On Encoding Constant Weight Words in HyMES

Final Report — September 4, 2011

Niklas Buescher and Jan Meub

TECHNISCHE UNIVERSITÄT DARMSTADT

Crypto Lab

Cryptography and Computer Algebra

Contents

1 Introduction

1.1 Abstract

This document is the result of a cryptographic lab during the summer semester 2011 at the Technische Universität Darmstadt. The task of this lab was to describe how exactly constant weight words are encoded in an implementation of the Hybrid McElice Encryption Scheme (HyMES) written in C by Bhaskar Biswas and Nicolas Sendrier at Project SECRET, INRIA, Rocquencourt in 2008. The encoding of constant weight words into binary information of fixed length is implemented as an algorithm consisting of three distinct components: Dichiotomic Model, an Enumerative Encoder and an Arithmetic Encoder. We give a short introduction into HyMES and then describe these components in detail.

1.2 Structure of this report

In the following chapters we will first introduce the McElice Cryptosystem and the changes made by HyMES. We will then describe the two used source coding approaches, Enumerative and Arithmetic Encoding in Chapter 3. Followed by Chapter 4 where we will give an overview of the implemented coding method. Then we continue with describing details of the implementation with direct reference to the source code in Chapter 5. Finally we summarize the work done in this lab and conclude our report.

2 Introduction into McEliece and HyMES

2.1 McEliece Cryptosystem

The first public key encryption scheme based on coding theory was proposed by Robert J. McEliece in 1978 [McE78]. The idea behind the McEliece cryptosystem exploits the hardness of decoding a linear code. For the encryption scheme a binary $(n \times k)$ linear code is selected that is able to detect t errors efficiently. This code is disguised as a general linear code through multiplication with a scrambler matrix. For encryption the cleartext is encoded and t errors are randomly introduced. Without knowledge of the linear code correcting these errors and the used scrambler matrix decryption of the message is NP hard. Robert J. McEliece proposed the usage of binary Goppa codes which can easily be decoded using Pattersons algorithm if the Goppa code is known. With sufficient parameters this variant has resisted cryptanalysis so far and is seen as secure against quantum attacks[OS09].

2.1.1 Linear codes

Linear codes are a form of forward error correcting codes. By lowering the information rate errors in the communication can be detected and corrected without resending the message. For linear codes every codeword is a linear combination of other codewords.
A linear code C with length n and rank k is a k-dimensional subspace of the vector space $\mathbb{F}_q^n$ where $\mathbb{F}_q$ is the finite field with q elements.
The encoding $\mathbb{F}_q^k \rightarrow C$ can be done using a generator matrix $G \in \mathbb{F}_q^{k \times n}$ such that $C = \{xG | x \in \mathbb{F}_q^k\}$. Any set of basis vectors of C forms such a generator matrix.
The basis vectors of the null-space of the linear code form the parity check matrix H: $\forall c \in C, Hc^T = 0$.
The syndrome S of a received vector y is $S = Hy^T$. Since the syndrome for all codewords $c \in C$ is 0 the syndrome only depends on the introduced error e: $Hy^T = H(c+e)^T = Hc^T + He^T = He^T$.
A t-bounded decoder is a mapping $decoder : \{0,1\}^n \rightarrow C$ that removes up to t errors: $decoder(y) = c$.

2.1.2 Binary Goppa Codes

A binary Goppa code $\Gamma(L, g)$ is defined by its support, an ordered subset $L = (a_1 ... a_n)$ of $\mathbb{F}_{2^m}$ with cardinality $n = 2^m$ and its generator, an irreducible monic polynomial g(x) of degree t in $\mathbb{F}_{2^m}[x]$.
The syndrome of a received vector $y \in \mathbb{F}_2^n = (y_1, y_2, ..., y_n)$ is

$$S_y(x) = \sum_{i=1}^{n} \frac{y_i}{x - a_i} \bmod g(x) \, .$$

Creating a t-bounded decoder can be achieved by finding a polynome σ whose roots are the positions of the ones in the error vector $e \in \mathbb{F}_2^n$:

$$\sigma(x) = \prod_{i \in I} (x - a_i) \text{ with } I = 0 < i \leq n | e_i = 1 \, .$$

Using the syndrome of a received vector this can be computed efficiently because of the relation

$$\sigma'(x) = \sigma(x)S_y(x) \bmod g(x) \,.$$

$\sigma(x)$ can be split in two parts $\alpha^2(x)$ and $x\beta^2(x) \in \mathbb{F}_{2^m}[x]$ so that $\sigma(x) = \alpha^2(x) + x\beta^2(x)$ where $\alpha^2(x)$ are the terms with even exponent and $x\beta^2(x)$ are the terms with odd exponent.
This is useful because deriving in $\mathbb{F}_{2^m}[x]$ eliminates all terms with even exponent. For terms with odd exponent the exponent is reduced by one. Therefore $\sigma'(x) = \beta^2(x)$.

Using the above mentioned relation between σ and S_y:

$$\begin{aligned} \beta^2(x) &= \big(\alpha^2(x) + x\beta^2(x)\big)S_y(x) \bmod g(x) \\ \beta^2(x)\big(S_y^{-1}(x) + x\big) &= \alpha^2(x) \bmod g(x) \\ \beta(x)T(x) &= \alpha(x) \bmod g(x) \text{ with } T(x) = \sqrt{S_y^{-1}(x) + x} \end{aligned}$$

α, β and therefore σ can now be computed with the extended euclidean algorithm

2.1.3 Key Generation

To generate a key pair for the McEliece Cryptosystem one first chooses the parameters m, t and a binary Goppa code $\Gamma(L, g)$ with length $n = 2^m$ and rank $k = 2^m - mt$ that can correct up to t errors and calculates its check matrix $H \in \mathbb{F}_2^{(n-k)\times n}$ and generator matrix $G \in \mathbb{F}_2^{k\times n}$.
Then an invertible scrambler matrix $S \in \mathbb{F}_2^{k\times k}$ and a permutation matrix $P \in \mathbb{F}_2^{n\times n}$ are chosen at random.

The public key is the tuple ($\hat{G}$, t) with $\hat{G} = SGP$, the private key is the tripel (S,G,P) and a t-bounded decoder d. The syndrome vector $s = yH^T \in \mathbb{F}_2^{k-n}$ can be transformed into the syndrome polynome $S_y(x)$. With H and g one has an efficient t-bounded decoder.

Generation of H and G:

$H_{2^m} = XYZ \in \mathbb{F}_{2^m}^{t\times n}$ where

$$X = \begin{pmatrix} g_t & & & & 0 \\ g_{t-1} & g_t & & & \\ g_{t-2} & g_{t-1} & g_t & & \\ \vdots & \vdots & \vdots & \ddots & \\ g_1 & g_2 & g_3 & \cdots & g_t \end{pmatrix}$$

$$Y = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ a_1 & a_2 & a_3 & \dots & a_n \\ a_1^2 & a_2^2 & a_3^2 & \dots & a_n^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_1^{t-1} & a_2^{t-1} & a_3^{t-1} & \dots & a_n^{t-1} \end{pmatrix}$$

$$Z = \begin{pmatrix} g(a_1)^{-1} & & & & 0 \\ & g(a_2)^{-1} & & & \\ & & g(a_3)^{-1} & & \\ & & & \ddots & \\ 0 & & & & g(a_n)^{-1} \end{pmatrix}$$

Each element of $\mathbb{F}_{2^m}$ consists of m bits. The check matrix $H \in \mathbb{F}_2^{mt \times n}$ ($mt = n-k$) is formed by splitting each column in H_{2^m} into m columns and applying a Gaussian elimination to reduce it to canonical (row echolon) form.

The canonical check matrix H is then transformed into systematic form $H_s = (E_{n-k}|A)$ where E_{n-k} is the identity-matrix of $\mathbb{F}_2^{(n-k)\times(n-k)}$. Using the systematic form the generator matrix can easily be created: $G_s = (A^T| - E_k)$. By applying the inverse of the transformation used to create H_s G_s is transformed into the generator matrix G [Ber11].

