots, p_0)$. To assess the rank profile of $\mathcal{D}_{m+1,n}$, we sample from $\mathcal{D}_{m,n}$ to obtain a matrix A, and add to A a row x in which every entry is chosen uniformly at random from $\mathbb{F}_q$. The resulting matrix, A', is a matrix that is sampled from $\mathcal{D}_{m+1,n}$. For $\ell \geq 1$, we have

$$\begin{aligned} \Pr[\mathrm{rk}(A') = \ell] &= \Pr[\mathrm{rk}(A) = \ell \& x \in \mathrm{Im}(A)] \\ &\quad + \Pr[\mathrm{rk}(A) = \ell - 1 \& x \notin \mathrm{Im}(A)], \\ &= \Pr[\mathrm{rk}(A) = \ell] \Pr[x \in \mathrm{Im}(A) \mid \mathrm{rk}(A) = \ell] \\ &\quad + \Pr[\mathrm{rk}(A) = \ell - 1] \Pr[x \notin \mathrm{Im}(A) \mid \mathrm{rk}(A) = \ell - 1], \end{aligned}$$

where $\mathrm{Im}(A)$ denotes the vector space generated by the rows of A. As x is chosen uniformly at random from $\mathbb{F}_q^n$, the probability that x belongs to a given subspace of dimension d is q^{d-n}, and the probability that it is not in this subspace is $1 - q^{d-n}$. Hence, we have

$$p'_\ell := \Pr[\mathrm{rk}(A') = \ell] = p_\ell q^{\ell-n} + p_{\ell-1}(1 - q^{d-1-n}).$$

For $\ell = 0$, we have

$$p'_0 := \Pr[\mathrm{rk}(A') = 0] = \frac{1}{q^{nm}},$$

as in this case A' has to consist of all zeros. Putting everything together, we see that

$$\begin{pmatrix} p'_n \\ p'_{n-1} \\ \vdots \\ p'_1 \\ p'_0 \end{pmatrix} = M_n(q) \cdot \begin{pmatrix} p_n \\ p_{n-1} \\ \vdots \\ p_1 \\ p_0 \end{pmatrix}.$$

The result follows. □

The above theorem is often used in the following more general form.

Theorem A.2. Let $\mathcal{D}$ be a probability distribution on $\mathbb{F}_q^{m\times n}$ and A be a matrix in $\mathbb{F}_q^{(m+t)\times n}$ of the form $A = (C/D)$, where C is sampled from $\mathcal{D}$ and the entries of D are chosen independently and uniformly over $\mathbb{F}_q$. Let $(p'_n, p'_{n-1}, \ldots, p'_0)$ denote the rank profile of $\mathcal{D}$, and $(p_n, p_{n-1}, \ldots, p_0)$ that of A. Then, we have

$$\begin{pmatrix} p_n \\ p_{n-1} \\ \vdots \\ p_1 \\ p_0 \end{pmatrix} = M_n(q)^h \begin{pmatrix} p'_n \\ p'_{n-1} \\ \vdots \\ p'_1 \\ p'_0 \end{pmatrix}.$$

This theorem is obviously a generalization of Lemma A.1: In the situation in the lemma $m = 0$. The proof of the theorem is essentially identical to that of the lemma. It also uses induction; however, the induction start is at the case where $A = D$ (i.e., $h = 0$).

B

Failure Probability of R10 and RQ

In this section, we will present overhead-failure curves of R10 and RQ for various loss rates and many different values of K.

B.1 Methodology

B.1.1 Computing the Failure Probability

For both R10 and RQ, we are going to provide results of a large number of simulations and report the observed failure rates. For R10, we are going to use the list of K-values given in Table B.1, whereas for RQ, we are going to use the K-values given in Tables B.2 and B.3.

For R10, we tested each of the K-values for loss probabilities 0.1, 0.2, 0.4, 0.6, and 0.85 (results for the last three loss rates are still in the making). The number of overhead symbols was between 0 and 40.

For RQ, we tested each of the K-values in Tables B.2 and B.3 for loss probabilities 0.1, 0.2, 0.5, 0.6, and 0.85, and with a number of overhead symbols between 0 and 10. The reason for a smaller number of overhead symbols in the case of RQ is the much smaller failure probability as compared with R10. With the number of experiments, we were running for each value of K, it is very unlikely to see an error event beyond 10 overhead symbols.

Table B.1. Values of K used for the failure probability estimation of R10.

10	40	70	100	130	160	190	230
260	290	320	350	380	410	440	470
500	530	560	590	620	670	700	730
760	790	820	850	880	910	940	970
1,000	1,030	1,060	1,090	1,122	1,155	1,188	1,222
1,258	1,294	1,332	1,371	1,410	1,452	1,494	1,538
1,583	1,629	1,677	1,726	1,777	1,829	1,883	1,939
1,996	2,055	2,115	2,178	2,242	2,308	2,377	2,448
2,520	2,595	2,672	2,751	2,833	2,917	3,004	3,094
3,186	3,281	3,379	3,480	3,584	3,691	3,801	3,915
4,032	4,153	4,277	4,405	4,537	4,673	4,813	4,958
5,107	5,261	5,419	5,582	5,749	5,922	5,981	6,100
6,284	6,473	6,668	6,869	7,076	7,289	7,508	7,734
7,967	8,126						

Table B.2. Values of K used for the failure probability estimation of RQ.

