Fountain Codes and Covert Channels

Ewelina Marciniszyn

May 20, 2022

Bachelorarbeit im Bachelorstudiengang Informatik

Matrikelnummer: 9711961

Prüfer: Prof. Dr. Jörg Keller

FernUniversität in Hagen

Contents

List of Symbols

K	Number of source blocks in the fountain code
K_S	Number of source blocks in the covert fountain code
T	Number of transmitted packets required to decode the message
T_S	Number of transmitted covert packets required to decode the secret message
ρ	Probability distribution for degrees
d^*	The degrees where the covert fountain code is embedded
S	The size of the secret message, in bit
b_S	The number of bit in the payload of the covert packets
p_S	The probability that the secret message can successfully be decoded
α	The ratio of covert packets to overt packets
s_i	A source block with index i
t_i	The i-th transmitted encoded packet

1 Introduction

Digital fountain codes, introduced in [1], are a way of transmitting data over a one-way communication channel, in a reliable way. The idea is that a message is encoded into an endless stream of packets that each contain a somewhat random part of the message. Once the receiver has received enough packets, the message can be decoded. The name *fountain code* stems from the analogy with a water fountain, which sprays water droplets in an endless stream, and a receiver can fill up a glass of water by collecting water droplets. It does not matter which droplets, or packets, are received, as long as enough of them are received in the end. The communication is one-way only, and any level of packet loss is acceptable.

The way fountain codes work is that the message is split into K source blocks. Each packet is then constructed by choosing a number of source blocks randomly (the number of blocks that are chosen is called the *degree*) and XOR-ing them together. These packets are sent over the communication channel. To reconstruct the original message the receiver can reconstruct each single block by inverting the XOR operation, which can be achieved by XOR-ing the received packets together in the correct way. The receiver can only know which packets to XOR when it knowns which source blocks were used to create the packet. So the sender also adds a *header* to the packet with this information.

In [6] it was shown that it is possible to embed a covert channel into a fountain code. This creates a secret, hidden way of sending messages from a covert sender to a covert receiver. An observer can not detect that any covert communication has taken place.

As the fountain code is one-way only, the covert receiver has no way of asking the covert sender for a retransmission in case of packet loss. To solve this problem the secret message that is transmitted in the covert channel is itself encoded as a fountain code. This deals with packet loss by providing redundancy, just like a normal fountain code. It also opens up the possibility of creating additional levels of fountain codes, secrets within secrets, as mentioned in [6], but we will not explore that idea in this work. Instead we focus on the first level of secrets only.

There is a downside of using a secret fountain code, which is that if the secret fountain code does not transmit enough packets for the covert receiver to decode the secret message, then the transmission of the secret will fail. This can happen when the normal receiver stops listening to packets after receiving enough normal fountain code messages. In the water analogy, the receiver will only collect water until the glass is full. If at that moment a covert receiver has not received enough “special” water drops, then the secret will not be decoded.

In this work we investigate the properties of the covert channel with secret fountain code that was proposed in [6]. We look in particular at the size of the secret message and at the chance of sucessfully transmitting a secret message over the covert channel. Then we discuss several modifications of the original approach, to increase the secret message size,

the success rate, or both.

We also introduce a new method, which we call Half-Degree, that gives a larger secret message size and a higher success rate than the original approach. We try to show this theoretically, and we also present many simulated results for all methods.

A large part of this work was the creation of Java code to test the behaviour of a secret fountain code embedded in a normal fountain code. The code was written in an Object-Oriented approach, and has been supplied together with this paper.

In the rest of this paper we first explain the concepts of a covert channel (section 2) and a fountain code (section 3) in more detail. We look in particular at the different types of headers, Luby's efficient decoding algorithm, and various distributions of degrees.

Then in section 4 we show how to embed a covert channel in a fountain code. We first explain the original method from [6] where only one degree is used. We then extend this method to multiple degrees and investigate different settings. Finally we introduce our Half-Degree method, and compare all methods theoretically and with experimental simulations.

Section 5 explains our implementation in Java via class diagrams, provides instructions on how to run the code, and discusses the motivation behind certain architectural choices. We also describe several algorithms that were used in the code.

We conclude this work with a final discussion and outlook to future work, section 6.

2 Covert Channel

Normal communication channels allow for the exchange of information (messages) between two parties, a sender and a receiver. The existence of the channel is known to everyone. The sender and receiver can hide the content of their messages via suitable forms of encryption, however the fact that communication takes place is known to any observer with access to the channel. An example is communication over Wi-Fi. As the communication packets travel over the air, any third party can "listen in" and find out that communication is happening. If the communication is not encrypted then the third party can even read the message content.

A covert channel is a hidden communication channel that attempts to hide the fact that communication is taking place. It does this by using methods that are not intended for information transfer at all [7]. This means that an observer will not be aware of the communication. For example, we could modify the bits in unimportant or unused parts of the headers of TCP/IP packets. The covert sender and the covert receiver communicate by interpreting these bits as secret messages. The observer does not suspect this, so is unaware of the communication. Another example of a covert channel is the introduction of artificial delays in the transmission of normal communication packets. The length of the delays denotes bits and thus forms a secret message.

Covert channels can be desirable to some and undesirable to others. A corporate whistle-blower or political dissident might want to exfiltrate information from within a high security network, and thus needs to use a covert channel to remain undetected. But the network administrator or the government would like to prevent this, by detecting the use of covert channels or by making certain that no covert channels can exist. For the example with the unused headers of TCP/IP packets, the network administrator could add filters that block packets that have non-0 bits in the unused headers. The network administrator could also rewrite those bits to become 0, thus destroying any potential covert channel. Many different covert channels and their countermeasures have been studied. See [15] or [11] for a classification and for many more examples.

It is important to note that the covert sender and covert receiver must agree in advance on which covert channel to use, and how to communicate over the covert channel. This is not different from a normal (overt) sender and receiver who also must agree in advance on the channel and protocol for communication to happen. For example, the agreement could be to communicate over electrical wires using the TCP/IP protocol. In this work we assume that there exists some way for the covert sender and covert receiver to come to this common understanding, so that when we design a covert channel protocol we can assume that both covert sender and covert receiver know exactly how the covert channel works.

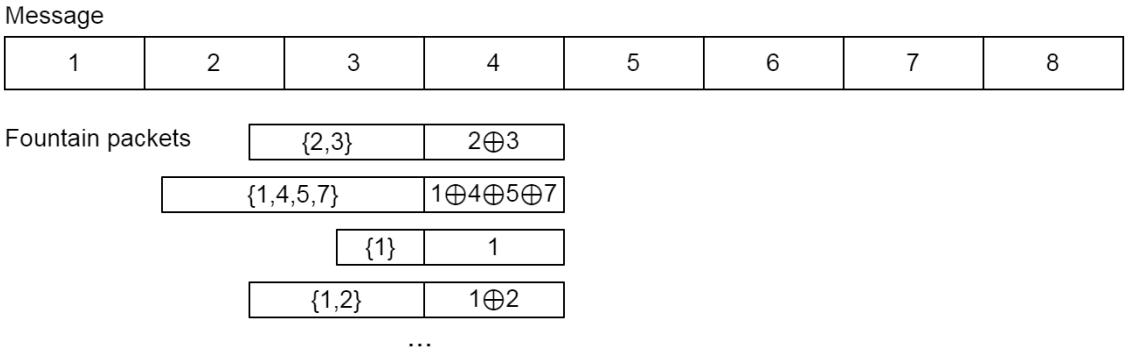


Figure 1: A message is split into equal-size source blocks, which are XOR-ed randomly to form a fountain code. A fountain code packet consists of a header and a payload. The header describes the content of the XOR-ed payload.

3 Fountain Code

When a message needs to be sent over a channel that does not allow two-way communication, data corruption or packet loss can be problematic. The receiver has no way to ask the sender to retransmit a certain packet. A common solution is to include forward error correction in the packets, for example by using Reed-Solomon error correcting codes. Each packet is extended with a number of extra bits that help with restoring the parts of the message that got lost or corrupted. The choice of how many extra bits to add is made based on the expected error rate of the channel. However when the error rate is not known, or when it fluctuates, then the choice of the number of extra bits will either be too high, resulting in wasted bits, or too low, resulting in data loss.

Fountain codes [8, 10] offer an alternative to forward error correction, by creating a potentially infinite stream of redundant data. The idea is that a message is first split into K equal-sized source blocks of integer length. Padding should be added to the message when its length is not a multiple of K. The source blocks are then randomly XOR-ed together to form the *payloads* of the packets. These payloads are prefixed with a header, describing which blocks were used in the XOR-ing. An unending stream of packets, each created from different combinations of source blocks, is then submitted over the channel. The receiver listens to the channel until enough packets have been received to reconstruct the message. This is illustrated in figure 1.

To construct a transmitted packet t from a set of source blocks $\{s_i\}$, first a degree d is chosen, with $d \in [1\,..\,K]$. The values of d are not chosen uniformly but instead follow a certain probability distribution $\rho(d)$. The choice of ρ influences the performance of the fountain code, and many different distributions have been proposed [8, 5, 4, 12, 9]. After d is chosen, a set of d integers is chosen uniformly without replacement from the set $1..K$. These are the indices of the source blocks that will be XOR-ed together.

For example, if we have $K = 16$ and d is chosen to be 4, and the source block indices

that are chosen are $\{3, 5, 11, 12\}$, then we will create the payload $t = s_3 \oplus s_5 \oplus s_{11} \oplus s_{12}$. Another example is that when $d = 1$ then the payload will be just a single source block. The degree distribution ρ determines how often each degree d is chosen.

3.1 Header formats

The payloads on their own do not contain enough information to reconstruct the original source blocks. The receiver also needs to know, for each payload, which source blocks were used in its creation. This information is transmitted as part of a header that is added as a prefix to each payload, see figure 1. The combination of header and payload is the transmitted packet. We want to keep the header short (in number of bits), to minimise the overhead in each transmitted packet.

We discuss 3 different ways to construct the header: bitvector (BV), enumerate (ENUM), and enumerate all (ENUMALL). In the following explanations we use the example of a packet $t = s_2 \oplus s_7 \oplus s_8 \oplus s_{13}$ with $K = 16$ to illustrate the header construction. Note that in this paper the source block indices start counting at 1 and run until K, whereas in the Java code we start counting at 0 and run until $K - 1$.

3.1.1 BV

This is the simplest header. It consists of K bits, representing the indices of the K source blocks. Each of those bits has a value of 0 if the source block at this index is not used, and a value of 1 if the source block is used in the transmitted packet.

In our example the source block indices are $\{2, 7, 8, 13\}$, so putting a binary `1` on those positions gives the BV header `0100001100001000`. Any given BV header can also easily be converted back to a set of source block indices by finding the indices of the bits with value `1`.

3.1.2 ENUM

To create a packet of degree d, we need to choose d out of K source blocks. We can use results from combinatorics to compute the number of different ways that d elements can be selected out of K without replacement,

$$\frac{K(K-1)\cdots(K-d+1)}{d(d-1)\cdots 1} = \frac{K!}{d!(K-d)!} = \binom{K}{d} \tag{1}$$

We can write down an ordered sequence of all the different unique combinations of d source blocks and then assign a value n to each combination based on its position in the sequence. The values of the enumeration range from 0 to $\binom{K}{d} - 1$. The enumeration gives an index number n for each header. For example for $K = 16$, $d = 4$ the first combination is $\{1, 2, 3, 4\}$, followed by $\{1, 2, 3, 5\}$, $\{1, 2, 3, 6\}$ and so on. These combinations correspond to the index numbers 0, 1, 2 and so on. In section 5.3.3 we discuss an algorithm to convert

directly between any combination and its corresponding index number n, without needing to construct the full ordered sequence.

The ENUM header consists of a binary representation of the degree d followed by a binary representation of the index number n, which we now explain in detail.

Degree number To encode the degree d we need to know how many bits to reserve. For this the maximum degree $d_{\max}$ is considered, and the number of bits required to represent $d_{\max}$ in binary is computed. There is no degree 0, so if we equate $d = 1$ with the binary representation of 0, and $d = d_{\max}$ with the binary representation of $d_{\max} - 1$, then the number of bits required is $b = \lceil \log_2 d_{\max} \rceil$. For example, encoding $d = 4$ when $d_{\max} = 16$ requires $b = 4$ bits and reads `0011`. Table 1 shows all the possible degree numbers in binary when $d_{\max} = 8$.

degree	bits	degree	bits
1	`000`	5	`100`
2	`001`	6	`101`
3	`010`	7	`110`
4	`011`	8	`111`

Table 1: Binary representation of the degree for the ENUM header, for dense degrees.

If not all degrees are present, for example if only degrees of powers of 2 are used (see section 3.3.3 for sparse degrees), then a different mapping between degree number and binary value can be created, needing less bits. For example, the degrees of powers of 2 up to $d_{\max} = 128$ can be encoded in just 3 bits using the mapping shown in table 2. The dense mode would require 7 bits.

degree	bits	degree	bits
1	`000`	16	`100`
2	`001`	32	`101`
4	`010`	64	`110`
8	`011`	128	`111`

Table 2: Binary representation of the degree for the ENUM header, for sparse degrees.

Index number To convert n to binary we need to consider the highest value $n_{\max}$ to determine the number of bits to reserve. For a given degree d, $n_{\max} = \binom{K}{d}$, so we should reserve $\left\lceil \log_2 \binom{K}{d} \right\rceil$ bits.

For our example, $\binom{16}{4} = 1820$, requiring $\lceil 10.829\ldots \rceil = 11$ bits. The set $\{2, 7, 8, 13\}$ has $n = 703$ (see section 5.3.3 for a calculation). Written in 11-bit binary this gives `01011000000`. Combined with the degree number `0011` (degree 4 in non-sparse mode) we get the full header `001101011000000`.

An important remark is that the length of the header depends on the degree. As seen, degree 4 requires 11 bits. Degree 1 would only need $\left\lceil \log_2 \binom{16}{1} \right\rceil = 4$ bits, so the length of the header is different. The receiver can dynamically figure out the length of the header based on the degree number that is given as the first bits of the header. However it is also possible to use the same fixed header length across all possible degrees, by reserving enough bits for the highest possible n value. With $K = 16$ the maximum n is reached at degree 8, being $n_{\max} = \binom{16}{8} = 12870$. This means that in this case up to 14 bits are required to represent all possible values of n, and our example set $\{2, 7, 8, 13\}$ will have the binary encoding `00001011000000`. Including the degree number as well gives `001100001011000000`.

In the Java code we use the second approach, i.e. with the additional zeros. This keeps the size of the transmitted packets consistent.

3.1.3 ENUMALL

This header format is similar to ENUM, but we do not start counting from zero for each degree. Instead, the degrees are ordered in ascending order and all possible combinations are enumerated for each degree without restarting from zero. The index number n is then encoded in binary to form the header. Because each number n uniquely represents a combination of any degree, the degree does not need to be explicitly specified in the header.

Enough bits should be reserved for being able to write the highest value $n_{\max}$. If all degrees from 1 to K are used, this value is

$$\sum_{i=1}^{K} \binom{K}{i} = 2^K, \tag{2}$$

which means that K bits are needed, just like in BV mode.

Considering our example again, there are $\binom{16}{1} = 16$ combinations for degree 1, $\binom{16}{2} = 120$ for degree 2, and $\binom{16}{3} = 560$ for degree 3, so the number n for the first combination in degree 4, $\{1, 2, 3, 4\}$, equals $16 + 120 + 560 = 696$. Previously we found that $\{2, 7, 8, 13\}$ has $n = 703$ for the ENUM header, so if we start counting from the degree 1 then this gives $n = 696 + 703 = 1399$. Expressed in binary we get the header `0000010101110111`.

The benefit of ENUMALL over BV and ENUM is that less bits are required when not all degrees are used. If only degrees of powers of 2 are used, as will be the case in this paper, then for $d_{\max} = 8$ we would have a total of only $\binom{16}{1} + \binom{16}{2} + \binom{16}{4} + \binom{16}{8} = 14826$ values, instead of $2^{16} = 65536$. So in that case 14 bit are enough to encode an ENUMALL header, saving 2 bit.

3.2 Decoding algorithm

To reconstruct the original source blocks the receiver needs to XOR the received encoded packets together in the correct way. For example, having received $t_1 = s_3 \oplus s_5$ and $t_2 = s_5$, the receiver can compute the source block $s_3 = t_1 \oplus t_2$. The hard part is to figure out which received encoded packets need to be XOR-ed together. We discuss two different algorithms: Gaussian Elimination (GE) and the decoder for LT codes.

3.2.1 Gaussian Elimination

In the Gaussian Elimination algorithm the indices in all received headers are first collected into a matrix $\mathbf{G}$, called the *generator matrix* [10], to create a transformation from the source blocks to the encoded packets. The matrix $\mathbf{G}$ contains only the values 1 and 0, where a 1 in row i and column j means that the source block s_j was used in the creation of transmitted packet t_i, and 0 if it was not used. If we received T transmitted packets and there are K source blocks, then $\mathbf{G}$ has size $T \times K$. Writing as G_{ij} the value of $\mathbf{G}$ at row i and column j, the transmitted packet t_i is then given by the formula

$$t_i = \bigoplus_{j=1}^{K} G_{ij} s_j. \tag{3}$$

By inverting $\mathbf{G}$ modulo 2 we can find an expression for s in function of t,

$$s_j = \bigoplus_{i=1}^{T} G_{ij}^{-1} t_i. \tag{4}$$

For example, if we have $K = 4$ and $T = 4$ and the received packets are $t_1 = s_1 \oplus s_2 \oplus s_3$, $t_2 = s_1 \oplus s_3$, $t_3 = s_2 \oplus s_4$ and $t_4 = s_2 \oplus s_3 \oplus s_4$, then we write

$$\begin{pmatrix} t_1 \\ t_2 \\ t_3 \\ t_4 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \end{pmatrix} \tag{5}$$

Solving this system of equations modulo 2 gives

$$\begin{pmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} t_1 \\ t_2 \\ t_3 \\ t_4 \end{pmatrix} \tag{6}$$

so that we compute the source blocks as $s_1 = t_2 \oplus t_3 \oplus t_4$, $s_2 = t_1 \oplus t_2$, $s_3 = t_3 \oplus t_4$ and $s_4 = t_1 \oplus t_2 \oplus t_3$.

Matrix inversion is only possible for square matrices, and in general the number of rows T will be greater than the number of columns K. For such non-square matrices the solution of the system of equations (3) can be obtained via LU decomposition. The matrix $\mathbf{G}$ is factorised into a lower triangular matrix L and an upper trianguler matrix U. The factorization is done via Gaussian elimination, so this way of decoding a fountain code is called Gaussian elimination (GE) [10].

Even with LU decomposition not all matrices $\mathbf{G}$ will be solvable, as $\mathbf{G}$ can be singular, i.e. there are rows that are linearly dependent on the other rows. This corresponds to the scenario where at least one of the received packets can be constructed from the other received packets via XOR operations, and thus does not contribute any new information. In that case the receiver will have to wait for additional packets that will hopefully add new information and make the system of equations (3) solvable.

In [10] the probability that a random binary matrix of size $K \times K$ is invertible, where each element of the matrix is either 1 or 0 with equal probability, was computed to be $(1-2^{-K})(1-2^{-K-1})\cdots(1-\frac{1}{4})(1-\frac{1}{2})$. This comes to a probability of 0.289 for K larger than 10. So this kind of fountain code transmission will only be solvable 28.9% of the time after receiving K packets. Receiving an extra E packets, so that $T = K + E$, rapidly increases the probability that the system can be solved. It was found that for any K the probability that the receiver will not be able to decode the transmission when given E extra packets, is bounded above by 2^{-E}. So receiving just 10 extra packets already gives a probability of about 0.999 that the transmission will succeed.

The upper bound of 2^{-E} does not depend on the number of source blocks K. If we measure the efficiency of the fountain code by the ratio ϵ of extra packets that needs to be transmitted, $T = (1+\epsilon)K$, then we can make the fountain code arbitrarily efficient, i.e. we can have very low values for ϵ by choosing high values for K.

The downside of the GE algorithm is that the time complexity of performing LU decomposition for a square matrix with K rows and columns is $O(K^3)$. This can be acceptable for small K, but becomes very computationally intensive for large K. Thus in practice a different algorithm is used.

3.2.2 LT Decoder

The LT decoder proposed by Luby in [8] reduces the number of computations required for decoding a fountain code, at the cost of increasing the number of extra packets E required. The decoding algorithm works by only XOR-ing received packets with source blocks that have already been extracted. Each time a new packet is received, XOR-ing removes the known source blocks that were used in the creation of the received packet. If this XOR-ing results in a single unknown source block remaining, then we add this new source block to the list of known source blocks, and apply this new knowledge to previously received packets. We stop when all source blocks are known.