2.1.4 Encryption

The encryption of a cleartext is done by splitting the cleartext in blocks of length k and then multiplying each block with the generator matrix $\hat{G}$ of the public key. The result is a codeword with length n of the disguised Goppa Code. This codeword is xored with randomly selected error e that has a Hamming weight of t or less forming a block of the crypttext.

$$\{0,1\}^k \mapsto \{0,1\}^n$$
$$m \to m\hat{G} \oplus e$$

2.1.5 Decryption

The decryption of a cryptext is done by splitting the crypttext in blocks of length n. Each block is first multiplied with the the inverse of the permutation P. The added error is then removed using the

t-bounded decoder d. And the scrambled plaintext of length k is recovered by multiplying with G^{-1} Finally the cleartext is recovered by multiplication with the inverse scrambler matrix.

$$\begin{aligned} \{0,1\}^n &\mapsto 0,1^k \\ c &\to d(cP^{-1})G^{-1}S^{-1} \end{aligned}$$

2.1.6 Proof of Decryption

From encryption we know that $c = m\hat{G} + e = mSGP + e$. Therefore:

$$\begin{aligned} d(cP^{-1})G^{-1}S^{-1} &= d((mSGP + e)P^{-1})G^{-1}S^{-1} \\ &= d(mSG + eP^{-1})G^{-1}S^{-1} \\ &= (mSG)G^{-1}S^{-1} \\ &= m \end{aligned}$$

The t-bounded-decoder d removes t errors. It obviously does not matter whether these are the t errors chosen during encryption or the t errors given by permutating the chosen error vector.

2.2 HyMES

McEliece has some advantages over other public key encryption schemes, for example the encryption and decryption are faster than the widely spread RSA algorithm. However some drawbacks exist, a large key (>100kb for reasonable security parameters) and a lower information rate compared to RSA.
Nicolas Sendrier and Bhaskar Biswas addressed these problems with HyMES (Hybrid McEliece Encryption Scheme). They increased the information rate by encoding information in the error and reduced the public key size by using a generator matrix in systematic form (ID | R).

Encoding information into the error is done using a function $\varphi : \{0,1\}^l \rightarrow W_{n,t}$.
$W_{n,t}$ is a set of binary words with length n and hamming weight t.

2.2.1 Cryptographic security

A discussion of the security aspects of the changes done in HyMES can be found in Bhaskar Biswas PhD Thesis "Implementation Details of code-based cryptography" [Bis10]. Bhaskar Biswas proves that HyMES is a One Way Encryption scheme under the assumptions that Goppa codes are hard to decode and pseudo random. By using a semantically secure conversation layer on top of the encryption the scheme is therefore resistant to adaptive chosen ciphertext attacks (IND-CCA2) in the random oracle model, even without using a scrambler matrix like the McElice Cryptosystem. Please note that the semantically secure layer is not yet part of the C implementation.

2.2.2 Key Generation

First the parameters m, t and a binary Goppa code $\Gamma(L, g)$ are choosen. Then the check matrix and corresponding generator matrix in systematic form (ID | R) are calculated.

The public key is the tuple (R, t), the private key is a t-bounded decoder d.

2.2.3 Encryption

The encryption of a cleartext is done by splitting the cleartext in blocks of length k+l The first part of length k of each block is multiplied with the generator matrix G of the public key. Since G is in row echolon form (Id | R) this is identical to concatenating x with xR. The result is a codeword with length n of the Goppa Code. The second part with length l is used to create the error e with length n and Hamming weight t using the φ function. These parts are then xored together to form a block of the crypttext. $(\{0,1\}^k, \{0,1\}^l) \rightarrow \{0,1\}^n$
$(x, e) \rightarrow (x, xR) \oplus \varphi(e)$

2.2.4 Decryption

The decryption of a crypttext is done by splitting the crypttext in blocks of length n. A t-bounded decoder is then used to remove the introduced error. The resulting vector is truncated to the first k bits to form the first k bits of the cleartext. The second part of length l is the result of the φ^{-1} function called

with the error positions. The two parts are then concatenated to form a cleartext block of length k+l.
$0, 1^n \rightarrow (0, 1^k, 0, 1^l)$
$c \rightarrow trunc(d(c)), \varphi^{-1}(c \oplus d(c))$

2.2.5 Proof of Decryption

From encryption we know that $c = (x, xR) \oplus \varphi(e)$. Therefore:

$$
\begin{aligned}
& trunc(d(c)), \varphi^{-1}(c \oplus d(c)) \\
= & trunc(d((x, xR) \oplus \varphi(e))), \varphi^{-1}((x, xR) \oplus \varphi(e) \oplus d((x, xR) \oplus \varphi(e))) \\
= & trunc(x, xR), \varphi^{-1}((x, xR) \oplus \varphi(e) \oplus (x, xR)) \\
= & x, \varphi^{-1}(\varphi(e)) \\
= & x, e
\end{aligned}
$$

3 Preliminaries

This chapter describes two different (source) coding approaches. Both are the base for the final implemented coding algorithm in HyMES. On the one side we give an introduction into *Enumerative coding* which is used to produce a bijective mapping from a constant weight word onto a binary string: $\varphi^{-1}: W_{n,t} \rightarrow: \{0,1\}^l$. We will show that under given parameters required for HyMES, Enumerative coding itself is too slow to encode constant weight words. Hence Bhaskar Biswas and Nicolas Sendrier introduced a combination of several techniques to deal with this problem. Therefor we describe the second coding technique, namely *Arithmetic Coding*, which is mainly used for data compression (source coding). The key idea behind arithmetic coding is, that every symbol is encoded in the interval $[0,1]$ according its probability of occurrence.

3.1 Enumerative Coding

Enumerative coding solves the problem to find an information optimal bijective mapping between a set of constant weight words $W_{n,t}$ and an interval $\left[0, \binom{n}{t}\right[$ that can be written as binary words of the length $k = \lfloor log_2(\binom{n}{t} - 1) + 1 \rfloor$. We denote $W_{n,t}$ the set of all binary words of length n with the Hamming weight t. Hence Enumerative coding is a concrete implementation of $\varphi: \{0,1\}^l \rightarrow W_{n,t}$ (and its inverse $\varphi^{-1}: W_{n,t} \rightarrow \{0,1\}^l$) where l = k-1.

Enumerative coding:

$$\begin{aligned} \varphi^{-1}: \quad W_{n,t} \quad &\longrightarrow \left[0, \binom{n}{t}\right[\\ (i_0, i_1, \ldots, i_{t-1}) \quad &\longmapsto \binom{i_0}{1} + \binom{i_1}{2} + \cdots + \binom{i_{t-1}}{t} \end{aligned}$$

Where $(i_0, i_1, \ldots, i_{t-1})$ represent all ascending ordered non-zero positions in the word $w \in W_{n,t}$. In principal all possible combinational configurations are "enumerated", so that every possible combination is taken into account. The inverse process is nearly as fast and searches for the largest binomial coeffcient, substracts it and continues until all i values a revealed. The search for the largest binomial coefficient might be implemented with a binary search for small sizes of (n,t) or for example with the inversion formula given in Bernstein2009.

Computational costs:

As seen before, Enumerative coding offers a perfect mapping without any information loss (or technically minimal loss of $2^k - (\binom{n}{t} - 1)$ which is the reason why only k-1 bits could be used if an enumerative encoder is used as φ.) but it has drawbacks regarding the computational costs. Each computation of φ^{-1} requires computing of t binomial coefficients. E.g. parameters used in McEliece may be $n = 1024$. $t = 50$ or above. This might result in binomials in the size of $\binom{1024}{50} \approx 10^{84}$. The calculation of such large numbers is rather slow in comparison to the aimed efficiency gain in HyMES. Moreover it has to be done t-times. It is possible to calculate the the larger binomials in advance, but this also requires up to $log_2(\binom{1024}{50}) \approx 284$ bits per binomial, given typical HyMES parameters. Hence this tradeoff easily exceeds any CPU cache size and is especially inconvenient for embedded devices.

Conclusions on Enumerative Coding:

Enumerative coding seems to be suited for small constant weight words $W_{n,t}$, since it offers a perfect bijective mapping. However larger values of n, t result in too high computational costs which do not fit

the requirements in HyMES.

