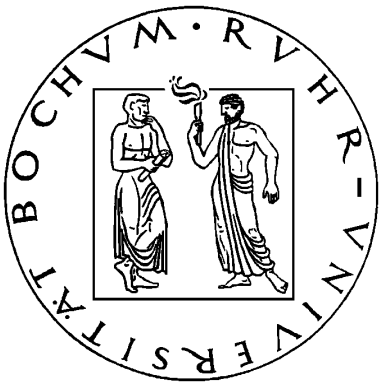
Practical Analysis of Information Set Decoding Algorithms

David Hobach
david.hobach@rub.de

Supervisor: M. Sc. Math. Alexander Meurer



Diploma Thesis

Chair for Cryptology and IT Security
Prof. Dr. Alexander May
Department of Electrical Engineering and Information Sciences
Ruhr-University Bochum

Für meine Großeltern.

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht sind und diese Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

Ort, Datum	Unterschrift (David Hobach)

Danksagungen

Ich möchte mich an dieser Stelle bei meinem Betreuer Alexander Meurer bedanken, der mir stets mit guten Ideen und Ratschlägen zur Seite stand.

Ein weiterer besonderer Dank gilt meinen Eltern, die mich während meines Lebens in Bildungsdingen immer unterstützt haben.

Contents

1 Introduction

The NP-hard problem of decoding random linear codes lies at the very heart of many constructions in code-based cryptography. Using information set decoding algorithms is the currently most promising way to solve instances of this problem. The effectiveness of these algorithms defines the security level at which many cryptosystems operate. Unfortunately many scientists only prove the asymptotic superiority of their information set decoding algorithm over previously developed algorithms, but do not provide exact formulas respecting polynomial factors to estimate the runtime complexity of their algorithms for concrete parameters in practice. Therefore it is hard for designers of code-based cryptosystems to choose durable parameters. Some scientists tried to improve this situation by developing generic lower bounds on the complexity of information set decoding; however such bounds tend to be beaten as soon as new ideas come up (e.g. the bound found in [26] was beaten in [19]).
In contrast our approach is to provide formulas that respect polynomial and – as far as possible – even constant factors for the currently most important information set decoding algorithms and hence help designers choose parameters, which are at least secure against these algorithms, at more ease. Concretely we present the following six information set decoding algorithms for binary finite fields in this thesis: Prange's algorithm [27], Lee-Brickell's algorithm [28], Stern's algorithm [22], Ball-Collision Decoding [19], FS-ISD [26] and BJMM [21]. We use a common model for all of them and apply various optimization techniques previously only seen in the context of Stern's algorithm to each of them to find precise formulas for their runtime and memory complexities. The formulas are implemented in the form of a C++ program for ease of use.
A particular focus is laid on the most recent BJMM algorithm: We investigate its so far unknown practical potential in comparison to the other algorithms, insofar that we exemplary compute the attack complexities for several relevant parameter sets of the McEliece cryptosystem [3] for all of them. Moreover we propose a conceptual change of the original version of the BJMM algorithm published in [21] that further improves the runtime of the algorithm by a polynomial factor. Our optimizations and improvements turn out to make the BJMM algorithm practically very relevant; in fact it seems to be the currently most efficient information set decoding algorithm not only asymptotically, but even in practice.

1.1 Overview

Section 2 explains the notation employed within this work and introduces important definitions and theorems required to understand the basics of code-based cryptography and in particular information set decoding. It also contains a short description of the McEliece cryptosystem [3] as a motivating example, which underlines the importance of information set decoding.

Section 3 first introduces a structure that all information set decoding algorithms share and discusses each of the aforementioned six algorithms separately and in detail afterwards: We provide a pseudocode description of each algorithm and explain the resulting runtime and memory complexities. Possible optimization techniques conclude each section.

In section 4 we display the results obtained from our C++ implementation of the runtime and memory complexity formulas developed in section 3 for four concrete McEliece parameter sets and different combinations of algorithmic optimizations. The section also features various tables meant as shorthand references to look up the aforementioned formulas.

After a short conclusion in section 5 follow several appendices with more detailed information for the interested reader. Most notably, appendix E contains the runtime and memory complexity formulas for the BJMM algorithm from [21], generalized to an arbitrary number of layers.

Literature references can be found at the end of this thesis.

1.2 Related Work

Bernstein, Lange and Peters describe most of the optimization techniques used within this thesis in the context of Stern's algorithm in [14] and propose some McEliece parameter sets for different security levels, that we put to the test in section 4.

Finiasz and Sendrier tried to provide lower bounds for the complexity of information set decoding in [26]. As proven in [19] these bounds do not always hold though. Nevertheless the paper [26] contains a description of the FS-ISD algorithm as well as other valuable ideas such as the application of the birthday paradox to information set decoding.

The paper [19] introduces the Ball-Collision Decoding algorithm and is one of the few that mention exact formulas for the computational complexity of the algorithm in question apart from the asymptotic behaviour. Bernstein et al. even discuss possible optimizations for their algorithm and propose another lower bound for the complexity of information set decoding.

In [21] May et al. describe the BJMM algorithm – an information set decoding algorithm in a divide-and-conquer structure, which incorporates the idea of using *representations* of vectorial sums. They prove its asymptotic superiority over the other algorithms discussed in this work and give an useful overview of worst-case complexities for many algorithms. The question of the practical break-even point of the BJMM algorithm in comparison to other information set decoding algorithms was left open in [21], but is addressed in this thesis.

Some information set decoding algorithms have been generalized to work over $\mathbb{F}_q$ instead of just over binary finite fields. In particular C. Peters made an implementation quite similar to ours available in combination with her paper [23]. Unfortunately this implementation only covers Stern's algorithm, whereas our implementation covers all of the six algorithms mentioned in this work (over $\mathbb{F}_2$ though). The program implemented in combination with this thesis additionally enables the user to automatically find the optimal parameter sets in a brute-force approach.

The more classical information set decoding algorithms were introduced in [27], [28] and [22].

2 Preliminaries

2.1 Notation

In this work we use $\mathbb{F}_q$ to denote a finite field/Galois field of order q. Usually we fix $q = 2$, i.e. we work with binary fields.
By $\vec{v} \in \mathbb{F}_q^n$ we mean a *column* vector with n elements $(v_1, \ldots, v_n)^T, v_i \in \mathbb{F}_q \; \forall \; i$. Thereby A^T means the transpose matrix of a matrix A.
In general matrices are denoted by uppercase Latin letters and the notation $\mathbb{F}_q^{n \times m}$ means the set of all matrices (n rows, m columns) with entries from the finite field $\mathbb{F}_q$, e.g. $\mathbb{F}_2^{n \times m}$ is the set of all $n \times m$ binary matrices. Special cases are $0^{[n \times m]}$ and $\mathrm{id}^{[n]}$, which denote the $n \times m$ all-zero matrix and the $n \times n$ identity matrix, respectively.
For a matrix $A \in \mathbb{F}_q^{n \times m}$ with column vectors $\vec{a}_1, \ldots, \vec{a}_m \in \mathbb{F}_q^n$ we employ the notation $A = (A_1 \mid A_2)$, $A_1 \in \mathbb{F}_q^{n \times (m-s)}$, $A_2 \in \mathbb{F}_q^{n \times s}$ to show that the concatenation of the column vectors of the matrices A_1 and A_2 form the matrix A. It is useful to recall that for any two compatible matrices $A = (A_1 \mid A_2)$ and $B = (B_1 \mid B_2)$ with splits A_1, A_2, B_1, B_2 of appropriate sizes (A_i and B_i must have the same number of columns) we have $AB^T = A_1 B_1^T + A_2 B_2^T$.
Moreover, given a matrix $A \in \mathbb{F}_q^{n \times m}$, A_I means the matrix, which consists of the column vectors of A, that are indexed by a set $I \subseteq \{1, \ldots, n\}$. Similarly we define the vector consisting of the I-indexed entries of the column vector $\vec{v} \in \mathbb{F}_q^n$ as $\vec{v}_I$ and $\vec{v}_{[r]} \in \mathbb{F}_q^r$ means the vector $\vec{v} \in \mathbb{F}_q^n$ restricted to its first r entries ($I = \{1, \ldots, r\}$). For $\vec{v}_1 \in \mathbb{F}_q^n, \vec{v}_2 \in \mathbb{F}_q^m$ the function $\vec{v}_3 \leftarrow \mathit{prepend}(\vec{v}_1, \vec{v}_2)$ creates a vector $\vec{v}_3 \in \mathbb{F}_2^{n+m}$, which contains the entries of $\vec{v}_1$ at the top and those of $\vec{v}_2$ below. In contrast the function $\vec{v}_2 \leftarrow \mathit{remove}(\vec{v}_3, n)$ returns a vector $\vec{v}_2$ that contains the entries of $\vec{v}_3$ apart from its first $n \in \mathbb{N}$ entries.
We use $\mathrm{wt}(\vec{v})$ to denote the *hamming weight* of a vector $\vec{v} \in \mathbb{F}_q^n$, which is defined as the number of nonzero elements of $\vec{v}$. The *hamming distance* between two vectors $\vec{v}_1, \vec{v}_2 \in \mathbb{F}_q^n$ is defined as $\mathrm{dist}(\vec{v}_1, \vec{v}_2) := \mathrm{wt}(\vec{v}_1 - \vec{v}_2)$, i.e. the number of coordinates where the two vectors differ.
The notation $s \in_r S$ means that we choose an element s uniformly at random from a set S and $Pr[e]$ indicates the probability of the occurrence of an event e.
By $\mathit{time}\{do()[o_1, o_2, \ldots]\}$ we mean the average runtime of an algorithm or function do() using the optimizations $o_1, o_2, \ldots$, whereas $\mathit{mem}\{do()[o_1, o_2, \ldots]\}$ denotes the average memory consumption of that algorithm or function with the same optimizations applied. These measures are meant to be statistically indistinguishable from average values of sufficiently many runtime and memory consumption experiments with the algorithm/function do() and uniformly chosen input parameters. $\mathit{mem}\{D\}$ may also be used to denote the memory consumption imposed by a data structure D.
We might also employ the *soft-O* notation, an extension of the Landau notation. $f(n) = \tilde{\mathcal{O}}(g(n))$ means $f(n) = \mathcal{O}(g(n) \log^k(g(n)))$ for some $k \in \mathbb{R}^+$; so basically we ignore logarithmic factors of $f(n)$ (which can be arbitrary polynomials, if $f(n)$ is an exponential function). Note that $\tilde{\mathcal{O}}(2^n) \leq 2^{(1+\epsilon)n}$ for some $\epsilon > 0$ and sufficiently large n.

2.2 Elementary Definitions and Theorems

This section introduces basic concepts used in code-based cryptography and contains many of the mathematical definitions and theorems employed within this work. Readers familiar with those concepts can easily skip this section.

2.2.1 Linear Codes

In 1948 Claude Elwood Shannon published "A Mathematical Theory of Communication" [1], which describes the transmission of messages over noisy channels and is one of the foundations of information theory. The basic idea is to add redundant information to a message, so that a receiver can recover the message even if a limited number of errors occurs during transmission. Obviously both sender and receiver must use the same predefined method to transform and recover messages - a so-called *code*. The process of applying redundant information to messages is called *encoding*, the process of removing that redundant information (and possibly correcting errors) is called *decoding*. The term *codeword* is used to describe an encoded message.
Over the years many codes were developed that featured different efficiency or error-correcting capabilities. Codes with a common structure can be grouped into *families*. The probably most simple family is the family of linear codes:

Definition 2.2.1 (Linear Code)**.** *Denote by $\vec{m} \in \mathbb{F}_q^k$ a message and by $G \in \mathbb{F}_q^{k \times n}$ a matrix of rank k. Then the set $C := \{\vec{c} \in \mathbb{F}_q^n \mid \vec{c}^T = \vec{m}^T G,\ \vec{m} \in \mathbb{F}_q^k\}$ is called a linear code of length n and dimension k. G is called a generator matrix.*

Remark 2.2.1. *The definition is slightly different from the standard definition, which defines a linear code C as a set of row vectors. Since we declared all vectors to be column vectors in section 2.1 though, we adapted the definition. Actually with our definition we have $C := \{\vec{c} \in \mathbb{F}_q^n \mid \vec{c}^T = \vec{m}^T G,\ \vec{m} \in \mathbb{F}_q^k\} = \{G^T \vec{m} \mid \vec{m} \in \mathbb{F}_q^k\}$.*

Remark 2.2.2. *Regarding the rows of the generator matrix G of a linear code C, note that a codeword $\vec{c}$ is just a linear combination of the rows of G; the rows of G form a basis of a k-dimensional subspace of the vector space $\mathbb{F}_q^n$ (since G has full rank).*

Remark 2.2.3. *If $q = 2$, we speak of binary linear codes, otherwise of q-ary linear codes. In the context of linear codes you will also often see the definition of the notion code rate or information rate as $R := \frac{k}{n}$. The error rate is usually defined as $W := \frac{w}{n}$, if the linear code allows for at most w errors to be corrected.*

We call such codes linear, because for any codewords $\vec{c}_1, \vec{c}_2 \in C$ we have: $\vec{c}_1 + \vec{c}_2 \in C$. This directly follows from the linearity of matrix multiplication and gives us an inefficient method to check whether a given code $\tilde{C}$ is indeed linear.
The minimum distance $d := \min\{\mathrm{dist}(\vec{c}_1, \vec{c}_2) \mid \vec{c}_1, \vec{c}_2 \in C, \vec{c}_1 \neq \vec{c}_2\}$ between any two codewords $\vec{c}_1, \vec{c}_2$ of such a code C leads to the usual notion of a *[n,k,d]-code* (although the minimum distance is sometimes left out). A generator matrix G of a linear code encodes messages of length k into codewords of length n. Obviously the receiver can correct at most $\lfloor \frac{d-1}{2} \rfloor$ errors in a unique way. There exist algorithms that try to correct more than $\lfloor \frac{d-1}{2} \rfloor$ errors in a codeword $\vec{c}$, but these just output lists of codewords close to $\vec{c}$ (so-called *list-decoding algorithms*). If that list contains more than one element, it remains a task for the receiver to decide which codeword was probably sent. In this work however we focus on linear codes with *error-correcting capability* $w \leq \lfloor \frac{d-1}{2} \rfloor$.
Decoding a linear code is usually done in the following way: The receiver of an encoded message may assume that some errors $\vec{e} \in \mathbb{F}_q^n$ might have occurred during transmission of the codeword $\vec{c}$ and thus expects to possess something in the form $\vec{y} = \vec{c} + \vec{e}$. Additionally he must assume that the number of possible errors is limited to $\mathrm{wt}(\vec{e}) \leq w \leq \lfloor \frac{d-1}{2} \rfloor$. Knowing the generator matrix G of the linear code C as well as a decoding algorithm $\mathrm{decode}_G()$, which takes $\vec{y}$ as input, he uses $\mathrm{decode}_G()$ to remove the errors $\vec{e}$ from the codeword $\vec{c}$ and retrieve $\vec{m}$ afterwards. Formally we define:

Definition 2.2.2 ($\text{decode}_G()$). *Given an [n, k, d] code C with generator matrix G, a codeword $\vec{c} = G^T\vec{m}$ and a vector $\vec{y} = \vec{c} + \vec{e}$ with $wt(\vec{e}) \leq w \leq \lfloor \frac{d-1}{2} \rfloor$, the algorithm $decode_G()$ takes $\vec{y}$ as input and outputs the message $\vec{m}$ in polynomial time (in n). It is parameterised by the generator matrix G.*

Usually such algorithms work for certain classes of linear codes and not just for one specific generator matrix G. However they heavily depend on the generator matrix that was used during the encoding process. An algorithm $\text{decode}_G()$ that works for any linear code in polynomial time (in n) is not known so far. In fact the general problem was proven to be NP-hard [4].
Nevertheless it is interesting to analyse, which linear codes have common properties that might be useful for decoding. As the main problem of an algorithm $\text{decode}_G()$ lies in correcting the errors in at most w positions (after having corrected the errors, retrieving $\vec{m}$ from $\vec{c}$ is trivial) and the capability to correct errors usually depends on the distance $\text{dist}(\vec{c_1}, \vec{c_2})$ between any two codewords $\vec{c}_1, \vec{c}_2 \in C$ only, it makes sense to define the notion of equivalent codes as follows:

Definition 2.2.3 (Equivalent Codes). *Two linear [n, k] codes C_1, C_2 are equivalent, if and only if there exists a bijective mapping function $f : \mathbb{F}_q^n \to \mathbb{F}_q^n$, s.th. $\forall\ \vec{c}_{1,1}, \vec{c}_{1,2} \in C_1$ there exist mappings $f(\vec{c}_{1,1}), f(\vec{c}_{1,2}) \in C_2$ with $dist(\vec{c}_{1,1}, \vec{c}_{1,2}) = dist(f(\vec{c}_{1,1}), f(\vec{c}_{1,2}))$.*

Equivalent codes have several interesting properties:

Lemma 2.2.1. *Given a linear code C_1 with a generator matrix $G_1 \in \mathbb{F}_q^{k\times n}$ and an equivalent linear code C_2 with a generator matrix G_2 the following statements hold:*

1. *C_1 is a linear [n,k,d]-code $\Leftrightarrow$ C_2 is a linear [n,k,d]-code*
2. *G_2 can be generated from G_1 using the following matrix operations:*
   (a) *elementary row operations (permutation of rows, adding a non-zero multiple of one row to another, multiplication of a row by a non-zero scalar)*
   (b) *permutation of columns*
   (c) *multiplication of a column by a non-zero scalar*

   More formally, we can write $G_2 = AG_1M$ with $A \in \mathbb{F}_q^{k\times k}, rank(A) = k$ and $M \in \mathbb{F}_q^{n\times n}$ being a monomial matrix.

Proof.

1. C_1 and C_2 are linear $[n, k]$-codes by definition. The minimum distance d is also the same, because C_1 and C_2 share the same set of distances between codewords in general.
2. Given a linear code C_1 with generator matrix G_1, the basic question is: Which operations on the set of codewords C_1 do not change the distances between any two codewords within C_1, i.e. which operations can be used to create an equivalent code C_2?
   First observe that we cannot change a single codeword $\vec{c}' \in C_1$ without changing all codewords, because we must retain the linearity of the resulting code. Hence we can only apply operations to all codewords at the same time. There are only 3 linear operations on all codewords of a linear code, that do not change the distance between any two of the codewords:

(a) Change the mapping $\vec{m} \to G^T\vec{m}$ of the message space to the code space, but do not change the code: This is equivalent to applying elementary row operations on G_1: Denote by G_1' the generator matrix G_1 with arbitrary elementary row operations applied, i.e. $G_1' = A \cdot G_1, rank(A) = k$. As already mentioned, the rows of G_1 form a basis of a subspace F_q^k in the vector space $\mathbb{F}_q^n$. It is well known that elementary row operations do not change the solution set of a system of linear equations. Therefore the rows of G_1' span the same subspace $\mathbb{F}_q^k$ in $\mathbb{F}_q^n$ as the rows of G_1 and we have $C_1 = C_1'$. It directly follows that we can define the mapping function f as the identity function. Obviously the distances between the codewords of C_1 and C_1' are the same as the codes are *identical* ($C_1 = C_1'$).

(b) Permute the positions of the entries of all codewords: In that case we would define f as the permutation function $\pi : \mathbb{F}_q^n \to \mathbb{F}_q^n$. Since $\text{wt}(\vec{c}) = \text{wt}(\pi(\vec{c}))$ for any $\vec{c} \in C_1$, the distances between any two codewords do not change after the permutation. Permuting the positions of the entries of all codewords obviously corresponds to a permutation of the columns of G_1.

(c) Multiply entries of all codewords in a fixed position by a non-zero scalar $r \in \mathbb{F}_q$: Without loss of generality let us choose to multiply all codewords of C_1 by $r \in \mathbb{F}_q$, $r \neq 0$ in the position $1 \leq i \leq n$. If we once again define f as the identity function, the only difference regarding the distance between any two codewords in C_1 and any two codewords in C_1' can come from the entries indexed by i. Denote these scalar entries by $c_1, c_2 \in \mathbb{F}_q$ (the i-indexed entries of any two different codewords of C_1) and $c_1' = r \cdot c_1, c_2' = r \cdot c_2$ (the i-indexed entries of the two corresponding codewords of C_1') respectively. We immediately get $c_1' - c_2' = r \cdot (c_1 - c_2)$ and see that the non-zero scalar r cannot make $c_1' - c_2'$ zero, if $c_1 - c_2$ is not zero. Thus the hamming distance between the codewords of C_1 and those of C_1' is the same (cf. section 2.1).

The latter two operations (b) and c)) can be described as a right-hand multiplication of the generator matrix G_1 by a so-called monomial matrix $M \in \mathbb{F}_q^{n \times n}$ (cf. remark 2.2.4). Using all possible linear operations a), b) and c) to create a generator matrix of another linear code C_2 we get $G_2 = AG_1M$ with $rank(A) = k$.
The only remaining possible operation "add a non-zero multiple of one column of G_1 to another column" does not preserve the distance between the codewords of C_1.

□

Remark 2.2.4. *A monomial matrix is a generalized form of a permutation matrix: A permutation matrix $P \in \mathbb{F}_q^{n \times n}$ is a matrix that can be created from the identity matrix $id^{[n]}$ by permutation of the columns. A monomial matrix $M \in \mathbb{F}_q^{n \times n}$ can be created from a permutation matrix P by additionally multiplying the columns of P by scalars $\alpha_1, \ldots, \alpha_n \in \mathbb{F}_q$, $\alpha_i \neq 0 \; \forall \, i$.*

Equivalent codes C_1 and C_2 with the same set of codewords, i.e. $C_1 = C_2$, are also called *identical codes*. In that case the generator matrix of the code C_1 can be transformed into the generator matrix of the code C_2 by only using elementary row operations. However note that the mapping $\vec{m} \to G^T\vec{m}$ of the message space to the code space is usually not the same for equivalent codes C_1 and C_2, even if we have $C_1 = C_2$ (i.e. identical codes). Instead, the notion of equivalence is based on the distance between the codewords of such codes, which is independent of the mapping of the message space to the code space.
The properties of equivalent codes from lemma 2.2.1 imply the following:

Theorem 2.2.1. *Every linear $[n, k, d]$ code C with generator matrix $G \in \mathbb{F}_q^{k\times n}$ has an equivalent (and even an identical) code C' with a generator matrix $G' = (id^{[k]} \mid R), R \in \mathbb{F}_q^{k\times(n-k)}$.*
Generator matrices in such a form are called systematic generator matrices.

Proof. Using the properties from lemma 2.2.1 we can apply Gaussian elimination to G to generate G'. As a byproduct we can also compute the matrix A, $rank(A) = k$ and the monomial matrix M. Note that Gaussian elimination does not necessarily require column operations, i.e. we can even make C and C' identical ($C = C', M = \mathrm{id}^{[n]}$). □

Using generator matrices in systematic form for encoding can be useful as it allows the sender to just append $n - k$ symbols to the message $\vec{m}$. For decoding, the receiver can directly read the message from the k first symbols - after having corrected possible errors using the redundant $n - k$ symbols of course.
As we will see, the notion of equivalent and/or identical codes is also useful in other scenarios.
However let us first have a look at parity check matrices: Instead of using a generator matrix G, a parity-check matrix H is often employed to define a linear code:

Definition 2.2.4 (Parity Check Matrix)**.** *Let $G \in \mathbb{F}_q^{k\times n}$ be a generator matrix of a code C. Any matrix $H \in \mathbb{F}_q^{(n-k)\times n}$ with $HG^T = 0^{[(n-k)\times k]}$ and $rank(H) = n - k$ is a parity check matrix of the code C.*

Remark 2.2.5. *The rows of H form a basis of $Ker(G)$. For example the Gaussian algorithm can be used to compute H from G in time polynomial in the size of G.*

Remark 2.2.6. *For any linear code C with generator matrix $G \in F_q^{k\times n}$ of the form $G = (id^{[k]} \mid R), R \in \mathbb{F}_q^{k\times(n-k)}$ (i.e. systematic form) we have that $H = (-R^T \mid id^{[n-k]})$ is a valid parity check matrix of C, because $HG^T = -R^T \cdot id^{[k]} + id^{[n-k]} \cdot R^T = 0^{[(n-k)\times k]}$.*
In combination with theorem 2.2.1 this allows us to compute the parity check matrix H for any generator matrix G of a linear code C since the parity check matrix of any identical code is also a parity check matrix of the original code C. A similar statement holds for equivalent codes (see the proof of lemma 2.2.2 for details).
Accordingly, we say that parity check matrices of the form $H = (Q \mid id^{[n-k]})$, $Q \in \mathbb{F}_q^{(n-k)\times k}$ are in systematic form.

The name "parity check matrix" probably comes from the fact, that one can easily check whether a received message $\vec{y} \in \mathbb{F}_q^n$ is an error-free codeword or not: If $\vec{y}$ is a valid codeword $\vec{c}$ of a code C with generator matrix G and parity check matrix H we have $H\vec{c} = H(\vec{m}^T G)^T = HG^T\vec{m} = \vec{0}$. The alternative definition of a linear code C for a parity check matrix $H \in \mathbb{F}_q^{(n-k)\times n}$ directly follows as $C := \{\vec{c} \in \mathbb{F}_q^n \mid H\vec{c} = \vec{0}\}$. Similar to theorem 2.2.1 one can see that using a different parity check matrix $H' = B \cdot H$ with $B \in \mathbb{F}_q^{(n-k)\times(n-k)}, rank(B) = n - k$ results in a code identical to C.
The resulting vector $\vec{s} = H\vec{y}$ is more generally defined as the syndrome of a vector $\vec{y} \in \mathbb{F}_q^n$:

Definition 2.2.5 (Syndrome)**.** *Denote by $H \in \mathbb{F}_q^{(n-k)\times n}$ a parity check matrix of a code C and $\vec{y} \in \mathbb{F}_q^n$ a vector. Then we call $\vec{s} = H\vec{y}$ the syndrome $\vec{s} \in \mathbb{F}_q^{n-k}$ of $\vec{y}$.*

Note that for any codeword with some added error vector $\vec{e}$, i.e. $\vec{y} = \vec{c} + \vec{e}$, we have

$$\vec{s} = H\vec{y} = H(\vec{c} + \vec{e}) = H\vec{e} \tag{2.1}$$

by linearity. This is often used to correct errors: Given a lookup table T, that maps all possible syndromes $\vec{s} = H\vec{e}$ to errors $\vec{e}$ with $\mathrm{wt}(\vec{e}) \leq \lfloor \frac{d-1}{2} \rfloor$ (note that H cannot be inverted), we can compute the original codeword from $\vec{y} = \vec{c} + \vec{e}$ as $\vec{c} = \vec{y} - T(\vec{s})$. This process is called *syndrome decoding*.
In particular $\vec{s} = 0$ implies $\vec{y} \in C$, i.e. a codeword without errors. From an information theoretical point of view the receiver got hold of $\lceil \log_2(q) \rceil (n-k)$ redundant bits of information in that case, justified by the possibility of occurring errors. If an appropriate code was used, the receiver should even be able to discard $n-k$ entries of the error-free $\vec{y}$ and still recover the message $\vec{m}$. More formally we define:

Definition 2.2.6 (Information Set (Generator Matrix))**.** *Given a linear $[n,k]$ code C with a generator matrix $G \in \mathbb{F}_q^{k \times n}$, a size-$k$ index set $I \subseteq \{1, \ldots, n\}$ is an information set, if and only if $\mathrm{rank}(G_I) = |I| = k$.*

If G_I is a $k \times k$ submatrix of G with full rank, its column vectors span a k-dimensional vector space and can thus be used to represent any possible non-redundant linear transformation on a message $\vec{m} \in \mathbb{F}_q^k$. In particular we know:

Theorem 2.2.2. *Given a vector $\vec{y} = \vec{c} + \vec{e}$ and an information set I with $\vec{e}_I = \vec{0}$ we can retrieve the message $\vec{m} \in \mathbb{F}_q^k$ corresponding to the codeword $\vec{c} \in C$ as $\vec{m}^T = \vec{y}_I^T G_I^{-1}$.*

Proof. First note that $G_I \in \mathbb{F}_q^{k \times k}$ is a square matrix with full rank and is thus invertible. For any codeword $\vec{c}^T = \vec{m}^T G$ we have: $\vec{y}_I^T G_I^{-1} = (\vec{c} + \vec{e})_I^T G_I^{-1} = \vec{c}_I^T G_I^{-1} = (\vec{m}^T G)_I G_I^{-1} = \vec{m}^T G_I G_I^{-1} = \vec{m}^T$.
$\vec{c}_I^T = (\vec{m}^T G)_I = \vec{m}^T G_I$ follows from the observation that the j'th column of $\vec{c}^T$ is only computed from linear combinations of the entries of the j'th column vector of G. □

Remark 2.2.7. *It is also interesting to see what we get for the case $\vec{e}_I \neq \vec{0}$: $\vec{y}_I^T G_I^{-1} G = (\vec{m}^T G_I + \vec{e}_I^T) \cdot G_I^{-1} G = \vec{m}^T G + \vec{e}_I^T G_I^{-1} G = \vec{c}^T + \vec{c}_x^T$ with $\vec{c}_x^T := (\vec{e}_I^T G_I^{-1}) \cdot G \in C$ as $\widehat{G} := G_I^{-1} G$ generates a code which is identical to the code generated by G according to lemma 2.2.1.*
Hence correcting errors in linear codes and retrieving the original message could be described as the problem of finding an information set, which does not contain any errors (or for which we know $\vec{e}_I$).

So the I-indexed entries of an error-free codeword $\vec{c}$ are sufficient to define the message $\vec{m}$. Therefore these entries are called *information symbols*, whereas the remaining $n-k$ redundant entries are called *parity check symbols*.
As we will mostly work with parity check matrices of linear codes, it is useful to have an alternative definition of information sets, if parity check matrices are used.