In detailed steps the Java code works as follows:

1. Store each incoming encoded packet in a list of known packets.
2. If the encoded packet has degree $d = 1$, then simply extract the source block. Add the source block to a list of known source blocks. No XOR operations are required.
3. If step 2 gave a previously unknown source block, then XOR this new source block to each of the encoded packets in the list of known packets that have $d > 1$ and that contain the new source block. The encoded packet thus loses the information of this new source block, since $s_i \oplus s_i = 0$. The set of the encoded packets that are affected by this operation is called the *ripple*.
4. Process each packet in the ripple by checking its current degree. The operation from step 3 reduced the degree of these encoded packets by one. If the ripple packet that is currently under consideration now has $d = 1$ then go to step 2. Remove each processed packet from the ripple.
5. The ripple is now empty.
   a If some source blocks are still unknown then wait for a new incoming encoded packet. Use XOR to remove known source blocks from this incoming encoded packet and go to step 1.
   b If all source blocks are known then we are done.

This algorithm is less complex than an LU decomposition and requires less computational resources, but it must have enough low-degree packets in order to get started. The number of additional packets E will also be bigger than for the LU decomposition method.

The example from the previous section (3.2.1) does not have any packets with $d = 1$, so step 1 in the LT decoder never starts. No source blocks are decoded. However if we receive a new packet, $t_5 = s_1$, then we know s_1 and can perform the following operations:

1. Remove s_1 from the packets that contain s_1:

$$t'_1 = t_1 \oplus s_1 = s_1 \oplus s_2 \oplus s_3 \oplus s_1 = s_2 \oplus s_3$$
$$t'_2 = t_2 \oplus s_1 = s_1 \oplus s_3 \oplus s_1 = s_3$$

2. We now know s_3 from t'_2, so we can now remove s_3 from the other packets:

$$t''_1 = t'_1 \oplus s_3 = s_2 \oplus s_3 \oplus s_3 = s_2$$
$$t'_3 = t_3 \oplus s_3 = s_3 \oplus s_4 \oplus s_3 = s_4$$
$$t'_4 = t_4 \oplus s_3 = s_2 \oplus s_3 \oplus s_4 \oplus s_3 = s_2 \oplus s_4$$

3. From t''_1 we learn s_2 and from t'_3 we get s_4, so all source blocks are known and we can stop.

3.3 Degree distributions

The LT Decoder algorithm relies on having enough packets of degree 1. However if only packets of degree 1 are transmitted, then the receiver might have to wait a long time for the last missing source block. As shown in [10], we can compute how many packets of degree 1 we need to receive on average (if we *only* receive packets of degree 1) by noting that each source block has a chance of $\frac{1}{K}$ to be chosen. Thus the probability that a particular block has not been chosen yet after N packets, is

$$\left(1 - \frac{1}{K}\right)^N \tag{7}$$

which approximates to $e^{-N/K}$ if N and K are high enough. It follows that the expected number of missing blocks is $Ke^{-N/K}$, because there are K blocks. To have no missing blocks we want this number to be smaller than 1, so

$$\begin{aligned} Ke^{-N/K} < 1 &\Rightarrow \ln\left(Ke^{-N/K}\right) < \ln 1 \\ &\Rightarrow \ln K + \ln e^{-N/K} < 0 \\ &\Rightarrow -N/K < -\ln K \\ &\Rightarrow N > K \ln K \end{aligned} \tag{8}$$

For $K = 64$ we would need to transmit 266 packets, and for $K = 1000$ we would need over 6,907 packets of degree 1, which is a very high overhead. Including packets with degrees higher than 1 improves this overhead, while still using the simple LT Decoder algorithm.

The probability of how often a certain degree is chosen to create a packet is determined by the degree distribution, written as $\rho(d)$. For example, $\rho(4) = 0.256$ means that degree 4 has a chance of 25.6% to be selected. The best distribution is the one that leads to the lowest overhead while still using the LT Decoder algorithm. It depends on the number of source blocks K and the set of allowed degrees (e.g. sparse or dense). Many papers have focussed on optimizing this distribution in various scenarios [5, 12, 9] after Luby first proposed the ideal and robust soliton distributions in [8].

3.3.1 Ideal soliton distribution

The LT Decoder algorithm needs only 1 packet of degree 1 to get started. Suppose that the source block of this degree-1 packet is also used in a packet of degree 2. Then these degree 1 and degree 2 packets can be XOR-ed to find another source block. If we also have a degree 3 packet that contains the two source blocks that we have so far, then we can XOR three packets together, and so on. In other words, the ripple is always 1.

The ideal soliton distribution is based on the expectation of a perfect ripple of size 1,

$$\rho_{\text{ideal}}(d) = \begin{cases} 1/K & d = 1 \\ \frac{1}{d(d-1)} & d > 1 \end{cases} \tag{9}$$

This is a distribution because $\sum_{i=1}^{K} \rho_{\text{ideal}}(i) = 1$. In practice this distribution does not lead to good (low) values for the overhead E, because small deviations from the expected behaviour stop the chain of packets, i.e. the ripple becomes 0 and the decoder needs to wait until a packet with the missing source block comes in.

3.3.2 Robust soliton distribution

To improve on the practical performance of the ideal soliton distribution, Luby proposed the robust soliton distribution [8]. The degree probabilities of (9) are modified by adding a function $\tau(d)$ and then renormalising,

$$\rho_{\text{robust}}(d) = \frac{\rho_{\text{ideal}}(d) + \tau(d)}{\sum_{i=1}^{K} \rho_{\text{ideal}}(i) + \tau(i)}. \tag{10}$$

The function $\tau(d)$ is defined as

$$\tau(d) = \begin{cases} \frac{1}{d}\frac{R}{K} & d = 1, 2, \ldots, K/R - 1 \\ \frac{1}{d}\ln(R/\delta) & d = K/R \\ 0 & d > K/R \end{cases} \tag{11}$$

Here the value R represents the expected ripple size, i.e. the expected number of degree-1 checks that are available at any step in the decoding process. It is computed from K and two additional parameters c and δ,

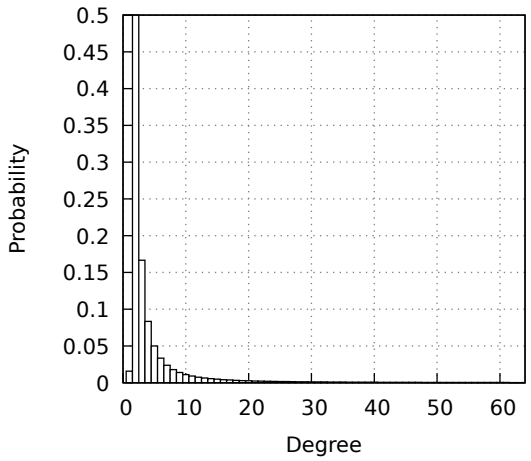
$$R = c \ln(K/\delta) \sqrt{K}. \tag{12}$$

The c and δ parameters can be chosen to obtain either a low average overhead E but a high variance of the overhead, or a high average overhead E with a low variance. See also the experiments in [10].

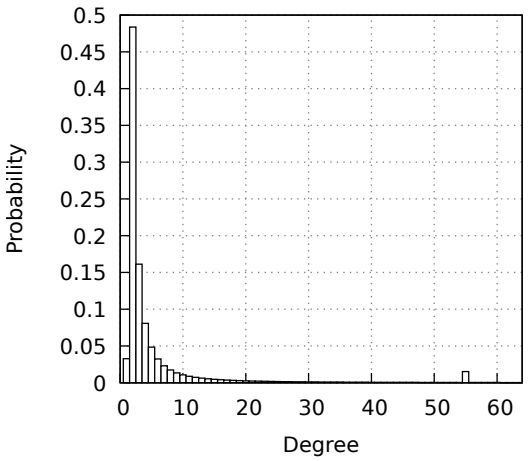
Figure 2 shows the ideal solition distribution and an example of the robust soliton distribution for $K = 64$. The main differences are that the robust version has a relatively high probability for degree 55, and also has a slightly higher probability for degree 1 than the ideal solition distribution. Intuitively, the extra packets of degree 1 help the decoding algorithm to get started, and the extra packets of degree 55 help the decoding algorithm to finish, by providing information about the last missing source blocks.

3.3.3 Sparse degree distributions

The distributions presented by Luby have the property that when K can be taken arbitrarily high, then the relative overhead will go to zero. However in practice we often can not make K arbitrarily high. In [5] it was shown that for low K values there exist better distributions than either the ideal or the robust solition, where “better” means that the



(a) Ideal soliton distribution



(b) Robust soliton distribution

Figure 2: The degree probabilities for the ideal and the robust solition distribution, for $K = 64$. For the robust solition we used $c = 0.03$ and $\delta = 0.5$.

K	16 [5]	64 [5]	16 [12]	64 [12]
$\rho(1)$	0.21	0.09	0.221	0.161
$\rho(2)$	0.47	0.49	0.457	0.4
$\rho(4)$	0.16	0.2	0.188	0.256
$\rho(8)$	0.16	0.13	0.134	0.101
$\rho(16)$	-	0.02	-	0.045
$\rho(32)$	-	0.07	-	0.037
Average T	22.5	81.9	22.6	82.7
Std $\hat{\sigma}(T)$	4.2	7.7	4.4	9.1

Table 3: The degree distributions from Hyytia et al. [5] and Rossi et al. [12] for $K = 16$ and $K = 64$.

choice of distribution results in fewer packets that need to be transmitted (on average) to successfully decode the fountain code message.

Optimal distributions for values of K lower than 10 were computed in [4]. It was also discovered that distributions which are close to the optimal distribution have nearly identical overheads as the optimal distribution. In addition, if some degree probabilities are set to zero so that not all degrees are used, i.e. creating a *sparse degree distribution*, then it is still possible to find degree probability values for the non-zero degrees so that the overhead is very close to optimal.

The benefit of using a sparse degree distribution is that we can use less bits in the header to encode the degree that was used in the fountain code packet. We discussed the example for sparse degrees of powers of 2 already in section 3.1.2. The use of only a sparse amount of non-zero degrees makes it possible to perform a numerical optimization to find the

optimal degree distribution, even for larger K values. Optimal sparse distributions of powers of 2 for $K = 16$, $K = 32$ and $K = 64$ were presented in Hyytia et al. [5] and in Rossi et al. [12]. Table 3 shows their sparse degree distributions, as well as the average and standard deviation of the number of packets T required to decode the fountain code message.

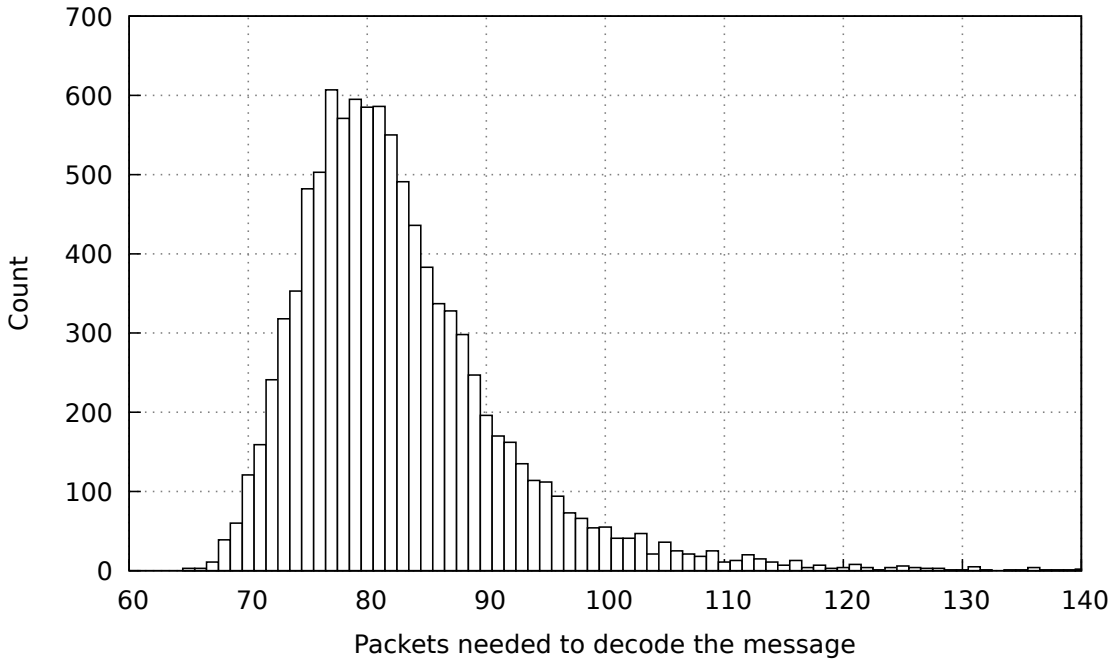


Figure 3: Histogram of the number of packets required to decode the fountain code for $K = 64$ and the optimal distribution from [12], as shown in table 3.

The average and standard deviation give only limited insight into the behaviour of the fountain code for these distributions. In figure 3 we present a histogram created from running 10,000 fountain codes with $K = 64$ and the optimal distribution from [12]. The histogram shows how many times in the 10,000 simulations a certain number of packets were required to decode the fountain code message. For example, in 585 out of the 10,000 simulations we needed to receive exactly 80 packets to decode the message. We see that the histogram does not follow a normal distribution. There are no histogram entries for fewer than 64 packets, and on the high end there is a long tail, meaning that occasionally a large number of packets is needed to successfully decode the message. Values above 140 packets exist, but we have omitted them in the graph as they are not very common.

We can also interpret the found histogram itself as a packet probability distribution. For example, from the histogram of figure 3 we can conclude that the probability that exactly 80 packets will be required is $585/10000 = 0.0585$.

If we experimentally determine the packet probability distribution of the two different degree distributions ρ in table 3, then we obtain the plot of figure 4. Even though the degree distributions in [5] and [12] are quite different, for example $\rho(4) = 0.2$ in [5] and

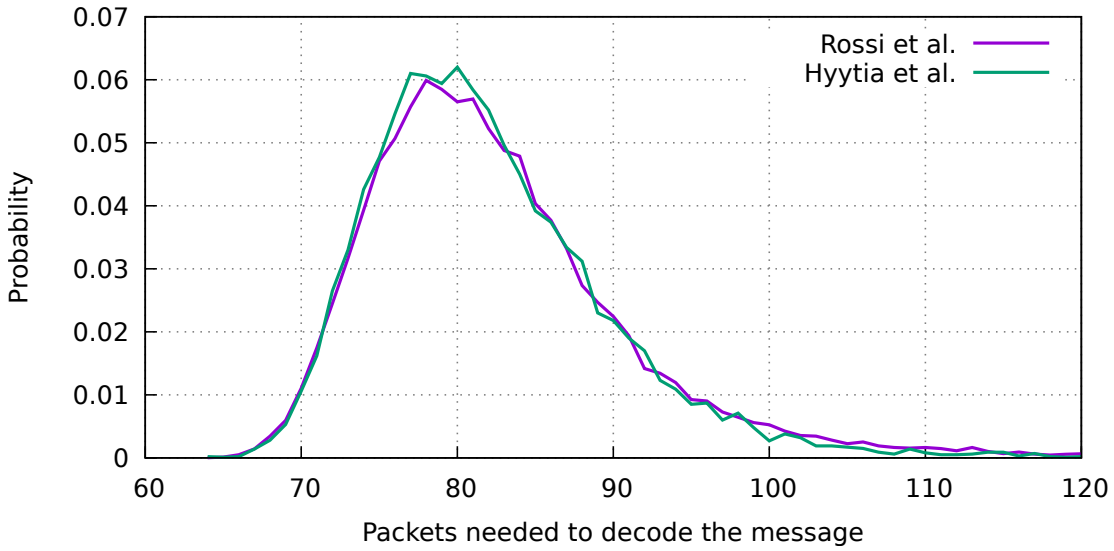
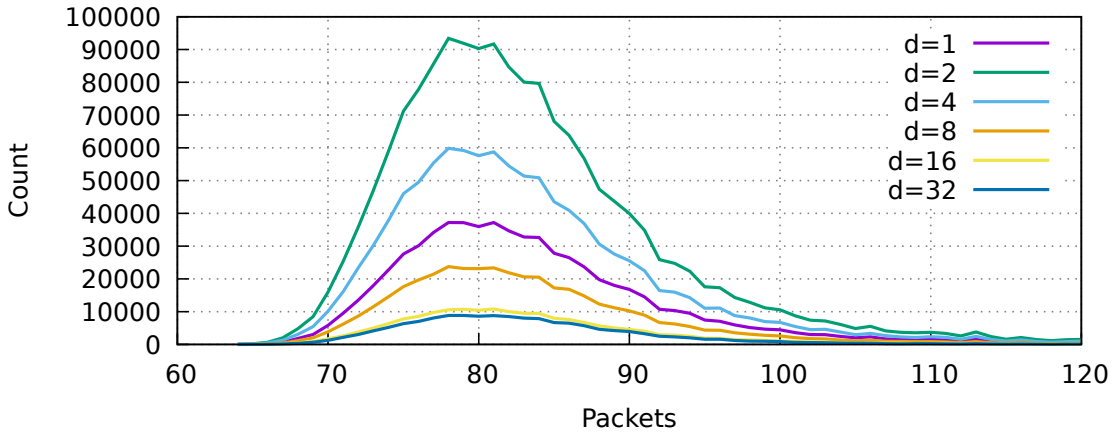


Figure 4: Comparison between the fountain code behaviour for $K = 64$ when using the distribution of Rossi et al. [12] and Hyytia et al. [5].

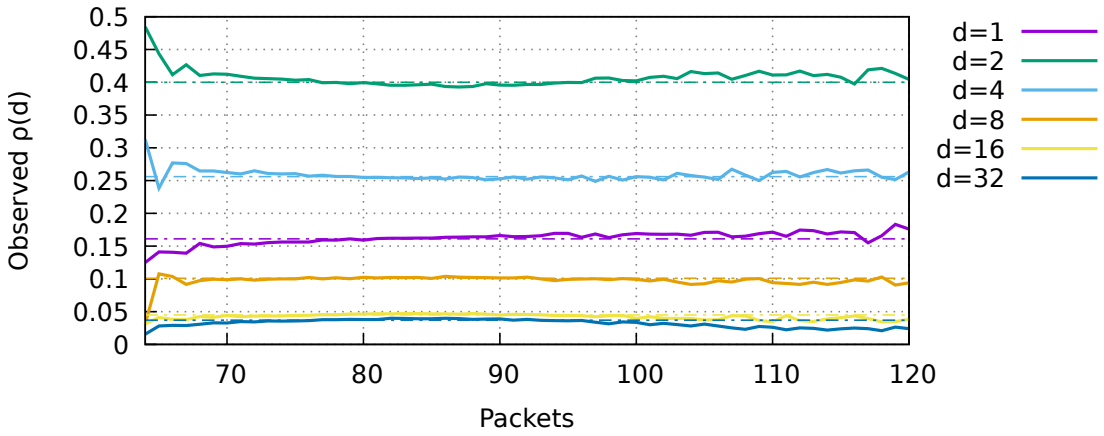
$\rho(4) = 0.256$ in [12], the behaviour of the fountain code is very similar. Hyytia et al. has a slightly higher concentration of packets near the peak than Rossi et al., and it happens slightly less often that a large number of packets are needed. This is reflected in both the average and standard deviation of Hyytia et al. being lower than those values of Rossi et al. However the difference is very small, and in practice either of these distributions is a good choice to use for a fountain code. We will continue to work with the distribution of Rossi et al., even though it is a bit worse than Hyytia et al., because the degree distribution in Rossi et al. favors the higher degrees, which is better for a covert channel (see next chapter).

To conclude our experimental analysis of the sparse degree fountain codes, we look at how the individual degrees are used for each amount of packets needed to decode the message. For example, it could be that when more than 90 packets are required to decode the message, that in this case there are relatively more packets of, say, degree 4 used. This would have an impact on the analysis of covert channels in the next chapter. To test this, we have repeated the experiment of figure 3 with 50,000 fountain code simulations, where for each successful decoding we also remember the number of packets of each degree that were used. Then we again combined the results into a histogram. This is shown in figure 5a. We see that the shape of the histogram of each individual degree follows the shape of the overall histogram of figure 3. In other words, it is not the case that the actual distribution $\rho(d)$ of the degrees is very different when many fountain code packets are required to decode the message.

To analyse this last statement in more detail, we compute the actual observed distribution for each of the number of required packets, i.e. for each vertical line in figure 5a. The distribution can be computed from the counts by simply dividing each count by the total



(a) Count of individual degrees. Degree 2 is highest because $\rho(2) = 0.457$.



(b) The observed $\rho(d)$ computed from the counts in graph (a).

Figure 5: Result of 50,000 simulations of a $K = 64$ fountain code, focusing on how often the various degrees are selected as a function of the number of packets required to successfully decode the message.

count per packet. The results are shown in figure 5b. The solid lines are the observed distributions, while the dashed horizontal lines were added to show the original $\rho(d)$ from table 3. We see that in the region between 75 and 90 packets the observed distribution follows the original distribution very closely. However when the fountain code needs less than 75 packets to finish, there are slightly more packets of degrees 2 and 4 used, and less packets of degree 1, than the original distribution would make us expect. A similar scenario happens when more than 90 packets are needed to finish.