SOURCE: [OS09], [Cov73]

3.2 Arithmetic Coding

Arithmetic coding is a coding algorithm that converts an input stream of symbols to a single number. Under the assumption that the used model of the distribution of symbols in the input stream matches the reality, arithmetic coding can be proven to almost reach a perfect compression ratio, which is limited by the entropy of the data. Also given a model, arithmetic encoding achieves the best possible compression ratio and therefore is far better than algorithms using a bit pattern to represent specific symbols. Furthermore the algorithm processes its input symbol-wise, can be implemented with finite precision integer arithmetic and runs in linear time with constant memory, which makes it a very popular choice in modern implementations.

In the following pages we will first show how encoding to real numbers is done, how an input stream is encoded as a sequence of bits and discuss the use of an adaptive model. Finally we will illustrate how adaptive arithmetic encoding is used in HyMES.

These pages are based on [Bod07].

3.2.1 Basic Definitions

Alphabet: A finite, nonempty set. $A = \{a_1, ..., a_m\}$ $|\mathrm{A}| = \mathrm{m} =$ Length or cardinality of A.
Symbol: The elements $a_i \in A$
Sequence: a series of symbols $S = (s_1, s_2, ...)$, $s_i \in A$, $|\mathrm{S}| = \mathrm{n} =$ Length of S
Probability: For a symbol a_i let $|S|_{a_i}$ be the frequency of a_i in S.

$$\begin{aligned} P(a_i) &:= \frac{|S|_{a_i}}{n},\ P(a_i) \in [0,1) \\ \sum_{i=1}^{m} P(a_i) &= 1 \end{aligned}$$

The interval is open ended as it would make no sense to encode a sequence of a single symbol.
Model: a mapping $M : A \to [0,1) : a_1 \to P_M(a_i)$
The probability used in arithmetic encoding might be estimated, i.e. because the input sequence is not fully known or calculating the exact probabilities is too expensive. Also the decoder has to use the same model as the encoder and if the model is based on the input sequence transmitting it together with the encoded number might be more expensive than using a well known model.

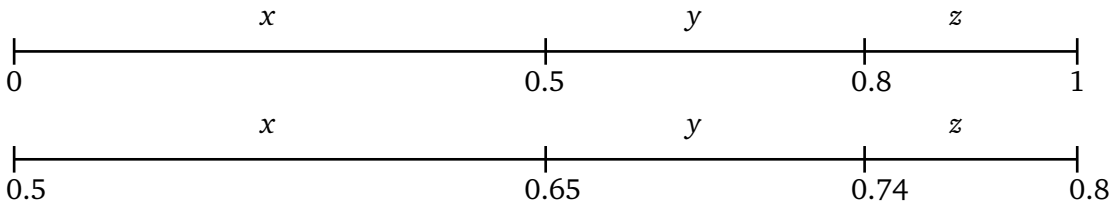
3.2.2 Encoding to real numbers

The amount of real numbers in the interval [0,1) is infinite. Therefore any given sequence can be mapped to one of them. This is achieved by partitioning the interval [0,1) according to the probabilities P_M:

$$[0,1) = [0, P_M(a_1)) + [0 + P_M(a_1), P_M(a_1) + P_M(a_2)) + ... + [P_M(a_1) + P_M(a_2) + ... + P_M(a_{m-1}), 1)$$

The first symbol s_1 can be mapped to any real number in its corresponding interval. This interval is then again partitioned according to the probabilities P_M. The sequence $s_1 s_2$ can be mapped to any real number in the partition corresponding to s_2 of this interval.

A graphical example might be easier to understand:
Given the alphabet $A = \{x, y, z\}$ and the model $M : P_M(x) = 0.5$, $P_M(y) = 0.3$, $P_M(z) = 0.2$ and the sequence yz can be encoded to any real number in the interval [0.74, 0.8).



3.2.3 Encoding as a sequence of bits

When using integer arithmetic we do not want to represent the probabilities as fractions. Instead we will use a model where every symbol a_i is mapped to two integers low_i and $high_i$. These values represent a partitioning of an interval $[low_0, high_m)$. We define $total :=$ size of this interval.
A model reflecting the probabilities $P(a_i)$ would be

$$low_1 = 0,\ low_i = \sum_{j=1}^{i-1} |S|_{a_j} \text{ for i} > 1 \text{ and } high_i = low_i + |S|_{a_i}.$$

Therefore $total = high_m = n$ and $high_i - low_i = n * P(a_i)$.
As mentioned a model based on the input sequence has some flaws: it needs to be transmitted to the decoder and it does not work for $n > 2^{29}$ when using 32 bit arithmetic, however it is sufficient for a short introduction to binary arithmetic encoding.

We will use an identical notation for the current interval where $mLow$ is the current lower bound and $mHigh$ the current upper bound. They are initialized with $mLow = 0$ and $mHigh = 7FFFFFFF$. Only 31 bits are used to prevent an overflow. An additional variable mStep is needed to partition the interval according to our model:

$$mStep = \lfloor (mHigh - mLow + 1)/total \rfloor$$

When a symbol a_i is encoded the boundaries are updated to

$$\begin{aligned} mHigh &= mLow + mStep * high_i - 1 \\ mLow &= mLow + mStep * low_i \end{aligned}$$

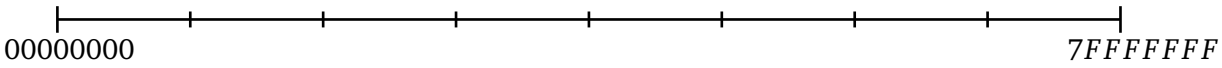
The range $mHigh - mLow + 1$ has to be greater than $total$, or $mStep$ would result in 0 and further encoding would not be possible. For a model with $total < 2^{29}$ this can be guaranteed by scaling the interval whenever it becomes smaller than one half of its initial size. Since the interval can only shrink this means we have one bit of information that will not change in the following iterations and can therefore be written to the output. However there are two different cases:
First: $mLow$ and $mHigh$ are in the same half. In this case the most significant bits of $mLow$ and $mHigh$ are identical and the MSB can be written to the output and shifted out of the boundaries. For $mLow$ a 0

is shifted in, for $mHigh$ a 1 because it is an open upper boundary.
Second: $mLow$ is in the second quarter of the initial interval, while $mHigh$ is in the third. The size of the interval can be scaled up, however we can not yet write a bit to the output, since we do not know if our final codeword will be in the first or the second half (= the second or third quarter before scaling). Instead we remember we did a second case scaling. Once a first case scaling is done again we can add the inverse of the MSB for every second case scaling we did.

Again, a graphical example might be helpful:
Given the alphabet $A = \{x, y, z\}$ and the model $M : low_x = 0, high_x = 3, low_y = 3, high_y = 6, low_z = 6, high_z = 8, total = 8$ and the sequence yz.

Initialization:

$mHigh$ = 7F FF FF FF
$mLow$ = 00 00 00 00
$mStep$ = ⌊ (7F FF FF FF - 0 + 1) / 8 ⌋ = 10 00 00 00

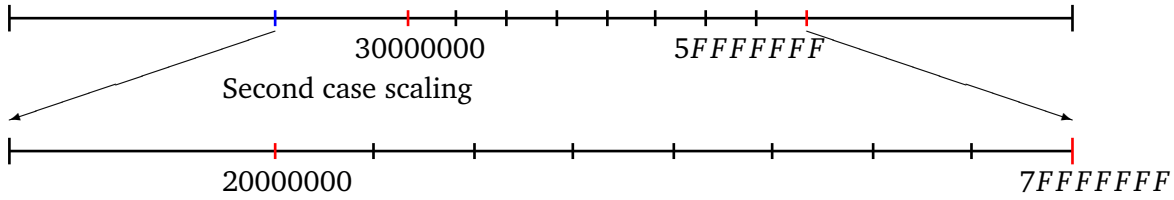


The first symbol is y:

$mHigh$ = 00 00 00 00 + 10 00 00 00 * 6 - 1 = 5F FF FF FF
$mLow$ = 00 00 00 00 + 10 00 00 00 * 3 = 30 00 00 00
$mStep$ = ⌊ (5F FF FF FF - 30 00 00 00 + 1) / 8 ⌋ = 06 00 00 00

Since $mLow$ is in the second and $mHigh$ in the third quadrant second case scaling is used.