10	11	12	13	18	19	20	21
26	27	30	31	32	33	36	37
42	43	46	47	48	49	50	55
56	60	61	62	63	69	70	75
76	84	85	88	89	91	92	95
96	97	98	101	102	114	115	119
120	125	126	127	128	138	139	140
141	149	150	153	154	160	161	166
167	168	169	179	180	181	182	185
186	187	188	200	201	213	214	217
218	225	226	236	237	242	243	248
249	257	258	263	264	269	270	280
281	295	296	301	302	305	306	324
325	337	338	341	342	347	348	355
356	362	363	368	369	372	373	380
381	385	386	393	394	405	406	418
419	428	429	434	435	447	448	453
454	466	467	478	479	486	487	491
492	497	498	511	512	526	527	532
533	542	543	549	550	557	558	563
564	573	574	580	581	588	589	594
95	600	601	606	607	619	620	633
634	640	641	648	649	666	667	675
676	685	686	693	694	703	704	718
719	728	729	736	737	747	748	759
760	778	779	792	793	802	803	811
812	821	822	835	836	845	846	860
861	870	871	891	892	903	904	913
914	926	927	938	939	950	951	963
964	977	978	989	990	1,002	1,003	1,020

(*Continued*)

Table B.2. (*Continued*)

1,021	1,032	1,033	1,050	1,051	1,074	1,075	1,085
1,086	1,099	1,100	1,111	1,112	1,136	1,137	1,152
1,153	1,169	1,170	1,183	1,184	1,205	1,206	1,220
1,221	1,236	1,237	1,255	1,256	1,269	1,270	1,285
1,286	1,306	1,307	1,347	1,348	1,361	1,362	1,389
1,390	1,404	1,405	1,420	1,421	1,436	1,437	1,461
1,462	1,477	1,478	1,502	1,503	1,522	1,523	1,539
1,540	1,561	1,562	1,579	1,580	1,600	1,601	1,616
1,617	1,649	1,650	1,673	1,674	1,698	1,699	1,716
1,717	1,734	1,735	1,759	1,760	1,777	1,778	1,800
1,801	1,824	1,825	1,844	1,845	1,863	1,864	1,887
1,888	1,906	1,907	1,926	1,927	1,954	1,955	1,979
1,980	2,005	2,006	2,040	2,041	2,070	2,071	2,103
2,104	2,125	2,126	2,152	2,153	2,195	2,196	2,217
2,218	2,247	2,248	2,278	2,279	2,315	2,316	2,339
2,340	2,367	2,368	2,392	2,393	2,416	2,417	2,447
2,448	2,473	2,474	2,502	2,503	2,528	2,529	2,565
2,566	2,601	2,602	2,640	2,641	2,668	2,669	2,701
2,702	2,737	2,738	2,772	2,773	2,802	2,803	2,831
2,832	2,875	2,876	2,906	2,907	2,938	2,939	2,979
2,980	3,015	3,016	3,056	3,057	3,101	3,102	3,151

Table B.3. Values of K used for the failure probability estimation of RQ.

3,152	3,186	3,187	3,224	3,225	3,265	3,266	3,299
3,300	3,344	3,345	3,387	3,388	3,423	3,424	3,466
3,467	3,502	3,503	3,539	3,540	3,579	3,580	3,616
3,617	3,658	3,659	3,697	3,698	3,751	3,752	3,792
7,93	3,840	3,841	3,883	3,884	3,924	3,925	3,970
3,971	4,015	4,016	4,069	4,070	4,112	4,113	4,165
166	4,207	4,208	4,252	4,253	4,318	4,319	4,365
4,366	4,418	4,419	4,468	4,469	4,513	4,514	4,567
5,68	4,626	4,627	4,681	4,682	4,731	4,732	4,780
4,781	4,838	4,839	4,901	4,902	4,954	4,955	5,008
009	5,063	5,064	5,116	5,117	5,172	5,173	5,225
5,226	5,279	5,280	5,334	5,335	5,391	5,392	5,449
450	5,506	5,507	5,566	5,567	5,637	5,638	5,694
5,695	5,763	5,764	5,823	5,824	5,896	5,897	5,975
976	6,039	6,040	6,102	6,103	6,169	6,170	6,233
6,234	6,296	6,297	6,363	6,364	6,427	6,428	6,518
519	6,589	6,590	6,655	6,656	6,730	6,731	6,799
6,800	6,878	6,879	6,956	6,957	7,033	7,034	7,108
109	7,185	7,186	7,281	7,282	7,360	7,361	7,445
7,446	7,520	7,521	7,596	7,597	7,675	7,676	7,770
771	7,855	7,856	7,935	7,936	8,030	8,031	8,111
8,112	8,194	8,195	8,290	8,291	8,377	8,378	8,474
475	8,559	8,560	8,654	8,655	8,744	8,745	8,837

(*Continued*)

Table B.3. (*Continued*)