Definition 2.2.7 (Information Set (Parity Check Matrix))**.** *Given a linear $[n,k]$ code C with a parity check matrix $H \in \mathbb{F}_q^{(n-k) \times n}$ any size-k index set $I \subseteq \{1, \ldots, n\}$ is an information set, if and only if $\mathrm{rank}(H_{I^*}) = |I^*| = n-k$ with $I^* := \{1, \ldots, n\} \setminus I$.*

Lemma 2.2.2. *The two definitions 2.2.6 and 2.2.7 are equivalent.*

Proof.

1. Starting with the generator matrix $G \in \mathbb{F}_q^{k \times n}$ of a linear code C and the index set I, first permute the k indexed columns to the left side of G (i.e. apply a corresponding monomial matrix M to the right side of G). Then we can use row operations to bring the permuted G in the systematic form $G' = AGM = (\mathrm{id}^{[k]} | Q)$, $Q \in \mathbb{F}_q^{k \times (n-k)}$; the code generated by G' is equivalent to the linear code generated by G (cf. theorem 2.2.1 and lemma 2.2.1).

2. Now we can create a parity check matrix H' from G' according to remark 2.2.6 as $H' = (-Q^T \mid \mathrm{id}^{[n-k]})$. H' is a parity check matrix for the code generated by G'.

3. Then $H := H'M^T$ is a parity check matrix for the code generated by G, because $H'G'^T = 0^{[(n-k)\times k]} \Leftrightarrow H'(AGM)^T = 0^{[(n-k)\times k]} \Leftrightarrow H'M^TG^TA^T = 0^{[(n-k)\times k]} \Leftrightarrow H'M^TG^T = 0^{[(n-k)\times k]} \Leftrightarrow HG^T = 0^{[(n-k)\times k]}$. Note that A and M are invertible matrices.

Obviously the identity matrix $\mathrm{id}^{[n-k]}$ in the second step represents the non-indexed columns ($I^* = \{1, \ldots, n\} \setminus I$) and $rank(\mathrm{id}^{[n-k]}) = n - k$. □

Roughly speaking, we can either choose k linearly independent columns of a generator matrix G to form an information set or choose $n - k$ linearly independent columns in H to (indirectly) do the same. If such an information set does not contain any errors, we can recover the message $\vec{m}$ using theorem 2.2.2.
If you are interested in a more thorough introduction into coding theory and linear codes, have a look at [2, 29].

2.2.2 McEliece Cryptosystem

The McEliece Cryptosystem is a public-key cryptosystem based on linear codes, which was originally published in [3]. Although encryption and decryption are extremely fast and one of its underlying problems was proven to be NP-hard (i.e. at least as hard as any NP-complete problem) [4], it is not widely used, because its keys are relatively large and the code rate is relatively low (increasing the code rate does not seem to be wise as [5] indicates). Moreover it has always been a problem to estimate the complexity of attacks on concrete McEliece parameter sets and thus choose secure parameters in practice. These problems however, as well as the fact that the McEliece cryptosystem seems to withstand known quantum attacks [6] at the cost of even larger key sizes [7], led to an increased research interest until today.
Let us review the McEliece Cryptosystem:

Algorithm 2.1: McEliece Key Generation

Input: $n, k, q \in \mathbb{Z}$
Output: the public key pk and the secret key sk

1 Choose a linear $[n, k, d]$ code C over the finite field $\mathbb{F}_q$ uniformly at random from a class of linear codes, so that a decoding-algorithm $\mathrm{decode}_G()$ exists, that can correct up to $w \leq \lfloor \frac{d-1}{2} \rfloor$ errors in polynomial time (cf. definition 2.2.2) for a structured generator matrix G.
2 Find a structured generator matrix $G \in \mathbb{F}_q^{k\times n}$ of the linear code C.
3 Choose $P \in_r \mathbb{F}_q^{n\times n}$, P is a permutation matrix.
4 Choose $A \in_r \mathbb{F}_q^{k\times k}$, $rank(A) = k$.
5 $G' := AGP$
6 $pk := (G', n, k, w, q)$
7 $sk := (G, P, A)$
8 **return** (pk, sk)

The basic idea is to hide the linear code C generated by the matrix G within an equivalent code C' generated by the matrix G'. We know from the previous section that C and C' are both linear $[n, k, d]$ codes (cf. lemma 2.2.1). Even though the codewords of C and C' are similar (cf. definition 2.2.3), the mapping $\vec{m} \to G^T\vec{m}$ from the message space to the codewords is entirely different for the codes C and C'.
Encryption is essentially the same as encoding a message $\vec{m}$ into the code C' and then

adding some random "transmission" error $\vec{e}$ with $\mathrm{wt}(\vec{e}) = w$:

Algorithm 2.2: McEliece Encryption

Input: a message $\vec{m} \in \mathbb{F}_q^k$ and the public key pk
Output: the corresponding ciphertext $\vec{y} \in \mathbb{F}_q^n$

$\vec{c}^T := \vec{m}^T G'$
Choose $\vec{e} \in_r \mathbb{F}_q^n, \mathrm{wt}(\vec{e}) = w$.
$\vec{y} := \vec{c} + \vec{e}$
return $\vec{y}$

For the security of the cryptosystem it is a good idea to set $w = \lfloor \frac{d-1}{2} \rfloor$ (if possible), i.e. to add an error vector with maximum weight during encryption.
Decryption is slightly more complicated:

Algorithm 2.3: McEliece Decryption

Input: a ciphertext $\vec{y}$ and the secret key sk
Output: the corresponding message $\vec{m}$

$\vec{d}^T := \vec{y}^T P^T = \vec{m}^T AG + \vec{e}^T P^T$ // or $\vec{d} := P\vec{y}$
$\vec{a}^T := \mathrm{decode}_G(\vec{d}) = \vec{m}^T A$
$\vec{m}^T := \vec{a}^T A^{-1}$
return $\vec{m}$

Note that P is a permutation matrix and thus $P^{-1} = P^T$ and $\mathrm{wt}(\vec{e}) = \mathrm{wt}\left((\vec{e}^T P^T)^T\right) = \mathrm{wt}(P\vec{e})$, i.e. multiplying $\vec{y}^T$ by P^T does not pose a problem for the decoding algorithm $\mathrm{decode}_G()$. It is also important to see that the columns and rows of $A \in \mathbb{F}_q^{k \times k}$ form a basis of $\mathbb{F}_q^{k \times k}$ as A has full rank. So $\vec{m}^T A$ cannot be distinguished from a normal message.
There are two possible types of attacks on the McEliece cryptosystem:

Structural Attacks: It must be hard to recover G (and thus A and P) from G' using the public key pk and $\vec{y}$. Also note that in practice the linear code C and thus G has a certain structure, so that one can define an efficient algorithm $\mathrm{decode}_G()$ for that linear code and any other generator matrix with that structure. An attacker could additionally use the structure of the code C to recover G: Given random input parameters to create the code C, it must be hard to distinguish the matrix G from a random matrix $\tilde{G} \in_r \mathbb{F}_q^{k \times n}$. For example the attack presented in [5] is based on that fact and only works for certain codes.

Decoding Attacks: It must be hard to recover $\vec{m}$ from $\vec{y}$ using the public key. Essentially this problem can be formulated as: Given a linear code C' generated by a random generator matrix $G' \in_r \mathbb{F}_q^{k \times n}$ and an erroneous codeword $\vec{y} = \vec{c} + \vec{e}, \mathrm{wt}(\vec{e}) = w, \vec{c}^T = \vec{m}^T G'$, find the message $\vec{m}$. This problem of decoding a random linear code was proven to be NP-hard in [4]. Note that even though C and C' are equivalent, we cannot use $\mathrm{decode}_G()$ in combination with G' and $\vec{y}$ as it is highly unlikely that G' has the same structure as G to make $\mathrm{decode}_G()$ work.

The security of the McEliece cryptosystem heavily depends on the structure of the linear code C as well as on the parameters n, k, w, q. In the original paper [3] McEliece proposed to use binary (i.e. $q = 2$) [1024,524] Goppa codes with error-correcting capability $w = 50$. Goppa codes are described in [8, 2]. An algorithm for efficient decoding of binary Goppa codes is described in [10]. However this set of parameters was found to be insecure by several researchers, for instance theoretically by Canteaut and Sendrier in [11] and practically by Bernstein et al. in [14]. Many other more secure parameter sets have been proposed.

Remark 2.2.8. *For binary Goppa codes the relation $k = n - t \cdot \log_2(n)$ is known to hold, if $n = 2^m$ for some $m \in \mathbb{N}$. The parameter t occurs in the context of Goppa codes. We only need to know that if Patterson's decoder as described in [10] is used, we have $t = w$, so that we obtain the number of errors as a function of the parameters n and k:*

$$w = \frac{n-k}{\log_2(n)}$$

This is particularly interesting for asymptotic observations as the error rate $W := \frac{w}{n}$ becomes $W = \frac{1-R}{\log_2(n)} = \Theta\left(\frac{1}{\log_2(n)}\right)$ in combination with Goppa codes, if $R = \Theta(1)$.
So one should remember that at least for the McEliece cryptosystem the often seen assumption "$W = \Theta(1)$ and $R = \Theta(1)$" does not hold.

It is worth mentioning that the original McEliece cryptosystem as it is presented here is not IND-CPA-secure: If we encrypt $\vec{0}$, we will always get a ciphertext $\vec{y}$ with $\mathrm{wt}(\vec{y}) = w$, whereas for almost all of the other plaintexts we have $\mathrm{wt}(\vec{y}) \neq w$. So we can simply choose the zero-vector and some other message as challenge plaintexts and can then distinguish the ciphertexts with high probability. More generally, we can distinguish any two ciphertexts $\vec{y}_1$ and $\vec{y}_2$ with probability 1, where $\mathrm{wt}(\vec{c}_1) - \mathrm{wt}(\vec{c}_2)$ is larger than $2w$ (note that this is not the distance between the codewords). An attacker does not know the weight of the codewords $\vec{c}_1$ and $\vec{c}_2$, but he knows that this event often occurs if he chooses the all-zero vector and some other vector as messages. Actually he can even check whether $\mathrm{wt}(\vec{y}_1) - \mathrm{wt}(\vec{y}_2) > 4w$ holds for any two ciphertexts $\vec{y}_1$ and $\vec{y}_2$ to ensure that $\mathrm{wt}(\vec{c}_1) - \mathrm{wt}(\vec{c}_2) > 2w$ holds for the corresponding codewords $\vec{c}_1$ and $\vec{c}_2$.
However there exist several suggestions to even achieve IND-CCA2-security for the McEliece cryptosystem. The interested reader is referred to [16, 17].
A relatively recent proposal was made by Bernstein, Lange and Peters in [18] to use wild Goppa codes instead of classical Goppa codes for the McEliece cryptosystem. Wild Goppa codes are a generalization of classical Goppa codes, which allow to correct a factor of roughly $\frac{q}{q-1}$ more errors than classical Goppa codes ($q > 2$). Since the increased error-correcting capability seems to allow for smaller key sizes at the same level of security, this proposal is quite interesting. This generalized form of the McEliece cryptosystem ($q = 2$ is identical to the original binary McEliece cryptosystem) is called *Wild McEliece*.

2.2.3 Information Set Decoding

Information set decoding tries to solve the following NP-hard problem: Given a random-looking generator matrix $G' \in \mathbb{F}_q^{k \times n}$ of a linear code C' and a vector $\vec{y}^T = \vec{m}^T G' + \vec{e}^T, \vec{e} \in \mathbb{F}_q^n, \mathrm{wt}(\vec{e}) = w$, recover $\vec{m}$. Alternatively, we could recover $\vec{e}$ and then solve the linear equation system $\vec{y}^T - \vec{e}^T = \vec{m}^T G'$ or use the method provided by theorem 2.2.2 to retrieve $\vec{m}$.
Roughly speaking, the problem is that of decoding a random linear code.
Information set decoding can be used to attack the McEliece cryptosystem presented in the previous section. So far it even seems to be the top threat against the McEliece cryptosystem (as well as against Wild McEliece). Therefore the security of a certain set of McEliece parameters against information set decoding is often used to estimate the overall security of these McEliece parameters. Several (exponential-time) information set decoding algorithms are discussed in section 3. Most of them use the parity check matrix $H' \in \mathbb{F}_q^{(n-k) \times n}$, that can be computed from G' using Gaussian elimination (cf. definition 2.2.4).
Then we can formulate the problem differently: Given a random-looking parity check matrix H', a vector $\vec{y} = \vec{c} + \vec{e}$ and a syndrome $\vec{s} = H'\vec{y} = H'\vec{e}$, find the error vector $\vec{e} \in \mathbb{F}_q^n$ with $\mathrm{wt}(\vec{e}) = w$ (cf. equation (2.1)). This problem - identical to the problem of decoding a

random linear code - is also called the *Computational Syndrome Decoding (CSD)* problem. The basic idea of any information set decoding algorithm is to uniformly choose an information set I (cf. definition 2.2.7), guess the I-indexed part $\vec{e}_I$ of the error vector according to a predefined method[1] in a brute-force approach and try to compute the whole error vector $\vec{e}$ from these assumptions. If the guess was correct, one would obtain an error vector $\vec{e}$ with $\mathrm{wt}(\vec{e}) = w$.

Remark 2.2.9. *Note that only the original error vector $\vec{e}$ satisfies $wt(\vec{e}) = w$ as long as $w \leq \lfloor \frac{d-1}{2} \rfloor$ holds.*

Proof. Let us assume that we find another error vector $\vec{e}_n \neq \vec{e}$ with $H'\vec{e}_n = \vec{s}$ and $\mathrm{wt}(\vec{e}_n) \leq \lfloor \frac{d-1}{2} \rfloor$. Then we have $H'\vec{e}_n = \vec{s} = H'\vec{y} = H'\vec{c} + H'\vec{e} \Rightarrow H'\vec{c} = H'(\vec{e}_n - \vec{e}) \Rightarrow \vec{e}_n - \vec{e} \in C'$. However the codeword $\vec{e}_n - \vec{e}$ has the weight $\mathrm{wt}(\vec{e}_n - \vec{e}) \leq \lfloor \frac{d-1}{2} \rfloor + \lfloor \frac{d-1}{2} \rfloor \leq d - 1$ and thus a distance of less than the minimum distance d to the all-zero codeword. $\Rightarrow$ Contradiction. □

For example we could use the observation from remark 2.2.7: Guessing an error $\vec{e}_I$ (or many different $\vec{e}_I$ according to a predefined method), we can compute $\vec{c}_x^T = (\vec{e}_I^T G_I'^{-1}) \cdot G'$. Then we can compute $\vec{c}^T = \vec{y}_I^T G_I'^{-1} G' - \vec{c}_x^T$ and $\vec{e} = \vec{y} - \vec{c}$. If such an error vector satisfies the equation $\mathrm{wt}(\vec{e}) \stackrel{?}{=} w$, we immediately found the original error vector $\vec{e}$. If we did not find the original error vector, we can still choose a different $\vec{e}_I$ or another information set I and try again.
Let us describe information set decoding with parity check matrices: First note that choosing an information set in the context of parity check matrices means choosing $n-k$ linearly independent columns of $H' \in \mathbb{F}_q^{(n-k)\times n}$ (cf. definition 2.2.7). For the sake of clarity after choosing an information set I, we apply the following steps to H':

1. We use a permutation matrix $P \in \mathbb{F}_q^{n\times n}$ to permute the columns of H' indexed by the set $I^* := \{1, \ldots, n\} \setminus I$ to the right side of H', i.e. we create $H'_p := H'P$. Observe that the parity check matrix H'_p corresponds to the generator matrix $\widehat{G} := G'P^{-1} = G'P^T$ as $H'_p \cdot \widehat{G}^T = H'P \cdot (G'P^{-1})^T = H'G'^T = 0^{[(n-k)\times k]}$ (cf. definition 2.2.4). One should keep in mind that $\widehat{G}$ generates a linear code, which is equivalent, but not identical to the code generated by G (cf. lemma 2.2.1).