$mHigh$ = (5F FF FF FF - 20 00 00 00) * 2 + 1 = 7F FF FF FF
$mLow$ = (30 00 00 00 - 20 00 00 00) * 2 = 20 00 00 00
$mStep$ = ⌊ (7F FF FF FF - 20 00 00 00 + 1) / 8 ⌋ = 0C 00 00 00



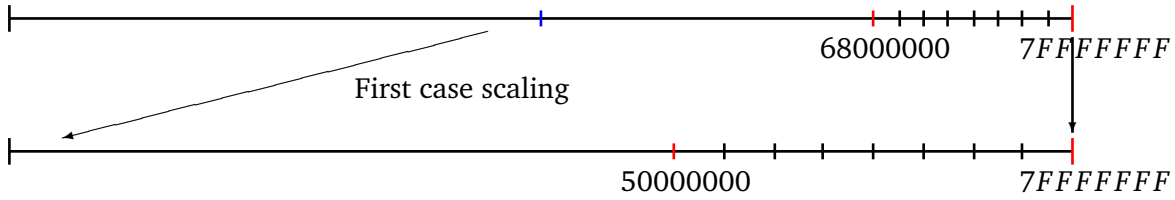
The second symbol is z:

$mHigh$ = 20 00 00 00 + 0C 00 00 00 * 8 - 1 = 7F FF FF FF
$mLow$ = 20 00 00 00 + 0C 00 00 00 * 6 = 68 00 00 00
$mStep$ = ⌊ (7F FF FF FF - 68 00 00 00 + 1) / 8 ⌋ = 03 00 00 00

Since both boundaries are in the same half first case scaling is used.

$mHigh$ = (7F FF FF FF - 40 00 00 00) * 2 + 1 = 7F FF FF FF
$mLow$ = (68 00 00 00 - 40 00 00 00) * 2 = 50 00 00 00
$mStep$ = ⌊ (7F FF FF FF - 50 00 00 00 + 1) / 8 ⌋ = 06 00 00 00

The MSB = 1 is written to the output.
For the second case scaling done before a 0 is now written as well.



As this is the end of the sequence we can use two more scalings so that the interval covers the whole space. The sequence yz is mapped to the code 1011.

3.2.4 Adaptive models

It is possible to change the used model, and the used alphabet, in every step of arithmetic encoding. The term *adaptive* means that the model adapts to the already encoded/decoded data. An adaptive model increases the information rate of the encoding by using knowledge about the structure of the encoded information. For example, when encoding a signed number base 10 one could assume a uniform distribution with the alphabet {+,-,0..9}. However an adaptive model that encodes the alphabet {+,-} in the first step and {0..9} in all later steps will provide a far better information rate.

4 Overview phi in HyMES

This chapters gives a first overview and works out the main components of the implemented coding method in HyMES. We will discuss the encoding and decoding together, since the main idea and procedure is almost the same. However this chapter does not refer to any part of the code.

HyMES φ function is very fast in comparison with Enumerative Coding and almost as information efficient as Enumerative Coding, the produced binary string has the length of $l = \lfloor log_2\binom{n}{t} \rfloor - 1$ bits. So the occurring question is, how does φ work?
The method implemented by Bhaskar Biswas and Nicolas Sendrier is based on three different components:

- The recursive Dichiotomic Model
- An adaptive Arithmetic Encoder
- An Enumerative Encoder

In our opinion, those components may be ordered in two different layers. These layers are not strict, but help to understand the principle idea behind the constant weight encoding. The first layer is a supporting structure, namely Arithmetic Coding, to store sequences of integers. Whereas the second layer is split into two different parts. The first step splits constant weight words in smaller pieces to encode those in the second step with an Enumerative Encoder. The sequence of the encoded pieces is then given to the first layer. We begin our overview with an introduction of layer one and the usage of an (adaptive) Arithmetic Coder.

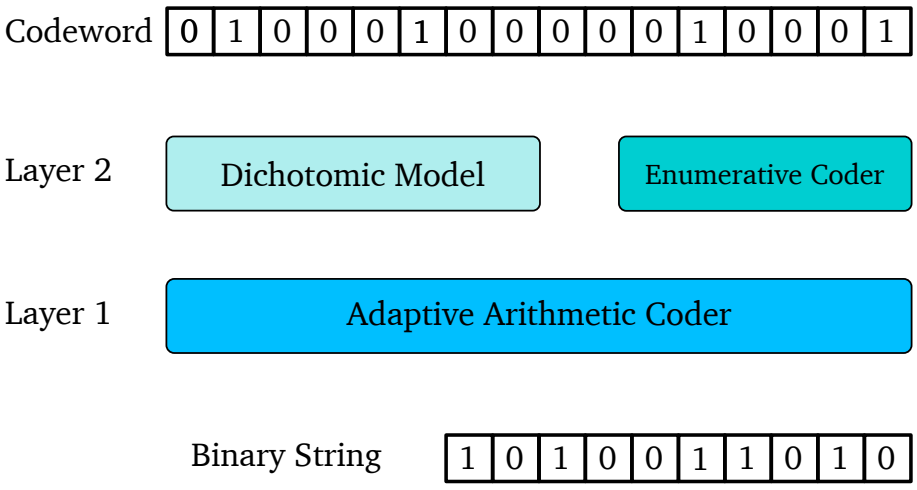


Figure 4.1: Overview of the three different components (Arithmetic Coder, Dichotomic Model and Enumerative Coder) in HyMES used to encode constant weight words into a binary string and vice-versa.

4.1 Layer 1: Adaptive Arithmetic Coder

The adaptive model used by HyMES is best described as the union of two distinct adaptive models, one using a distribution based on binomials, the other a uniform distribution. We will later discuss how a model is selected for each step of the Arithmetic Encoding when we discuss how Arithmetic Encoding is embedded in φ. For now we will consider the adaptive model as a lookup table from which the specific model is selected by a parameter.

The adaptive Arithmetic Encoder main purpose in HyMES is to store a sequence of (mainly uniform distributed) integers in an efficient way. To explain the purpose we consider the following example: We want to encode the sequence of four integers $< x_1, x_2, x_3, x_4 >$ and we know $x_1 \in X_1 = \{0,1,2,3,4\}, x_2 \in X_2 = \{3,4,5,6\}, x_3 \in X_3 = \{0,1,\ldots,8\}, x_4 \in X_4 = \{0,1,2\}$. If $|X_i| = 2^t, t \in \mathbb{N}$, it is obviously easily possible to encode $x_i \in X_i$ with $l = log_2(|X_i|)$ bits. For all other sizes of $|X_i|$ either a (complex) rule setting is required or each x_i has to be encoded with $l = \lceil log_2(|X_i|) \rceil$ bits, which results in a loss of information efficiency. Considering the example sequence $s_b =< 3,5,1,2 >$:

$$\begin{aligned} x_1 = 3 &= 011_b \\ x_2 = 5 &= 10_b \ (offset = 3) \\ x_3 = 1 &= 00001_b \\ x_4 = 2 &= 10_b \end{aligned}$$

The concatenated string $s_b = 011100000110$ requires $b = 12$ bits to be encoded. But only $5 \cdot 4 \cdot 9 \cdot 3 = 540$ different configurations for $< x_1, x_2, x_3, x_4 >$ exists, which may be stored in a more efficient way with $b \leq log_2(540) = 10$ bits. This problem is solved in HyMES with the usage of the Arithmetic Encoder, which ensures perfect information efficiency in a real number implementation. However as seen in Section 3.2, a good implementation on bit-level just introduces a rather small information loss. Hence the sizes of the different X_i's does not matter anymore. Moreover the following sections will show that the range of each X_i of the integers in the sequence produced by the adaptive Arithmetic Decoder is expected by the encoder, so no further information on the sequence itself has to be stored. Summing up, the main purpose of the Arithmetic Encoding in HyMES is not to compress data in the sense of information theory, in fact, its purpose is to avoid losing any bits by converting the sequence of integers into a bit string representation. The Arithmetic Coder is not able to compress anything, since up to know we assumed uniform distributed integers $x_i \in X_i$. Later we will see that the Arithmetic Coding is also used for non-uniform distributions in HyMES.
In conclusion the Arithmetic Coder represents a first layer in the coding process to efficiently encode a sequence integers given by the second layer. Thus we will further regard the Arithmetic Coder as a black-box for reading and writing a sequence of integers.