Even though this analysis shows minor deviations from the original distribution in the case of low or high packet values, the deviations are relatively small. We will ignore these deviations in the rest of this paper, and assume that if a total of T packets are transmitted, then an amount of $\rho(d) \times T$ of those packets is of degree d (on average), independent of the actual value of T. For example, if 82 packets are needed, then on average 21 of those will be of degree 4 when using the distribution from [12] for $K = 64$ where $\rho(4) = 0.256$.

3.4 Variations of Luby fountain codes

Many variations of fountain codes have been proposed. We will analyse covert channels only for the fountain codes from Luby [8], but in this section we mention some additional approaches to reduce the transmission overhead.

Raptor codes [13] are a variation of LT fountain codes, where each of the K source blocks is modified to add information of the other source blocks to it, i.e. an outer code that can restore erasures [10]. This way not all K source blocks need to be received before the message can be decoded, because the missing blocks can be reconstructed from the extra information that was added to each source block. For example, receiving 90% of all augmented source blocks suffices to restore the remaining missing 10% of source blocks. This means that less encoded packets will be needed, although each packet will be larger. Another consequence is that Raptor codes can use a lower average degree number (an average of $d = 3$ is mentioned in [10]), which reduces the encoding and decoding complexity.

Further variations were made by observing that in LT codes the choice of source blocks is made uniformly. Each source block is equally likely, at all times. In addition, the degree distribution does not change.

Sorensen et al. [14] modifies the degree distribution depending on how many packets were sent already. The sender keeps track of the number of transmitted packets, and slowly modifies the degree distribution to favor higher degrees when many packets were sent. Having higher degrees at the end of the transmission helps the decoder to figure out the last remaining unknown source blocks. This leads to a lower overhead, as there is less waste of sending low-degree packets when the decoder already knows most source blocks and is waiting to receive the last few missing source blocks. The downside is that if the communication channel has a high packet loss rate, then the sender will modify the distribution too fast, because the sender can not know if a packet failed to arrive. This could lead to even higher overheads than without this modification.

This modification has no impact on the covert channel. As we will see, higher degrees are actually better for the covert channel, so it is likely that the method of [14] will lead to a higher success rate for the secret transmission.

Work by Hayajneh et al. [3] has investigated improvements to the fountain code performance by having the sender remember which source blocks it already sent. Unsent source blocks are temporarily given a higher probability to be selected. This way the sender helps the receiver, by wasting less packets sending information that the receiver already knows. The downside is again that packet loss is not known by the sender and can not be predicted. The sender will think that the receiver already knows a certain source block, while actually that packet was lost in transmission. This will again cause higher overheads.

Another downside, from our point of view of trying to embed a covert channel, is that the proposal of [3] makes the sampling of source blocks no longer uniform. As our covert

channel relies on random source block choices, a protocol that does not sample randomly will break our covert channel.

4 Covert Channel in Fountain Code

The main idea for having a covert channel in a fountain code, as first presented in [6], is to modify the choices of the source blocks when constructing encoded packets. Instead of choosing the source blocks randomly, they are chosen in such a way that the information contained in the header of the encoded packet can be interpreted as a secret message.

As a conceptual example, let us transmit a secret message inside a fountain code with $K = 16$. For simplicity we only use degree 1, and we transmit the secret text `"test"`. We first convert the text to numbers by counting the position of the letter in the alphabet, giving $(20, 5, 19, 20)$. Converting the numbers to 5-bit binary gives `10100|00101|10011|10100`. The header of the fountain code has $\binom{16}{1}$ options, which equates to $\log_2 \binom{16}{1} = 4$ bit. So we split the bitstring into groups of 4 bits, `1010|0001|0110|0111|0100`, and convert back to numbers, giving $(10, 1, 6, 7, 4)$. This tells us that we should send five fountain code packets containing the source blocks 10, 1, 6, 7, and 4, in that order. The encoded packets consist of a prefix (the header) describing the source block number, followed by the fountain code payload. The payload does not matter for the covert receiver, but the header does. The covert receiver extracts the numbers from the consecutive headers and runs the presented process in reverse, obtaining the secret message `"test"`.

The procedure in the above example illustrates how the header of encoded packets could be used to create a covert channel, but it has numerous practical problems that need to be solved:

1. The choice of random source blocks follows a uniform distribution. So the choice of well-chosen source blocks should also follow a uniform distribution, otherwise it would be possible for an observer to notice that something strange is going on in this fountain code and detect the covert channel.
2. The header should still make sense as a header when degrees higher than 1 are used. A source block should not occur multiple times in a header, e.g. the choice $\{2, 2, 4, 5\}$ is not valid.
3. The secret message transmission should still work when there is packet loss, when there is no return channel, or when there is out-of-order arrival of packets.
4. The secret message should have a high probability p_S to be successfully decoded.
5. It should be possible for the size S (in bit) of the secret message to be large.

Problems 1, 2 and 3 were solved in [6] by embedding a secret fountain code in the header of the normal fountain code. We will first present their approach in section 4.1. Then we investigate solutions to problems 4 and 5. We will first modify the approach from [6] in section 4.2 and then propose a new way of embedding the covert channel in section 4.3, followed by a comparison in section 4.4.

4.1 Covert Channel in one header degree

4.1.1 Approach

In [6] a covert channel was created in the headers of packets of a single degree d^*. The covert degree d^* is chosen in advance and known by both the covert sender and the covert receiver. With K source blocks there are $\binom{K}{d^*}$ valid possibilities for the header of degree d^*. If no covert channel is used then the source blocks in each header are chosen uniformly without replacement. This implies that the $\binom{K}{d^*}$ different headers are also uniformly distributed. The covert channel should maintain this property.

The bits of a secret message are in general not uniformly distributed. For example in the secret message `"this is a secret"` some letters occur more frequently than other letters. This can cause a non-uniform distribution of the headers. One solution is to first encrypt the secret message, e.g. using AES encryption. A good encryption ensures that frequency analysis can not be used to break the encryption, implying that each byte will occur with the same probability after encryption. Encryption solves problem 1.

Problem 2 can be solved by viewing all the different possible headers as unique symbols with a certain base. Only those symbols can be used in the transmission. A binary message, which is in fact a series of symbols $\{0, 1\}$ with base 2, is first converted to a series of symbols with base $\binom{K}{d}$. Each symbol then corresponds to a header. For example the fountain code of $K = 16$ has 1820 possible headers in degree $d^* = 4$. The message `"secret"`, encoded in 8-bit ASCII, is `01110011|01100101|01100011|01110010|01100101|01110100` base 2. Interpreting this sequence of bits as a big decimal number gives 126879297332596 base 10. This can be converted to base 1820 by writing the number as $11 \times 1820^4 + 1026 \times 1820^3 + 620 \times 1820^2 + 833 \times 1820 + 536$. So the message `"secret"` is encoded as the five symbols $\{11, 1026, 620, 833, 536\}$ base 1820. These five symbols could be embedded as five headers of consecutive fountain code packets of degree $d^* = 4$. For example to obtain a header with the symbol 11 we must select the source blocks $\{1, 2, 3, 15\}$ (assuming the fountain code uses ENUM mode). This solves problem 2.

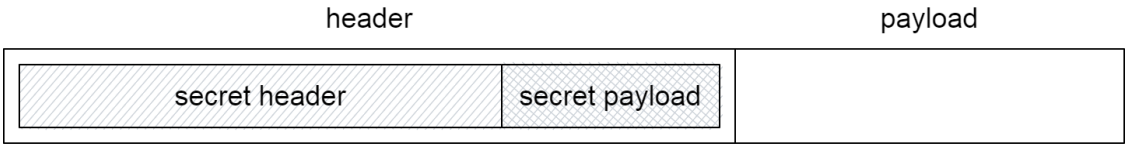


Figure 6: Embedding a secret fountain code in a covert channel of a normal fountain code.

Solving the third problem is more difficult, because the properties of the channel (unknown amount of packet loss, one-way communication only, possible out-of-order packet arrival) are what originally motivated the use of a fountain code. The solution of [6] is to also send the secret message itself with a fountain code. The secret message is split into K_S secret source blocks which are XOR-ed together and prefixed with a secret header, creating a secret encoded packet. The bits of the secret encoded packet are then interpreted as a symbol base $\binom{K}{d^*}$ and used as a header of the original, overt fountain code. The covert

receiver then applies the LT decoding algorithm on the received headers, and needs to wait until enough secret encoded packets have been received to reconstruct the secret message. Figure 6 illustrates this.

4.1.2 Additional problems

The use of a covert fountain code solves the problems created by an unreliable communication channel, but it creates a new problem because fountain codes are based on probabilities. The normal, overt fountain code will *eventually* finish successfully, because the receiver will continue listening until enough encoded packets have been received to decode the message. However the covert fountain code does not have this luxury. When the normal, i.e. overt receiver stops listening, no more covert packets will be obtained by the covert receiver. If through bad luck the covert receiver did not obtain enough covert packets, or is unable to decode the secret from the received covert packets, then the covert transmission will fail.

We define as p_S the probability that the covert transmission succeeds, i.e. that the covert receiver gets enough secret packets to successfully decode the secret message. Although our argument in the previous paragraph shows that p_S is always lower than 1, we should try to design the covert fountain code so that p_S is close to 1. This is problem 4 in the list.

Not all normal fountain code headers will contain a covert packet. For example, headers of degree 1 do not have enough available bits to embed a secret encoded packet. Only a subset of the T packets that are transmitted in the normal fountain code will contain packets from the covert fountain code. We introduce a factor α to describe the ratio of covert packets to normal packets. The expected number of transmitted covert fountain code packets $\overline{T_S}$, given an expected number of transmitted overt packets $\overline{T}$, is then

$$\overline{T_S} = \alpha \overline{T} \tag{13}$$

The use of a covert fountain code also adds overhead due to the need for secret headers, see figure 6. This means that not all available bits of the overt header can be used as secret payload. The covert fountain code limits the size of the secret message further, via the choice of the number of secret source blocks K_S. This is problem 5. Each secret payload is a combination of secret source blocks, so if we write as b_S the number of bits of the secret payload (which is also the number of bits of each secret source block), then the size S of the secret message is

$$S = b_S K_S \tag{14}$$

To illustrate problems 4 and 5, we look at the particular example of $K = 64$ that was analysed in [6], with optimal sparse distribution ρ from [12]. The degree $d^* = 4$ was chosen to contain the covert channel, with $\rho(4) = 0.256$. So the covert fountain code has $\log_2 \binom{64}{4} \approx 19.28$ bit available for the covert data, consisting of a secret header plus a secret

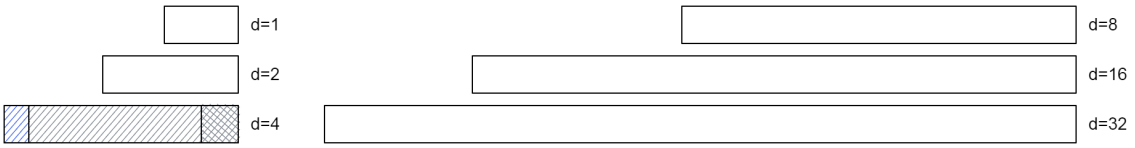


Figure 7: Using only degree $d^* = 4$ to embed the covert fountain code, as in the example of [6]. The secret packet consists of a secret header and a (small) secret payload. The blue part of the header is the integral factor.

payload. A covert fountain code with $K_S = 16$ with sparse degrees 1, 2, 4 and 8 was used. The header of the covert fountain code was encoded using a variant of ENUMALL with integral factors, which we explain in detail in the next section. The covert header was sized at 16 bit, leaving $b_S = 3$ bit available for the payload. Using these values in equation (14) gives $S = 48$ bit. This may be sufficient for certain applications, but it seems very constrained. Figure 7 illustrates this example.

To form an idea of the success rate p_S for this example we look at equation (13). All normal packets of degree 4 will have a covert packet, so

$$\overline{T_S} = \rho(d^*)\overline{T} = 0.256\overline{T} \tag{15}$$

In other words, $\alpha = 0.256$.

Is $\overline{T_S}$ large enough to decode the secret? From table 3 we read the expected values for T for the distribution used in [6], which is the distribution presented in [12]. The expected number of transmitted packets for $K = 64$ and $K = 16$ are

$$\overline{T_{64}} = 82.7, \quad \overline{T_{16}} = 22.6 \tag{16}$$

The covert channel contains a (secret) fountain code with $K = 16$, so we can use equation (15) to estimate how many secret packets $\overline{T_S}$ we expect to receive, given that we expect to get $\overline{T_{64}} = 82.7$ normal packets. This gives an expected value of $\overline{T_S} = 0.256 \times 82.7 = 21.2$ secret packets. This is smaller than $\overline{T_{16}}$, so it is not unlikely that we will sometimes be unable to decode the secret message before the normal fountain code terminates.

We verified this theoretical analysis via a simulation, shown in figure 8. A secret fountain code of $K_S = 16$ with secret message "`SECRET`" (6 bytes = 48 bit long) was embedded in the headers of degree 4 of a normal fountain code of $K = 64$. When running 10,000 simulations with different random seeds, the secret message was decoded only 4368 times, giving $p_S = 0.4368$.

In this experiment with 10,000 simulations the average number of normal fountain code packets that were required was 82.6959, with a standard deviation of $\sigma = 9.14$. This corresponds very well to the value of $\overline{T_{64}} = 82.7$ from table 3 that we expect to get when

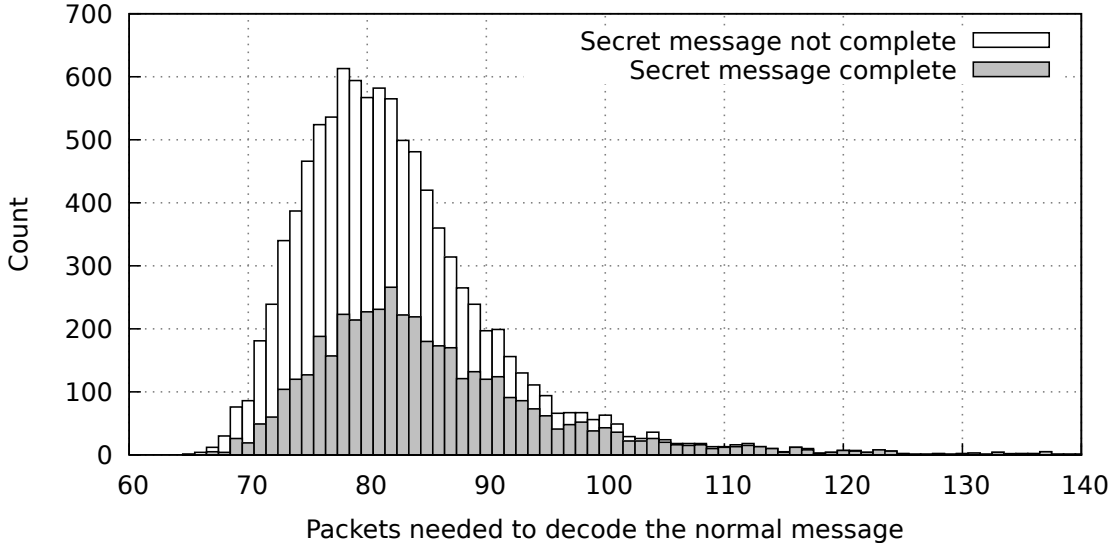


Figure 8: Stacked bar chart showing how often the secret message transmission succeeds, created from 10,000 simulations. In each simulation the secret message is either completely transmitted or not. In 4368 simulations the secret transmission was successful. In the other 5632 simulations the normal fountain code finished before the secret fountain code.

there is no covert channel present, and thus shows that the addition of the covert channel does not impact the normal fountain code (this was achieved by using *integral factors*, see the next sections).

4.1.3 Detectability and efficiency

There are two additional complications of using a (secret) fountain code in the covert channel rather than a direct embedding of the secret message:

- There are more different possibilities for the headers of the normal fountain code than there are different possibilities for the secret packets. This means that some headers of the normal fountain code might never get used.
- The headers of the secret fountain code packets do not follow a uniform distribution. For example in $K_S = 16$ a certain header of degree 1 will happen more often than a certain header of degree 8. The reason is not because $\rho(1)$ may be bigger than $\rho(8)$, but because there are much fewer different headers of degree 1 than there are of degree 8. E.g. there are $\binom{16}{1} = 16$ headers of degree 1, so each header of degree 1 has a chance of $\rho(1)/16 = 1.38\%$ to be selected. However there are $\binom{16}{8} = 12870$ headers of degree 8, so each header of degree 8 has a chance of only $\rho(8)/12870 = 0.001\%$ to be selected.

These complications have two effects. First, it makes the normal fountain code less efficient because not all source block combinations will be used, and second, it could expose the covert channel to detection. The solution is to introduce integral factors. Before we

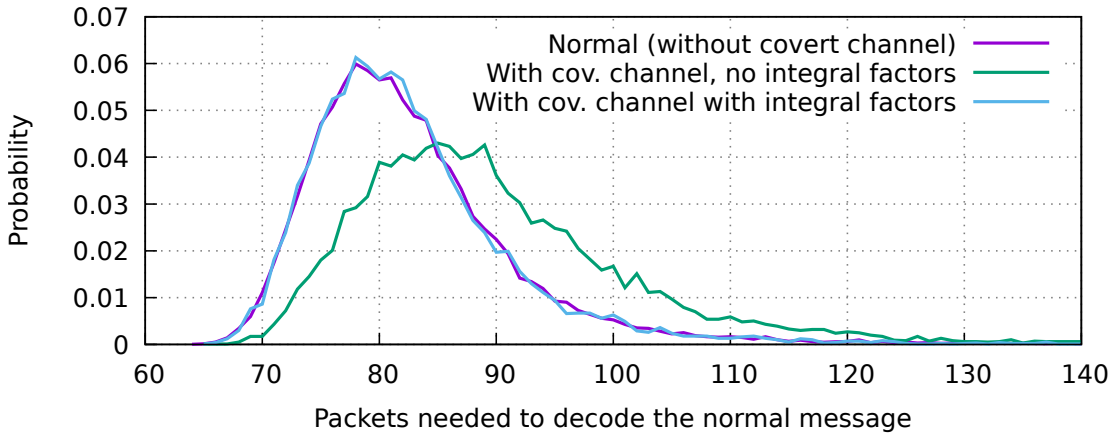


Figure 9: Comparison showing the effect of (not) using integral factors when embedding a covert fountain code inside a normal fountain code. Each probability distribution is created from the histogram of 10,000 fountain code simulations.

describe integral factors in the next section, we can ask whether these issues are really that important to justify using integral factors.

Efficiency When a covert channel is embedded in the normal fountain code at degree $d = 4$ without the use of integral factors, the normal fountain code will need to transmit more packets (on average) to finish. This is shown in figure 9 using a simulation of 10,000 experiments with and without integral factors, for a normal fountain code of $K = 64$ and a secret fountain code of $K_S = 16$. The average number of packets needed to decode the normal fountain code message is normally $\overline{T}_{64} = 82.7$, but when the headers of degree $d = 4$ are modified by a covert channel, without using integral factors, this number increases to $\overline{T}_{64} = 89.9$.

The reason is that there are $\binom{64}{4} = 635376$ different unique combinations of 4 source blocks out of 64 total source blocks. However the secret fountain code only has 14826 different headers, and a payload size of $b_S = 3$, giving each header 8 possible payloads. We use only $14826 \times 8 = 118608$ unique combinations out of the total number of 635376 unique combinations, leaving $635376 - 118608 = 516768$ combinations unused. In addition, the used combinations will be the low ENUM values. These correspond to the first source blocks only. In other words, higher source blocks will be much less likely to occur in the normal fountain code packets of degree 4, which means that the receiver will need to wait longer for those particular blocks.

Interestingly the probability p_S of successfully transmitting the secret message is higher when no integral factors are used. It is now $p_S = 0.5566$ instead of 0.4368. The reason is simply that there are on average more normal fountain code packets available, and thus also more covert packets, because the normal fountain code is less efficient. We could in theory design a normal fountain code that is very inefficient, to make the probability p_S arbitrarily close to 1. However a very inefficient normal fountain code is not desirable for

the normal sender and receiver. In addition, a normal fountain code that operates below the expected optimal efficiency level of $\overline{T} = 82.7$ packets could look suspicious, although many observations of different fountain code messages would be required to notice this inefficient behaviour. However an observer has more ways to inspect the normal fountain code than just counting the total number of sent or received packets. In the next paragraph we show that observing a single fountain code is already enough to detect a covert channel that does not use integral factors.