2. We apply elementary row operations to H'_p to bring it into the systematic form $\widehat{H} := (Q \mid \mathrm{id}^{[n-k]}), Q \in \mathbb{F}_q^{(n-k)\times k}$. As $\widehat{H}$ generates the same code as H'_p, it is still a valid parity check matrix for the code generated by $\widehat{G}$ (also see remark 2.2.6). We introduce the matrix R to represent the elementary row operations on H'_p.

More formally we define algorithm 2.4 to do exactly those operations mentioned above and additionally define a new syndrome $\vec{\varsigma} := R\vec{s}$ and a new (unknown) error vector $\vec{\epsilon} := P^{-1}\vec{e} = P^T\vec{e}$. If you carefully look at the operation $P^T\vec{e}$, you will notice that a left-handed multiplication by a permutation matrix means a permutation of the rows of the (column) vector $\vec{e}$. As P comes from step 1 above, we know that it only changes the positions of the I-indexed rows of a matrix or (in this case) a vector. Observe that $P\vec{e}$ would mean a permutation of the I-indexed rows with the rows at the very bottom of $\vec{e}$; however here we have the inverse permutation $P^{-1} = P^T$, i.e. a permutation of the I-indexed rows of $\vec{e}$ with the top rows of $\vec{e}$. Putting it all together we can see that the I-indexed entries from the error vector $\vec{e}$ are all shifted to the top of $\vec{\epsilon}$ by P^T. One could write $\vec{\epsilon}_I := \vec{\epsilon}_{[|I|]}$.

[1] The concrete method depends on the specific algorithm. Details can be found in section 3.

Algorithm 2.4: GenerateSystematicParityCheckMatrix

Input: parity check matrix $H' \in \mathbb{F}_q^{(n-k)\times n}$, $\vec{y} = \vec{c} + \vec{e}$, information set I

Output: parity check matrix $\widehat{H}$ in systematic form, permutation matrix P, syndrome $\vec{\varsigma}$

1 $\vec{s} := H'\vec{y}$ (syndrome of the original problem)

2 apply the column operations mentioned in step 1 above to H' to obtain $H'_p = H'P$, save the permutation matrix P

3 apply the elementary row operations mentioned in step 2 above to H'_p to obtain $\widehat{H} = (Q \mid \mathrm{id}^{[n-k]})$ in systematic form, save the matrix R

4 $\vec{\varsigma} := R\vec{s}$

5 **return** $(\widehat{H}, P, \vec{\varsigma})$

Remark 2.2.10. *The permutation of the columns of H' and thus the matrix P is not really necessary for the idea presented here to solve the CSD problem. It would suffice to bring the I-indexed entries of H' into the form of an identity matrix using elementary row operations only. Nevertheless we use P to bring H' into completely systematic form to make important observations more obvious.*

Remark 2.2.11. *In the context of information set decoding the information set I is usually chosen uniformly at random, so that in practice you will not see algorithm 2.4 as described above, but in a more optimised version: Instead of taking the random information set I as input, one just randomly permutes the columns of H' and chooses the k leftmost columns as the information set I (as long as the set $I^* = \{1, \ldots, n\} \setminus I$ indexes $n-k$ linearly independent columns to satisfy the definition of an information set). The rest is done as in algorithm 2.4.*
Algorithm 3.1 takes this remark into account.

We have $\widehat{H}\vec{\epsilon} = RH'P \cdot P^T\vec{e} = RH'P \cdot P^{-1}\vec{e} = RH'\vec{e} = \vec{\varsigma} = R\vec{s} \Leftrightarrow H'\vec{e} = \vec{s}$, i.e. the formulation of the computational syndrome decoding problem with a matrix in systematic form is the same (just with $\widehat{H}$ instead of H', $\vec{\varsigma}$ instead of $\vec{s}$ and $\vec{\epsilon}$ instead of $\vec{e}$). Thus, if we are able to solve the problem for a parity check matrix in systematic form, we obtain an error vector $\vec{\epsilon}$ and can then solve the problem in the original form by computing $\vec{e} = P\vec{\epsilon}$. Now to possibly solve the computational syndrome decoding problem with the parity check matrix $\widehat{H}$ using information set decoding, we once again guess the I-indexed entries of the error vector $\vec{\epsilon}$, i.e. $\vec{\epsilon}_I$ - presumably many times and according to a predefined method. From our previous observations we know that the equation $\widehat{H}\vec{\epsilon} = \vec{\varsigma}$ has a structure as depicted in figure 2.1. Note that $\widehat{H} \in \mathbb{F}_q^{(n-k)\times n}, Q \in \mathbb{F}_q^{(n-k)\times k}, i := \mathrm{id}^{[n-k]}, \vec{\epsilon} \in \mathbb{F}_q^n, \vec{\epsilon}_I \in \mathbb{F}_q^k, \vec{\epsilon}_{I^*} \in \mathbb{F}_q^{n-k}$ and $\vec{\varsigma} \in \mathbb{F}_q^{n-k}$. $\vec{\epsilon}_{I^*}$ is used to denote the entries of $\vec{\epsilon}$, that are not indexed by I. We aim to compute those entries.

$$\left(\begin{array}{c|c} Q & i \end{array}\right) \left(\begin{array}{c} \vec{\epsilon}_I \\ \hline \vec{\epsilon}_{I^*} \end{array}\right) = \vec{\varsigma}$$

Figure 2.1: Structure of $\widehat{H}\vec{\epsilon} = \vec{\varsigma}$ with $i := \mathrm{id}^{[n-k]}$

This gives us the idea that it might be useful to split $\vec{\epsilon}$ even more, so that we get an

equation as in figure 2.2:

$$\left(\begin{array}{c|c} Q & i \end{array}\right) \left(\begin{pmatrix} \vec{\epsilon}_I \\ \hline \vec{0} \end{pmatrix} + \begin{pmatrix} \vec{0} \\ \hline \vec{\epsilon}_{I^*} \end{pmatrix} \right) = \vec{\varsigma}$$

Figure 2.2: Transformed structure of $\widehat{H}\vec{\epsilon} = \vec{\varsigma}$

Let us define $\vec{\epsilon}_{I,0} \in \mathbb{F}_q^n$ to be the vector $\vec{\epsilon}_I$ with $n-k$ zeros padded at the bottom and $\vec{\epsilon}_{I^*,0}$ the vector $\vec{\epsilon}_{I^*}$ with k zeros padded at the top (as in figure 2.2). Then we get

$$\begin{aligned} \widehat{H}\vec{\epsilon} = (\ Q \mid i\ ) \cdot (\vec{\epsilon}_{I,0} + \vec{\epsilon}_{I^*,0}) &= \vec{\varsigma} \\ \Leftrightarrow Q\vec{\epsilon}_I + \vec{\epsilon}_{I^*} &= \vec{\varsigma} \qquad (2.2) \\ \Rightarrow \vec{\epsilon}_{I^*} &= \vec{\varsigma} - Q\vec{\epsilon}_I \qquad (2.3) \end{aligned}$$

and we can extract $\vec{\epsilon}_{I^*}$ from equation (2.3). To check whether our guess regarding $\vec{\epsilon}_I$ was correct, we could now test if $\mathrm{wt}\,(\vec{\epsilon}_{I^*}) \stackrel{?}{=} w - \mathrm{wt}\,(\vec{\epsilon}_I)$ holds. If it does not, we can just choose a different information set I or guess a different $\vec{\epsilon}_I$ and try again. If it does, we compute $\vec{\epsilon} = \vec{\epsilon}_{I,0} + \vec{\epsilon}_{I^*,0}$ and obtain the unique error vector $\vec{e} = P\vec{\epsilon}$ with $\mathrm{wt}\,(\vec{e}) = w$.

2.2.4 Comparing Information Set Decoding Algorithms

Comparing information set decoding algorithms is by no means trivial: Although all of the algorithms presented in section 3 have runtimes that can easily be modeled by equations containing several binomial coefficients, most of these equations cannot be used to directly compare the algorithms for improvements over one another. In practice one would probably just compute the results of these equations for a concrete parameter set (n, k, w) and choose the algorithm with the best runtime. This is the approach presented in section 4. In theory though it is desirable to know which algorithm is superior to another by what degree and for which range of parameters.
An usual approach to do so is to rewrite the aforementioned equations containing the binomial coefficients in a way that reveals their exponential nature, i.e. by writing them as $2^{(\alpha(R,W)+o(1))n} = \tilde{\mathcal{O}}(2^{\alpha(R,W)n})$. Thereby it is common to write the exponent as a function $\alpha(R, W)$ of the code rate $R := \frac{k}{n}$ and the error rate $W := \frac{w}{n}$. Note that a statement such as $\tilde{\mathcal{O}}(2^{\alpha(R,W)n})$ regarding the runtime of an information set decoding algorithm hides any possible polynomial $p(n, k, w)$ as $p(n, k, w)2^{\alpha(R,W)n} = \tilde{\mathcal{O}}(2^{\alpha(R,W)n})$, which makes such a statement only useful for asymptotic observations $(n \to \infty)$.
For asymptotic observations it is even possible to relate the code rate and the error rate of random binary linear codes via the Gilbert-Varshamov bound (an introduction can be found in [29], chapter 2) and thus remove the parameter w or W from the equations. By maximising the resulting statement of the form $\tilde{\mathcal{O}}(2^{\alpha(R)n})$ with regard to R $(0 \leq k \leq n)$ whilst choosing the optimal algorithmic parameters, one can even obtain a worst case complexity of the form $\tilde{\mathcal{O}}(2^{an})$ for some constant $0 < a < 1$. For example this is done by May et al. in [20] and [21]. We call these bounds that allow for a straightforward asymptotic comparison of information set decoding algorithms *rough bounds*. A nice overview of the

rough bounds of the most important information set decoding algorithms is presented in [20, 21].

Remark 2.2.12. *May et al. often use the handy formula* $\binom{an}{bn} = \tilde{\mathcal{O}}(2^{aH_2(b/a)n})$*, where* $H_2(x) := -x\log_2(x) - (1-x)\log_2(1-x)$ *is the binary entropy function. The formula follows from Stirling's formula (cf. definition 2.2.9); it can even be used to quickly obtain conservative parameter choices for any information set decoding algorithm.*
In [21] they additionally use the fact that $\tilde{\mathcal{O}}(2^{an}) < 2^{(a+\epsilon)n}$ *for some* $\epsilon > 0$ *and sufficiently large* n *(i.e. they round up at some decimal) to get rid of the Landau notation. We will also provide the rough bounds that way.*

The overall advantages and disadvantages of this bound are displayed in figure 2.3.

Advantages	**Disadvantages**
+ allows for a straightforward comparison of the asymptotic runtimes	- polynomial differences remain invisible
+ short	- no usable statement for practical applications, i.e. for fixed parameters (n, k, w)

Figure 2.3: Advantages and disadvantages of the *rough bound*

As the rough bound is an asymptotic bound that ignores polynomial factors we can only use it for a statement such as: "For sufficiently large n, algorithm X performs better than algorithm Y." However algorithm Y might perform better for smaller n as algorithm X might have polynomials in its runtime description that are much larger than those of algorithm Y. These observations are important in practice. Therefore it is desirable to provide explicit runtime descriptions of information set decoding algorithms of the form $p(n,k,w)2^{\alpha(R,W)n}$ for a polynomial $p(n,k,w)$. Note that the $o(1)$ in the exponent is missing this time.
It is common in literature (e.g. see [15]) to even fix $\alpha(R,W)$ to the exponent that occurs in the runtime description of the oldest information set decoding algorithm, namely Prange's algorithm (cf. section 3.2), which exposes the exponential term $2^{\alpha(R,W)n}$ with the following function $\alpha(R,W)$ for $n \to \infty$ (i.e. $time\{Prange\} = \tilde{\mathcal{O}}(2^{\alpha(R,W)n})$):

Definition 2.2.8.

$$\alpha(R,W) := (1-R-W)\log_2(1-R-W) - (1-R)\log_2(1-R) - (1-W)\log_2(1-W)$$

Remark 2.2.13. *To verify this claim, we can start with* $time\{Prange\} = \tilde{\mathcal{O}}\left(\binom{n}{w}\binom{n-k}{w}^{-1}\right) = \tilde{\mathcal{O}}\left(\binom{n}{Wn}\binom{(1-R)n}{Wn}^{-1}\right)$ *(inverse of equation (3.9)) and use the formula from remark 2.2.12 to obtain* $time\{Prange\} = \tilde{\mathcal{O}}(2^{\alpha(R,W)n})$ *with* $\alpha(R,W)$ *as defined above.*

Then we can try to rewrite the runtime of any algorithm ALG as $time\{ALG\} = q(n,k,w)\cdot 2^{\alpha(R,W)n}$, where $q(n,k,w)$ is a possibly (inverse) exponential function. Thereby it is useful to write $q(n,k,w) = q'(n,k,w)\cdot err(n,k,w)$, where $\lim_{n\to\infty} err(n,k,w) = 1$ for asymptotic observations (i.e. $time\{ALG\} = \Theta(q'(n,k,w)2^{\alpha(R,W)n})$). Unfortunately we usually cannot compute $err(n,k,w)$ exactly due to the problem of writing the binomial coefficients and factorials as exponential functions. Nevertheless it is normally possible to provide tight bounds on $err(n,k,w)$ by using the refined Stirling approximation below.

Definition 2.2.9 (Stirling Formula)**.**

$$m! = \sqrt{2\pi}\cdot m^{m+1/2}\cdot e^{-m+\epsilon(m)}$$

where $\epsilon : \{1,2,3,\ldots\} \to \mathbb{R}$ *is a function to make the equation hold.*

The classic Stirling approximation is $\epsilon(m) \approx 0$, but we require the refined approximation:

Lemma 2.2.3 (Refined Stirling Approximation)**.**

$$\frac{1}{12m+1} < \epsilon(m) < \frac{1}{12m}$$

Proof. A proof can be found in [30]. □

Hence we cannot only use $q(n,k,w)$ for asymptotic observations as with the rough bound, but also for comparisons that hold for any n (with the upper and lower bound on $err(n,k,w)$ providing a certain error range for these comparisons). Figure 2.4 sums up the advantages and disadvantages of this method, that results in an equation which we define as the *explicit bound* of the runtime of an information set decoding algorithm. Several explicit bounds for different algorithms are provided in [15, 25]; however sometimes only for the success probability of an algorithm.