4.2 Layer 2: Dichotomic Model and Enumerative Coder

As mentioned before the main idea in HyMES constant weight encoding is to split the constant weight words into several chunks which are small enough to be efficiently encoded with an Enumerative Encoder. Now given the possibility to store sequences of integers, HyMES is able to save and restore these sequence without any loss in information efficiency. HyMES splits the constant weight word in a dynamic way to keep information efficiency and encoding speed. This has the consequence that not only the chunks have to be saved but also the position and size of the chunks. The following paragraphs will discuss these processes in detail.

4.2.1 Chunking — Dichotomic Model

Bhaskar Biswas and Nicolas Sendrier describe their Dichotomic Model as follows[Bis10]: Each constant weight word $x = (x^L \,||\, x^R) \in W_{n,t}$ with the length $n = 2^m$ can be recursively split into two words x^L and x^R with the length $n/2 = 2^{m-1}$ and the Hamming weights $i = w_H(x^L)$ and thereby be encoded as a finite sequence of integers:

$$F_{m,t}(x) = \left\{ \begin{matrix} nil & if\, t \in \{0, 2^m\} \\ i, F_{m-1,i}(x^L), F_{m-1,t-i}(x^R) & else \end{matrix} \right\} \text{where a,nil = nil,a = a}$$

Now the recursive produced sequence of Hamming weights is encoded with an adaptive Arithmetic Encoder according the distribution of i, which is given by

$$Prob(i) = \frac{\binom{n/2}{i}\binom{n/2}{t-i}}{\binom{n}{t}} \,.$$

For the moment we will ignore the distribution and Arithmetic Encoder and explain the idea of the Dichotomic Model along an example. Given the constant weight word

$$cw = 01110001$$

the recursive encoder splits this word into two halves

$$cw_l = 0111$$
$$cw_r = 0001$$

and begins not only to recursively work on both halves but also starts to write the output sequence beginning with the Hamming weight of the left half: $w_H(cw_l) = w_H(0111) = 3$

$$s =< 3 >$$

Followed by the next recursive step, splitting cw_l and adding $w_H(cw_{ll}) = w_H(01) = 1$ to the sequence

$$s =< 3, 1 > \,.$$

The recursion ends when the chunks are small enough, in our example with the size of one bit. So the last integer in the produced sequence will be $w_H(cw_{rrl}) = w_H(cw_{rrr} = 1)$, cw_{rrl} and cw_{rrr} describe the

same bit. At the end of the recursion a sequence with the Hamming weights of the halves is produced. These sequence may also regarded as a traversed tree and is hence a representation of a tree.

Looking at the code of HyMES a slightly different version of the Dichotomic Model than the above described is applied. The key idea is the same, the codeword is split into the left and right side and the Hamming weight is stored. However the recursive process stops dynamically. It stops iff $\binom{n}{t} < 2^{32}$ or $n <= 1$ or $t <= 1$ where n is the size in bits of the actual part of the codeword and t the Hamming weight of the same part. As seen earlier, the sequence encodes a tree with Hamming weight as nodes and leafs. Since the recursion may stop on different levels, the concrete HyMES implementation constructs a tree where the leafs may be on different heights. However this non perfect tree is still revertible in the decoding process by checking the same size boundaries as during the encoding. Finally the sequenced tree is written out to the binary output string as the first part of the encoding process. Summing up, the first encoding step from constant weight word to binary string, is to split the word into different chunks and write the Hamming weight of those chunks into the beginning of the binary output string.

Coming back to the Arithmetic Encoder, the sequenced tree is written out through the Arithmetic Encoder. Opposing the introduction of the Arithmetic Encoder in this chapter, this time the *Binomial Model* instead of the uniform distribution is used. The reason for this decision is the decoding process. A given binary string should produce uniform distributed constant weight word. Hence the Hemming weights should be naturally distributed. And exactly this distribution is used to encode the sequenced tree.

Binomial Model:

Parameter: A distribution $D_{m,t}$ = (min $\in \mathbb{N}$, max $\in \mathbb{N}$, P $\in \mathbb{R}^{max-min+1}$)
Alphabet: $A = \{a \in \mathbb{N} \mid min \leq a \leq max\}$
Model: $\forall a \in A\ low_a = P_a\ high_a = P_{a+1}$

The distributions $D_{m,t}$ are precomputed according the requirements above for all values m and t needed by HyMES with:

$$P_{min} = 0, P_a = \frac{\sum_{j=min}^{a} \binom{2^{m-1}}{j}\binom{2^{m-1}}{t-j}}{\sum_{j=min}^{max} \binom{2^{m-1}}{j}\binom{2^{m-1}}{t-j}}.$$

min and *max* are chosen to guarantee that no information is lost when using finite precision integer arithmetic.
It should be noted that with $min = 0$ and $max = t$ the difference $P_a - P_{a-1} = \frac{\binom{2^{m-1}}{a}\binom{2^{m-1}}{t-a}}{\binom{2^m}{t}}$ is the probability mentioned in the description of the Dichiotomic Model. This means that this part of the Arithmetic Encoder expects constant weighted words with at least *min* and at most *max* ones in the first half.

The use of min and max seem to be a problem at first, since they mean that not every possible word in $W_{n,t}$ can be encoded. The solution to this problem is easily seen when we look how the encoding is used in HyMES: A constant weight word is constructed when we encrypt a message. Since we have to expect every possible binary sequence of length l we need a set $W_{n,t}$ that has at least 2^l members: $2^l \leq \binom{n}{t}$. HyMES selects the length l from the constant parameters m and t with $l = \lfloor log_2 \binom{2^m}{t} \rfloor$. The precomputed min and max values therefore cut $\binom{2^m}{t} - 2^l$ constant weight words off.

4.2.2 Encoding Chunks — Enumerative Encoding

Now given the tree of Hamming weights and so also the chunking, the final step of the encoding may take place. Every leaf/chunk of the constant weight word is encoded with an Enumerative Encoder and written out in the order of the sequenced tree to the binary string. This time the Uniform Model (See 4.2.2) is used to write out integers. Hence the Arithmetic Encoder is only used to store a sequence of integers, as described above. Now is this process revertible during decoding? The answer is: Yes it is. The reason is that Hamming weight and size of the chunks (or layout of the tree) is decoded as first. Given these information it is possible to get the max value for every chunk in the sequence. By using the max value the Arithmetic Decoder is able to restore the integers and hence an Enumerative Decoder is able to fully decode the chunk.

Uniform Model:

Parameter: $n \in \mathbb{N}$
Alphabet: $A = \{a \in \mathbb{N} \mid 0 \leq a < n\}$
Model: $\forall a \in A \; P(a) = \frac{1}{n}$

4.3 Summary: Overview

Encoding: The Encoding begins with the Dichiotomic Model, where the constant weight word is recursively split into smaller parts and a sequence of Hamming weights of those is constructed. This recursion stops when the size of the parts (chunks) is small enough to be efficiently encoded with an Enumerative Encoder. The produced sequence of Hamming weights is written out at the beginning of the output string by using an Arithmetic Encoder. The Arithmetic Encoder encodes the Hamming weights according a normal distribution in a constant weight word. In the final step, all chunks are sequentially encoded with an Enumerative Encoder and added to the binary string with the Arithmetic Encoder this time using a uniform distribution. Figure 4.3 illustrates a possible layout of a constructed binary string.

Decoding: The Decoding process is almost the same. Given a binary string, at first the tree/sequence of Hamming weights is read through the Arithmetic Decoder using a normal distribution of Hamming weights. After reading the tree, the Decoder knows the sizes of the chunks and sequentially reads those from the binary string using the Uniform Version of the Arithmetic Decoder. In the final step, all chunks are concatenated and the constant weight word is constructed.

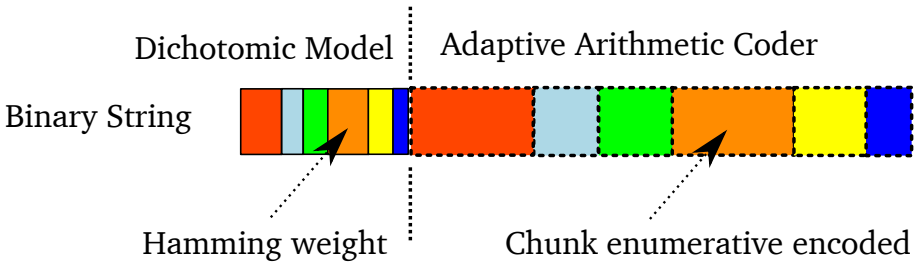


Figure 4.2: The layout and origin of every part of a binary output string constructed from a constant weight word in HyMES φ^{-1}.