Detectability Keeping with the example from the previous section where $K = 64$, the header at $d^* = 4$ has $\binom{64}{4} = 635376$ unique combinations of 4 source blocks. We will call a unique combination a *symbol*. A symbol is written in binary using one of the header representations of section 3.1. It does not matter which header representation is chosen, as we can simply continue thinking in symbols. In binary, we need $\log_2 655376 = 19.28$ bit to write down all unique symbols. So the header will use 20 bit as we can not write fractional bits, but not all 2^{20} different values of 20 bit will be used.

Each of the 635376 symbols can represent a unique encoded secret packet, where a secret packet is the combination of a secret header and a secret payload. Supposing that the payload size $b_S = 3$, then every unique secret header can have $2^3 = 8$ different payloads, depending on the content of the secret message. Every secret header thus requires space for 8 payloads, so it follows that there are $635376/8 = 79422$ spaces for the headers. We call these spaces the *header symbols*. An observer expects that all 635376 symbols, and thus all 79422 header symbols, will occur with equal probability.

In our example with $K_S = 16$ there can be $16 + 120 + 1820 + 12870 = 14826$ different secret headers, and there are 79422 header symbols available to use. If we map the 14826 secret headers onto the first 14826 header symbols in a direct one-to-one mapping, then the sender will only send normal fountain code packets (of degree 4) that have one of those 14826 headers. The remaining 64596 header symbols of degree 4 will never occur in the normal fountain code. This is certainly suspicious if the observer can see a very large number of transmitted packets, however in our example we expect to send only $\overline{T_S} = 21.2$ secret packets. So what is the likelihood that an observer will actually notice the fact that a block of header symbols are never used if only about 20 samples are given?

To estimate this, we note that the observer can collect the received header symbols and can expect these to be sampled from a uniform distribution. The observer can perform a *goodness of fit* calculation to detect whether the actual, observed distribution in fact follows a uniform distribution. A common way of testing goodness of fit is Pearson's χ^2 test, however this is not a good choice in this case because of the low number of samples. A more appropriate goodness of fit test is the Kolmogorov-Smirnov test. Alternatives are the Cramér-von Mises criterion and the Anderson-Darling test, but we do not explore these further.

The Kolmogorov-Smirnov test compares the cumulative distribution function (CDF) of the

α	0.001	0.01	0.02	0.05	0.1
$D_{20,\alpha}$	0.42085	0.35240	0.32866	0.29407	0.26473

Table 4: Critical values of $D_{n,\alpha}$ for $n = 20$, copied from [17].

observed distribution to the CDF of the expected distribution. When the two CDF are very different, then it is likely that the observations do not follow the expected distribution. To perform the Kolmogorov-Smirnov test we first find the value of the largest difference between both CDF. Writing the expected CDF as $F(x)$ and the observed CDF as $F_n(x)$, where n denotes the number of samples and x is the random variable, then the Kolmogorov-Smirnov statistic D_n is

$$D_n = \max_x |F_n(x) - F(x)| \tag{17}$$

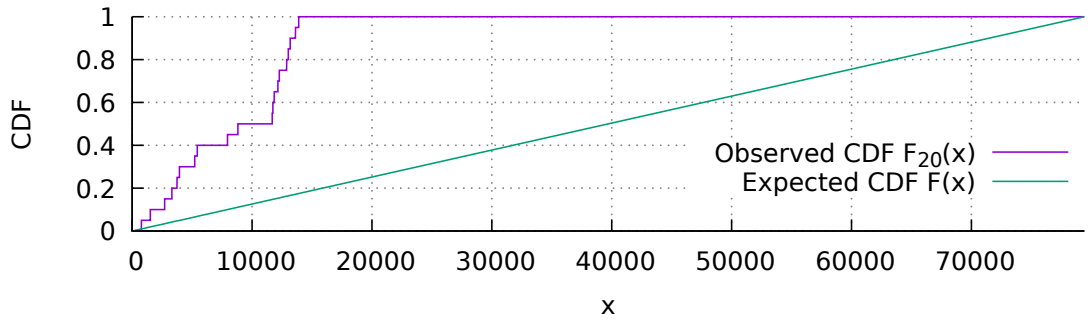
The value of D_n must then be compared to the Kolmogorov distribution to see if its value is not larger than what we expect it to be. This is easiest by using tabulated critical values $D_{n,\alpha}$ for a chosen critical value α. The relation between D_n and $D_{n,\alpha}$ is that if the observed distribution is in fact sampled from the expected distribution, then $P(D_n \leq D_{n,\alpha}) = 1-\alpha$, see [16]. In other words, if $D_n > D_{n,\alpha}$ then we can be quite certain (with the "certainness" depending on the choice of α) that the observed distribution does *not* follow the expected distribution.

Table 4 shows an extract from the table of [17] for various critical values $D_{20,\alpha}$. We copy only the $n = 20$ values because we will run tests only for $n = 20$. If a simulation results in a D_{20} value that is higher than the listed $D_{20,\alpha}$ value for a certain choice of α, then Kolmogorov-Smirnov rejects the hypothesis that the observed distribution matches the expected distribution.

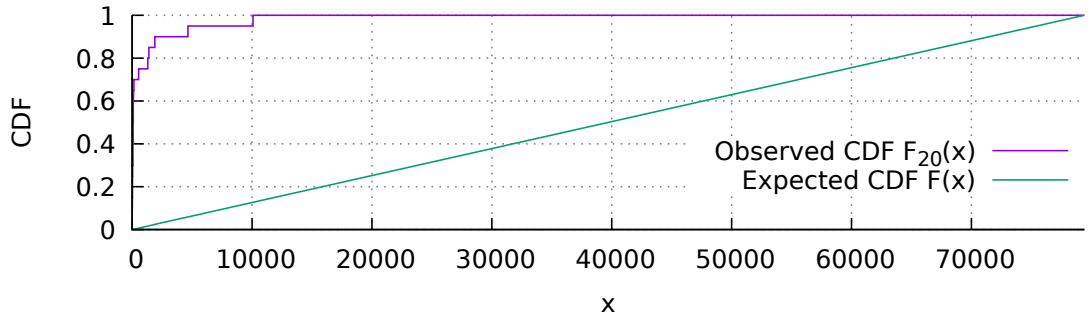
We run simulations where we take $n = 20$ samples in three different ways: uniformly, via degree distribution, and using integral factors (introduced in the next section). We plot the resulting CDF for each of the three tests in figure 10. The first 2 plots, (a) and (b), have a value of D_{20} that is much larger than $D_{20,\alpha}$ for even the highest confidence level α from table 4. In (a) we have $0.825 > 0.42085$ and in (b) we have $0.891 > 0.42085$. An observer will conclude with a very high level of certainty that in (a) and (b) the fountain code does not behave as expected. Only in test (c) will the observer think that everything is normal. Of course this experiment is only a single fountain code simulation, but repeating the test with different random seeds gives similar results. This shows that integral factors are necessary when embedding a covert channel, even when only a single fountain code message is transmitted.

4.1.4 Integral factors

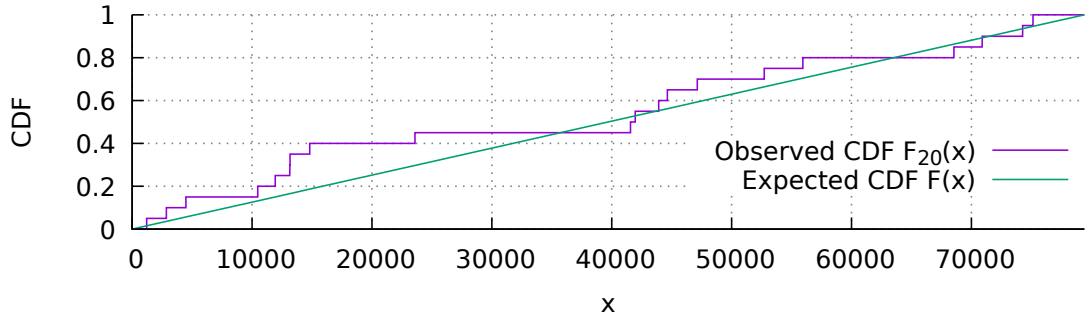
As shown in the previous section, integral factors are a necessary addition to the secret fountain code to prevent an observer from easily detecting the covert channel. We will



(a) Taking uniform samples between 0 and 14826 gives $D_{20} = 0.825$.



(b) Using the degree distribution $\rho(d)$ from table 5 gives $D_{20} = 0.891$.



(c) Applying integral factors and degree distribution $\rho^*(d)$ from table 5 gives $D_{20} = 0.214$

Figure 10: Simulations where 20 secret packets are transmitted when there are 79422 header symbols available, for three different sampling approaches. In (a) and (b) the observed CDF looks very different from the expected CDF, and the Kolmogorov-Smirnov test will reject the hypothesis that the 20 samples were selected from a uniform distribution between 0 and 79422. An observer will conclude that the fountain code behaves strangely. In (c) the Kolmogorov-Smirnov test will not reject this hypothesis, and an observer will not suspect a covert channel.

describe the integral factors by considering the same example as before in section 4.1.2.

Each symbol is equally likely to occur in a normal random fountain code, i.e. without a covert channel. So we should make certain that after creating a covert channel, each symbol still occurs with probability of e.g. 1 / 635376.

d	$\rho(d)$	#headers $\binom{16}{d}$	header probability	f_d	$\rho^*(d)$
1	0.221	16	0.0138125	1059	0.213
2	0.457	120	0.0038083	292	0.441
4	0.188	1820	0.0001033	8	0.183
8	0.134	12870	0.0000104	1	0.163

Table 5: Integral factors to make each header symbol equally likely, for embedding a $K_S = 16$ covert channel fountain code into a $K = 64$ overt fountain code in degree $d^* = 4$, with the amount of header symbols $H = 79422$.

In the covert fountain code of size K_S there are $\binom{K_S}{d}$ different headers for a certain degree d. Each header has a payload of b_S bit added to it, to form an encoded packet. In our example with $K_S = 16$ and $b_S = 3$, there are $(16 + 120 + 1820 + 12870) \times 2^3 = 14826 \times 8$ different encoded packets.

The set of all encoded secret packets should be mapped to all available symbols of the normal fountain code. We typically have more symbols available than there are secret packets. So each secret packet should be mapped to one or more symbols, but each symbol corresponds to only one secret packet.

In general, with $\mapsto$ meaning mapping, the number of symbols mapped is

$$2^{b_S} \sum_d \binom{K_S}{d} \mapsto \binom{K}{d^*}. \tag{18}$$

We know that all payloads are equally likely to occur (from solving problem 1), so we can divide both sides of (18) by 2^{b_S}, the number of different payloads per header. This simplifies the formula, as now the right hand side of (18) takes the meaning of number of available *header symbols* as we introduced earlier. We can write this value as H, so

$$H = \frac{1}{2^{b_S}} \binom{K}{d^*} \tag{19}$$

and the mapping is then more simply written as

$$\sum_d \binom{K_S}{d} \mapsto H. \tag{20}$$

In our example, we need to map 14826 secret headers onto 79422 header symbols.

As mentioned before, the secret headers do not all occur with the same probability. A single header of a degree with probability $\rho(d)$ happens with a probability of $\rho(d)/\binom{K_S}{d}$. Table 5 shows values for $K_S = 16$.

We introduce integral factors f_d for each degree, which has the effect of mapping each single header onto f_d different symbols with equal probability. This means that for a

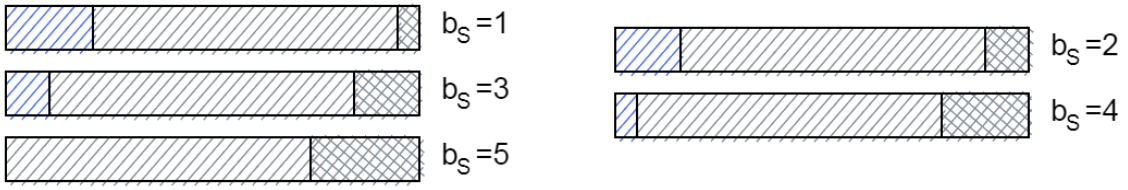


Figure 11: The effect of the choice of payload size b_S on the space available for the integral factors in the header of $d^* = 4$ using $K = 64$.

given header each of the f_d mapped symbol has a chance of $1/f_d$ to be chosen. We want each symbol choice to be equally likely across all degrees, so

$$\frac{\rho(d)}{\binom{K_S}{d}} \frac{1}{f_d} = \frac{1}{H} \tag{21}$$

Additional constraints on f_d are that only integer values are allowed, $f_d \in \mathbb{N}$ and $f_d \geq 1$. We also should not map onto more header symbols than we have available, i.e. in our example $16f_1 + 120f_2 + 1820f_4 + 12870f_8 \leq 79422$. From these considerations we can compute the values for f_d. In section 5.3.4 we present an algorithm to compute f_d.

Due to the constraints on f_d the target probability of 1 / 79422 can not be matched exactly. We can however modify the secret degree distribution $\rho(d)$ slightly as well by solving equation (21) for $\rho(d)$, giving the modified secret degree distribution $\rho^*(d)$ as

$$\rho^*(d) = \frac{\binom{K_S}{d} f_d}{H} \tag{22}$$

With $\rho^*(d)$ we achieve a near-perfect probability matching. The change from $\rho(d)$ to $\rho^*(d)$ has a small impact on the performance of the secret fountain code, but ensures that all symbols occur with equal probability. Table 5 gives the resulting f_d for our example, as well as the $\rho^*(d)$ to achieve the target probability exactly.

When the integral factors alone do not provide a good match to the target probabilities then $\rho^*(d)$ might be very different from the optimal $\rho(d)$. A non-optimal secret degree distribution results in a higher average number $\overline{T_S}$ of secret packets that need to be transmitted before the covert receiver can decode the secret message. In other words, p_S will be lower.

The choice of $b_S = 3$ bit in our example (taken from [6]) was made as a compromise between good integral factors and a reasonable size S of the secret message. Other choices for b_S are possible. Figure 11 shows the impact of this choice visually.

Decreasing b_S to 2 bit gives 158844 available header symbols for the mapping, but decreases the secret message size to just $2 \times 16 = 32$ bit. Increasing b_S to 4 bit increases the secret message size to 64 bit, but gives only 39711 header symbols. In table 6 we show the integral factors for the cases of b_S from 1 bit to 5 bit as well as the modified $\rho^*(d)$. The f_d values were found using the algorithm of section 5.3.4.

		$b_S = 5$ $H = 19855$		$b_S = 4$ $H = 39711$		$b_S = 3$ $H = 79422$		$b_S = 2$ $H = 158844$		$b_S = 1$ $H = 317688$	
d	$\rho(d)$	f_d	$\rho^*(d)$	f_d	$\rho^*(d)$	f_d	$\rho^*(d)$	f_d	$\rho^*(d)$	f_d	$\rho^*(d)$
1	0.221	105	0.085	436	0.176	1059	0.213	2119	0.213	4463	0.225
2	0.457	29	0.175	120	0.363	292	0.441	584	0.441	1230	0.465
4	0.188	1	0.092	3	0.137	8	0.183	16	0.183	33	0.189
8	0.134	1	0.648	1	0.324	1	0.163	2	0.163	3	0.121

Table 6: An expanded version of the integral factors f_d from table 5 for different payload sizes b_S and number of header symbols H. The integral factors f_d map 14826 headers onto H header symbols. Notice how $\rho^*(d)$ approaches the original $\rho(d)$ for higher values of H.

Figure 12 shows the impact of the non-optimal $\rho^*(d)$ distributions on the number of fountain code packets that need to be transmitted before the message can be decoded. The choice of $b_S = 3$ gives almost the same shape as $b_S = 1$ or as the optimal distribution. On the other hand, $b_S = 4$ already is slightly worse, but could perhaps still give acceptable outcomes and has a higher secret message size S. However the additional packets that are required to successfully decode the secret message will have an impact on the success rate of any secret fountain code that uses this distribution. The normal fountain code, which uses an optimal distribution, will not send more packets just to accomodate the secret fountain code. From this observation it is clear that $b_S = 5$ is not a good choice. The distribution $\rho^*(d)$ is now far from optimal, and the large number of extra packets will make it unlikely that enough secret fountain code packets will be transmitted before the normal fountain code finishes.

b_S	1	2	3	4	5
p_S	0.4551	0.4371	0.4368	0.3227	0.0177
$\overline{T}$	82.96	82.87	82.69	82.94	83.00

Table 7: Success rates for various choices of b_S in the header of $d^* = 4$ using $K = 64$. The presence of the covert channel does not impact the normal fountain code, as $\overline{T}$ is always close to the expected value of 82.7.

We experimentally determine the values for p_S by running 10,000 simulations similar to the one from figure 8 for each choice of b_S. The results are shown in table 7, and confirm what we anticipated from looking only at the required number of packets in figure 12. A secret payload size of $b_S = 5$ has a success rate of only 1.77% and should thus not be used, whereas values of $b_S = 3$ and lower are still reasonable. We also show the success rate of the secret message in histogram-form in figure 13.

4.1.5 Packet loss in the channel

In all of the above discussion we have assumed that the channel over which the packets are transmitted has a 100% success rate, i.e. all packets always arrive. In practice this is

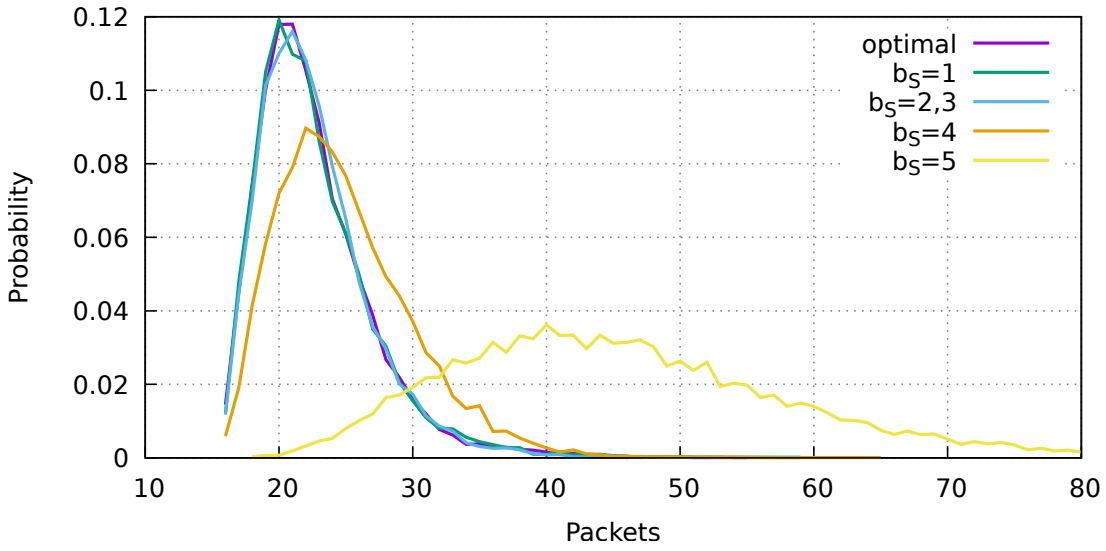


Figure 12: The number of packets required to decode the fountain code message of $K = 16$, for the distributions $\rho^*(d)$ given in table 6. A total of 10,000 experiments were run for each b_S value. Each experiment gave a single packet count. These counts were then collected in a histogram, after which the histogram values were divided by 10,000 to give the probability that the number of packets on the x-axis are needed to decode the fountain code message.

usually not the case. In fact, one of the main reasons to use fountain codes is to deal with channels that have an unpredictable loss rate. The question now is whether the covert channel will be impacted by this unpredictable packet loss.

We have tested this by simulating a channel with packet loss. Each time the sender creates a fountain code packet and attempts to transmit it over the channel, there is a chance that the packet will be lost and thus the receiver will not receive it. The fountain code packets contain secret fountain code packets in the covert channel at $d^* = 4$. We test loss chances of 0%, 10%, 20% and 90%.

The results show that the covert channel is not impacted. The normal fountain code requires additional packets to be sent, but the normal receiver still needs to receive on average 82.7 packets in order to decode the normal fountain code message. In other words, the normal receiver will continue to wait for packets until it can decode the message. Those 82.7 packets (on average) contain the covert channel with the information about the secret message. Because the covert channel has a secret fountain code in it, the same principle applies there. It does not matter that some packets are dropped, as long as eventually enough packets arrive at the receiver and the covert receiver.