Advantages	**Disadvantages**
+ polynomial differences are visible	- equations are long and complicated
+ more usable for practical applications and concrete parameter sets	
+ asymptotic runtime is still visible	

Figure 2.4: Advantages and disadvantages of the *explicit bound*

Now to compare an algorithm ALG1 with an algorithm ALG2 we define the *advantage* of ALG1 over ALG2 as:

Definition 2.2.10 (Advantage of an algorithm over another)**.**

$$Adv(ALG1[o_{1,1}, o_{1,2}, \ldots] > ALG2[o_{2,1}, o_{2,2}, \ldots]) := \frac{time\{ALG1[o_{1,1}, o_{1,2}, \ldots]\}}{time\{ALG2[o_{2,1}, o_{2,2}, \ldots]\}}$$

Thereby $o_{1,1}, o_{1,2}, \ldots$ are the optimizations applied to algorithm ALG1 and $o_{2,1}, o_{2,2}, \ldots$ are those applied to algorithm ALG2.

For $time\{ALGi\} = q_i(n,k,w)2^{\alpha(R,W)n}$, $q_i(n,k,w) = q_i'(n,k,w) \cdot err_i(n,k,w)$ and $\lim_{n\to\infty} err_i(n,k,w) = 1 \ \forall \ i$ we can then use $Adv(ALG1 > ALG2) = \Theta\left(\frac{q_1'}{q_2'}\right)$. This way we can even see polynomial differences between two algorithms for a fixed $\alpha(R,W)$ (cf. definition 2.2.8).
All in all it must be admitted though that none of these methods are very useful in practice. In practice it is desirable to have exact implementations of the runtime and memory complexity formulas for the existent information set decoding algorithms. Such an implementation for the algorithms discussed in this thesis is provided as part of this work (cf. section 4).

3 Algorithms

Information set decoding algorithms aim to solve the computational syndrome decoding problem: Given a random-looking parity check matrix $H \in \mathbb{F}_q^{(n-k)\times n}$ of a linear $[n,k,d]$ code C, a vector $\vec{y} = \vec{c} + \vec{e}$ and a syndrome $\vec{s} = H\vec{y} = H\vec{e}$, find the error vector $\vec{e} \in \mathbb{F}_q^n$ with $\mathrm{wt}(\vec{e}) = w$.
This section first presents several classical algorithms to cope with the problem, followed by some more recent algorithms. All of these algorithms are analysed for their average number of iterations, the number of binary operations per iteration and their memory consumption. We also provide both rough and sometimes even explicit bounds for their runtime (cf. section 2.2.4) and consider possible optimizations of the algorithms. The ultimate goal is to provide formulas for each algorithm that enable us to estimate the runtime complexities for concrete parameter sets (n,k,w) (including polynomial factors) and thus help to pick an algorithm for a specific practical application. In contrast most scientists only analyse the asymptotic runtime ($n \to \infty$) - which does not really help in practical applications where n is fixed.
Note that we use $q = 2$ from now on, i.e. we limit our observations to binary finite fields.

3.1 General Structure

All of the information set decoding algorithms presented in this section are Las Vegas algorithms and therefore share a common structure. This structure was already indicated in section 2.2.3. Algorithm 3.1 presents it in more detail:

Algorithm 3.1: isd$(H, \vec{y}, w)$; General structure of information set decoding algorithms

Input: parity check matrix $H \in \mathbb{F}_2^{(n-k)\times n}$, $\vec{y} = \vec{c} + \vec{e}$, $w = \mathrm{wt}(\vec{e})$
Output: error vector $\vec{e}$

1 $\vec{s} := H\vec{y}$
2 **while** *true* **do**
3 $(\widehat{H}, P, \vec{\varsigma}) \leftarrow$ `randomize(`$H, \vec{s}$`)`
// optional optimization: $H := \widehat{H}$, $P := P_{last} \cdot P$
4 $(success, \vec{\epsilon}) \leftarrow$ `searchALG(`$\widehat{H}, \vec{\varsigma}, w$`)` // defined by the specific algorithm ALG
5 **if** $success = true$ // $\mathrm{wt}(\vec{\epsilon}) = w$
6 **then**
7 $\vec{e} := P\vec{\epsilon}$
8 **return** $\vec{e}$
9 **end**
10 **end**

11 **function** `randomize(`$H, \vec{s}$`)`
12 **begin**
13 permute the columns of H uniformly at random to obtain $H_p = HP := (K \mid T), T \in \mathbb{F}_2^{(n-k)\times(n-k)}$, save the permutation matrix $P \in \mathbb{F}_2^{n\times n}$
14 **if** $rank(T) \neq n - k$ **then goto** line 13 // check for an information set
15 apply elementary row operations to H_p to obtain $\widehat{H} = RH_p = (Q \mid \mathrm{id}^{[n-k]})$ in systematic form, save the matrix $R \in \mathbb{F}_2^{(n-k)\times(n-k)}$ (or directly apply the same operations to $\vec{s}$ to obtain $\vec{\varsigma}$)
16 $\vec{\varsigma} := R\vec{s}$
17 **return** $(\widehat{H}, P, \vec{\varsigma})$
18 **end**

The function randomize() is basically a slight variation of algorithm 2.4 incorporating remark 2.2.11, i.e. we simply choose the k leftmost columns of $\widehat{H}$ as information set I, which is checked to be valid in line 14. Obviously one would not use the explicit check of line 14 in practice, but rather check the rank of T during the Gaussian elimination performed in line 15 anyway. The separation of these two lines in algorithm 3.1 is only done for illustrative purposes and can be ignored for the runtime analysis.
The function searchALG() is different for each information set decoding algorithm ALG and tries to find an error vector $\vec{\epsilon}$, so that $\mathrm{wt}(\vec{\epsilon}) = w$ holds. If it is able to find such a vector, it sets the variable *success* to true and returns the error vector $\vec{\epsilon}$. Otherwise it sets *success* to false.
As with any Las Vegas algorithm the basic idea is to randomize the original input and hope that the randomized input of searchALG() features exactly those properties that make the function searchALG() work. The "good properties" of $\widehat{H}$ and $\vec{\varsigma}$ are defined by the implementation of searchALG() and thus by the algorithm used.
To estimate the runtime and memory consumption of algorithm 3.1, we introduce the following model:

- Given two binary vectors $\vec{v}_1, \vec{v}_2 \in \mathbb{F}_2^n$ addition of these vectors can be achieved in time $\mathcal{O}(n)$. Note that this assumption only makes sense for $q = 2$, where an addition of binary vectors of length n can be represented as n XOR-operations. These operations can easily be parallelized in hardware.
  If a computational architecture works on words of size b, a more proper complexity would be $\mathcal{O}(\frac{n}{b})$.

- Storing a binary vector $\vec{v} \in \mathbb{F}_2^n$ consumes $\mathcal{O}(n)$ units of memory space (usually exactly n bits). Storing a matrix $M \in \mathbb{F}_2^{l \times n}$ accordingly results in a memory consumption of $\mathcal{O}(ln)$.

- Memory access times (read, write) are neglected. Neglecting this measurement is common in literature to obtain a relatively simple model that only requires a minimal set of assumptions with regard to the underlying computational architecture. Memory access times can be quite relevant in practice though.

The inverse success probability of the algorithm ALG, namely $Pr_{ALG}[success = true]^{-1}$ can be used to estimate the average number of iterations of algorithm 3.1. Note that in reality $Pr_{ALG}[success = true]$ heavily depends on the implementation of searchALG() and the concrete error vector $\vec{e}$, which we do not know. Therefore $Pr_{ALG}[success = true]$ is replaced by a probability over averages of $\vec{e}$, namely $\overline{PR}_{ALG}[success = true]$. As [25] points out, $\overline{PR}_{ALG}[success = true]^{-1}$ is not necessarily identical to the real average number of iterations for a concrete error vector[2]. Nevertheless it is a reasonable measure often seen in literature (e.g. [22, 25]). Then we have:

$$\begin{aligned} time\,\{isd()\} &= \overline{PR}_{ALG}[success]^{-1}(time\,\{randomize()\} + time\,\{searchALG()\}) \quad (3.1) \\ mem\,\{isd()\} &= \mathcal{O}(n^2) + mem\,\{randomize()\} + mem\,\{searchALG()\} \end{aligned}$$

Regarding the function randomize() one can easily see that its runtime is dominated by the binary Gaussian elimination in line 15 of algorithm 3.1. Therefore it makes sense to recall how this Gaussian elimination on $H \in \mathbb{F}_2^{(n-k) \times n}$ works: We iterate over the columns indexed by $I^* = \{k+1, \ldots, n\}$. Then we choose a pivot for the specific column (which might involve some row swapping) and iterate over all of the $n-k$ rows of H, excluding the pivoted row. For each "1"-entry in the current column vector we add the pivoted row, so that the column vector only contains exactly one "1"-entry in the end (at the diagonal of $\mathrm{id}^{[(n-k)]}$). So during the first column iteration we work on $\leq n-k$ rows and the addition of the pivoted row to the current row requires $n-1$ binary additions (the first column can be left out as we know that it becomes zero). During the second iteration, we once again iterate over all of the $n-k$ rows, but the addition of the pivoted row to the current row only requires $n-2$ additions (mod 2) as both the first and the second column are known to have a single "1"-entry only (at the diagonal of $\mathrm{id}^{[(n-k)]}$); and so on through the last column iteration, which involves $\leq k$ binary additions/operations per row (for this column). So each row of H involves at most $b_r := \sum_{i=1}^{n-k} n - i = n(n-k) - \frac{1}{2}(n-k)(n-k+1)$ binary operations (with i iterating over the columns). All in all we get $(n-k) \cdot b_r = (n-k) \cdot [(n-k)^2 + (k - \frac{1}{2})(n-k) - \frac{1}{2}(n-k)^2] = \frac{1}{2}(n-k)^3 + (k-\frac{1}{2})(n-k)^2 = \mathcal{O}(n^3)$ bit operations.

Remark 3.1.1. *One might also observe that a row addition only occurs with probability $\frac{1}{2}$, which is ignored here.*

[2] An example for Lee-Brickell's algorithm can be found on page 85 of [25].

Due to the fact that randomize() stores at least the matrix $P \in \mathbb{F}_2^{n \times n}$, we may use $mem\{randomize()\} = \mathcal{O}(n^2)$. If we do not store the matrix R in line 15, we could even use $mem\{randomize()\} = \mathcal{O}(n \log(n))$, because we only need to save the positions of the 1's in each column of P. By applying optimization 3.1.1 we can pretty much ignore the repetitions implied by line 14 of algorithm 3.1 and thus get:

$$\begin{aligned} time\{isd()\} &= \overline{PR}_{ALG}[success = true]^{-1} \cdot \left(\mathcal{O}(n^3) + time\{searchALG()\}\right) \\ mem\{isd()\} &= \mathcal{O}(n^2) + mem\{searchALG()\} \end{aligned} \tag{3.2}$$

In theory one would probably use the rough estimate $time\{randomize()\} = \mathcal{O}(n^3)$, whereas in practice the more concrete estimate $time\{randomize()[o_{3.1.1}]\} = \frac{1}{2}(n-k)^3 + (k - \frac{1}{2})(n-k)^2$ for the number of bit operations during the Gaussian elimination process is often relevant. This number can even be reduced by using more of the optimizations below.

Optimizations

As almost all of the optimization techniques presented in this work, most of the following optimizations come from [14, 25]. They are meant to speed up the Gaussian elimination process in the randomize() function of algorithm 3.1.

Optimization 3.1.1 (Adaptive information sets)**.** *As already mentioned, in practice we compute line 14 and line 15 of algorithm 3.1 at the same time by applying Gaussian elimination. Returning to line 13, if we cannot produce an identity matrix $id^{[n-k]}$ from the $n-k$ rightmost columns of H_p, seems inefficient though: Instead, we could just adapt our choice of the $n-k$ columns and swap those columns that seem to introduce the linear dependence between the $n-k$ selected columns (at the right) with some of the k deselected columns (at the left). The permutation matrix P needs to be updated accordingly. Then we could continue with the Gaussian elimination and benefit from almost all of the work done before. Although these operations might result in a matrix P, that is not entirely uniform and might thus affect the success probability of searchALG(), no noticeable effects have been observed in practice so far [25].*
If we do not want to employ this optimization for algorithm 3.1, we can estimate the probability that a random binary matrix $T \in \mathbb{F}_2^{(n-k)\times(n-k)}$ only contains linearly independent row vectors in line 14 as follows: If we choose the first row vector of T uniformly at random, it can only be linearly dependent of the zero-vector, i.e. we have a probability of $\frac{1}{2^{n-k}}$ to obtain a linearly dependent vector for the first row. For the second row the probability is $\frac{2}{2^{n-k}}$, for the third row it is $\frac{2^2}{n-k}$ as there are 2^2 linear combinations of the previous two rows and so on. All in all we have a probability of $\prod_{i=1}^{n-k}(1 - \frac{2^{i-1}}{2^{n-k}}) = \prod_{i=1}^{n-k}(1 - 2^{-i})$ to obtain a matrix T of full rank in the end. It holds that $\frac{1}{4} < \lim_{n\to\infty} \prod_{i=1}^{n-k}(1 - 2^{-i}) < \frac{1}{4} + \frac{1}{2^5} + \frac{1}{2^7}$ according to the pentagonal number theorem, so that the number of iterations due to line 14 of algorithm 3.1 is asymptotically constant and thus negligible.