8,838	8,928	9,019	9,111	9,206	9,303	9,400	9,497
601	9,708	9,813	9,916	10,017	10,120	10,241	10,351
10,458	10,567	10,676	10,787	10,899	11,015	11,130	11,245
1,358	11,475	11,590	11,711	11,829	11,956	12,087	12,208
12,333	12,460	12,593	12,726	12,857	13,002	13,143	13,284
3,417	13,558	13,695	13,833	13,974	14,115	14,272	14,415
14,560	14,713	14,862	15,011	15,170	15,325	15,496	15,651
5,808	15,977	16,161	16,336	16,505	16,674	16,851	17,024
17,195	17,376	17,559	17,742	17,929	18,116	18,309	18,503
8,694	18,909	19,126	19,325	19,539	19,740	19,939	20,152
20,355	20,564	20,778	20,988	21,199	21,412	21,629	21,852
2,073	22,301	22,536	22,779	23,010	23,252	23,491	23,730
23,971	24,215	24,476	24,721	24,976	25,230	25,493	25,756
6,022	26,291	26,566	26,838	27,111	27,392	27,682	27,959
28,248	28,548	28,845	29,138	29,434	29,731	30,037	30,346
0,654	30,974	31,285	31,605	31,948	32,272	32,601	32,932
33,282	33,623	33,961	34,302	34,654	35,031	35,395	35,750
6,112	36,479	36,849	37,227	37,606	37,992	38,385	38,787
39,176	39,576	39,980	40,398	40,816	41,226	41,641	42,067
2,490	42,916	43,388	43,840	44,279	44,729	45,183	45,638
46,104	46,574	47,047	47,523	48,007	48,489	48,976	49,470
9,978	50,511	51,017	51,530	52,062	52,586	53,114	53,650
54,188	54,735	55,289	55,843	56,403			

Testing was done by performing the following procedure for a given pair of K and loss probability p. The parameter N will be discussed below, and the parameter O is 40 for R10 and 10 for RQ.

(1) Create a vector `nfail` of length 11 and set its entries to zero.
(2) For $j = 1, 2, \ldots, N$, do
(3) Create $K + O$ ESIs according to the following process: go through the list of all possible ESIs, and drop each one independently with probability p until $K + O$ ESIs $(e_0, \ldots, e_{K+9})$ are created.
    (a) For $i = 0, 1, \ldots, O$, do:
        (i) Test whether the decoder works when input with the set of ESIs $(e_0, e_1, \ldots, e_{K+i})$.
        (ii) If not, then increase `nfail`$[i]$ by one.
        (iii) If decoding works, then increase j by one and go to Step 3.

At the end of this procedure, $\texttt{nfail}[i]$ gives the number of runs in which a code overhead of i was not sufficient for recovery. The plots below give for every value of K the estimated failure probability, which, for an overhead of i symbols, is $\texttt{nfail}[i]/N$.

For R10, the value of N is always between 10^7 and 10^8. More precisely, the value of N was chosen according to the following rule in order to keep the running time of the whole experiment manageable:

$$N = \begin{cases} 10^8, & \text{if } K < 500, \\ 5 \times 10^7, & \text{if } 500 \leq K < 2{,}000, \\ 10^7, & \text{if } K \geq 2{,}000, \end{cases}$$

In total, $33,145,200,000$ tests were performed for the R10 code.[1]

For RQ, the value of N is always between 10^7 and 2×10^7. In total, $39,901,000,000$ tests were performed for the RQ code.

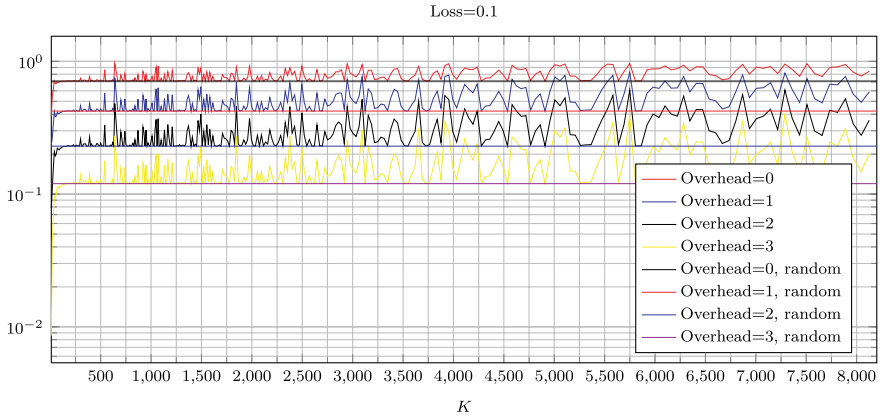
More details on the results are given in the following sections.

B.2 The Failure Probability of R10

We used the methodology described in Section B.1.1 to estimate the failure probability of R10. Results are gathered in the following sections.