Because this test shows that it does not matter what the packet loss rate is from the point of view of the covert channel, we will leave the packet loss rate at 0% in the rest of this work. This speeds up the simulations, as generating fountain code packets and then

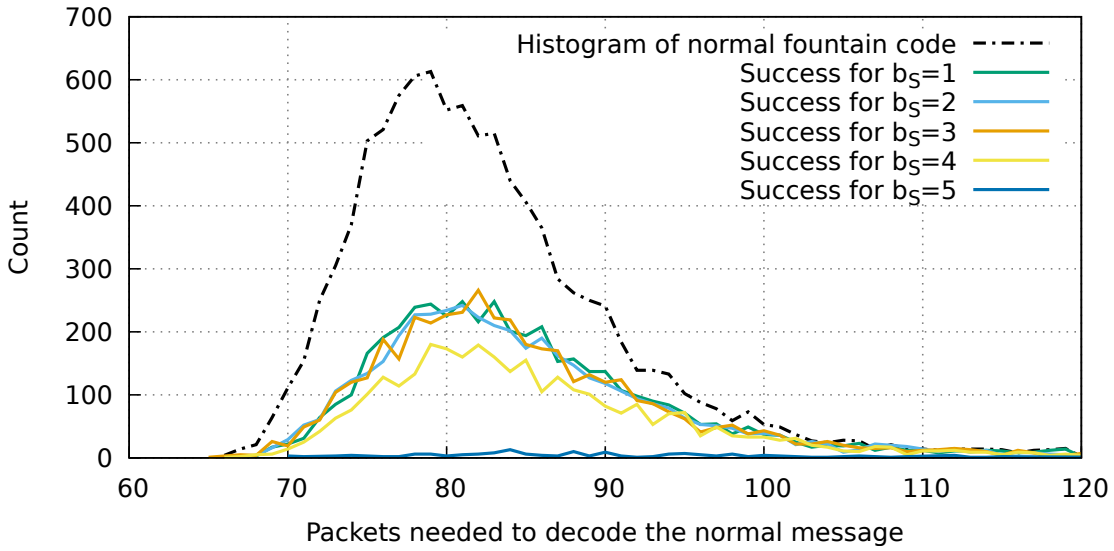


Figure 13: Histogram of how often the secret message was successfully decoded for different choices of b_S, when embedding a secret fountain code of $K_S = 16$ into degree $d^* = 4$ of a normal fountain code of $K = 64$.

simply discarding them wastes some computation time.

4.2 Covert Channel in multiple degrees

The approach presented in [6] to embed a covert channel in a fountain code has a success rate p_S of 43.68% for a secret message size S of 48 bit. We wish to increase both the size S of the secret message and the probability p_S that the secret transmission is successful. In this section we will investigate different modifications of the original approach that was presented in [6].

4.2.1 Using degrees 4, 8, 16 and 32

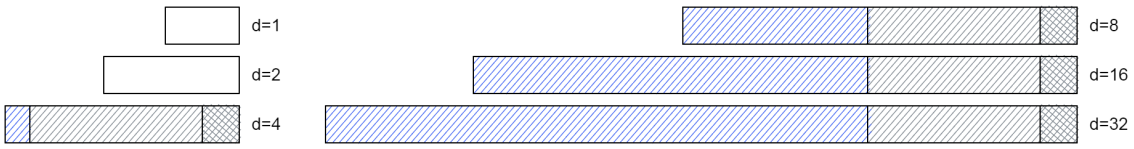


Figure 14: Using degrees $d^* = \{4, 8, 16, 32\}$ for embedding the secret fountain code.

As a first step we try to use more degrees than just $d^* = 4$ to embed packets of the secret fountain code. Table 8 shows the number of available bits in the normal fountain code header for different degrees. Packets of the secret fountain code with $K_S = 16$ and $b_S = 3$ take up 16.86 bit (not counting integral factors), so they fit well within normal fountain code headers of degree 4, 8, 16 and 32. This means that we can also use the degrees above 4 to embed a covert channel. Only the headers of degrees 1 and 2 are not large enough to contain an encoded packet of 16.86 bit. We show this visually in figure 14.

d	$\rho(d)$	ENUM bits	fits covert packet of 16.86 bit
1	0.161	6.00	no
2	0.4	10.98	no
4	0.256	19.28	yes
8	0.101	32.04	yes
16	0.045	48.80	yes
32	0.037	60.67	yes

Table 8: Degree distribution for $K = 64$ and the number of bits in the ENUM header, calculated using $\log_2 \binom{K}{d}$. All headers except degrees 1 and 2 have enough space to store a covert packet of 16.86 bit.

The difference between 16.86 bit and the actual number of bit available in the header is filled up by the integral factors. As a reminder, we need to use all the available header symbols. We can not simply round down to the nearest whole number of bits. For example, degree 8 has $\binom{64}{8} = 4426165368$ different combinations of source blocks, and the covert channel should be able to generate all of them. Rounding down to 32 bit instead of 32.04 bit would give only 4294967296 different combinations, so that 131198072 combinations would never be used, degrading the performance of the normal fountain code.

We have already computed the integral factors for degree 4. To compute the integral factors for the higher degrees we again use the algorithm from section 5.3.4. The resulting values are shown in table 9.

d	f_d for degree 8	f_d for degree 16	f_d for degree 32
1	7641972	843472289867	3164140118346202
2	2107017	232559177356	872405450427883
4	57150	6307902759	23663004017666
8	5761	635806231	2385116888950

Table 9: Integral factors for $b_S = 3$ into degrees 8, 16 and 32. The numbers get very large.

The integral factors are much higher than before, because the degrees of 8 and above have a lot more header symbols available. In the Java code we use the `BigInteger` type to store these values exactly, as the values are too large for an `int` or a `long`, and a `float` or `double` would lead to a loss of precision. These integral factors lead to a very close approximation of the optimal degree distribution for $K_S = 16$. So instead of having to use a modified secret degree distribution $\rho^*(d)$ as for $d^* = 4$ from table 6, we can now simply use the optimal degree distribution $\rho(d)$ for a fountain code with $K_S = 16$ from table 3. In other words, for $d^* = \{8, 16, 32\}$ we have $\rho^*(d) = \rho(d)$.

Using multiple degrees to store secret packets increases the factor α from equation (13), giving

$$\alpha = 0.256 + 0.101 + 0.045 + 0.037 = 0.439$$

This α is almost two times higher than only using degree $d^* = 4$. Using the values for $\overline{T}$ from equation (16) we compute the expected number of transmitted covert packets as $\overline{T_S} = 0.439 \times 82.7 = 36.3$, which is higher than the average required amount of $\overline{T_{16}} = 22.6$ to decode the secret message.

Adjusting the Java code to incorporate these changes, and repeating the experiment with 10,000 simulations, gives a success rate of $p_S = 97.47\%$ to decode the secret message. The number of required packets for the normal fountain code is on average $T = 82.7$ during the simulations, showing that using additional degrees for the covert channel in this way does not impact the normal fountain code.

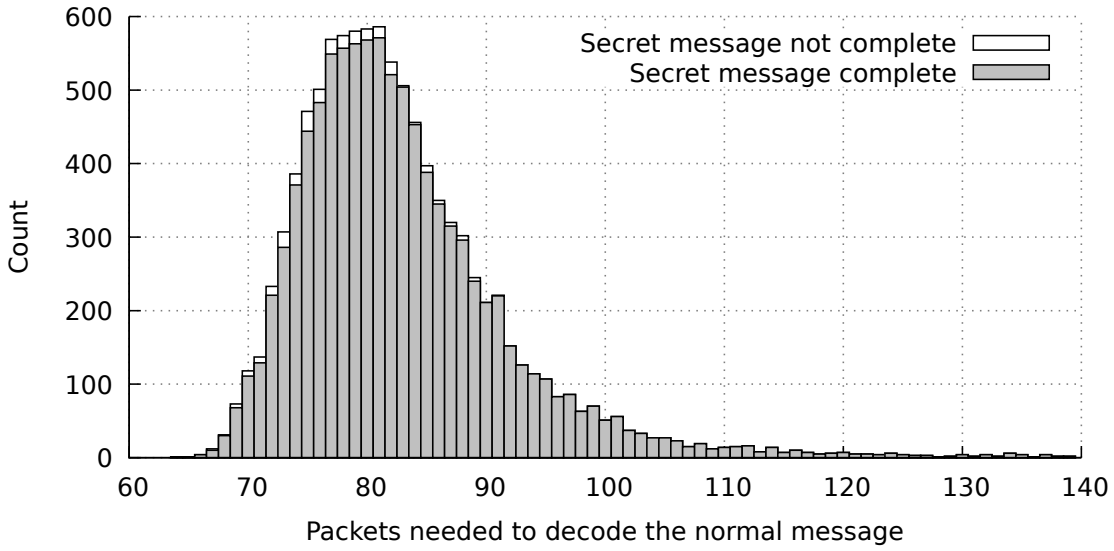


Figure 15: Stacked bar chart showing when the secret message transmission succeeded when using degrees 4, 8, 16 and 32 for the covert channel. Out of 10,000 simulations there were 9,747 successes.

Figure 15 is similar to figure 8 from before and shows where the secret messages succeed and where they fail. The figure shows that the covert channel now has a high chance of success, which is a positive improvement. However the size S did not change, i.e. we are still limited to a secret message of only 48 bit.

Additional packets in degrees 16 and 32 The headers of degrees 16 and 32 have enough bit available to store more than one covert packet. In the header of degree 16, with 48.80 bit space, we could store two covert packets of 16.86 bit and still have room for integral factors. The header of degree 32 could even store three covert packets. We can compute the value of α as

$$\alpha = 1 \times 0.256 + 1 \times 0.101 + 2 \times 0.045 + 3 \times 0.037 = 0.558$$

This further increase in α gives an expected number of transmitted covert packets of $\overline{T_S} = 0.558 \times 82.7 = 46.1$, which means that in this case p_S will be very close to 1.

We do not simulate this idea here, as p_S was already close to 1 and adding additional packets does not increase the secret message size S, because the secret payload size b_S stays the same. We will however come back to this idea later in this paper.

4.2.2 Using only degrees 8, 16 and 32

Using all the degrees of 4 and higher improves p_S substantially, but it does not increase the secret size S. Since equation (14) tells us that $S = b_S K_S$, we can either increase the number of secret source blocks K_S or increase the number of bits b_S of the secret payload in order to increase S. In the section on integral factors, 4.1.4, we found that $b_S = 3$ bit is the best we can do while keeping reasonable integral factors. Choosing $K_S = 32$ is also not an option, as then the header of the secret fountain code becomes too large: $\sum_{d=\{1,2,4,8,16\}} \log_2 \binom{32}{d} = 29.2$ bit, which does not fit in the header of $d^* = 4$. So it seems that neither increasing b_S nor increasing K_S is possible for $d^* = 4$.

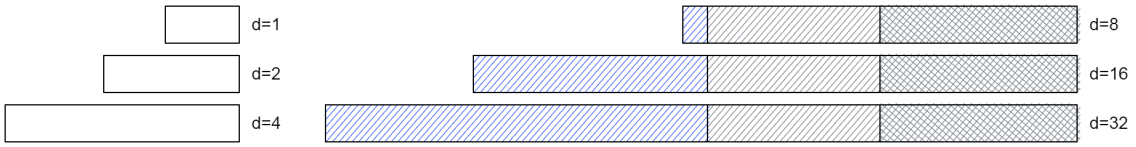


Figure 16: Using only degrees $d^* = \{8, 16, 32\}$ for embedding the secret fountain code.

If we choose $d^* = \{8, 16, 32\}$ however, then we have 32.04 bit available in degree 8 (see table 8). For a $K_S = 16$ secret fountain code we can now choose $b_S = 16$ and still have enough header symbols available for the secret fountain code headers. This gives a secret message size of $S = 16 \times 16 = 256$ bit, which is more than the original $S = 48$ bit. Figure 16 shows the effect.

The downside is that α decreases. Our secret packets are now about 32 bit long so we can not put multiple packets in the higher degree numbers, giving $\alpha = 1 \times 0.101 + 1 \times 0.045 + 1 \times 0.037 = 0.183$.

d	f_d for degree 8	$\rho^*(d)$ for degree 8	f_d for degree 16	f_d for degree 32
1	850	0.201	102962928	386247573044
2	236	0.419	28388573	106494805960
4	7	0.189	770008	2888550295
8	1	0.191	77613	291151964

Table 10: Integral factors for $d^* = \{8, 16, 32\}$ with $b_S = 16$. We have omitted the $\rho^*(d)$ for degrees 16 and 32, as it is identical to the ideal distribution $\rho(d)$ for $K_S = 16$.

The lower α has a negative impact on p_S. To run simulations we again need to calculate appropriate integral factors. They are shown in table 10 for $b_S = 16$. Figure 17 shows the results. The probability of success is very low, at $p_S = 9.65\%$.

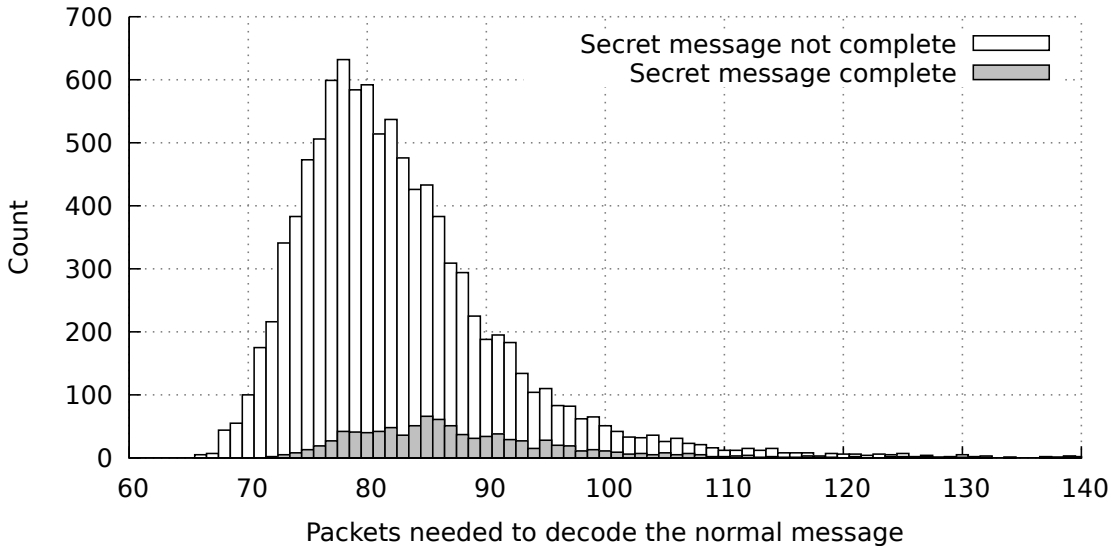


Figure 17: Stacked bar chart showing when the secret message transmission succeeded when using degrees 8, 16 and 32 for the covert channel, with a payload size of $b_S = 16$. Out of 10,000 simulations only 965 succeeded.

4.2.3 Degrees 4, 8, 16 and 32, with $K_S = 8$

The modifications of the previous two sections increase either S or p_S, but not both. However they give confidence that something can be done as a middle ground. A solution is counter-intuitively to decrease K_S to 8, and keeping $d^* = \{4, 8, 16, 32\}$. When $K_S = 8$ we use only secret degrees 1, 2 and 4. The size of the secret header is smaller, and thus there is more space available for the secret payload. For $K_S = 8$ there are $\binom{8}{1} + \binom{8}{2} + \binom{8}{4} = 106$ different headers, requiring $\log_2(106) = 6.7$ bit in ENUMALL mode. For $d^* = 4$ there are 19.28 bit in the normal fountain code header, so if we reserve between 7 to 10 bit for the header (to account for integral factors) then this leaves between 12 to 9 bit for the payload. The secret message size is then either $S = 96$, $S = 88$, $S = 80$ or $S = 72$ bit. The exact choice depends on the p_S results from the simulation, as smaller headers lead to worse integral factors and thus a worse performance of the secret fountain code. Figure 18 shows this idea.

Unfortunately neither [12] nor [5] provide optimised sparse degree distributions for $K = 8$, so we do not know yet which degree probabilities to use in the secret fountain code. However a dense distribution for $K = 8$ was presented in [4] in Table III. We can use this dense distribution to derive two candidate sparse distributions as follows:

1. Either discard the dense distribution values for degrees not equal to 1, 2 or 4 and then renormalise the remaining values.
2. Or compute the sparse degree probability values by summing the given dense distribution values in the groups (1), $(2, 3)$ and $(4, 5, 6, 7, 8)$.

The resulting distributions for these methods are shown in table 11, together with the op-

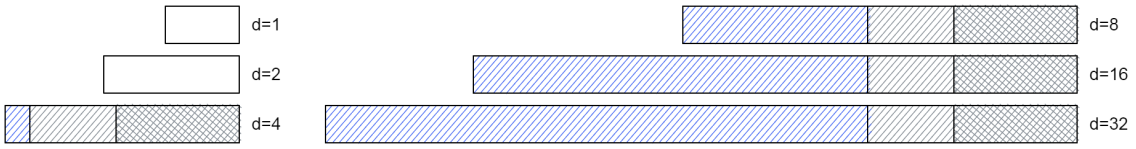


Figure 18: Using degrees $d^* = \{4, 8, 16, 32\}$ with $K_S = 8$ gives smaller headers and thus more space for the payload. In this figure we use $b_S = 10$.

	[4]	method 1	method 2
$\rho(1)$	0.268	0.312	0.268
$\rho(2)$	0.491	0.572	0.576
$\rho(3)$	0.085	-	-
$\rho(4)$	0.099	0.116	0.156
$\rho(5)$	0.013	-	-
$\rho(6)$	0.027	-	-
$\rho(7)$	0.010	-	-
$\rho(8)$	0.007	-	-
$\overline{T}$	11.565	12.157	11.865
σ	-	3.599	3.335

Table 11: Sparse degree distributions for $K = 8$, derived from the optimised dense distribution in [4], Table III, with two different methods.

timal dense distribution of [4]. We determine the overhead for the new sparse distributions experimentally by running 50,000 fountain codes (without covert channel) and computing the average and standard deviation of the required number of fountain code packets to decode the message. These results are also shown in table 11. We see that the difference in the average $\overline{T}$ between the optimal dense distribution of [4], and our not-optimised sparse distribution of the second method is small. We conclude that our probability values for the sparse degree distribution of method 2 are close to the optimum, so we can use these for the secret fountain code with $K_S = 8$.

			$b_S = 9$ $H = 1240$		$b_S = 10$ $H = 620$		$b_S = 11$ $H = 310$		$b_S = 12$ $H = 155$	
d	$\binom{8}{d}$	$\rho(d)$	f_d	$\rho^*(d)$	f_d	$\rho^*(d)$	f_d	$\rho^*(d)$	f_d	$\rho^*(d)$
1	8	0.268	41	0.264	23	0.297	9	0.232	3	0.155
2	28	0.576	25	0.564	13	0.587	6	0.542	2	0.361
4	70	0.156	3	0.172	1	0.116	1	0.226	1	0.484

Table 12: Integral factors f_d and modified $\rho^*(d)$ for different payload sizes b_S, for a secret fountain code with $K_S = 8$ into degree $d^* = 4$ of a normal fountain code with $K = 64$.

In order to simulate this secret fountain code and decide which b_S value is best, we also need integral factors. Table 12 shows our computation for b_S ranging from 9 to 12, similar

to table 6 from before. This table only shows the integral factors for embedding the secret fountain code packets into normal fountain code degree $d^* = 4$. We also use normal fountain code degrees 8, 16 and 32, which also need integral factors for each choice of b_S. We have listed these in appendix A.

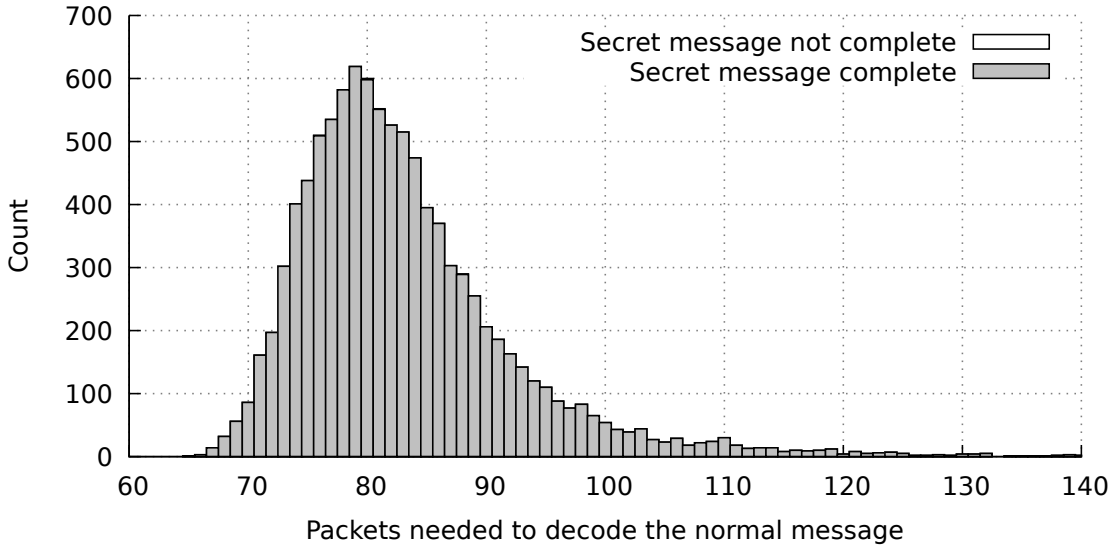


Figure 19: Stacked bar chart showing when the secret message transmission succeeded for the method with $K_S = 8$ and $b_S = 9$. In 10,000 simulations there were 9,995 successes.