Optimization 3.1.2 (Reusing existing pivots)**.** *By including the comment between the lines 3 and 4 of algorithm 3.1 we can reduce the number of bit operations required by the Gaussian elimination: As the matrix $\widehat{H}$ from the previous iteration already contains an identity matrix $id^{[n-k]}$, we can assume that the permutation of H in line 13 will result in a matrix of the form $H_p = (K \mid T), K \in \mathbb{F}_2^{(n-k)\times k}, T \in \mathbb{F}_2^{(n-k)\times(n-k)}$, where T contains at about $\frac{(n-k)^2}{n}$ columns from $id^{[n-k]}$, because the probability of a single column to be part of the previous identity matrix of $\widehat{H}$ is $\frac{n-k}{n}$. These columns from $id^{[n-k]}$ just contain a single "1"-entry, which is simply (row) swapped to the appropriate position by the Gaussian*

algorithm. No more work is done for these columns. The real work is left over for the remaining $n-k-\frac{(n-k)^2}{n}=\frac{k(n-k)}{n}$ *columns of* H.
When using this optimization we can basically ignore $s:=\frac{(n-k)^2}{n}$ *columns of* H. *So for the first column that we cannot ignore we get at most* $n-s-1$ *binary additions/operations per row, for the second column* $n-s-2$ *and so on. All in all we get at most* $\sum_{i=s+1}^{n-k}(n-i)=\sum_{i=1}^{n-k}(n-i)-\sum_{i=1}^{s}(n-i)=b_r-\sum_{i=1}^{s}(n-i)$ *binary operations per row. Recall that* b_r *is the number of binary operations per row without applying this optimization.* $\sum_{i=1}^{s}(n-i)$ *represents the number of binary operations that we save by this optimization per row. We have*

$$\begin{aligned}\sum_{i=1}^{s}(n-i) &= sn-\frac{1}{2}s(s+1)\\ &= s(n-\frac{1}{2})-\frac{1}{2}s^2\\ &= \frac{(n-k)^2(n-\frac{1}{2})}{n}-\frac{(n-k)^4}{2n^2}\\ &= \frac{(n-k)^2(2n^2-n)-(n-k)^4}{2n^2}\\ &= \frac{(n-k)^2[2n^2-n-(n-k)^2]}{2n^2} \quad =: b_{opt-3.1.2}\\ &= \frac{(n-k)^2[n^2+2nk-n-k^2]}{2n^2}\end{aligned}$$

Altogether we need $\frac{(n-k)^3[n^2+2nk-n-k^2]}{2n^2}$ *binary operations less than the original algorithm 3.1 needs for the Gaussian elimination (*$\frac{1}{2}(n-k)^3+(k-\frac{1}{2})(n-k)^2$ *bit operations). Note that we once again ignored remark 3.1.1.*

Optimization 3.1.3 (Force more existing pivots)**.** *The basic idea of this optimization technique is not only to reuse those pivots that come from the identity matrix in* $\widehat{H}=(Q\mid id^{[n-k]})$ *as described above, but to even drop the uniform permutation of the columns of* H *(line 13 of algorithm 3.1) and only artificially select* x*-many columns from* $Q\in\mathbb{F}_2^{(n-k)\times k}$ *and* $n-k-x$ *columns from* $id^{[n-k]}$ *in random order for the Gaussian elimination in line 15. For any* $x<\frac{k(n-k)}{n}$ *the Gaussian elimination step takes less time in comparison to the previous optimization technique. Actually we can calculate the number of binary operations that we save for one Gaussian elimination in comparison to the original Gaussian elimination similarly to the calculations for optimization 3.1.2: We can ignore* $s':=n-k-x$ *columns of* H *during the Gaussian elimination process. For the first of the columns that we cannot ignore we need* $n-s'-1$ *bit operations per row, for the second* $n-s'-2$ *and so on. All in all we need* $\sum_{i=s'+1}^{n-k}n-i$ *bit operations per row. Observe the following for* $s:=\frac{(n-k)^2}{n}$ *as defined in the text of optimization 3.1.2:*

$$\begin{aligned}\sum_{i=s'+1}^{n-k}n-i &= \underbrace{\sum_{i=s+1}^{n-k}(n-i)}_{\substack{\#\text{binary operations per row}\\ \text{with optimization 3.1.2}}}-\sum_{i=s+1}^{s'}(n-i) \qquad (s'>s)\\ &= \sum_{i=s+1}^{n-k}(n-i)-\Bigg[\underbrace{\sum_{i=1}^{s'}(n-i)}_{b_{opt-3.1.3}}-\underbrace{\sum_{i=1}^{s}(n-i)}_{b_{opt-3.1.2}}\Bigg] \qquad (3.3)\end{aligned}$$

$$b_{opt-3.1.3} := \sum_{i=1}^{s'}(n-i)$$
$$= s'(n-\frac{1}{2}) - \frac{1}{2}(s')^2 \qquad (s' := n-k-x)$$

$b_{opt-3.1.3}$ is the number of bit operations per row that we save in comparison to the original Gaussian elimination process. Using the formulas above we can also easily compare this optimization with optimization 3.1.2 with regard to the number of binary operations required for one Gaussian elimination process. The overall comparison is far more complicated though:
Artificially swapping x-many columns instead of uniformly selecting $n-k$ columns from $\widehat{H}$ is identical to changing only x entries in the corresponding information set I. As we know from section 2.2.3 this implies that only x entries of the I-indexed entries of the real error vector $\vec{\epsilon}$ change, i.e. $\vec{\epsilon}_I$ and thus $\vec{\epsilon}$ (cf. equation (2.3)) cannot be assumed to be uniform anymore during each iteration of algorithm 3.1. So we cannot use $\overline{PR}_{ALG}[success = true]^{-1}$ anymore to model the average number of iterations. The average number of iterations will actually increase in comparison to the original algorithm. The difficulty lies in optimizing the parameter x, so that the fewer work for the Gaussian elimination step outperforms the more work from the additional iterations. As the work per iteration heavily depends on the concrete implementation of searchALG(), this optimization can only be done per algorithm. Nevertheless it is possible to do a general analysis of this optimization technique, which uses statistical properties of the implementation of searchALG() as parameters. This is done in appendix B. To use this general analysis in practical applications, one would have to determine those parameters and write a program to do a lot of matrix computations to find the optimal x for a concrete parameter set (n,k,w). This is for example done in [24] and as part of this thesis as well.
The idea of this optimization was originally introduced by Canteaut et al. in [12, 11] for $x = 1$, but generalized for arbitrary x by Bernstein, Lange and Peters in [14] (for Stern's algorithm only though). Note that $x = 1$ is often not optimal and can result in a worse overall number of binary operations for algorithm 3.1.
Applying this optimization can even have negative effects only, if the runtime of the Gaussian elimination process is negligible in comparison to the runtime of searchAlg(), which often happens to be the case in practice (cf. section 4).

Optimization 3.1.4 (Faster pivoting)**.** *The next optimization technique is an application of the Four-Russians speedup [9] to the Gaussian elimination process and a typical time-memory trade-off: We introduce a new parameter $r \geq 2$ and split the parity check matrix $H \in \mathbb{F}_2^{(n-k)\times n}$ into column blocks of size r (the last block may be smaller). Then we do the following:*

1. *We iterate over the blocks using the iteration index $l = 1, \ldots, \lceil \frac{n-k}{r} \rceil$ and apply our standard Gaussian elimination process to the current block $B_l \in \mathbb{F}_2^{(n-k)\times r}$ only. After the Gaussian elimination during iteration step l we obtain a matrix of the form $\begin{pmatrix} 0^{[((l-1)r)\times r]} \\ id^{[r]} \\ 0^{[(n-k-(l+1)r)\times r]} \end{pmatrix} \in \mathbb{F}_2^{(n-k)\times r}$. We save whatever row additions we applied per row symbolically in a bit array $a[i][j] = true/false$, where i is the index of the row that we changed ($1 \leq i \leq n-k$) and j indicates the row that we added ($\Leftrightarrow a[i][j] = true$). Since we only use the r rows indexed by $J = \{(l-1)r+1, \ldots, lr\}$ for row additions, we have $1 \leq j \leq r$. In comparison to the original algorithm we get an additional memory consumption of $mem\{a[i][j]\} = \mathcal{O}(r(n-k))$. If none of the aforementioned*

optimizations were applied, this step would imply at most $(n-k)\sum_{i=1}^{r}(r-i) = \frac{1}{2}(n-k)(r^2-r)$ *bit operations.*

2. *We compute all of the possible* 2^r-1 *sums of the* r *rows indexed by* J *(row swapping must be considered) for their remaining length* $n-lr$*. Saving those sums requires another* $\mathcal{O}(2^r(n-r))$ *of memory (more precisely:* $\mathcal{O}(2^r(n-lr))$ *during iteration* l*) and means a computational complexity of roughly* $2(2^r-1)(n-lr) \approx 2^{r+1}(n-lr)$ *binary operations, if each sum is computed from two summands only (we can first compute all sums of 2 rows, then all sums of 3 rows by using the sums of 2 rows and so on). Alternatively we could also compute the sums "on demand", i.e. whenever we need them in the next step, but were not computed before. This is even more efficient and should always be done, if other optimization techniques such as optimization 3.1.2 or 3.1.3 are applied as we do not need to do a lot (or even nothing at all) for many blocks with these optimizations. Without these optimizations however, one would expect every possible sum to occur anyway (for sufficiently small* r*).*

3. *Now we can use the array* $a[i][j]$ *in combination with the* 2^r-1 *precomputed sums to compute the matrix* H *as it would look like after applying the original Gaussian elimination to the first* lr *columns of* H *(apart from the columns of* X*). The advantage lies in the fact that we avoid multiple computations of the same subset sum. For example in the original row addition step of the Gaussian elimination process, every row above the pivoted row may be assumed to have a chance of* $\frac{1}{2}$ *to be added to another row (as long as* H *looks random). So any two rows above the pivoted row are added to another row with probability* $\frac{1}{4}$*. This implies that we may expect to compute the same sum of these two rows every 4 rows of* H*. Therefore it is worth saving the intermediate sums.*
   In this step $n-k$ *binary vectors of length* $n-lr$ *are added, i.e. we need* $(n-k)(n-lr)$ *binary operations.*

It is probably easier to understand this method together with figure 3.1: During the first iteration, we apply the standard Gaussian elimination process to both R *and* T *in step 1. It makes sense to also include* T*, because we need* r *linearly independent row vectors to be able to create the identity matrix* $id^{[r]}$ *within* R *and the all-zero matrix within* T*. During the Gaussian elimination we save whatever operations we applied to the original rows of* R *and* T*. This does not cost us any additional computational power, we just require some more memory. As second step we compute all of the possible* 2^r-1 *sums of the rows of* S *(over the length of* S *and the upper part of* X*). In step 3 we add these sums to the rows of* S *and* U *(over the length* $n-r$*, excluding* T*, but including* X*) according to the row operations that we saved during the Gaussian elimination process. This way no sum is computed twice neither for* S *nor for* U*. Afterwards we proceed to the second iteration.*
Figure 3.2 additionally shows the parity check matrix H *after three such iterations, i.e. during the fourth.*
The analysis of the runtime advantage of this optimization is non-trivial, especially if we compute the intermediate sums "on demand", possibly using previously computed intermediate sums. It also heavily depends on the parity check matrix H*, which cannot be assumed to have uniformly distributed entries anymore, if the optimizations 3.1.2 or 3.1.3 are used. In particular only* $r \leq x$ *makes sense, if optimization 3.1.3 is applied, because we would want to use "Faster pivoting" only on those* x *columns that really require row additions.*
Nevertheless let us simply assume that none of the aforementioned optimizations were applied (except for optimization 3.1.1) and that we do not compute intermediate sums "on demand". Then we can roughly estimate the overall number of binary operations of the

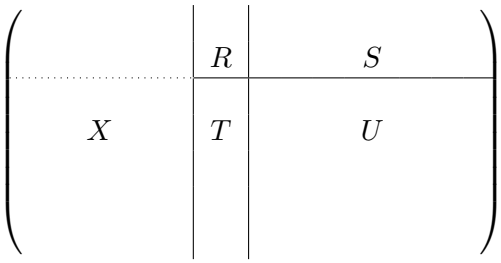


Figure 3.1: Parity Check Matrix H before the first iteration with $X \in \mathbb{F}_2^{(n-k)\times k}, R \in \mathbb{F}_2^{r\times r}, T \in \mathbb{F}_2^{(n-k-r)\times r}, S \in \mathbb{F}_2^{r\times(n-k-r)}, U \in \mathbb{F}_2^{(n-k-r)\times(n-k-r)}$; after the last iteration it will be in the form $\widehat{H} = (Q \mid \mathrm{id}^{[n-k]})$, where Q results from the computations on X.

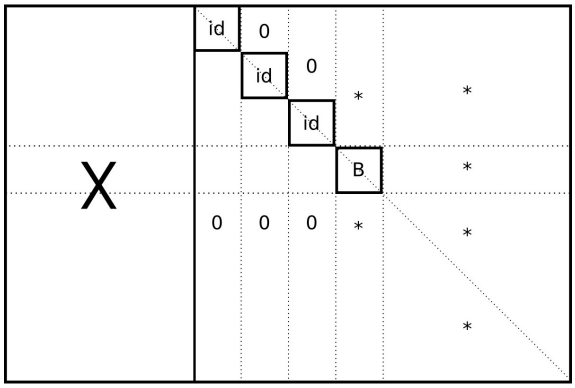


Figure 3.2: Parity Check Matrix H during the fourth iteration. $\mathrm{B} \in \mathbb{F}_2^{r\times r}$ is the block that the Gaussian elimination process is currently working on. $\mathrm{id} := \mathrm{id}^{[r]}$ are finished blocks.

Gaussian elimination process in combination with this optimization as follows:

$$
\begin{aligned}
&time\{randomize()[o_{3.1.1}, o_{3.1.4}]\} \\
&\approx \sum_{l=1}^{\lceil\frac{n-k}{r}\rceil} \frac{1}{2}(n-k)(r^2-r) + 2^{r+1}(n-lr) + (n-k)(n-lr) && (3.4)\\
&= \frac{(n-k)^2(r-1)}{2} + \frac{n-k}{r}\left[2^{r+1}n + (n-k)n\right] - \sum_{l=1}^{\lceil\frac{n-k}{r}\rceil} lr(2^{r+1}+n-k) \\
&= \frac{(n-k)^2(r-1)}{2} + \frac{(n-k)2^{r+1}n}{r} + \frac{(n-k)^2 n}{r} - \left(\frac{(n-k)^2}{2r} + \frac{n-k}{2}\right)(2^{r+1}+(n-k)) \\
&\approx \frac{2^{r+1}(n-k)^2}{r} + \frac{(n-k)^3}{r} - \frac{2^{r+1}(n-k)^2}{2r} - \frac{(n-k)^3}{2r} && (3.5)\\
&= \frac{1}{2}\left(\frac{2^{r+1}(n-k)^2}{r} + \frac{(n-k)^3}{r}\right) && (3.6)
\end{aligned}
$$

Note that equation (3.5) is a rough estimate of the previous equation neglecting all polynomials equal to $\mathcal{O}((n-k)^2)$. However equation (3.6) is sufficient to get an idea of the number of binary operations of one Gaussian elimination process with this optimization. As one might see, choosing $r = \log_2(n-k) - 1$ is probably optimal, because any other choice would make one part of the sum in equation (3.6) become larger. This observation even holds for every one of the l iterations. For this choice of r we get $time\{randomize()[o_{3.1.1}, o_{3.1.4}]\} \approx \frac{(n-k)^3}{\log_2(n-k)-1}$, which seems significantly better than the

runtime of the original Gaussian elimination.
Bernstein, Lange and Peters did a similar, but possibly more precise analysis and state in [14] that "the optimal choice of r is roughly $\log_2(n-k) - \log_2(\log_2(n-k))$ but interacts with the optimal choice of x" for optimization 3.1.3.
Also observe that any choice of $2 \leq r \leq x \leq n-k$ is beneficial for the runtime of algorithm 3.1, if we compute the intermediate sums whenever we need them ("on demand"), but also cache them. In that case we gain an advantage over the original runtime whenever any intermediate sum was previously computed (neglecting the time to look up the precomputed sums, e.g. from an array).
Regarding the memory consumption of algorithm 3.1 in combination with this optimization we have $mem\{randomize()[o_{3.1.4}]\} = \mathcal{O}(n^2 + r(n-k) + 2^r(n-r))$.

Table 4.1 contains an overview of the previous three optimization techniques.
All of the optimization techniques explained here can help to improve the runtime of any of the algorithms presented in the next sections. Any more optimization techniques to speed up the Gaussian elimination process in general may also be considered.