B.2.1 Failure Probability Over the Entire Range

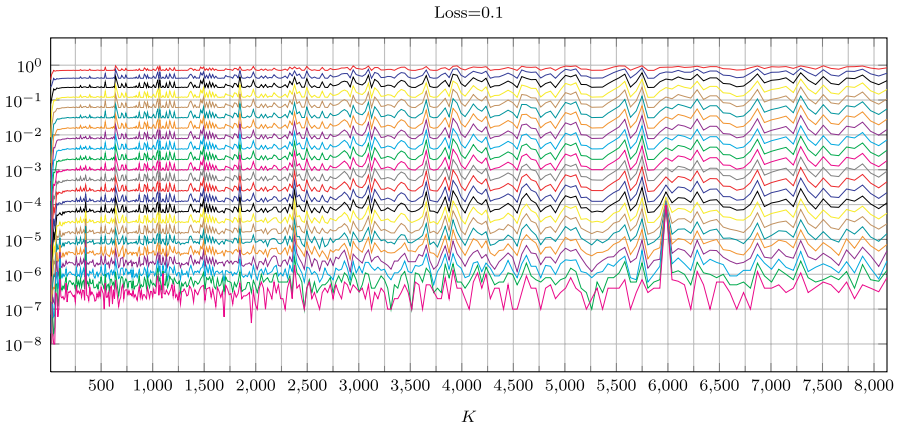
The following plots show the failure probability of R10 over the range of values of K given in Table B.1.



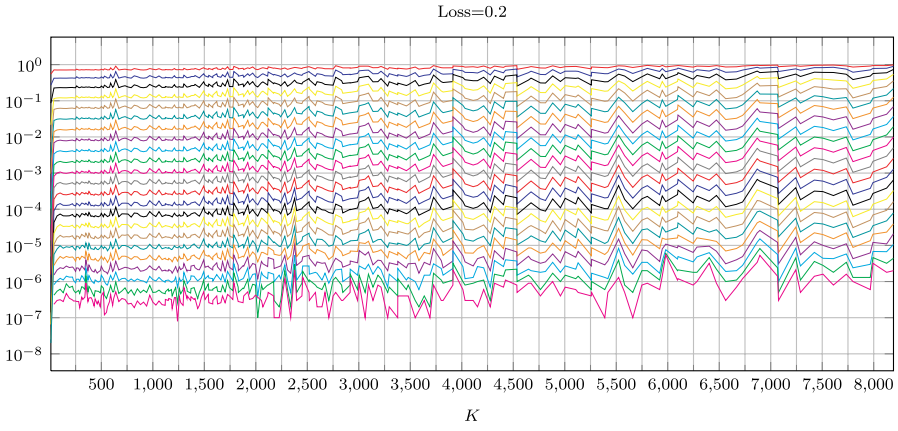
[1] For loss rates 0.1 and 0.2, we performed tests for sets of K-values that included those in Table B.1, but were substantially larger.

As can be seen, the failure probability is not quite smooth, though for quite a lot of values it is very close to the failure probability of the random binary fountain.

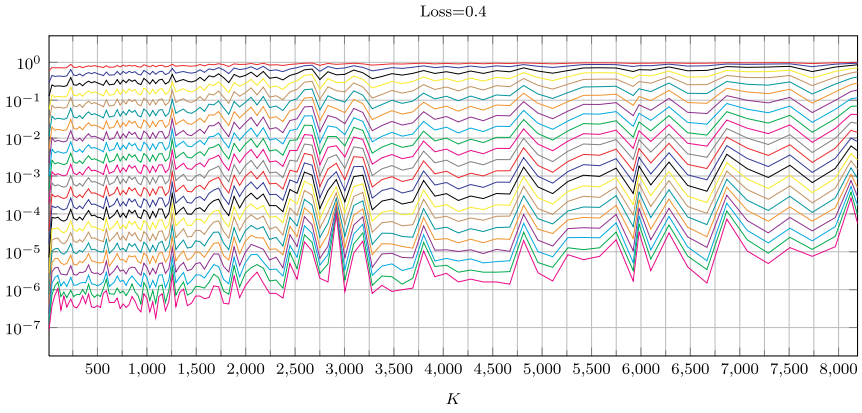
The following plot gives the failure probabilities for overhead symbols between 0 and 22:



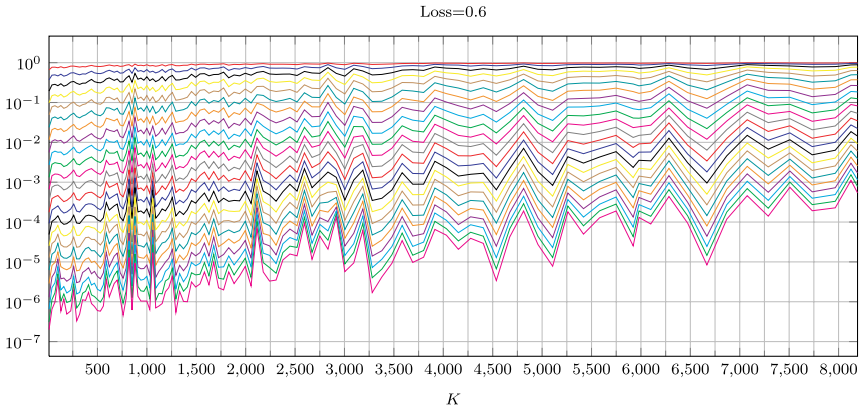
Some of the K-values behave worse than others, as can be seen. The plot below shows the performance of R10 when the loss rate is 0.2. The overhead is between 0 and 22:



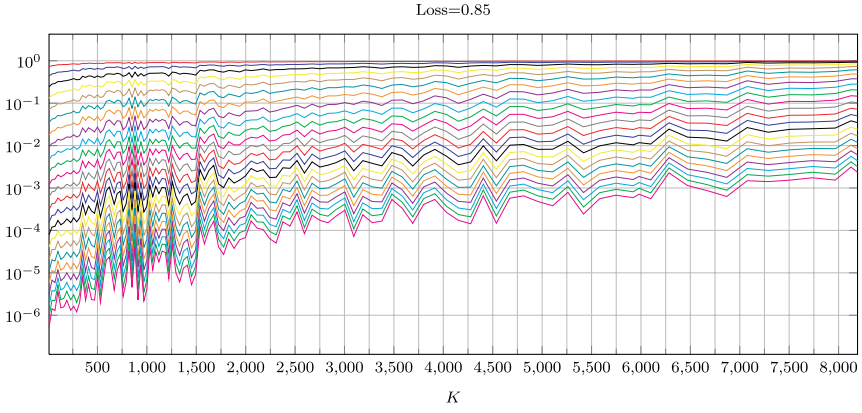
The plot below shows the performance of R10 when the loss rate is 0.4.



The plot below shows the performance of R10 when the loss rate is 0.6.



The plot below shows the performance of R10 when the loss rate is 0.85.



B.3 The Failure Probability of RQ

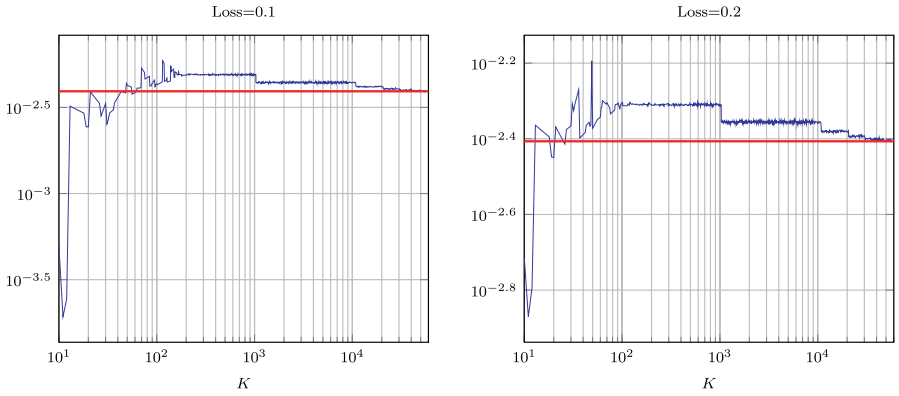
In this section, we provide results of a large number of simulations of the RQ code and report the observed failure rates. We refer the reader to Section B.1.1 for the methodology of our failure tests.

The number of failure events seen depends on the overhead used: for an overhead of zero, the number of failure events was of the order of 4×10^4, which gives us a fairly good confidence in the estimated failure rates. For a code overhead of one symbol, the number of failure events was of the order of 2×10^2, which still provides a good confidence in the estimated values. The number of failure events seen for a code overhead of 2 was never larger than 8, and in fact equal to 0 most of the time; hence, the numbers do not give a confident estimate of the failure probability with two overhead symbols.

More details on the results are given in the following sections.

B.3.1 Failure Probability at Zero Overhead

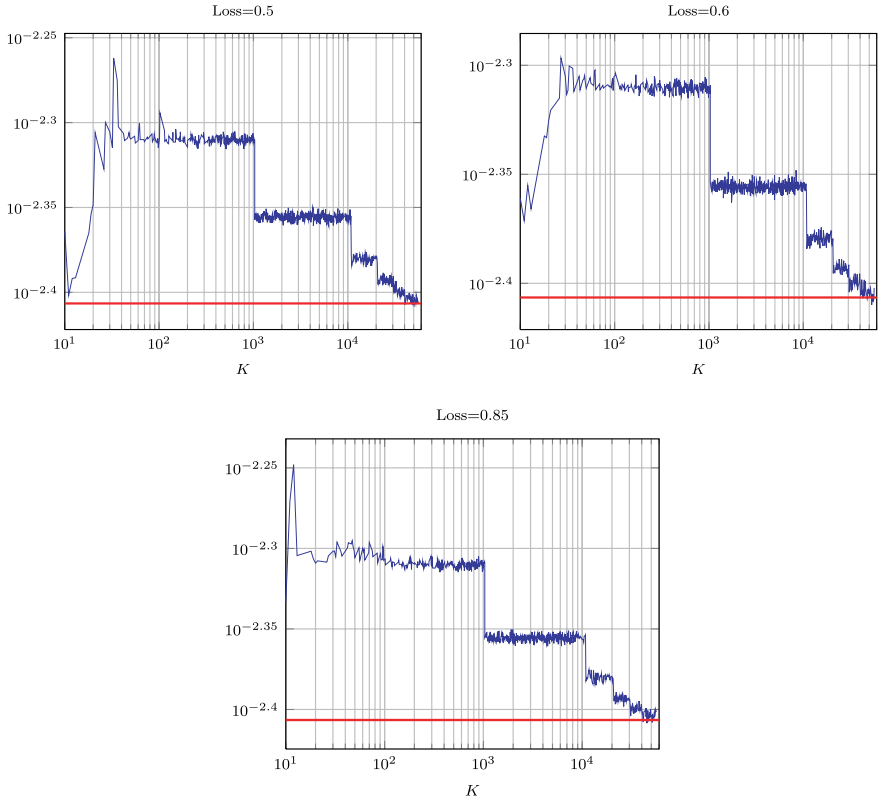
We first discuss the failure probability of RQ when the overhead is zero, that is, the number of received symbols equals K, the number of source symbols. In all cases, we compare the performance against that of the random fountain over $\mathbb{F}_{256}$ for which the failure probability is the red line.