Simulations give a very high success rate of $p_S = 99.95\%$ for the secret message transmission. Figure 19 shows the histogram of successes for $b_S = 9$. The histograms for higher values of b_S look almost identical. Even for $b_S = 12$, which has a $\rho^*(d)$ distribution that is very different from the optimal one, the success rate is still $p_S = 99.95\%$.

An explanation for this high success rate is that the number of transmitted secret packets is on average $\overline{T_S} = \alpha\overline{T_{64}} = 36.3$, while only 11.86 secret packets are required on average (see table 11) to decode the secret message when using $\rho(d)$ values that are close to optimal. Still, even in this very favourable setup it happened 5 times in 10,000 simulations that the secret transmission failed.

The simulations show that both S and p_S are higher when using $K_S = 8$ compared to using $K_S = 16$. One idea could be to go even further, to $K_S = 4$, in the hope of further improvements. However we can reject this quickly by noting that the value of b_S will not grow very much, while K_S halves. The reason that we cannot at least double b_S is that the total number of available bits in the normal fountain code at $d^* = 4$ remains 19.28, which we need to divide between the secret header and the secret payload. So even if we could make $b_S = 15$, this still only gives $S = 15 \times 4 = 60$ bit which is worse than what we found for $K_S = 8$.

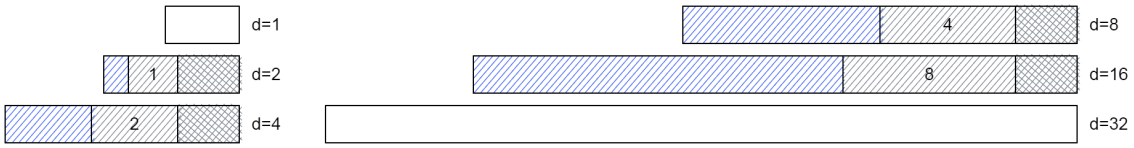
A final remark about this method is that we still can not use the headers of degree 2. These happen often since $\rho(2) = 0.4$ and have about 10 bit of space in the header. Attempting

to put a covert fountain code packet of $K_S = 8$ into the header of $d^* = 2$ leaves only 1 or 2 bit for the payload, so leads to a very small S.

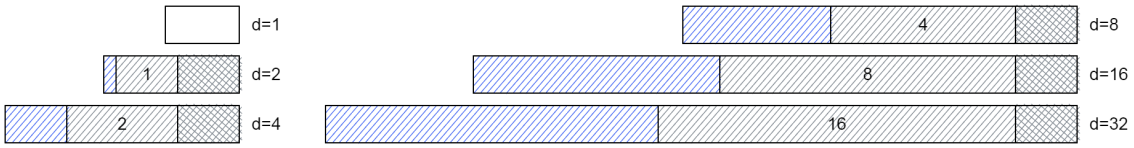
4.3 Half-Degree covert channel

4.3.1 Basic approach

In the methods that we have discussed so far, the degree of the secret packet is chosen randomly according to a secret degree distribution. Any secret degree can be used in each normal header. The secret headers must encode this secret degree choice, and must also be able to encode the secret source block indices for the highest possible secret degree. So even for a secret packet of degree 1 we still need to reserve a lot of bits. By removing the flexibility to store any secret degree in each overt degree, we could spare more bits for the secret payload and potentially get better results.



(a) $K_S = 16$, $b_S = 5$



(b) $K_S = 32$, $b_S = 5$

Figure 20: The Half-Degree method for two choices of K_S. The number written inside the secret header is the (fixed) secret degree that the covert sender and covert receiver agree on in advance. The secret header size is no longer always the same, but now depends on the value of the secret degree. The overt degree 32 in $K_S = 16$ does not have a covert channel, because secret degree 16 does not occur when $K_S = 16$.

We propose a method which we call Half-Degree, where each overt degree starting from 2 contains secret fountain code packets from a degree that is half the overt degree. In other words, if the normal fountain code creates a packet of degree 2, then we will embed a secret packet of degree 1. An overt packet of degree 4 gets a secret packet of degree 2, and so on. This is illustrated in figure 20, for both $K_S = 16$ and $K_S = 32$. We will only consider those two K_S values. Higher K_S have a low chance of success p_S, and lower K_S have a smaller secret message size.

Removing the choice of secret degree does not remove the need for integral factors, as figure 20 clearly shows. The combination of secret payload and secret header does not fill the available space. However the integral factors are now much simpler to determine and to implement. For each overt degree there are $\binom{K}{d}$ spaces, and the Half-Degree covert channel can create $\binom{K_s}{d/2}$ different combinations, each of which can have 2^{b_S} different secret

d	$\rho(d)$	secret d	$\rho^*(d)$ for $K_S = 16$	$\rho^*(d)$ for $K_S = 32$
1	0.161	-	-	-
2	0.4	1	0.499	0.477
4	0.256	2	0.319	0.305
8	0.101	4	0.126	0.120
16	0.045	8	0.056	0.054
32	0.037	16	-	0.044

Table 13: Actual degree distribution for Half-Degree secret fountain code for various K_S. The left columns show the distribution values for the normal fountain code. The $\rho^*(d)$ are the actual distribution of the degrees of the secret fountain code, as received by the covert receiver, and are calculated by renormalising the $\rho(d)$ values.

payloads. So the integral factor that maps all the possible secret packets onto the available normal header spaces is

$$f_d = \binom{K}{d} \Big/ 2^{b_S} \binom{K_s}{d/2}. \tag{23}$$

Using the Half-Degree method for a secret payload size of $b_S = 5$, as shown in figure 20, gives a secret message size of $S = 80$ bit when $K_S = 16$, and $S = 160$ bit when $K_S = 32$. We could use a higher b_S value by reducing the space for the integral factors, but as we will see in the simulations this has a negative effect on the normal fountain code. Even $b_S = 5$ already has a negative effect.

Before we look at the experimental results, we can discuss what we expect from a theoretical point of view. As we use all overt degrees except for $d = 1$, the factor α is much higher now. For $K_S = 16$ we have $\alpha = 0.802$ and for $K_S = 32$ we have $\alpha = 0.839$. However the secret degree distribution is far from optimal, because it depends on the degree distribution of the normal fountain code. We can no longer choose the secret degree probabilities to minimise the number of required secret packets. Table 13 lists the computed probability values for the secret degrees. These are the secret degree probability values that will occur in practice, forced by the degree distribution of the normal fountain code.

The non-optimal secret degree distribution increases the average number $\overline{T_S}$ of packets that need to be transmitted to successfully decode the secret. Earlier we found that the overt fountain code of $K = 64$ will have $\overline{T} = 82.7$. We can thus expect that $\alpha\overline{T} = 69.4$ packets contain information about the secret for $K_S = 32$. This is unlikely to be enough when $K_S = 64$ (so we do not consider this choice for K_S), but might be sufficient when $K_S = 32$ or $K_S = 16$.

To test these ideas experimentally we have run 10,000 simulations for both $K_S = 16$ and $K_S = 32$, for sizes b_S of 3, 4 and 5. The results are shown in table 14 and figures 21 and 22. When $K_S = 16$ the secret message is successfully transmitted with a very high probability, similar to the $K_S = 8$ approach from the previous section, i.e. when not using

	$K_S = 16$			$K_S = 32$		
b_S	3	4	5	3	4	5
p_S	0.998	0.9993	0.9994	0.9208	0.9348	0.9745
$\overline{T}$	84.76	87.44	94.65	84.85	88.51	102.29
σ	9.13	9.76	11.07	9.50	10.09	12.83

Table 14: Success rate for Half-Degree method for various K_S and b_S, and the negative impact on the normal fountain code.

Half-Degree. For $K_S = 32$ the success rate is above 90%, which is also not bad considering the larger size of the secret message.

However this method has a negative impact on the normal fountain code. The average number of transmitted normal fountain code packets $\overline{T}$, also listed in table 14, increases when b_S increases. Further increases of b_S have an even bigger negative impact on the normal fountain code. Interestingly, the increase in transmitted normal packets causes p_S to increase as well. As the normal fountain code behaves worse, more secret packets are transmitted, which helps to decode the secret message. That is why the value of p_S for $K_S = 32$ is higher when $b_S = 5$ than when $b_S = 3$.

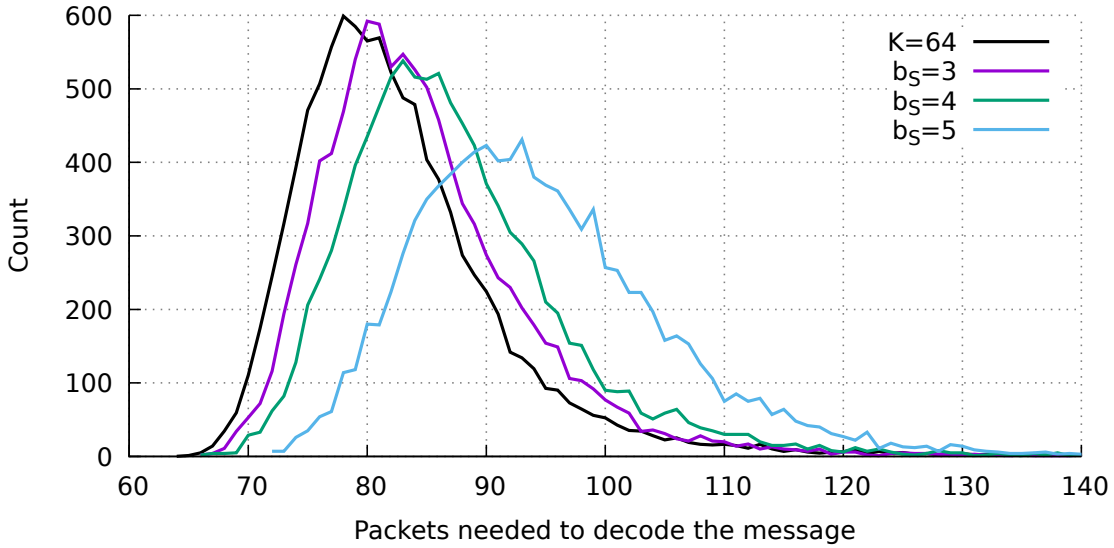


Figure 21: Histogram of the number of packets needed to decode the normal fountain code message, when embedding a Half-Degree covert channel with $K_S = 16$ and various b_S. Higher b_S have a more negative impact on the normal fountain code behaviour.

Figures 21 and 22 compare the behaviour of the transmitted packets to the ideal behaviour for $K = 64$ when no covert channel is embedded. A value of $b_S = 3$ might still be acceptable, but higher values for b_S have the risk of raising suspicion in an observer, especially if the observer can see multiple full fountain code transmissions. The covert channel should not adversely impact the normal fountain code, so the choices $b_S = 4$ and

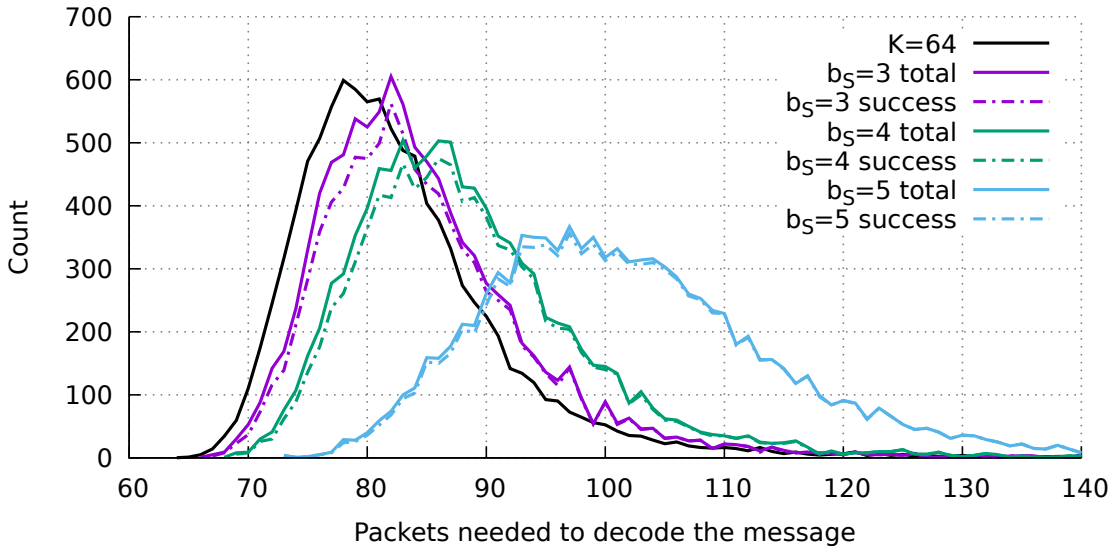


Figure 22: The same as figure 21 but for $K_S = 32$. Also shown in dashed lines are the counts for the successfull transmissions of the secret, as p_S is now not close to 1 anymore.

$b_S = 5$ are not recommended. However if we limit ourselves to using $b_S = 3$, then the secret message size is only $S = 48$ for $K_S = 16$, which is worse than what we were able to achieve in the previous section. So it would seem that we have not made any improvement by introducing the Half-Degree method, despite no longer needing to encode the value of the secret degree.

In order to find a solution we note that the cause for the deterioration of the normal fountain code lies in what goes on in the normal header of degree 2. The secret header is variable but the secret payload is not. Even though 2^{b_S} different combinations are possible for the secret payload, in practice not all these combinations will occur, as the secret message is fixed at the start of the fountain code. So for $K_S = 16$ there will be only $\binom{16}{1} = 16$ different secret payloads that happen at secret degree 1.

Worse, each secret payload is coupled to a single secret header, so the number of unique secret packets of degree 1 will be very small. If $b_S = 5$ and $K_S = 16$ (requiring 4 bit for the secret header of degree 1) then the $5 + 4 = 9$ bit that are taken by the secret packet will only have 16 different values instead of $2^9 = 512$ different values. The integral factors add some more freedom, but degree 2 of the normal fountain code only has 11 bit available, leaving only 2 bit for the integral factor. This means that the total number of different combinations in the normal fountain code header of degree 2 will be not larger than 64, as opposed to $\binom{64}{2} = 2016$ possible combinations when no covert channel is present.

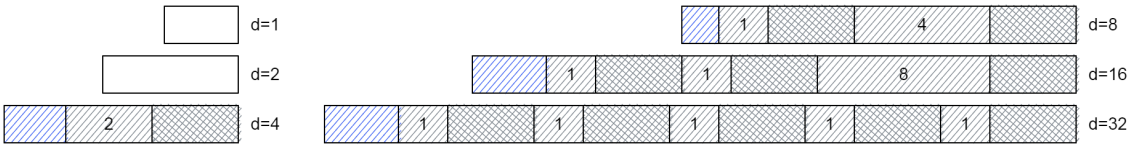
The low number of used normal source block combinations will cause the normal fountain code to often send identical packets in normal degree 2, which does not help the normal receiver. The normal receiver will thus need to receive more packets before it can decode the normal message. We did not have this problem before, as there were typically enough

bits in the normal header that were not used by the secret payload, thus allowing enough different combinations of normal source blocks.

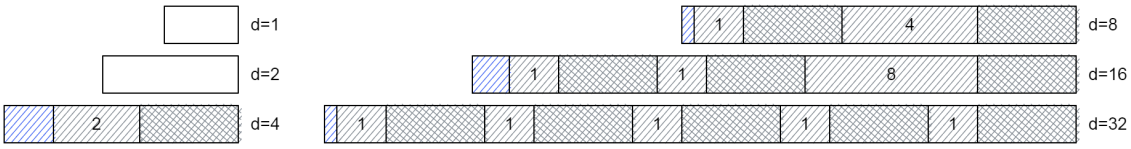
Note that the normal fountain code works fine if there are enough source block combinations available, even if a covert channel reduces this amount from the maximum available amount. No extra packets are needed as long as the receiver will, with high likelihood, always receive different packets. The trick is thus to design a covert channel that does not reduce the number of available source block combinations too much.

4.3.2 Improvement by adding more secret packets

A solution to the problem of insufficient choice in degree 2 when embedding a covert channel is to simply not use degree 2 in the Half-Degree method. This solves one problem but creates another, because we now no longer send secret packets of degree 1, so the LT decoding never starts. However we do have extra space available in the headers of higher degrees, which we are currently using only for the integral factors. So by making a puzzle of secret packets, we can also send secret packets of degree 1.



(a) $b_S = 7$



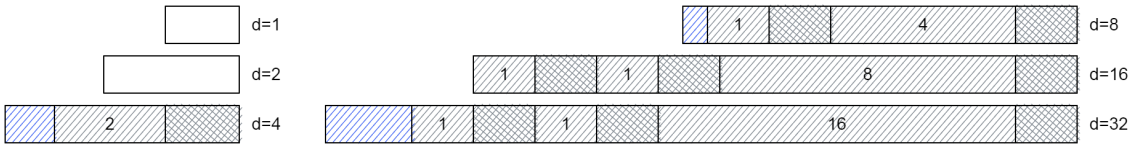
(b) $b_S = 8$

Figure 23: Puzzle for $K_S = 16$. The secret packets of the Half-Degree are augmented by secret packets of degree 1 everywhere there is space available in the normal header.

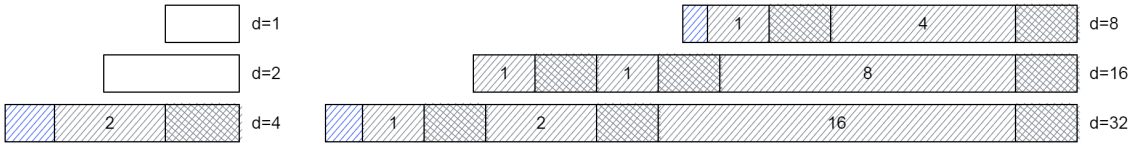
In figure 23 we show what we mean, for puzzles with $K_S = 16$ using $b_S = 7$ and $b_S = 8$. The header of normal degree 8 contains the secret packet of degree 4, and also has an extra secret packet of degree 1. Normal degree 16 has space for two extra secret packets of degree 1. Normal degree 32 was not used in $K_S = 16$, so we fill this with five secret packets of degree 1. We also force these five packets to be created from different secret source blocks, as it does not make sense to send the same secret packet multiple times if we can avoid it.

We can compute the value of α as before, giving $\alpha = 1 \times 0.256 + 2 \times 0.101 + 3 \times 0.045 + 5 \times 0.037 = 0.778$.

We can make a similar puzzle for $K_S = 32$. The secret headers are bigger now, which leaves less space to add packets of degree 1, see figure 24. To fit enough secret packets we reduce the secret payload size to $b_S = 5$. We also present a slight variation, by changing



(a) Filling the extra space with packets of degree 1.



(b) Filling the extra space with packets of both degree 1 and degree 2.

Figure 24: Puzzle for $K_S = 32$ and $b_S = 5$.

one of the degree 1 packets to a degree 2 packet. The idea is that this might improve the secret fountain code success rate, as the actual secret degree distribution is now closer to optimal. In both cases we have $\alpha = 1 \times 0.256 + 2 \times 0.101 + 3 \times 0.045 + 3 \times 0.037 = 0.704$.

Combining multiple secret packets into a single header We have not yet embedded two or more secret packets into a single normal fountain code header, so we should first describe how to do this in practice. A simple approach would be to divide the bits of the normal header into groups based on the number of bits that each secret packet requires. This is possible when a secret packet takes up an integer number of bits, as is the case in the secret packets of degree 1 for $K_S = 16$ and $K_S = 32$. In the case of $K_S = 16$ the secret header needs exactly $\log_2 \binom{16}{1} = 4$ bit, and in the case of $K_S = 32$ the secret header needs exactly 5 bit. The secret payload is of course also an integer number of bit, so the total secret packet fits exactly into an integer number of bit. This means that we could for example say that bits 10 to 20 of the normal header of degree 32 are reserved for a secret packet of degree 1.

A more flexible approach for embedding multiple secret packets into a single normal fountain code header, that also works when the secret packet does not fit exactly into an integer number of bits, is to think of the secret packets as big decimal numbers instead of as bitstrings. This is the approach that we have implemented in the Java code. Each secret packet is first converted to a decimal number, and then all the decimal numbers are combined via multiplication and addition. Suppose that we have two secret packets, A and B, both of degree 2 in a $K_S = 16$ secret fountain code, and both with a payload size of $b_S = 5$. This means that the secret packets will have a secret header value between 0 and $\binom{16}{2}$ and a secret payload value between 0 and 2^5. The secret packet consisting of both secret header and secret payload will thus have a decimal value between 0 and $\binom{16}{2} \times 2^5 = 3840$. The value 3840 does not fit exactly into an integer number of bits, so the simple approach from the previous paragraph does not apply.