3.2 Prange

In 1962 Eugene Prange described the first information set decoding algorithm in [27], which is basically algorithm 3.1 with the function searchALG() as defined by algorithm 3.2. He observed that for $\vec{\epsilon}_I = \vec{0}$, i.e. if the k entries of the error vector $\vec{\epsilon}$ indexed by the information set I all happen to be zero, then the syndrome $\vec{\varsigma}$ reveals the entries of $\vec{\epsilon}_{I^*}$ ($I^* := \{1, \ldots, n\} \setminus I$). We can easily verify this observation from equation (2.3). Also note that all of the w-many errors must occur in $\vec{\epsilon}_{I^*}$, if $\vec{\epsilon}_I = \vec{0} \in \mathbb{F}_2^k$. In that case we have $\vec{\epsilon}_{I^*} = \vec{\varsigma}$ according to equation (2.3) and thus $\mathrm{wt}(\vec{\epsilon}_{I^*}) = \mathrm{wt}(\vec{\varsigma}) = w$, which is exactly the condition that we test in line 1 of algorithm 3.2.

Algorithm 3.2: searchPrange()

Input: parity check matrix $\widehat{H} = (Q \mid \mathrm{id}^{[n-k]}) \in \mathbb{F}_2^{(n-k)\times n}$, syndrome $\vec{\varsigma} \in \mathbb{F}_2^{n-k}$, $w = \mathrm{wt}(\vec{e})$

Output: success indicator (true/false), error vector $\vec{\epsilon} \in \mathbb{F}_2^n$

1 **if** $wt(\vec{\varsigma}) = w$ **then**
2 $\vec{\epsilon}_I := \vec{0} \in \mathbb{F}_2^k$
3 $\vec{\epsilon} \leftarrow \texttt{prepend}(\vec{\epsilon}_I, \vec{\varsigma})$ // $\vec{\varsigma} = \vec{\epsilon}_{I^*}$
4 **return** $(true, \vec{\epsilon})$
5 **else**
6 **return** $(false, \vec{0})$
7 **end**

Remark 3.2.1. *Prange's algorithm is sometimes also called "plain information set decoding".*

Remark 3.2.2. *Algorithm 3.2 uses the fact that* $wt(\vec{\epsilon}_{I^*}) = w \Rightarrow wt(\vec{\epsilon}_I) = 0 \Rightarrow \vec{\epsilon}_I = \vec{0} \in \mathbb{F}_2^k$. *However the zero-vector is the only vector, where the weight directly defines the vector. If we wanted to check for a different pattern* $\vec{\epsilon}_I$ *with a different weight* $0 \leq p \leq w$, *this would imply* $\binom{k}{p}$ *possibilities for* $\vec{\epsilon}_I$. *Choosing* $p = 0$ *seems a natural choice to keep this number low. Nevertheless this choice is not optimal as will be explained in the next section.*

Obviously we have

$$time\{searchPrange()\} = \mathcal{O}(1) \tag{3.7}$$

$$mem\{searchPrange()\} = \mathcal{O}(n) \tag{3.8}$$

$$\overline{PR}_{Prange}[success = true] = \frac{\binom{n-k}{w}}{\binom{n}{w}} \tag{3.9}$$

The asymptotic runtime is evaluated in the next section as Prange's algorithm is just a special case ($p = 0$) of Lee-Brickell's algorithm.
Even though the other algorithms have significantly better success probabilities than Prange's algorithm, none of them exposes the nice feature of not adding any additional runtime during each iteration ($time\{searchPrange()\} = \mathcal{O}(1)$). So whenever we execute an algorithm with a structure as in algorithm 3.1, it does not hurt to additionally check the *Prange condition* $\mathrm{wt}(\vec{\varsigma}) \stackrel{?}{=} w$. This might be interesting in practice, but is negligible in theory due to the comparatively small success probability of Prange's algorithm.
Other than that, Prange's algorithm is only interesting for historical reasons.

3.3 Lee-Brickell

Lee-Brickell's algorithm is a generalization of Prange's algorithm for arbitrary patterns of $\vec{\epsilon}_I$ with $\mathrm{wt}(\vec{\epsilon}_I) = p$, $0 \leq p \leq w$. The algorithm was published by Pil Joong Lee and Ernest F. Brickell in [28] and can be described as algorithm 3.1 with the function searchALG() as defined by algorithm 3.3.
It is based on the idea that it is unlikely for $\vec{\epsilon}$ to spread *all* 1's over its last $n-k$ coordinates (i.e. over $\vec{\epsilon}_{I^*}$ only) as with Prange's algorithm. Instead, Lee and Brickell decided to allow exactly p many 1's in the first k coordinates of $\vec{\epsilon}$, i.e. during each iteration of 3.1 they hoped for an error vector $\vec{\epsilon}$ with $\mathrm{wt}(\vec{\epsilon}_I) = p$.
Since for $p > 0$ there are $\binom{k}{p}$ many possible vectors $\vec{\epsilon}_I$ with $\mathrm{wt}(\vec{\epsilon}_I) = p$, algorithm 3.3 iterates over all of them in line 1.

Remark 3.3.1. *Iterating over all of these vectors can be achieved very efficiently. We can start with the vector $\vec{\epsilon}_{I,1} = (1, 1, 1, \ldots, 0, 0, 0)^T$ with p many 1's at the front. Assuming a vector $\vec{\epsilon}_{I,j}$ we can then generate the next $\vec{\epsilon}_{I,j+1}$ as follows:*

1. *Find the rightmost 1 followed by a 0 at position i in $\vec{\epsilon}_{I,j}$ (or keep track of that position). If this is not possible, stop.*
2. *Count the number of 1's at the positions $\geq i$ and set all of them to 0 (including the 1 at position i). Store this number in the variable β.*
3. *Set the entries at the positions $i+1, \ldots, i+1+\beta$ to 1. Use the resulting vector as $\vec{\epsilon}_{I,j+1}$.*

For each of the vectors $\vec{\epsilon}_I$ with $\mathrm{wt}(\vec{\epsilon}_I) = p$, equation (2.3) (in $\mathbb{F}_2$) is used to compute the corresponding $\vec{\epsilon}_{I^*}$ in line 2. If $\mathrm{wt}(\vec{\epsilon}_{I^*}) \stackrel{?}{=} w - p$ holds in line 3, we know that the resulting error vector $\vec{\epsilon}$ has weight w. As already explained in section 2.2.3 only the *real* error vector $\vec{\epsilon}$ can have that weight, because the existence of any other error vector with the same weight would prevent us from uniquely decoding the linear code.
If none of the $\vec{\epsilon}_{I^*}$'s happen to satisfy the condition in line 3 of algorithm 3.3, $success = false$ is returned in line 8 and we can simply hope for a different information set I that exposes the desired property $\mathrm{wt}(\vec{\epsilon}_I) = p$ during the next iteration of algorithm 3.1.

Algorithm 3.3: searchLB()

Input: parity check matrix $\widehat{H} = (Q \mid \mathrm{id}^{[n-k]}) \in \mathbb{F}_2^{(n-k)\times n}$, syndrome $\vec{\varsigma} \in \mathbb{F}_2^{n-k}$, $w = \mathrm{wt}(\vec{e})$, algorithmic parameter $0 \leq p \leq w$
Output: success indicator (true/false), error vector $\vec{\epsilon} \in \mathbb{F}_2^n$

```
1 foreach ε_I ∈ F_2^k, wt(ε_I) = p do
2     ε_{I*} := ς + Qε_I
3     if wt(ε_{I*}) = w − p then
4         ε ← prepend(ε_I, ε_{I*})
5         return (true, ε)
6     end
7 end
8 return (false, 0)
```

The parameter p needs to be optimized with regard to the overall runtime of algorithm 3.1 in combination with algorithm 3.3. To do so, we first need to define the important algo-

rithmic properties:

$$time\left\{searchLB()\right\} = \binom{k}{p} \cdot (p(n-k) + \mathcal{O}(1)) \tag{3.10}$$

$$mem\left\{searchLB()\right\} = \mathcal{O}(n) \tag{3.11}$$

$$\overline{PR}_{LB}[success = true] = \frac{\binom{k}{p}\binom{n-k}{w-p}}{\binom{n}{w}} \tag{3.12}$$

Thereby $time\left\{searchLB()\right\} = \binom{k}{p} \cdot (p(n-k) + \mathcal{O}(1))$ follows from the observations that each line of algorithm 3.3 is executed exactly $\binom{k}{p}$ times and that most work is done during the multiplication $Q\vec{\epsilon}_I$ in line 2 of algorithm 3.3. Multiplying the matrix $Q \in \mathbb{F}_2^{(n-k)\times k}$ by the vector $\vec{\epsilon}_I \in \mathbb{F}_2^k$ with $\mathrm{wt}\,(\vec{\epsilon}_I) = p$ means a selection of exactly p columns of Q and an addition of these columns that each have $n-k$ entries. Hence the complexity of this multiplication is $p(n-k)$ per iteration of algorithm 3.3.
In practice it is wise to compute $time\left\{isd()\right\} = time\left\{\text{Lee-Brickell}\right\}$ for a concrete parameter set (n,k,w) according to equation (3.1) for any choice of $p \in \{0,1,2,3,4\}$ and simply use the one that features the best runtime of algorithm 3.1. Alternatively it is possible to use the following explicit bound on $\overline{PR}_{LB}[success = true]$ provided by [15, 25] in combination with equation (3.1) and any number of optimizations to (roughly) determine the p that results in the best runtime for a parameter set (n,k,w):

$$\overline{PR}_{LB}[success = true] = 2^{-\alpha(R,W)n}\frac{1}{p!}\left(\frac{RWn}{1-R-W}\right)^p \frac{1}{\beta(R,W)} err_{LB}(n,k,w,p) \tag{3.13}$$

$R = \frac{k}{n}$ is the code rate, $W = \frac{w}{n}$ the error rate and $\alpha(R,W)$ is defined as in definition 2.2.8. Regarding the other unknowns we have

$$\beta(R,W) := \sqrt{(1-R-W)/((1-R)(1-W))} \tag{3.14}$$

$$\left(\frac{(1-\frac{p}{k})(1-\frac{p}{w})}{1+\frac{p}{n-k-w}}\right)^p e^{-\frac{1}{12n}\left(1+\frac{1}{1-R-W}\right)} < err_{LB}(n,k,w,p) < e^{\frac{1}{12n}\left(\frac{1}{1-R}+\frac{1}{1-W}\right)}$$

$$\text{with} \lim_{n\to\infty} err_{LB}(n,k,w,p) = 1$$

Note that we assume $0 < W < 1-R < 1$, which implies $0 < \beta(R,W) < 1$.
Although we cannot determine $err_{LB}(n,k,w,p)$ exactly, the bound above might be sufficient to compute the optimal p for a parameter set (n,k,w) in combination with the equations from section 3.1.
In theory one usually determines the optimal p as the one that features the best asymptotic ($n \to \infty$) runtime of algorithm 3.1. This p however is not necessarily optimal for all parameter sets (n,k,w). In any case it is important to respect the amount of work coming from the randomize() function in equation (3.1) including possible optimizations. Without $time\left\{randomize()\right\}$ in equation (3.1) $p = 0$, i.e. Prange's algorithm, would always be the optimal choice. For the overall runtime of Lee-Brickell's algorithm we know from equation (3.1) and the previously mentioned explicit bound that

$$time\left\{\text{Lee-Brickell}\right\} = 2^{\alpha(R,W)n} \cdot p! \left(\frac{1-R-W}{RWn}\right)^p \beta(R,W) \cdot err_{LB}(n,k,w,p)^{-1} \cdot \left(\mathcal{O}(n^3) + \binom{Rn}{p} \cdot (pn(1-R) + \mathcal{O}(1))\right) \tag{3.15}$$

$$time\left\{\text{Lee-Brickell } (p=0)\right\} = 2^{\alpha(R,W)n} \cdot \beta(R,W) \cdot err_{LB}(n,k,w,0)^{-1} \cdot \mathcal{O}(n^3)$$

$$
\begin{aligned}
time\,\{\text{Lee-Brickell } (p=1)\} &= 2^{\alpha(R,W)n}\cdot\frac{1-R-W}{RWn}\beta(R,W)\cdot err_{LB}(n,k,w,1)^{-1}\\
&\quad\cdot\left(\mathcal{O}(n^3)+Rn^2(1-R)\right)\\
&= 2^{\alpha(R,W)n}\cdot\frac{1-R-W}{RW}\beta(R,W)\cdot err_{LB}(n,k,w,1)^{-1}\cdot\mathcal{O}(n^2)\\
time\,\{\text{Lee-Brickell } (p=2)\} &= 2^{\alpha(R,W)n}\cdot\frac{(1-R-W)^2}{(RWn)^2}\beta(R,W)\cdot err_{LB}(n,k,w,2)^{-1}\\
&\quad\cdot\left(\mathcal{O}(n^3)+2Rn^2(Rn-1)(1-R)\right)\\
&= 2^{\alpha(R,W)n}\cdot\frac{(1-R-W)^2}{(RW)^2}\beta(R,W)\cdot err_{LB}(n,k,w,2)^{-1}\cdot\mathcal{O}(n)\\
time\,\{\text{Lee-Brickell } (p=3)\} &= 2^{\alpha(R,W)n}\cdot\frac{(1-R-W)^3}{(RWn)^3}\beta(R,W)\cdot err_{LB}(n,k,w,3)^{-1}\\
&\quad\cdot\left(\mathcal{O}(n^3)+3Rn^2(Rn-1)(Rn-2)(1-R)\right)
\end{aligned}
\tag{3.16}
$$

As $err_{LB}(n,k,w,p)=\Theta(1)$, these equations enable us to easily determine the asymptotically optimal p: It seems as if for $p\in\{0,1\}$ the Gaussian elimination step from randomize() in algorithm 3.1 is dominant, whereas for $p\geq 2$ the $\binom{k}{p}$ iterations in searchLB() become more important. $p=2$ looks like the asymptotically optimal choice, although $p=3$ might be better for certain parameters (R,W). Note that these asymptotic observations are independent of any of the optimizations mentioned in section 3.1. The effect of optimization 3.3.1 is also negligible for the choice of an asymptotically optimal p as the most important factor $\binom{k}{p}$ does not vanish.

May, Meurer and Thomae provide a rough bound for Lee-Brickell's algorithm of $2^{0.05752n}$ for $p=0$ in [21][3].

From the explicit bounds above we can also compute the factor that we asymptotically gain from using Lee-Brickell's algorithm ($p=2$) instead of Prange's algorithm as

$$
\begin{aligned}
Adv(\text{Lee-Brickell } (p=2) > \text{Lee-Brickell } (p=0)) &= Adv(\text{Lee-Brickell } (p=2) > \text{Prange})\\
&= \frac{(1-R-W)^2}{(RW)^2}\cdot\Theta(1)\cdot\mathcal{O}\left(\frac{1}{n^2}\right)\\
&= \begin{cases}
\mathcal{O}\left(\frac{\log_2(n)^2}{n^2}\right) & \text{for } W=\Theta\left(\frac{1}{\log_2(n)}\right),\ R=\Theta(1)\\
\mathcal{O}\left(\frac{1}{n^2}\right) & \text{for } W=\Theta(1),\ R=\Theta(1)
\end{cases}
\end{aligned}
$$

In order to understand those equations, recall from remark 2.2.8 that $W=\Theta(1/\log_2(n))$ is true for Goppa codes and thus the McEliece cryptosystem, whereas $W=\Theta(1)$ is true for many other linear codes.