It is interesting to see that for a loss rate of 0.1 and when K is below 100, the failure probability of RQ is often better than that of the random fountain. The reason for this behavior lies in the choice of the systematic indices, and the fact that for a loss rate of 0.1 the received

symbols are very close to the original source symbols, which would lead to a failure probability of 0.

The following plots give the failure rates for loss rates of 0.5, 0.6, and 0.85. For these loss rates, we do not see the above effect anymore. Moreover, as we can see, the performance of the RQ-code is almost exactly like that of a random fountain over $\mathbb{F}_{256}$, especially for larger values of K.

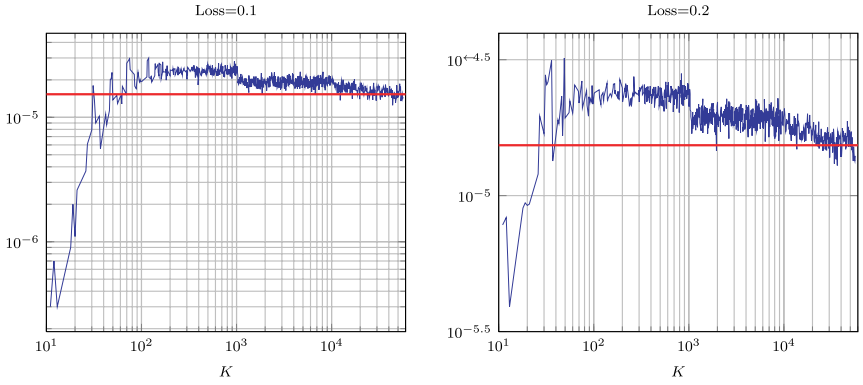


The failure probability approaches that of a random fountain over $\mathbb{F}_{256}$ (red line) as K grows; however, it does so in "steps." The reason for these steps lies in the different numbers of HDPC symbols for different values of K. The switching points for this value are given below. They coincide with the observed drops of the failure probability:

K	10	1,032	10,899	20,778	30,654	40,398	50,511
# HDPC	10	11	12	13	14	15	16

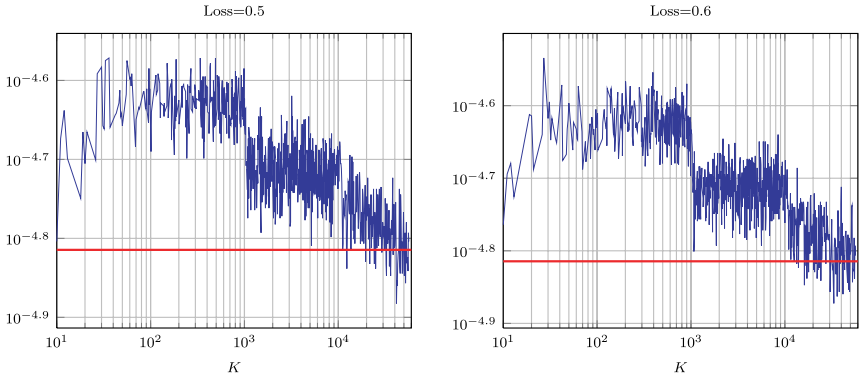
B.3.2 Failure Probability with One Overhead Symbol

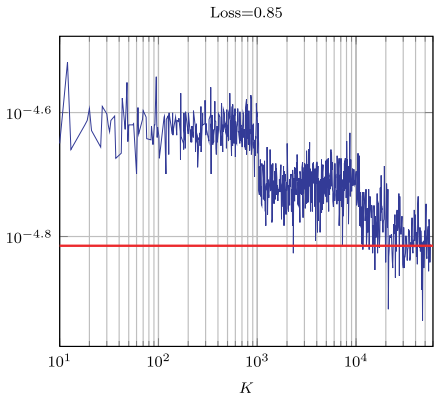
As was discussed before, our failure probability estimates when the number of overhead symbols is one are less reliable than the corresponding estimates for zero overheads. This is simply because the absolute numbers of error events observed are almost 200 times smaller. In the plots below, this is manifested by the rather strong "jitter" in the curves.



These two plots are very similar to their corresponding plots in the case of zero overhead symbols. As before, the red line is the failure probability of the random fountain over $\mathbb{F}_{256}$ when there is one overhead symbol. In some cases, e.g., at $K = 1032$, we see a drop of the failure probability as a result of a change in the number of HDPC symbols. The next drops are not prominent, though, because of the strong jitter in the plots.

The following three plots depict the behavior for loss rates equal to 0.5, 0.6, and 0.85.