To combine secret packets A and B into a single decimal value, where B takes on values

between 0 and 3840, we compute

$$x = A \times 3840 + B. \tag{24}$$

We then multiply x with an integral factor, like before, to fill all the available spaces in the normal fountain code header. This final value is then used as the normal fountain code header. The receiver can extract A and B from the received x value by computing $A = \lfloor x/3840 \rfloor$ and $B = x \mod 3840$. This multiplication approach is very flexible as the degree of the secret packet does not matter. It is also safe, as all the available space in the normal header is used, so an observer will not notice any abnormal behaviour in the normal fountain code. Finally, the multiplication approach is easy to extend to scenarios where there are more than 2 secrets packets, potentially of different degrees, that need to be combined, as is for example the case in normal degree 32 in figure 24.

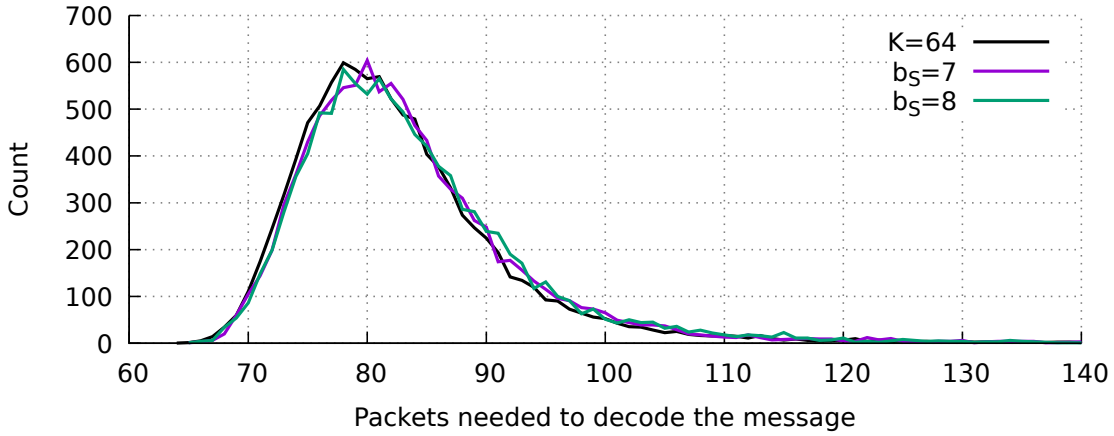


Figure 25: Comparison between a fountain code of $K = 64$ that does not have a covert channel, and the fountain code with covert channel using the improved Half-Degree method with $K_S = 16$ for either $b_S = 7$ or $b_S = 8$. The number of packets required to decode the message is not impacted by the presence of the covert channel. The secret message is transmitted with a success rate of almost 100%.

Simulations We run 10,000 simulated fountain codes, for each of the scenarios discussed above. The results for $K_S = 16$ for both $b_S = 7$ and $b_S = 8$ are shown in figure 25. The normal fountain code is now no longer negatively affected by the existence of the covert channel, thus redeeming the Half-Degree method. The success rate of the secret message is $p_S = 0.9961$ for $b_S = 7$, and is $p_S = 0.9961$ (the same) for $b_S = 8$. Reserving less space for the integral factors does not negatively impact the performance in this case, because there are enough free bits left in the normal header, even when $b_S = 8$. This shows that the improved Half-Degree method with $K_S = 16$ and $b_S = 8$, giving a secret message size of $S = 128$ bit, achieves a success rate of 99.61%.

The simulation results for the puzzles with $K_S = 32$ and $b_S = 5$ are shown in figure 26. The normal fountain code is again no longer negatively impacted. The success rate is

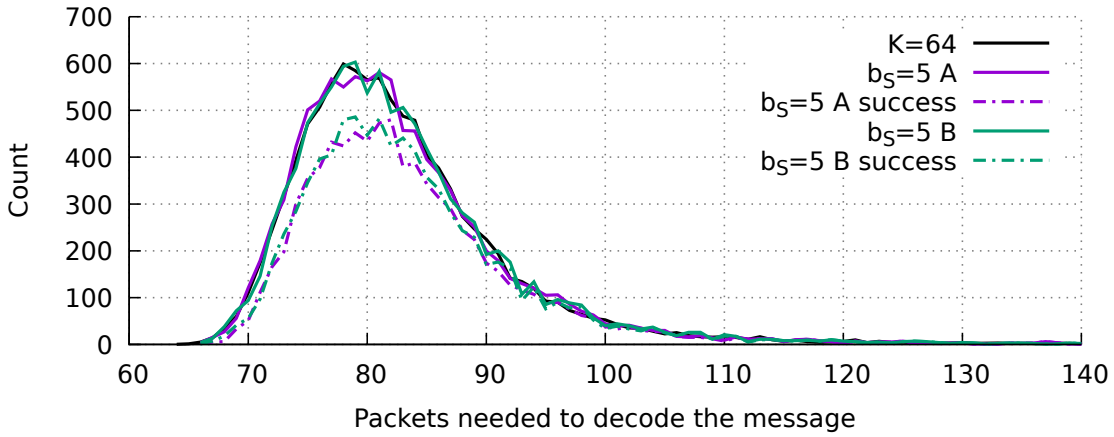


Figure 26: Similar to figure 25 but now with $K_S = 32$. Only $b_S = 5$ is tested, but both variations of the puzzle from figure 24 are shown. The dashed lines show the success rate of the secret message, as it is no longer close to 100%.

$p_S = 0.7959$ for the first puzzle (figure 24 (a)), and $p_S = 0.8234$ for the second puzzle, where the single degree 1 secret packet was replaced with a degree 2 secret packet (figure 24 (b)). This experiment shows that it is possible to transmit a secret message of size $S = 32 \times 5 = 160$ bit with a reasonably high success chance of 82.34%.

4.4 Comparison

In the previous sections we have analysed many different ways of embedding a covert fountain code in the headers of a normal fountain code. Table 15 summarises all these results and compares them for the parameters that we are most interested in, namely the secret message size S and the secret transmission success chance p_S. The parameter $\overline{T}$ is included to show the impact that embedding the covert fountain code has on the normal fountain code. A value close to 82.7 indicates that there is no impact. We have included the results for the unmodified Half-Degree method for completeness, however the negative impact on the normal fountain code is quite large and this could expose the existence of the covert channel. The improved Half-Degree method works better in any case, achieving higher values for S compared to the non-improved version of the Half-Degree method.

If a small secret message of up to 96 bit should be sent, then the method from [6] with $K_S = 8$ behaves best, giving a success rate of almost 100%. Larger secret messages can be sent with the improved Half-Degree method, either with $K_S = 16$ for 128 bit messages, or with $K_S = 32$ for 160 bit messages. However the $K_S = 32$ method suffers from a lower success rate, so overall the improved Half-Degree method with $K_S = 16$ provides the best balance of S and p_S.

Further improvements are perhaps possible by continuing with the idea of the improved Half-Degree method and creating different puzzles. As long as the covert sender and covert receiver both know how the fountain code headers are structured, any combination

Method	K_S	b_S	S (bit)	degrees	α	p_S	$\overline{T}$
[6]	16	3	48	4	0.256	0.4368	**82.696**
[6] with more degrees	16	3	48	4, 8, 16, 32	0.439	0.9747	**82.915**
[6] with $d^* = \{8, 16, 32\}$	16	16	**256**	8, 16, 32	0.183	0.0965	**82.811**
[6] with $K_S = 8$	8	12	96	4, 8, 16, 32	0.439	**0.9995**	**84.234**
Half-degree	16	5	80	2, 4, 8, 16	0.802	**0.9994**	94.65
Half-degree	32	4	**128**	2, 4, 8, 16, 32	0.839	0.9348	88.51
Half-degree improved	16	8	**128**	4, 8, 16, 32	0.778	**0.9961**	**83.791**
Half-degree improved	32	5	**160**	4, 8, 16, 32	0.704	0.8234	**82.920**

Table 15: Comparison between different methods to use a fountain code (of various K_S) as a covert channel inside a fountain code with a fixed $K = 64$. High S and p_S values are better, while keeping $\overline{T}$ close to 82.7 to avoid making the normal fountain code less efficient.

of secret packets inside normal headers can work. A triple trade-off exists between the choice of K_S, the choice of b_S, and the impact on the normal fountain code, which can make determining the overall best puzzle very challenging.

5 Implementation

We have created Java code that implements fountain codes with optional embedding of a covert fountain code. The Java code has been structured using an Object-Oriented approach. We have created a total of 26 classes to perform all the simulations that were described in the previous sections. In this section we will explain how the classes form a coherent whole, and how to get started with compiling and running them. This should hopefully allow other people to easily verify the results that we obtained.

5.1 Instructions

We have used Java 8 with Eclipse 2021-09, version 4.21.0. Only standard Java libraries were used (JRE System Library JavaSE-1.8), so no external libraries need to be downloaded or installed to run the code. All classes are part of the same package, called "fountaincode". So it should be very straightforward to add all Java code files to an Eclipse project to compile and run the code.

The entrypoint to the code is the `Test.java` file. The Java function `public static void main` in `Test.java` lists all the tests that can be run. By default all tests will run in sequence. This might take a while and requires a lot of computation power, so we recommend to comment out all the tests that should not be run, and perhaps only run a single test at a time.

Running a test will generate an output text file with a name like `results_***.txt`. Each test will generate a differently named results file, to avoid overwriting results from other tests. For most tests the output written to the results files is the histogram data that we used to create the plots shown in this work. They include computations of $\overline{T}$, σ and p_S at the top of each file.

Tests can be modified by changing the parameters in the `Test.java` class. One useful value is the variable `numberOfExperiments`. The precision of the resulting histogram will be higher if a large number of experiments are run. We have used a value of 10,000 for almost all experiments, but to run the tests faster, or to confirm that the tests work, a lower value can be used.

On our machine, an Acer Swift laptop with an AMD Ryzen 5 4500U CPU with 8GB RAM running Windows 10 64-bit, a single run of 10,000 simulations took between 5 to 10 minutes to complete, depending on the complexity of the test.

5.2 Architecture

As the implementation of the Java code was a core part of this work, we go into some detail of how the code is structured. We first present a high-level overview of the classes and how they work together, and then dive deeper into each class. An overview of our architecture is shown in figure 27.

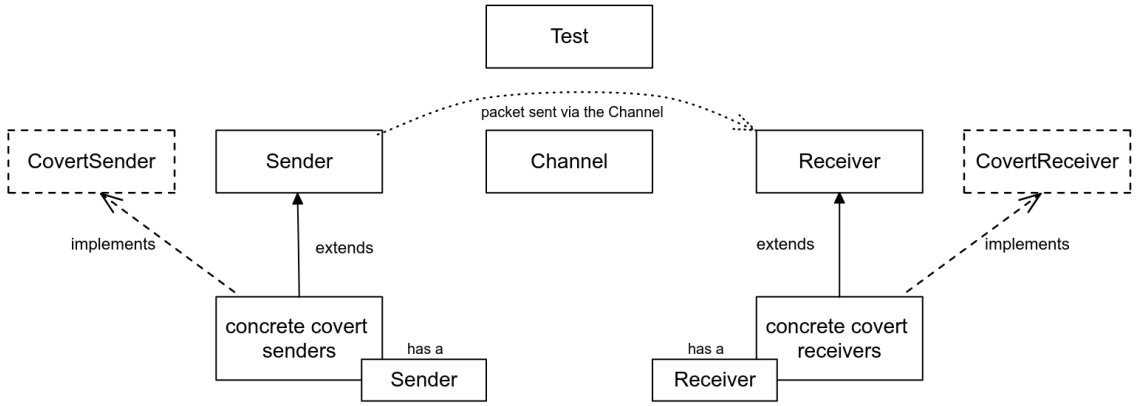


Figure 27: Simplified architecture diagram for the fountain code and covert channel implementation. See the text for a description.

Central to everything are the `Sender`, `Receiver` and `Channel` classes. A `Channel` object is constructed by passing a reference to a `Sender` and a `Receiver`. The `Channel` has a function called `sendPacket` that asks the `Sender` for a fountain code packet, and then gives this packet to the `Receiver`. A packet itself is a Java `String` object consisting of only the characters 0 and 1. This way of passing the packet ensures that no other information can leak from the `Sender` to the `Receiver` by accident. The `Channel` can also decide to discard this packet with a certain configured probability, to simulate packet loss. The `Receiver` then receives this packet and applies the LT decoding algorithm. All the test functions in the `Test` class call the `sendPacket` function in a loop, which runs until the `Receiver` has successfully decoded the (normal) fountain code message.

For the covert channel, the main idea is that a covert sender is first of all still a normal sender. Without normal fountain code packets there can be no secret message. So a covert sender must have all the functions necessary to create normal fountain code packets, like a normal sender. For this reason all covert senders extend from `Sender`.

The special thing about a covert sender is that it will no longer choose the (normal) source blocks randomly. Instead, the source blocks will be chosen so that the header that is created will be a secret fountain code packet. It does this by overriding the `selectPackets` function. Inside the `selectPackets` function the covert sender will set the source block indices non-randomly. To know which source blocks to select it needs to know secret packets from the secret fountain code. To generate these secret fountain code packets a covert sender owns an extra `Sender` object as a private attribute, which creates secret fountain code packets from the secret message. There are thus two objects of type `Sender` involved: the covert sender itself, since it extends from `Sender` to create the (modified) normal fountain code packets, and the `Sender` object that is owned by the covert sender, to create the secret fountain code packets that are used to determine the source block indices to use.

To group the different types of covert senders, and to also make it clear to the orchestration

functions in `Test` that the given `Sender` object can be a covert sender, all covert senders implement the interface `CovertSender`. Note that this structure also can allow multi-level covert channels [2]. To achieve multi-level covert channels a special kind of covert sender should be created which owns an object of type `CovertSender` instead of owning an object of type `Sender`.

We used a similar setup for the receiver. Each covert receiver extends from `Receiver`, and overrides the function `decode`. First the received packet will be given to the parent class `Receiver`, to perform the normal fountain code decoding. Then, instead of stopping, like a normal `Receiver` would, the covert receiver also extracts the normal header and interprets it as a secret fountain code packet. It gives this secret fountain code packet to an extra object of type `Receiver`, owned as a private attribute, which will attempt to decode the secret message. The interface CovertReceiver is similar to `CovertSender` in that it groups all the covert receivers together. It also exposes the private `Receiver` object, so that the test function can check whether the secret message has been successfully decoded or not.

This symmetrical setup of `Sender` and `Receiver` makes it hopefully easier to understand the code. The following covert sender and receiver pairs are provided:

- `CovertSenderDegree4` and `CovertReceiverDegree4`: the approach from [6] with a covert channel in just $d^* = 4$. This was covered in section 4.1.
- `CovertSenderMoreDegrees` and `CovertReceiverMoreDegrees`: the approach from [6] with more degrees, i.e. $d^* = \{4, 8, 16, 32\}$. This was covered in section 4.2.1.
- `CovertSenderHigherDegrees` and `CovertReceiverHigherDegrees`: the approach from [6] with only higher degrees, i.e. $d^* = \{8, 16, 32\}$. This was covered in section 4.2.2.
- `CovertSenderMoreDegreesK8` and `CovertReceiverMoreDegreesK8`: the approach from [6] with $K_S = 8$. This was covered in section 4.2.3.
- `CovertSenderHalfDegree` and `CovertReceiverHalfDegree`: the initial Half-Degree method, which negatively impacts the normal fountain code. This was covered in section 4.3.1.
- `CovertSenderHalfDegreePuzzle` and `CovertReceiverHalfDegreePuzzle`: the modified Half-Degree method with $K_S = 16$, where degree $d^* = 2$ is no longer used. This was covered in section 4.3.2.
- `CovertSenderHalfDegreePuzzle32` and `CovertReceiverHalfDegreePuzzle32`: the modified Half-Degree method with $K_S = 32$, where degree $d^* = 2$ is no longer used. This was covered in section 4.3.2.

Each of the static functions in the `Test` class create a certain concrete `CovertSender` and `CovertReceiver` with certain parameters, e.g. the choice of b_S. The covert sender and covert receiver must match, e.g. one should not create a test where a `CovertSenderMoreDegrees` object sends packets to a `CovertReceiverDegree4` object. That will lead to errors. The

parameters that are given to the constructors likewise need to match as well, e.g. the choice of secret payload size b_S must be identical. This requirement makes it clear that these choices are what the covert sender and covert receiver need to somehow agree on in advance, outside of the fountain code transmission.

The classes `Channel`, `Sender`, `Receiver`, `CovertSender`, `CovertReceiver` and `Test` that we have already discussed make use of the following classes that provide shared functionality. These shared classes were not included in the diagram of figure 27 to keep the figure simple.

- `Distribution`: a collection of degree distributions ρ. To sample degrees according to a distribution we need to know the cumulative distribution function, see section 5.3.1, so we also compute the CDF in this class. The `Distribution` class also has code that computes the ideal and robust soliton distributions, although these are not used in tests.
- `Header`: functions to create a header of BitVector, Enum of EnumAll type, given a list of source block indices. The tests only use Enum and EnumAll, but we have included the BitVector header type for completeness.
- `Headertype`: a Java enum with types BITVECTOR, ENUM and ENUMALL.
- `IntegralFactors`: all the integral factors used by the various covert senders. The numbers are very large so we use `BigInteger` to store them without loss of precision.
- `KolmogorovSmirnov`: independent class to perform the Kolmogorov-Smirnov tests from section 4.1.3. This class is not used by the fountain code classes.
- `Utilities`: static functions for computing a binomial and for XOR-ing bitstrings.

5.3 Algorithms

The architecture diagram and class structure explanation glances over the fact that some algorithms are not straightforward. Especially the ENUM algorithm and the integral factors algorithm required careful analysis and testing. In this section we explain the implementation of these two algorithms, as well as the degree distribution sampling approach and the computation of large binomial factors in Java.

5.3.1 Distribution algorithm

When creating a fountain code packet, the degree d is chosen according to a given probability distribution, which is not necessarily uniform. To create a random number that is sampled according to a certain probability distribution, we first generate a uniformly distributed random number using the `Random.nextDouble()` function in Java. Then we convert this number, which lies between 0 and 1, to a degree value via the cumulative distribution function (CDF) of the degree probability distribution. The chosen degree

d	$\rho(d)$	CDF(d)
1	0.161	0.161
2	0.4	0.561
4	0.256	0.817
8	0.101	0.918
16	0.045	0.963
32	0.037	1.0

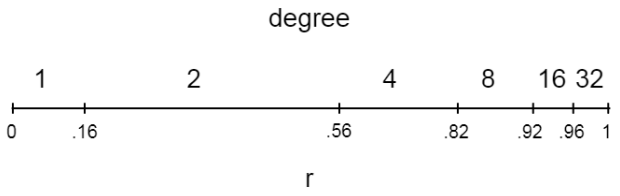


Figure 28: Illustration of the procedure to generate the degree number for a given distribution $\rho(d)$ for $K = 64$. Starting from a random number r with uniform distribution, the degree number is found by looking at the interval of the CDF where r is located. For example $r = 0.75$ is between 0.561 and 0.817 and thus gives $d = 4$.

value depends on where the random number r falls in the range of the CDF. Figure 28 illustrates the method with an example.

5.3.2 Binomial coefficients algorithm

The value of the binomial $\binom{K}{d}$ can become very large. To keep precision we use the Java built-in type `BigInteger`. Unfortunately Java does not have a built-in way to compute large binomial numbers, so we must compute this ourselves. The standard factorial form $\frac{K!}{d!(K-d)!}$ is not efficient as it leads to a fraction that is difficult to evaluate, and slow to compute in Java. Instead we use the multiplicative formula

$$\binom{K}{d} = \prod_{i=0}^{k-1} \frac{n-i}{i+1}. \tag{25}$$

This gives precise results and is fast for our values of K and d.

5.3.3 ENUM algorithm

From source block indices to enum number The ENUM header was described in section 3.1.2.

A straightforward approach to find the index number n that corresponds to a certain set of source block indices $\{s_i\}$ is to enumerate all possible sets in a loop. The loop stops when the required set is reached, and n is then equal to the loop counter. For example to find the value of n for $\{2, 7, 8, 13\}$ we can start with $0 = \{1, 2, 3, 4\}$, followed by $1 = \{1, 2, 3, 5\}$, $2 = \{1, 2, 3, 6\}$, $\ldots$, $703 = \{2, 7, 8, 13\}$. The same method can be used to find the set of source block indices that correspond to a given n.