5 Implementation Details

In this chapter we will discuss some details of the implementation next to code references. The key idea behind this chapter, is to give the reader the possibility to understand the C implementation[HyMES Sourcecode] of HyMES. We begin with the entry points in the source code for the coding process and give details about internal data structures and their representation. This should offer the reader the possibility to navigate and also modify the source code of HyMES. Then we will go over to the implemented optimizations and modifications of the principle idea. Those are optional and some may even be deactivated in the configuration of HyMES. Finally we will discuss the precompution which is also done to speed up the coding process in HyMES.

5.1 Source Code

The φ and φ^{-1} coding takes mainly place in *dicho.c*. Access to the Arithmetic Coder are done through *arith.c*, which itself is connected to a binary string buffer *buffer.c*. We will begin our discussion of the source code with the encoding process:

5.1.1 Encoding codeword to binary string

The entry point for φ^{-1} in the source code of HyMES is the function `dicho_cw2b()`. The codeword `cw` $\in W_{n,t}$ is represented by an integer array `int cw[t]` with the ordered positions of the errors in the codeword. Example: Given an constant weight word with $n = 8$ $t = 3$ errors, the size of the array `cw[]` will be 3:

$$\begin{aligned} cw &= 00101010_b \\ \texttt{cw[0]} &= 2 \\ \texttt{cw[1]} &= 4 \\ \texttt{cw[2]} &= 6 \end{aligned}$$

Besides some optimizations (see Section 5.2) `dicho_cw2b()` initializes the Arithmetic Encoder and aligns uneven message bits. The $\varphi^{-1}-$encoding algorithm itself starts with a function call to `dicho()` with only three parameters. These are the codeword `cw[]`, the pointer to the Arithmetic Encoder and a pointer to precomputed distributions of Hamming weights.
As described in Chapter 4, the encoding starts with splitting the codeword into different chunks. Therefor the recursive function `dicho_rec()` is called, which does the explained recursive splitting according the Dichiotomic Model. `dicho_rec()` checks if the actual chunk, which is the whole codeword on the first call, is small enough to be encoded with an Enumerative Encoder. Therefor function `is_leaf()` is called, which determines if $\binom{n}{t}$ fits into a word with the lengths of 32 bits. If this is not the case, the codeword is split into two halves, `dicho_rec()` is recursively called on both halves in the order left to right and the Hamming weight of the left node is encoded with the Arithmetic Encoder. If a leaf is found, the recursion stops at this chunk and the enumerative encoded value (`cw_coder(chunk)`) of this chunk is attached to a global linked list (`liste_t liste_todo`) together with theoretical maximal possible value. Given a chunk with length n and hamming weight t, the theoretical maximum an Enumerative Encoder

Table 5.1: Encoding functions in HyMES and their purpose

Function	Purpose
`dicho_cw2b()`	Entrypoint for the encoding process
`dicho()`	The actual φ^{-1} encoding takes place here
`dicho_rec()`	Dichiometic Model, builds Hamming weight tree
`cw_coder()`	Enumerative Coder
`coder()`	Arithmetic Coder

can output is $\binom{n}{t} - 1$.
When `dicho_rec()` reaches the last leaf, the control flow returns to `dicho()` which continues and starts to work through the list `liste_t liste_todo`. Important at this point is, that every information about the distribution of the nodes (Hamming weights) is already encoded and written out to the binary string. Now `dicho()` iterates over the list in the same order as the prior written tree and encodes every value of the chunk with a uniform distribution and the Arithmetic Encoder. Thus, the mentioned maximum value is needed to encoded the actual value of the chunk. Finally the buffer and Arithmetic Encoder flush out to disk and the φ^{-1}-encoding process is finished. Table 5.1.1 describes the different coding functions, which all have a different purpose but a similar naming.

5.1.2 Decoding binary string to codeword

The decoding process in HyMES is the reversed encoding process. The entry point for φ in the source code of HyMES is the function `dicho_b2cw()`. This function expects a message as a binary string to decode and also expects the parameters for the constant weight word. Ignoring the later discussed optimizations, `dicho_b2cw()` initializes the Arithmetic Decoder (`decoder()`) with the message as binary input string. As soon as the Arithmetic Decoder is set up, the inverse Dichotomic algorithm is called `dicho_inv()`.
`dicho_inv()` expects the Arithmetic Encoder and a precomputed distribution as input and an allocated `cw[t]` array as output. Given those parameters it begins to decode the binary string by calling the recursive `dichoinv_rec()` function. These function reconstructs the chunk sizes out of the binary string, by following the same Dichiotomic Model as in the encoding process. This approach may demand more further explanation. As described in Chapter 4, the Dichiotomic Model dynamically creates a tree of chunks, which are small enough to be decoded through an Enumerative Decoder. Hence the φ-decoder in `dichoinv_rec()` reverse engineers the codeword by reading the Hamming weights out of the binary string. The decoder begins with the full size constant weight word, even so he has no details about its appearance at the moment. So the decoder begins by splitting it into two halves and reads the Hamming weight of the left half out of the Arithmetic Encoder, which was initialized with the binary string. The Arithmetic Encoder uses the randomly and uniformly precomputed distribution for the total given length and Hamming weight of the whole codeword. Now having the weight of the left half, the decoder just derives the weight for the right half by subtracting the left from the total Hamming weight.
This procedure repeats itself recursively until the size of the chunks is small enough (`is_leaf()` `evaluate to true`) to be efficiently decoded with an Enumerative Decoder. Similar to the encoding process, the `dichoinv_rec()` functions creates a `liste_t liste_todo` with position, size and Hamming weight of every chunk. When the recursive process is finished the `dichoinv()` function continues by iterating over the list and decoding every chunk. This is done by combining the information stored in the list with the Arithmetic Decoder. The Arithmetic Decoder decodes according the Uniform Model every chunk and the `dichoinv()` runs the Enumerative Decoder to construct a constant weight chunk out of

Table 5.2: Decoding functions in HyMES and their purpose

Function	Purpose
`dicho_b2cw()`	Entrypoint for the decoding process
`dichoinv()`	The actual φ decoding takes place here
`dichoinv_rec()`	Dichiometic Model, builds Hamming weight tree
`cw_decoder()`	Enumerative Decoder
`decoder()`	Arithmetic Decoder

the decoded value. Since its final position in the constant weight word is known (noted in the list), the `dichoinv()` function is able to determine the exact error positions for every chunk. As soon as the end of the list is reached, the total constant weight word is constructed. Table 5.1.2 sums up the purposes of every decoding function.

5.1.3 Complex Data Structures

`liste_t liste_todo`

Table 5.3: Structure of `liste_t liste_todo`

Function	Purpose
`liste_t element`	Pointer to element(s) in codeword
`int nombre`	The Hamming weight of the chunk
`int pos`	Offset in Codeword
`int valeur`	Value to be encoded (only Encoding)
`int maximum`	The chunks maximal value which can be encoded with an Enumerative Encoder
`int taille`	Size of chunk to base 2

`precomp_t cwdata`

Table 5.4: Structure of `precomp_t`

Function	Purpose
`int m`	Codeword size ($n = 2^m$) used in the precomputed distributions
`int t`	Hamming weight used in the precomputed distributions
`int real_m`	Actual size used in coding, without *Reduction Optimization*
`int real_t`	Actual Hamming weight used in coding,without *Reduction Optimization*
`int offset`	Offset used for accessing distributions, saves space
`distrib_t ** distrib`	Precalculated distribution of Hamming weights, accessed by `[m][t]`
`leaf_info_t ** leaf_info`	Splitting Information for *Enumerative Cleartext Optimization*

5.2 Optimization

Even so the given description about the φ implementation in HyMES points out the main coding process, Bhaskar Biswas and Nicolas Sendrier implemented several performance optimization which modify the original idea and improve the coding speed. Those optimizations are tightly coupled with the main idea, but we split them from the main idea to reduce the complexity of the main idea. In this section we want to discuss the four main optimization which are necessary to get a better understanding of the HyMES source code. These are *Inversion*, *Reduction*, *Enumerative Cleartext* and *Precomputation*. We gave those optimization names, which may not occur in the actual code. We will begin with an overview of the *Inversion* optimization.