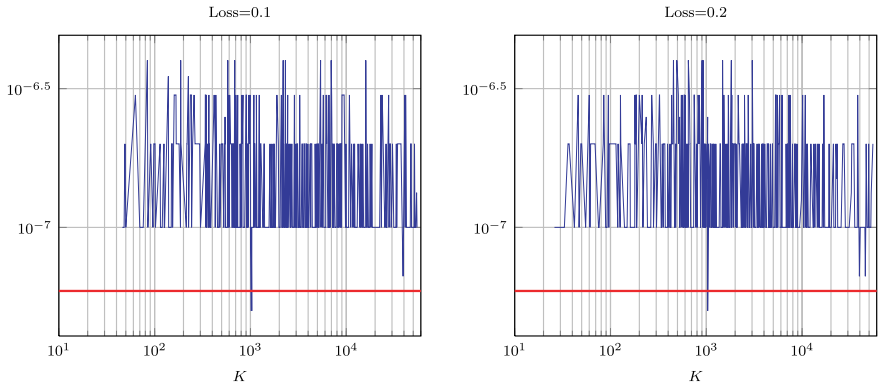


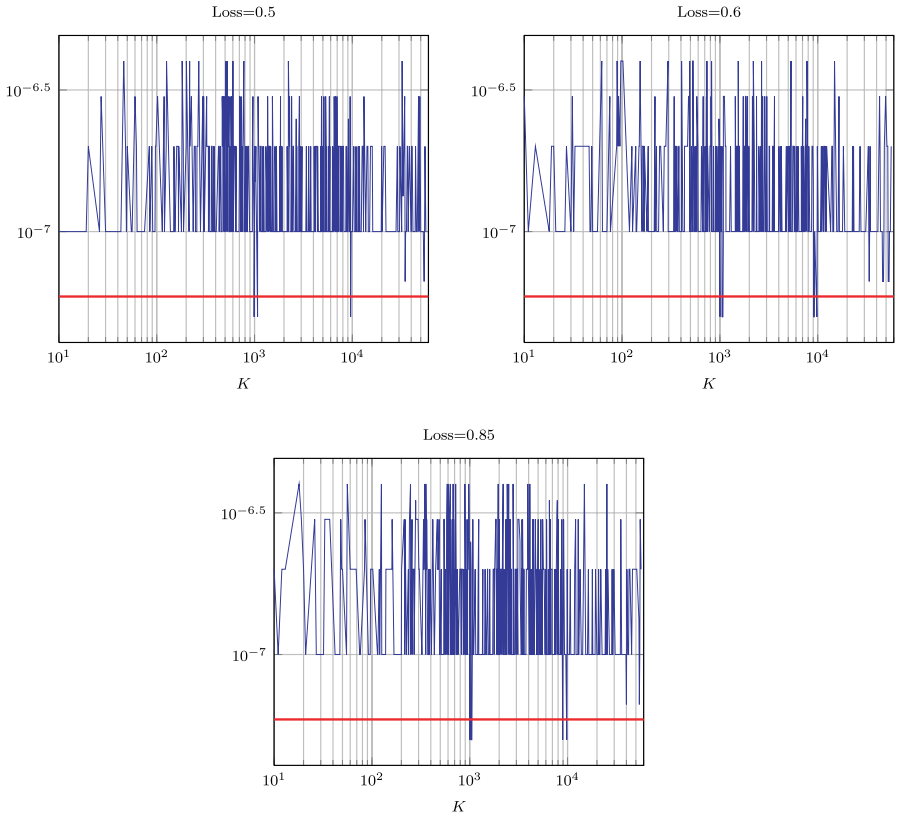


Here, too, the behavior is similar to the case in which the overhead is 0. However, we see a number of cases where the failure probability is lower than that of a random fountain, even when the loss rate is large. This is mainly due to the increased jitter of the curves, caused by the relatively small number of failure events observed. Looking at the average of these curves, it is reasonable to expect that the failure probability is practically the same as that of a random fountain, especially when K is large.

B.3.3 Two Overhead Symbols

With two overhead symbols, we expect to see 0.598 errors in 10^7 runs; hence, the number of runs that we have chosen (between 10^7 and 2×10^7) is too small to yield a reliable estimate of the failure probability. Nevertheless, we include the graphs for completeness.





These graphs only plot the cases where the number of observed failures is larger than zero. Among the total number of 3,865 cases considered, we observed 2,427 cases in which the number of failures was larger than zero. Based on these results, we expect that the failure probability with two overhead symbols is close to that of a random fountain over $\mathbb{F}_{256}$.

Unfortunately, the number of cases tested does not allow an accurate estimate of the failure probability when the number of overhead symbols is larger than 2.

Acknowledgments

The work reported on in this document has been a collaborative effort of many and the authors would like to take the opportunity to thank them in alphabetical order:

Andrew Brown and *Giovanni Cangiani* for helping to create the first implementation of R10.

Steve Chen for creating the first commercial implementation of R10 and RQ, optimizations thereof, and for commercialization of RQ.

Christian Foisy for his support for the R10 and RQ code, and his support for all questions regarding the commercial implementations of R10 and RQ.

Eric Holm for his help in creating the first commercial implementations of R10.

Richard Karp for his help in creating the concept of inactivation decoding and inactivation strategies.

Ranganathan Krishnan for running some of the large simulations and identifying some of the design weaknesses with the earlier versions of the RQ code.

Søren Lassen for his first implementations of Raptor codes, many optimizations, and data structures that are used in today's commercial

implementations of R10 and RQ, and for his help in creating inactivation decoding.