This method is very inefficient, because it needs to construct all the possible sets up to the required set. We saw earlier that there are $\binom{K}{d}$ different unique combinations, i.e. sets. In other words, the runtime complexity is equal to $O(N)$ with $N = \binom{K}{d}$. This grows at a phenomenal rate, for example for the reasonably small values of $K = 64$ and $d = 32$

the loop needs to construct up to $\binom{64}{32} = 1832624140942590534$ different sets, which is not feasible.

A better approach is to calculate n directly. We introduce the approach with the example set $\{2, 7, 8, 13\}$ with $K = 16$, and then generalise the procedure.

In order to reach $\{2, 7, 8, 13\}$ in the loop that enumerates all possible sets, we need to have constructed all the sets that start with a 1, i.e. all $\{1, \square, \square, \square\}$. For the positions denoted by the $\square$ symbols, we can choose any set of 3 elements out of 15 possible values (ranging from 2 to 16, because the value 1 has been chosen already). This gives $\binom{15}{3} = 455$ different sets that have a value of 1 for the first source block index. So we could start the enumeration loop already from $\{2, 3, 4, 5\}$ with $n = 455$ instead of from $\{1, 2, 3, 4\}$ with $n = 0$, thus saving some effort.

We can continue this reasoning for the second position. In order to reach the value 7 on the second position, we need to have constructed all the sets that have the value 3, 4, 5 and 6 on the second position. For the set $\{2, 3, \square, \square\}$ there are $\binom{13}{2}$ possibilities, for $\{2, 4, \square, \square\}$ there are $\binom{12}{2}$ possibilities, and then $\binom{11}{2}$ and $\binom{10}{2}$ for 5 and 6 respectively. Summing these binomials together gives $\binom{13}{2} + \binom{12}{2} + \binom{11}{2} + \binom{10}{2} = 244$, which means that $n = 455 + 244 = 699$ for $\{2, 7, 8, 9\}$. Continuing the procedure gives the expected $n = 703$.

Generalizing, if the set looks like $\{A, B, C, D\}$ for $d = 4$ and general K then we first need to count the number of sets where the first position has a value that is lower than A. This gives the sum of binomials $\binom{K-1}{3} + \binom{K-2}{3} + \cdots + \binom{K-A+1}{3}$, or generally

$$\sum_{i=1}^{A-1} \binom{K-i}{d-1} \tag{26}$$

possibilities before reaching a value A in the first position.

Then we need to count the number of sets that have A in the first position and that have a value greater than A and lower than B in the second position. This gives another sum of binomials, similar to the previous one. Continuing until D, we get

$$n = \sum_{i=1}^{A-1} \binom{K-i}{d-1} + \sum_{i=A+1}^{B-1} \binom{K-i}{d-2} + \sum_{i=B+1}^{C-1} \binom{K-i}{d-3} + \sum_{i=C+1}^{D-1} \binom{K-i}{d-4} \tag{27}$$

The above formula is valid for $d = 4$. If we try to generalise d as well then we need to create a sum of sums. Writing as s_j the value of the set at position j, we get the following expression

$$n = \sum_{j=1}^{d} \left(\sum_{i=s_{j-1}+1}^{s_j-1} \binom{K-i}{d-j} \right) \tag{28}$$

In Java this double sum is implemented as a double loop and extensively tested for correctness. The binomials are calculated using the algorithm from section 5.3.2.

From enum number to source block indices The opposite direction, where n is given and the set needs to be found, is done by evaluating formula (28) in reverse. Starting from n we subtract consecutive binomials from n, continuing until the next subtraction would give a number below 0. The amount of binomials that we can subtract before n falls below 0 is then the first value of the source block indices. We continue subtracting smaller binomials from the reduced n to find the second source block value and continue until there are no more binomials to subtract. The following example clarifies this process.

We want to compute the source block indices corresponding to the enum number $n = 1293$ with $K = 16$ and $d = 4$. The first index can not be 1, because there are only $\binom{15}{3} = 455$ different sets that start with 1, and n is higher than 455. We thus subtract 455 from n and check if the first source block could be 2.

$$1293 - \binom{15}{3} = 838 \qquad 1$$

The value for n is now 838. The first index can also not be 2, because $838 > \binom{14}{3}$, so we subtract $\binom{14}{3}$ from the current value of n. Continuing this gives

$$838 - \binom{14}{3} = 474 \qquad 2$$
$$474 - \binom{13}{3} = 188 \qquad 3$$
$$188 - \binom{12}{3} < 0 \qquad \underline{4}$$

So we see that the first source block index is 4. To find the next source block indices we reduce the degree number, as the first source block has been determined. If we simply keep increasing the counter (written on the right), then we can read off all the source block indices by noting where the sum would drop below 0.

$$188 - \binom{11}{2} = 133 \qquad 5$$
$$133 - \binom{10}{2} = 88 \qquad 6$$
$$88 - \binom{9}{2} = 52 \qquad 7$$
$$52 - \binom{8}{2} = 24 \qquad 8$$
$$24 - \binom{7}{2} = 3 \qquad 9$$
$$3 - \binom{6}{2} < 0 \qquad \underline{10}$$

The second index is thus 10. The next indices are

$$\begin{aligned} 3 - \binom{5}{1} &< 0 && \underline{11} \\ 3 - \binom{4}{0} &= 2 && 12 \\ 2 - \binom{3}{0} &= 1 && 13 \\ 1 - \binom{2}{0} &= 0 && 14 \\ 0 - \binom{1}{0} &< 0 && \underline{15} \end{aligned}$$

We conclude that $1293 = \{4, 10, 11, 15\}$. Using formula (28) to go in the other direction confirms that this is correct.

5.3.4 Integral factors

Computing integral factors The integral factors f_d help to create a mapping from X different secret headers to H different header symbols, with $H \geq X$. Not all X secret headers are equally likely, as first a secret degree is chosen according to a degree distribution, and each degree has a different number of secret headers. The effect of the mapping should be that all H header symbols are equally likely, despite the X secret headers not being equally likely. This is achieved by mapping each element of X to a different amount of header symbols, depending on the likelihood of the element of X.

We explain the algorithm to find such integral factors first with an example, and then try to formulate a general approach. We use the example of mapping the secret headers for $K_S = 16$, with degree distribution from table 3, onto $H = 79422$ header symbols. Our algorithm starts from finding the integral factor for the highest degree and then works its way down to the lowest degree. We start with the highest, because that has the smallest integral factors and thus has the largest impact from rounding.

Degree 8 has a chance of $\rho(8) = 0.134$ to occur. So out of the 79422 header symbols, there should be $79422 \times 0.134 \approx 10642$ header symbols mapped from degree 8. There are $\binom{16}{8} = 12870$ different headers of degree 8, so the ideal integral factor is $10642/12870 \approx 0.827$. However integral factors can only be integer and must be at least 1, so we round this ideal value up to 1. The integral factor of degree 8 is $f_8 = 1$.

To determine the integral factor of degree 4, we first note that already 1×12870 header symbols have been reserved by degree 8. So $79422 - 12870 = 66552$ header symbols remain to be divided between degrees 1, 2 and 4 with relative weights $\rho(1)$, $\rho(2)$ and $\rho(4)$. The relative chance of degree 4 is $\frac{\rho(4)}{\rho(1)+\rho(2)+\rho(4)} = 0.217$, and there are $\binom{16}{4} = 1820$ headers of degree 4, so the ideal integral factor is $66552 \times 0.217/1820 = 7.94$. Rounding to the nearest integer gives $f_4 = 8$.

With f_4 determined only $66552 - f_4 \times 1820 = 51992$ header symbols remain to be distributed over degrees 1 and 2. The ideal integral factor for degree 2 is $51992 \times \frac{\rho(2)}{\rho(1)+\rho(2)} / \binom{16}{2} = 292.04$. The nearest integer gives $f_2 = 292$.

Finally for f_1 there are $51992 - f_2 \times \binom{16}{2} = 16952$ header symbols remaining. The ideal integral factor is $16952/16 = 1059.5$. For this last integral factor we can not round up, because then we would map to more symbols than there are available. This gives then finally $f_1 = 1059$.

Generally, the algorithm runs in iterations. If H header symbols are still available, then

$$f_d = \left[H \frac{\rho(d)}{\sum_k \rho(k)} / \binom{K_S}{d} \right]. \tag{29}$$

where [ ] means rounding to the nearest integer.

Then H is updated, with new $H' = H - f_d \binom{16}{d}$. The algorithm starts with the highest d and continues until $d = 1$. For $d = 1$ the rounding is down rather than to the nearest integer.

When all f_d have been found there exists a mapping from X to H. However because we rounded the ideal integral factors to the nearest integer, the relative probability of the header symbols is still wrong. We can fix this by updating the original distribution $\rho(d)$ to a new distribution $\rho^*(d)$ via formula (22),

$$\rho^*(d) = \frac{\binom{K_S}{d} f_d}{H}. \tag{30}$$

In our example from before this gives for e.g. degree 8, $\rho^*(8) = 1 \times 12870/79422 = 0.162$, which is a bit higher than the optimal value 0.134.

A final problem to fix is that the sum of all the ρ^* should give 1, so that ρ^* is a proper distribution. Due to rounding it can happen that the values of ρ^* do not sum nicely to 1. We fix this problem by simply adding the difference to the highest degree. An example where this happens is for the integral factors of $K_S = 8$ with $b_S = 12$ into $d^* = 4$.

Using integral factors An integral factor maps a single secret header onto a range of possible target header symbols, e.g. a single header of degree 1 in $K_S = 16$ is mapped to $f_1 = 1059$ different header symbols. The size of the range is exactly the value of the integral factor. The actual header symbol that will be used to represent the secret header is a random choice within this range.

In the covert sender we first determine the start and end of the range, for a given secret header expressed as a decimal value. Then we select a random value between the start and end, and use this value as our header symbol, i.e. this value combined with the secret payload becomes the header of the normal fountain code packet.

In the covert receiver we first separate the payload from the normal header, and then we separate the secret payload from the secret header. This secret header we then simply divide by the integral factor and round down, which gives us the original secret header.

6 Conclusion

We have presented several ways to embed a covert channel in a fountain code, building on the work of [6]. We have first evaluated the chance that the original approach of [6] can successfully transmit a secret message over the covert channel. We then proposed several improvements, to increase both the chance of a successful transmission and to increase the size of the secret message that can be embedded in the covert channel. The best approach that we found was a puzzle of secret fountain code packets embedded in headers of degree 4 and higher, based on the Half-Degree method. This approach achieves a success rate of over 99.5% for a secret message size of 128 bit.

We have also made Java code available to test all this work and to verify the results. The Java code has been written in an Object-Oriented approach to simplify understanding the program and the algorithms.

An important result from all our simulations is that there is always a chance that the secret message fails to transmit. We can make this chance small, but we can not make it zero, because the secret fountain code could be very unlucky and accidentally not transmit the packets required by the covert receiver. Unlike the normal receiver, the covert receiver does not have the luxury to continue to wait until enough packets have been received. So if absolute certainty that the covert transmission will succeed is required, then a different approach than the ones presented in this work should be used.

When it comes to future work, in section 3 we made the decision to use the distribution for $K = 64$ from Rossi et al. [12] instead of the better distribution from Hyytia et al. [5], because the difference in required normal fountain code packets is small while the ρ for the higher degrees is higher in Rossi et al. As we have seen, the higher degrees have more bits available for the covert channel. A possible further avenue for investigation is then how much the distribution of the normal fountain code could be modified to favor higher degrees, without increasing the number of required packets too much. If the normal fountain code is too sub-optimal then this might indicate the existence of a covert channel. This analysis might be interesting for future work, to make a comparison between the covert transmission success rate and secret message size on the one hand, and the probability that an observer detects that the normal fountain code has been tampered with because it behaves inefficiently.

In practice more advanced variants of fountain codes are used, such as Raptor codes which we mentioned in section 3.4. Because Raptor codes also use uniform random sampling of source blocks, the possibility exists to embed a covert channel in Raptor codes as well, using methods similar to what we described in this paper. The overhead is smaller for Raptor codes, meaning less packets will be transmitted for a given K, so we will need a secret fountain code that requires less packets as well. The presented methods that have a very high p_S value, such as Keller's method [6] with $K_S = 8$ or Half-Degree with $K_S = 16$, will potentially still perform well in Raptor codes. On the other hand, Raptor

codes typically use a different degree distribution than LT codes and have an average degree value of about 3 [10]. As the covert channel relies on the higher degrees to transmit the secret fountain code packets, the presented methods might have a low success rate. An option could be to use Raptor codes for the secret fountain code as well. This could lead to higher success rates.

In addition, extentions to beyond $K = 64$ could be investigated as well. The proposed Half-Degree method and puzzle of secret fountain code packets lends itself well to being embedded into normal fountain codes with a different value for K, perhaps with a dense degree distribution instead of a sparse one. A puzzle image similar to figure 24 should be created and then implemented in Java, in order to evaluate the practical results for different puzzle variations.

References

[1] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege. "A Digital Fountain Approach to Reliable Distribution of Bulk Data". In: *Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM '98. Vancouver, British Columbia, Canada: Association for Computing Machinery, 1998, pp. 56–67. ISBN: 1581130031. DOI: 10.1145/285237.285258. URL: https://doi.org/10.1145/285237.285258.

[2] W. Fraczek, W. Mazurczyk, and K. Szczypiorski. "Multi-Level Steganography: Improving Hidden Communication in Networks". In: *Computing Research Repository - CORR* 8 (2011), pp. 2551–2567. DOI: 10.3217/jucs-018-14-1967.

[3] K. F. Hayajneh, S. Yousefi, and M. Valipour. "Left degree distribution shaping for LT codes over the binary erasure channel". In: *2014 27th Biennial Symposium on Communications (QBSC)*. 2014, pp. 198–202. DOI: 10.1109/QBSC.2014.6841213.

[4] E. Hyytia, T. Tirronen, and J. Virtamo. "Optimal Degree Distribution for LT Codes with Small Message Length". In: *IEEE INFOCOM 2007 - 26th IEEE International Conference on Computer Communications*. 2007, pp. 2576–2580. DOI: 10.1109/INFCOM.2007.324.

[5] E. Hyytia, T. Tirronen, and J. Virtamo. "Optimizing the degree distribution of LT codes with an importance sampling approach". In: *6th International Workshop on Rare Event Simulation, RESIM*. 2006.

[6] J. Keller. "Multilevel Network Steganography in Fountain Codes". In: *EICC: European Interdisciplinary Cybersecurity Conference* (Nov. 2021), pp. 72–76. DOI: 10.1145/3487405.3487420.

[7] B. W. Lampson. "A Note on the Confinement Problem". In: *Commun. ACM* 16.10 (Oct. 1973), pp. 613–615. ISSN: 0001-0782. DOI: 10.1145/362375.362389.

[8] M. Luby. "LT Codes". In: *Proceedings of The 43rd Annual IEEE Symposium on Foundations of Computer Science*. 2002, pp. 271–282.

[9] P. Luo, H. Fan, W. Shi, X. Qi, Y. Zhao, and X. Zhou. "An ECSO-based approach for optimizing degree distribution of short-length LT codes". In: *EURASIP Journal on Wireless Communications and Networking* 76 (2019). DOI: 10.1186/s13638-019-1376-6.

[10] D. J. C. Mackay. "Fountain Codes". In: *IEE Communications* 152 (2005), pp. 1062–1068. DOI: 10.1049/ip-com:20050237.

[11] W. Mazurczyk, S. Wendzel, S. Zander, A. Houmansadr, and K. Szczypiorski. *Information Hiding in Communication Networks: Fundamentals, Mechanisms, and Applications*. Mar. 2016. ISBN: 978-1-118-86169-1.

[12] M. Rossi, G. Zanca, L. Stabellini, R. Crepaldi, A. F. Harris III, and M. Zorzi. "SYNAPSE: A Network Reprogramming Protocol for Wireless Sensor Networks Using Fountain Codes". In: *2008 5th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks*. 2008, pp. 188–196. DOI: 10.1109/SAHCN.2008.32.

[13] A. Shokrollahi. "Raptor Codes". In: *IEEE Transactions on Information Theory* 52.6 (2006), pp. 2551–2567. DOI: 10.1109/TIT.2006.874390.

[14] J. H. Sørensen, P. Popovski, and J. Østergaard. "Design and Analysis of LT Codes with Decreasing Ripple Size". In: *IEEE Transactions on Communications* 60.11 (2012), pp. 3191–3197. DOI: 10.1109/TCOMM.2012.091112.110864.

[15] S. Wendzel, S. Zander, B. Fechner, and C. Herdin. "Pattern-Based Survey and Categorization of Network Covert Channel Techniques". In: *ACM Comput. Surv.* 47.3 (2015), 50:1–50:26. DOI: 10.1145/2684195.

[16] C. Zaiontz. *Kolmogorov-Smirnov Normality*. https://www.real-statistics.com/tests-normality-and-symmetry/statistical-tests-normality-symmetry/kolmogorov-smirnov-test/, Last accessed on 2022-03-29.

[17] C. Zaiontz. *Kolmogorov-Smirnov Table*. https://www.real-statistics.com/statistics-tables/kolmogorov-smirnov-table/, Last accessed on 2022-03-29.

A Appendix: integral factors for $K_S = 8$ into degrees of $K = 64$

When embedding a secret fountain code with $K_S = 8$ into the headers of degrees $d^* = \{4, 8, 16, 32\}$ of a normal fountain code with $K = 64$, integral factors need to be used in order to keep the performance of the normal fountain code and to avoid detection of the covert channel. We have already presented the integral factors for $d^* = 4$ in the main text, in table 12. The following tables show the (sometimes very large) integral factors for the remaining degrees d^*. These are used in the Java code when running simulations.

For $d^* = 8$			$b_S = 9$ $H = 8644854$		$b_S = 10$ $H = 4322427$		$b_S = 11$ $H = 2161213$		$b_S = 12$ $H = 1080606$	
d	$\binom{8}{d}$	$\rho(d)$	f_d	$\rho^*(d)$	f_d	$\rho^*(d)$	f_d	$\rho^*(d)$	f_d	$\rho^*(d)$
1	8	0.268	289603	0.268	144801	0.268	72401	0.268	36200	0.268
2	28	0.576	177836	0.576	88918	0.576	44460	0.576	22230	0.576
4	70	0.156	19266	0.156	9633	0.156	4816	0.156	2408	0.156

Table 16: Integral factors f_d and modified $\rho^*(d)$ for different payload sizes b_S, for a secret fountain code with $K_S = 8$ into degree $d^* = 8$ of a normal fountain code with $K = 64$.

For $d^* = 16$			$b_S = 9$ $H = 954154173983$		$b_S = 10$ $H = 477077086991$	
d	$\binom{8}{d}$	$\rho(d)$	f_d	$\rho^*(d)$	f_d	$\rho^*(d)$
1	8	0.268	31964164825	0.268	15982082413	0.268
2	28	0.576	19628314436	0.576	9814157219	0.576
4	70	0.156	2126400731	0.156	1063200365	0.156
			$b_S = 11$ $H = 238538543495$		$b_S = 12$ $H = 119269271747$	
d	$\binom{8}{d}$	$\rho(d)$	f_d	$\rho^*(d)$	f_d	$\rho^*(d)$
1	8	0.268	7991041207	0.268	3995520604	0.268
2	28	0.576	4907078608	0.576	2453539305	0.576
4	70	0.156	531600183	0.156	265800091	0.156

Table 17: Integral factors f_d and modified $\rho^*(d)$ for different payload sizes b_S, for a secret fountain code with $K_S = 8$ into degree $d^* = 16$ of a normal fountain code with $K = 64$.

For $d^* = 32$			$b_S = 9$ $H = 3579344025278497$		$b_S = 10$ $H = 1789672012639248$	
d	$\binom{8}{d}$	$\rho(d)$	f_d	$\rho^*(d)$	f_d	$\rho^*(d)$
1	8	0.268	119908024846829	0.268	59954012423413	0.268
2	28	0.576	73632219948585	0.576	36816109974293	0.576
4	70	0.156	7976823827764	0.156	3988411913882	0.156
			$b_S = 11$ $H = 894836006319624$		$b_S = 12$ $H = 447418003159812$	
d	$\binom{8}{d}$	$\rho(d)$	f_d	$\rho^*(d)$	f_d	$\rho^*(d)$
1	8	0.268	29977006211708	0.268	14988503105855	0.268
2	28	0.576	18408054987146	0.576	9204027493574	0.576
4	70	0.156	1994205956941	0.156	997102978470	0.156

Table 18: Integral factors f_d and modified $\rho^*(d)$ for different payload sizes b_S, for a secret fountain code with $K_S = 8$ into degree $d^* = 32$ of a normal fountain code with $K = 64$.

B Appendix: USB stick content

The Java code and this paper have been submitted on a USB stick. The folder structure of the USB stick is as follows:

- `/bin/fountaincode`: The compiled java code as .class files.
- `/paper`: The pdf version of this paper as well as the *Selbständigkeitserklärung*.
- `/src/fountaincode`: The .java source files.
- `readme.txt`: Instructions on how to run the java code.