5.2.1 Inversion

Inversion is a powerful and often used optimization in HyMES. Whenever the Hamming weight t of the current codeword is greater than half of its length n the codeword is inverted. This behavior can be observed on many different levels of the encoding and decoding process. Since the internal representation of a codeword is based on error position and every error position increases the computational cost for the enumerative encoding this inversion is a clever idea. To give an example, a codeword $cw = 01101111_b$ will be inverted to $cw_{inv} = 10010000_b$ to increase the encoding performance. But the occurring question now is, does this approach change the output value. Yes of course is does, but important is that the Hamming weights for every chunk and also for the total codeword are known, so the decoder is able to revert the inversion on every level of the coding process.

Looking at the actual code implementation, there are mainly two different versions of the inverting necessary. One is realized in the `dicho_cw2b()` and `dicho_b2cw()` function. Here the total codeword will be inverted. Since this part of the code is the beginning of the encoding and decoding process, only the initial parameters have an influence. If $t > n/2$ an new codeword with $t_{new} = n - t$ errors is allocated and filled with the positions of all zeros in the original codeword and vice-versa.
The other implementation takes place in the recursive functions `dicho_rec()` and `dichoinv_rec()`. Here every chunk with $t > n/2$ is inverted to a new chunk and the function calls itself to run on the inverted chunk to improve the coding performance. The decoding process works in the same way and knows when to invert because all Hamming weights are known.
In conclusion the inversion is a very clever way to improve the coding performance without any loss in information efficiency.

5.2.2 Reduction

In the beginning of the encoding `dicho_cw2b()` function and in the end of the decoding function `dicho_b2cw()` another optimization takes place. Looking at the source code a variable

```
int reduc = p.m - m
```

is important to mention. `p.m` is the chosen security parameter and in the default setup `reduc` becomes 2. Now the idea behind the *Reduction Optimization* is to run the described φ^{-1}-encoding (and decoding) process on a shortened codeword and hence shortened binary string by extending the codeword with cleartext bits.
The least significant `reduc` bits are removed from each error position and directly copied to the binary

output string. After shifting the error positions, the dichotomic encoding *dichio()* takes place on a codeword of the length $2^{m-reduc}$ which is $2^{reduc} = 4$ times smaller than we expected. Figure 5.2.2 explains this behavior along an example:

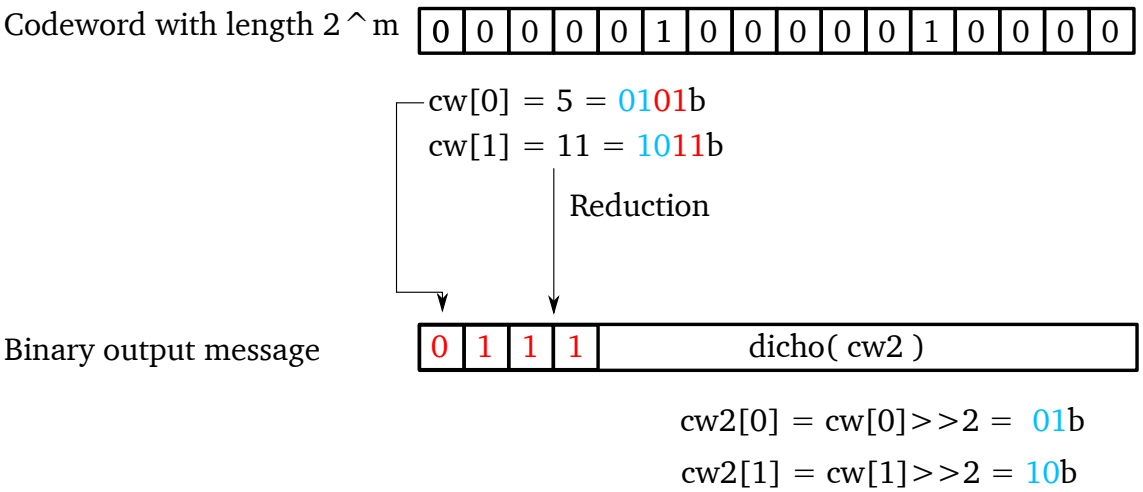


Figure 5.1: Reduction of an exemplary codeword in dichio_cw2b()

Given the codeword

$$\begin{aligned} cw_2^{16} &= 0000\ 0100\ 0001\ 0000_b \\ \texttt{cw[0]} &= 5 \\ \texttt{cw[1]} &= 11 \end{aligned}$$

and reduction parameter of

$$\texttt{reduc} = 2$$

the last `reduc = 2` bits of every error position:

$$\begin{aligned} \texttt{cw[0]} &= 01\,|\,\mathbf{01}_b \\ \texttt{cw[1]} &= 10\,|\,\mathbf{11}_b \end{aligned}$$

are directly copied into the binary output string. Now the φ^{-1} encoding starts on a four times shortend codeword:

$$\begin{aligned} cw2_2^4 &= 1100_b \\ \texttt{cw2[0]} &= 01_b \\ \texttt{cw2[1]} &= 10_b \end{aligned}$$

This optimization is obviously reversed during the decoding process by running the φ decoding on a shortened binary string and then adding the cleartext bits to the error positions at the end of the

decoding process. And it is also good to see that this approach improves the coding performance, since copying bits is rather cheap compared the whole coding process. The only problem which may occur is how this optimization of the coding procedure has an influence on the information efficiency and security. In principle the number of bits saved in the codeword reduces from

$$length = \left\lfloor log_2 \left(\binom{2^m}{t} \right) \right\rfloor$$

to

$$length = \left\lfloor log_2 \left(\binom{2^{m-reduc}}{t} \right) \right\rfloor + t \cdot reduc \ .$$

Since we are only reverse engineering the code, we just can make really vague assumptions. But given useful parameters this optimization seems legit and reasonable, but unfortunately we are unable to give a founded proof.

5.2.3 Enumerative Cleartext

We want to discuss another optimization which is more hidden inside the source code and we will further refer to it as *Enumerative Cleartext*. This optimization is located within the functions `dichio()` and `dichoinv()` and the intension is to reduce the workload of the Arithmetic Encoder. Every chunk is encoded with an Enumerative Encoder to a binary string/value. And as explained this value is written to the final output buffer by encoding it through the Arithmetic Encoder. Now the key idea behind the *Enumerative Cleartext* is to split this chunk's binary string into two parts. The first part of the chunk's binary string is encoded with the Arithmetic Encoder, whereas the second one is directly copied into the output buffer. Of course the lengths has to be reasonable large enough. This process is again invertible by applying the inverse approach in the same order and concatenating the results. This means that the decoder reads the first part of every chunk through the Arithmetic Decoder and concatenates the second part by reading cleartext from the buffer. Important to note is that the cleartext part of the binary string should have another location in the buffer than the Arithmetic encoded part, else the Arithmetic Encoder would not work information efficient enough. To ensure this property the sizes of the parts are precalculated for every combination of lengths and Hamming weights of chunks. Hence the encoder and decoder will use the same information to split the chunks.
Some more code references: The `int accel` parameter enables the optimization in `dicho()` and `dichoinv()`. Each list element in `liste_t liste_todo` (See Table 5.1.3) contains a parameter `int taille` which describes how many bits are directly copied. This parameter is set during the recursive Dichotomic process by accessing the precalculations. The maximal value which is used for the Enumerative Decoder is as well based on the precalculations and only describes the maximum for the first part of the binary chunk. If `accel` is not set, `dicho()` and `dichoinv()` will only use Enumerative Encoding and Decoding for both parts of the chunk where the first part uses the maximum value from the list, whereas the second part maximum is derived by $2^{taille} - 1$. Figure 5.2.3 illustrates this optimization along an graphical example.