Lorenz Minder for helping to create the first implementation of R10, research on a subsequent version, his first implementation of RQ, commercialization of RQ, and his help in obtaining systematic indices for RQ, which led to the results reported in Section B.3.

Thomas Stockhammer for his help in standardizing R10 and RQ.

Mark Watson for his help in implementing and optimizing R10 and a subsequent version thereof, for the commercialization of R10 including standardization efforts, and for the systematic indices used for R10.

An anonymous reviewer for constructive comments on a first version of the paper.

Last but by no means least, the authors would like to express their gratitude to *Roberto Padovani* who initiated the idea of writing this document and facilitated its creation.

References

[1] 3GPP TS 26.346 V6.1.0, "*Technical Specification Group Services and System Aspects; Multimedia Broadcast/Multicast Service; Protocols and Codecs*," June, 2005.

[2] J. R. Bitner, G. Ehrlich, and E. M. Reingold, "Efficient generation of the binary reflected Gray code and its applications," *Communications of the ACM*, vol. 19, no. 9, pp. 517–521, 1976.

[3] J. Byers, M. Luby, M. Mitzenmacher, and A. Rege, "A digital fountain approach to reliable distribution of bulk data," in *Proceedings of ACM SIGCOMM '98*, 1998.

[4] DARPA Internet Program, "Internet protocol," September 1981, Internet Engineering Task Force, RFC 791. Available at http://www.ietf.org/rfc/rfc0793.txt?number=791.

[5] DARPA Internet Program, "Transmission control protocol," September 1981, Internet Engineering Task Force, RFC 793. Available at http://www.ietf.org/rfc/rfc0793.txt?number=793.

[6] ETSI TS 102 472 v1.2.1, "*IP Datacast over DVB-H: Content Delivery Protocols*," March 2006, Technical Specification. Available at http://www.dvb-h.org.

[7] R. G. Gallager, *Low Density Parity-Check Codes*. Cambridge, MA: MIT Press, 1963.

[8] B. A. Lamacchia and A. M. Odlyzko, "Solving large sparse linear systems over finite fields," in *Proceedings CRYPTO'90*, pp. 109–133, Springer, 1991.

[9] M. Luby, "LT-codes," in *Proceedings of the 43rd Annual IEEE Symposium on the Foundations of Computer Science (FOCS)*, pp. 271–280, 2002.

[10] M. Luby and V. Goyal, "Wave and equation based rate control," April 2004, Internet Engineering Task Force, RFC 3738. Available at http://tools.ietf.org/html/rfc3738.

[11] M. Luby, V. Goyal, S. Skaria, and G. Horn, "Wave and equation based rate control," in *Proceedings of SIGCOMM*, pp. 191–204, 2002.

[12] M. Luby, M. Mitzenmacher, and 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.

[13] M. Luby, M. Mitzenmacher, A. Shokrollahi, and D. Spielman, "Efficient erasure correcting codes," *IEEE Transactions Information Theory*, vol. 47, pp. 569–584, 2001.

[14] M. Luby, M. Mitzenmacher, A. Shokrollahi, D. Spielman, and V. Stemann, "Practical loss-resilient codes," in *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pp. 150–159, 1997.

[15] M. Luby, A. Shokrollahi, M. Watson, and T. Stockhammer, "Raptor forward error correction scheme for object delivery," September 2007, Internet Engineering Task Force, RFC 5053. Available at http://tools.ietf.org/html/rfc5053.

[16] M. Luby, A. Shokrollahi, M. Watson, T. Stockhammer, and L. Minder, "RaptorQ forward error correction scheme for object delivery," August 2010, Internet Engineering Task Force. Available at http://tools.ietf.org/html/draft-ietf-rmt-bb-fec-raptorq-03.

[17] C. Pomerance and J. W. Smith, "Reduction of huge, sparse matrices over finite fields via created catastrophes," *Experimental Math*, vol. 1, pp. 89–94, 1992.

[18] J. Postel, "User datagram protocol," August 1980,Internet Engineering Task Force, RFC 768. Available at http://www.ietf.org/rfc/rfc0768.txt?number=768.

[19] T. Richardson and R. Urbanke, "Efficient encoding of low-density parity-check codes," *IEEE Transactions Information Theory*, vol. 47, pp. 638–656, 2001.

[20] A. Shokrollahi, "Raptor codes," *IEEE Transactions Information Theory*, vol. 52, no. 6, pp. 2551–2567, 2006.

[21] A. Shokrollahi, "Theory and applications of raptor codes," in *Proceedings of MathKnow*, pp. 59–89, 2009.

[22] A. Shokrollahi, S. Lassen, and R. Karp, "Systems and processes for decoding chain reaction codes through inactivation," U.S. Patent number 6,856,263. February 15, 2005.

[23] A. Shokrollahi, S. Lassen, and M. Luby, "Multi-stage code generator and decoder for communication systems," U.S. Patent 7,068,729. June 27, 2006.

[24] A. Shokrollahi and M. Luby, "Systematic encoding and decoding of chain reaction codes," U.S. Patent 6 909 383. June 21, 2005.

[25] V. V. Zyablov and M. S. Pinsker, "Decoding complexity of low-density codes for transmission in a channel with erasures," *Probl. Information Transmission*, vol. 10, 1974.