5.2.4 Precomputation

As already mentioned the coding process HyMES makes use of precomputated values. When running the *./configure* script and compiling, HyMES calculates some values into its program code to improve

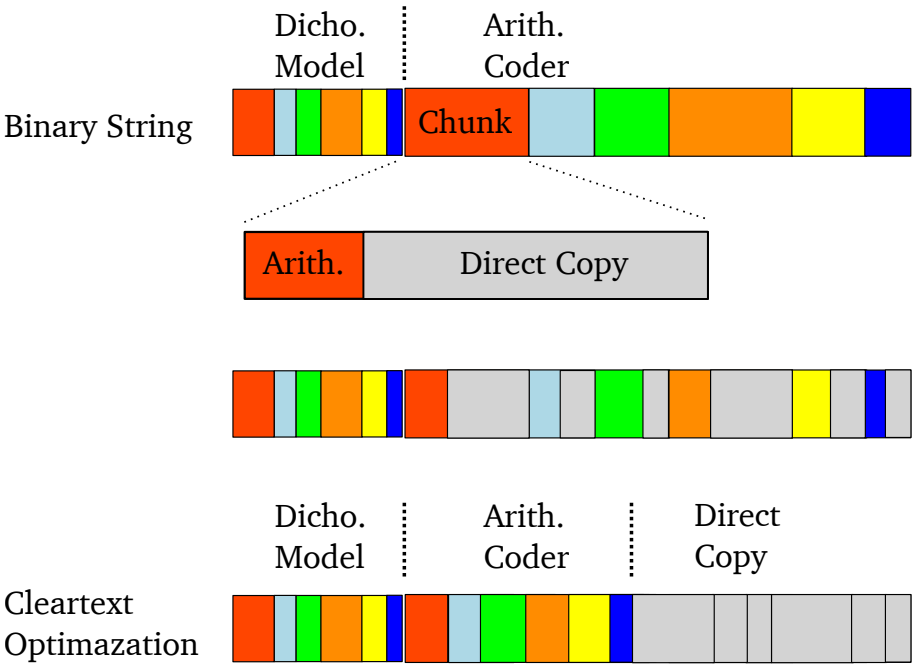


Figure 5.2: Cleartext Optimization explained graphical.

the coding performance for the given parameters. The file *precomp.c* is responsible for precalculating. It creates the file *cwdata.c* and fills it with a datastructure named `precomp_t cwdata`. Table 5.1.3 gives a detailed explanation of this datastructure, but in principle it is filled with distributions (`distrib_t`) of Hamming weights for the given parameters (n and t) and the splitting information required for the Enumerative Cleartext optimization (`leaf_info`). The distributions are exactly calculated as explained in Chapter 4, the only difference is, that some extreme probabilities will be disregarded, if they are not reasonable large enough to occur in a real setup. Hence a min and max value is added to avoid those probabilities of Hamming weights. Unfortunately we were not able to fully understand the precomputation of the `leaf_info` structure. We assume that the goal is not to get a large cleartext part but also to return values which are suitable for an Arithmetic Encoder based on Integers. But as mentioned these are only assumptions.
In conclusion the *Precomputation* speeds up the coding process by precalucating the distributions which are necessary for the Dichiotomic Model.

5.3 Information Efficiency

As stated before φ is used to encode a binary sequence of length l into a constant weight word of length n and weight t. There is a total of $\binom{n}{t}$ such words. We can therefore use at most $\lfloor log_2(\binom{n}{t} - 1) \rfloor$ bits.

As discussed in section 4.2.1 the possible constant weight words are limited by the values min and max in the precomputed distributions. These values are created to circumvent errors due to finite precision integer arethmetic, however from all possible combinations those with a minimal loss of information are chosen. In the computation of l the precomputed distributions are traversed to compute how many constant weight words are realy reachable. We will call the set of these words $W'_{n,t}$ and its size $|W'_{n,t}|$. The maximum possible length l is $l = \lfloor log_2(|W'_{n,t}| - 1) \rfloor$. Due to the limited precision the information loss increases with the size of n and t.

If HyMES is allowed to introduce a reduc factor the length is deliberately lowered by one bit and the maximal reduc factor is chosen so that

$$length - 1 = \lfloor log_2(W'_{m-reduc,t}) \rfloor + t \cdot reduc$$

There has one exception: if the total plaintext length k+l mod 8 = 0 the length is not reduced by 1, but an optimal reduc is still chosen. We assume that the gain in speed is just not worth destroying a blocksize in full bytes.

5.4 Security

As stated in 2.2.1 Bhaskar Biswas proves that HyMES is a One Way Encryption scheme under the assumptions that Goppa codes are hard to decode and pseudo random. It should be noted that φ is public. If someone is able to break the Goppa Code and therefore optains the error positions he can compute the cleartext. Biswas proof uses adversaries that assume a uniform distribution of e in the image of φ. This is a clever notion because with zero knowledge of the plaintext a uniform distribution has to be assumed and due to the bijectivity between the input and the image of φ a uniform distribution in the image is guaranteed.

6 Conclusions

We hope that we were able to give a detailed overview about the insights of HyMES φ coding function. We showed which different components are needed and how they interact to become a really efficient coding algorithm. Moreover we explained those components and pointed out their application within the practical implementation.
Looking back at the performance of HyMES φ-function, the encoding speed is quite impressive compared to algorithms fulfilling similar requirements. Those are either only able to do a variable length encoding or those are too slow for a practical hybrid McElliece implementation. The idea itself, to split up the problem in smaller chunks, which are efficiently encoded through an Enumerative Encoder and adaptive Arithmetic Encoder is really clever. But there are also some minor disadvantages to mention. On the one side the lack of documentation. In the whole thesis about HyMES [Bis10], the Dichiotomic Model only fills a half page and just gives a clue what the encoding is about, even so the encoding process is a large (the largest) and complex part in the practical implementation. Nevertheless in our opinion the code itself is very well written, shows a really good code quality and we could not find any indication of an implementation mistake. On the other side, we do not have seen any proof or reason about the information efficiency of φ, which shows that the decoding produces uniform distributed code words accomplish necessary security requirements. We just can make guesses about the information efficiency. Important to mention is that φ of HyMES decodes binary sequences of length $l = \lfloor log_2(|W'_{2^m-reduc,t}| - 1)\rfloor - 1$ whereas pure enumerative encoding would decode up to $l = \lfloor(log_2\binom{n}{t} - 1)\rfloor$ bits. This is a tradeof between resource needs and information rate. Furthermore the reduction factor optimization directly transfers clear text bits into the constant weight word, which may be a hook for an adversary. So in total we were just able to take assumptions regarding the security of φ and how the constant weight encoding is ensured.

However every detail and assumption we made is based on reverse engineering of several thousands lines of source code. We may have overseen parameters or have a different view on HyMES than the authors themselves. Summing up, it was not only a quite interesting but also challenging project for us, since we started without any background knowledge into the field of code-based cryptography. Thus we thankfully acknowledge Dr. Pierre-Louis Cayrel for offering this lab project to us and we want to thank him and especially Gerhard Hoffmann for all their advices during the course of the project.

References

[HyMES Sourcecode] Bhaskar Biswas, Nicolas Sendrier. The Hybrid McEliece Encryption Scheme. Project SECRET, INRIA, Rocquencourt, 2008
www-rocq.inria.fr/secret/CBCrypto/index.php?pg=hymes 5

[McE78] Robert J. McEliece. A public-key cryptosystem based on algebraic coding theory. In *Deep Space Network Progress Report 44* Pages 114-116, 1978 2.1

[OS09] Raphael Overbeck, Nicolas Sendrier. Code-based cryptography. In Johannes Buchmann et al. (editors) *Post-quantum cryptography*, Pages 95-145, 2009 2.1, 3.1

[Bis10] Bhaskar Biswas. Implementation details of code-based cryptography. PhD Thesis, L'École Polytechnique, 2010
pastel.archives-ouvertes.fr/docs/00/52/30/07/PDF/thesis.pdf 2.2.1, 4.2.1, 6

[Ber11] Björn Berezowski. Effiziente Implementierung des McEliece-Kryptosystems. Master Thesis, Hochschule Darmstadt, 2011
fbi.h-da.de/fileadmin/gruppen/FG-IT-Sicherheit/Publikationen/2011/Abschlussarbeiten/11_04_Berezowski-Masterarbeit.pdf" 2.1.3

[Cov73] Thomas M. Cover. Enumerative source encoding. IEEE Transactions on Information Theory, IT19(1), Pages 73-77, 1973 3.1

[Bod07] Eric Bodden, Malte Clasen, Joachim Kneis. Arithmetic coding revealed. Sable Technical Report No.2007-5, McGill University 3.2