Actually these equations are even exact asymptotic bounds (i.e. Θ instead of $\mathcal{O}$), if only optimization 3.1.1 is used. Otherwise the exact bounds might be better. The same is true for equation (3.15) and the other equations.

Optimizations

Optimization 3.3.1 (Reusing additions of the $(n-k)$-bit vectors)**.** *As already mentioned the probably most expensive computation in algorithm 3.3 is $Q\vec{\epsilon}_I, wt(\vec{\epsilon}_I)=p$ in line 2, which we have to do for each set of p out of the k columns of Q, i.e. $\binom{k}{p}$ times. $Q\vec{\epsilon}_I$ basically means an addition of exactly those columns of Q that are selected by the entries of $\vec{\epsilon}_I$. The*

[3] Their analysis ignores $time\,\{randomize()\}$, which makes $p=0$ optimal.

naive way would mean $p-1$ additions of column vectors $\vec{v}_i \in \mathbb{F}_2^{n-k}$. As chances are high that the next selection of p columns contains some of the previously selected columns, most of these additions reoccur. We can prevent this by first computing all $\binom{k}{2}$ possible sums of 2 columns of Q; each sum meaning one addition in $\mathbb{F}_2^{n-k}$. Then we can add an extra column to the previous results and thus compute all $\binom{k}{3}$ sums of 3 columns of Q; and so on until we compute the desired $\binom{k}{p}$ sums of p columns of Q.
In this case we have

$$\begin{aligned} time\{searchLB()[o_{3.3.1}]\} &= (n-k)\left(\binom{k}{2}+\binom{k}{3}+\ldots+\binom{k}{p}\right)+\mathcal{O}(1)\cdot\binom{k}{p} \\ &= \binom{k}{p}\cdot\left((n-k)\left(1+\frac{\binom{k}{p-1}}{\binom{k}{p}}+\ldots+\frac{\binom{k}{2}}{\binom{k}{p}}\right)+\mathcal{O}(1)\right) \\ &= \binom{k}{p}\cdot\left((n-k)\left(1+\underbrace{\frac{p}{k-p+1}+\ldots+\frac{(k-p)!p!}{(k-2)!2!}}_{\delta(k,p)}\right)+\mathcal{O}(1)\right) \end{aligned}$$

So for the operation $Q\vec{\epsilon}_I$ we gain a runtime factor of $\frac{p}{1+\delta(k,p)}$ over the naive way of doing it. Note that $\delta(k,p) \ll 1$ as long as $k \gg p$. In contrast we require more memory as we compute the $\binom{k}{p}$ sums all at once and need to save them in order to iterate over them in line 1 of algorithm 3.3. Thus we get a memory consumption of $mem\{searchLB()[o_{3.3.1}]\} = \mathcal{O}(\binom{k}{p}(n-k)+n)$, which makes this optimization a classical time-memory trade-off.

Optimization 3.3.2 (Early abort)**.** *The operation $Q\vec{\epsilon}_I$ in line 2 of algorithm 3.3 means a selection and addition of p columns of Q, i.e. we need to compute $p-1$ column sums on a length of $n-k$ bits. An early abort strategy can reduce this number: Since we are only interested in an $\vec{\epsilon}_{I^*}$ with $wt(\vec{\epsilon}_{I^*}) = w-p$ (line 3), we can iteratively compute the entries of $\vec{\epsilon}_{I^*} \in \mathbb{F}_2^{n-k}$ (i.e. we know $(\vec{\epsilon}_{I^*})_{[i]}$ during iteration i) and check their weight during each iteration. If we see a weight larger than $w-p$, we can already stop our computations for that candidate for the real $\vec{\epsilon}_{I^*}$. As Q and thus $\vec{\epsilon}_{I^*}$ may be assumed to be uniformly distributed, we may expect such an early abort after roughly $2(w-p+1)$ iterations. So all in all we can compute every $\vec{\epsilon}_{I^*}$ with at about $2(w-p+1)(p-1) \approx 2(w-p+1)p$ instead of $(n-k)(p-1) \approx (n-k)p$ binary operations. This results in*

$$time\{searchLB()[o_{3.3.2}]\} := \binom{k}{p}\cdot(2p(w-p+1)+\mathcal{O}(1))$$

whilst the memory consumption remains the same.
Note that the "+1" in the factor $(w-p+1)$ can become significant for the choice $p=w$.

Remark 3.3.2. *Note that the previous two optimizations can be used together, if we apply optimization 3.3.1 for columns of length $\approx 2(w-p+1)$ only.*

Other than that there are methods to refine Lee-Brickell's algorithm by additionally requiring that $\vec{\epsilon}_{I^*}$ has weight 0 on l positions. This idea is meant to reduce the number of binary operations required in line 2 of algorithm 3.3 as we can first compute $(\vec{\epsilon}_{I^*})_{[l]} = \vec{\varsigma}_{[l]} + (Q\vec{\epsilon}_I)_{[l]}$ (pl binary operations), check whether $(\vec{\epsilon}_{I^*})_{[l]} = \vec{0} \in \mathbb{F}_2^l$ holds and do more expensive operations such as the one of line 2 only in that case. However fixing l-many zeroes within the error vector $\vec{\epsilon}$ reduces the success probability and thus increases the overall number of iterations. A similar but more advanced idea is employed for Stern's algorithm, which is presented in the next section. Therefore we do not discuss it in depth for Lee-Brickell's algorithm; the interested reader is rather referred to [31].

3.4 Stern

To understand Stern's algorithm from [22] it is necessary to reconsider equation (2.2): Denote by $\vec{\epsilon}_{I_1}$ and $\vec{\epsilon}_{I_2}$ the entries of the error vector $\vec{\epsilon}$ indexed by the sets $I_1, I_2 \subseteq I$. As usual, $I = \{1, 2, \ldots, k\}$ is the currently selected information set. Assuming $I_1 \cup I_2 = I$ and $I_1 \cap I_2 = \emptyset$ we can then rewrite the term $Q\vec{\epsilon}_I$ as $Q(\vec{\epsilon}_{I_1,0} + \vec{\epsilon}_{I_2,0})$, where $\vec{\epsilon}_{I_1,0} \in \mathbb{F}_2^k$ and $\vec{\epsilon}_{I_2,0} \in \mathbb{F}_2^k$ are the vectors $\vec{\epsilon}_{I_1}$ and $\vec{\epsilon}_{I_2}$ with zeros padded at the bottom and top respectively. A visual explanation is given by figure 3.3.

$$Q\vec{\epsilon}_I = \left(\begin{array}{c|c} Q_1 & Q_2 \end{array} \right) \left(\begin{array}{c} \left(\begin{array}{c} \vec{\epsilon}_{I_1} \\ \hline \vec{0} \end{array} \right) + \left(\begin{array}{c} \vec{0} \\ \hline \vec{\epsilon}_{I_2} \end{array} \right) \end{array} \right)$$

Figure 3.3: $Q\vec{\epsilon}_I = Q(\vec{\epsilon}_{I_1,0} + \vec{\epsilon}_{I_2,0})$

Let us define Q_1 as those columns of Q indexed by I_1 and Q_2 as those indexed by I_2. It immediately follows that

$$Q\vec{\epsilon}_I = Q_1\vec{\epsilon}_{I_1} + Q_2\vec{\epsilon}_{I_2} \tag{3.17}$$

To be precise, Stern decided to choose two uniform subsets I_1, I_2 of (almost) identical size from the information set I. Since the information set I was already chosen uniformly at random by the column permutations of randomize() in algorithm 3.1, we can simply define $I_1 := \{1, \ldots, \lceil \frac{k}{2} \rceil\}$ and $I_2 := \{\lceil \frac{k}{2} \rceil + 1, \ldots, k\}$. Stern adapted Lee-Brickell's idea to look for error vectors $\vec{\epsilon}$ with $\mathrm{wt}(\vec{\epsilon}_I) = p$ by looking for the vectors $\vec{\epsilon}_{I_1}$ and $\vec{\epsilon}_{I_2}$ as defined above with $\mathrm{wt}(\vec{\epsilon}_{I_1}) = \mathrm{wt}(\vec{\epsilon}_{I_2}) = \frac{p}{2}$ in the lines 1 and 2 of algorithm 3.4. Note that this change implies less possibilities[4] for $\vec{\epsilon}_I$ than in Lee-Brickell's algorithm (cf. section 3.3). However even if we used every combination of $\vec{\epsilon}_{I_1}$ and $\vec{\epsilon}_{I_2}$ to express $\vec{\epsilon}_I$ and then compute $\vec{\epsilon}_{I^*}$ as $\vec{\varsigma} + Q\vec{\epsilon}_I$ (as in Lee-Brickell's algorithm), we would still get lots of iterations without any means to control that number. That's probably why Stern introduced the additional parameter $0 < l \leq n - k$ and demanded that $(\vec{\epsilon}_{I^*})_{[l]} = \vec{0}$, i.e. that the first l coordinates of $\vec{\epsilon}_{I^*}$ all be zero. So all in all he decided to look for an error vector $\vec{\epsilon}$ with a weight distribution as depicted in figure 3.4. A comparison of the weight distributions of the error vector $\vec{\epsilon}$ for various information set decoding algorithms is shown in figure 4.2 on page 69.

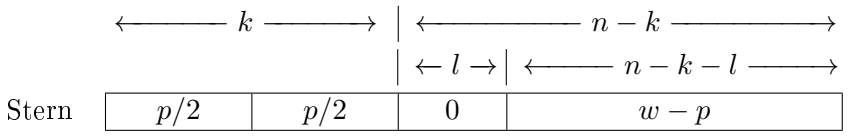


Figure 3.4: weight distribution of $\vec{\epsilon}$ as demanded by Stern

If we assume $(\vec{\epsilon}_{I^*})_{[l]} = \vec{0}$, we require from equation (2.3) in combination with equation (3.17) that

$$\vec{\varsigma}_{[l]} + (Q\vec{\epsilon}_I)_{[l]} = \vec{\varsigma}_{[l]} + (Q_1\vec{\epsilon}_{I_1})_{[l]} + (Q_2\vec{\epsilon}_{I_2})_{[l]} = (\vec{\epsilon}_{I^*})_{[l]} \stackrel{!}{=} \vec{0} \qquad (\text{in } \mathbb{F}_2) \tag{3.18}$$

So basically we can look for vectors $\vec{\epsilon}_{I_1}$ and $\vec{\epsilon}_{I_2}$ so that

$$\vec{\varsigma}_{[l]} + (Q_1\vec{\epsilon}_{I_1})_{[l]} \stackrel{!}{=} (Q_2\vec{\epsilon}_{I_2})_{[l]}$$

[4] A factor of $\binom{k}{p}^{-1}\binom{k/2}{p/2}^2$ for this change only (as long as we choose p to be identical for both algorithms).

Whenever such a *collision* between $\vec{\varsigma}_{[l]} + (Q_1\vec{\epsilon}_{I_1})_{[l]}$ and $(Q_2\vec{\epsilon}_{I_2})_{[l]}$ occurs, we found an error vector that exposes the desired weight distribution from figure 3.4 at least for the first $k + l$ entries. Equation (2.3) enables us to compute the whole $\vec{\epsilon}_{I^*}$ and thus check the weight distribution for the remaining $n - k - l$ entries.
In line 1 of algorithm 3.4 we compute $\vec{\varsigma}_{[l]} + (Q_1\vec{\epsilon}_{I_1})_{[l]}$ for every possible $\vec{\epsilon}_{I_1}$ with weight $\frac{p}{2}$ and store the results in a list $\mathcal{L}_1$. In line 2 we compute the first l entries of $Q_2\vec{\epsilon}_{I_2}$ for every $\vec{\epsilon}_{I_2}$ with weight $\frac{p}{2}$ and store the results in a list $\mathcal{L}_2$. Finally we look for the aforementioned collisions in line 3 of algorithm 3.4. Whenever such a collision is found, $\vec{\epsilon}_{I^*}$ is computed according to equation (2.3) and Stern's algorithm tests whether $\vec{\epsilon}_{I^*}$ has the correct weight $w - p$ in line 6. If it does, the real error vector $\vec{\epsilon}$ was found and $success = true$ is returned. If it does not, we can hope for another collision or better luck with another information set during the next iteration of algorithm 3.1.

Remark 3.4.1. *The list $\mathcal{L}_2$ is not really needed: We could also check for collisions (line 3 of algorithm 3.4) directly after computing a single entry of $\mathcal{L}_2$ (line 2). Nevertheless we introduced $\mathcal{L}_2$ for the sake of clarity as it makes runtime observations easier to understand. Not using $\mathcal{L}_2$ also reduces the overall memory consumption, which is reflected by equation (3.20).*

Algorithm 3.4: searchStern()

Input: parity check matrix $\widehat{H} = (Q \mid \mathrm{id}^{[n-k]}) \in \mathbb{F}_2^{(n-k)\times n}$, syndrome $\vec{\varsigma} \in \mathbb{F}_2^{n-k}$, $w = \mathrm{wt}(\vec{e})$, algorithmic parameter $0 \leq p \leq w$, algorithmic parameter $0 \leq l \leq n - k - w + p$
Output: success indicator (true/false), error vector $\vec{\epsilon} \in \mathbb{F}_2^n$

1 **foreach** $\vec{\epsilon}_{I_1} \in \mathbb{F}_2^{\lceil k/2 \rceil}, wt(\vec{\epsilon}_{I_1}) = \frac{p}{2}$ **do** $\mathcal{L}_1[\vec{\epsilon}_{I_1}] \leftarrow \vec{\varsigma}_{[l]} + (Q_1\vec{\epsilon}_{I_1})_{[l]}$
2 **foreach** $\vec{\epsilon}_{I_2} \in \mathbb{F}_2^{\lfloor k/2 \rfloor}, wt(\vec{\epsilon}_{I_2}) = \frac{p}{2}$ **do** $\mathcal{L}_2[\vec{\epsilon}_{I_2}] \leftarrow (Q_2\vec{\epsilon}_{I_2})_{[l]}$
3 **foreach** $\vec{\epsilon}_{I_1}, \vec{\epsilon}_{I_2}, \mathcal{L}_1[\vec{\epsilon}_{I_1}] = \mathcal{L}_2[\vec{\epsilon}_{I_2}]$ **do**
4 $\vec{\epsilon}_I \leftarrow \texttt{prepend}(\vec{\epsilon}_{I_1}, \vec{\epsilon}_{I_2})$
5 $\vec{\epsilon}_{I^*} := \vec{\varsigma} + Q\vec{\epsilon}_I$
6 **if** $wt(\vec{\epsilon}_{I^*}) = w - p$ **then**
7 $\vec{\epsilon} \leftarrow \texttt{prepend}(\vec{\epsilon}_I, \vec{\epsilon}_{I^*})$
8 **return** $(true, \vec{\epsilon})$
9 **end**
10 **end**
11 **return** $(false, \vec{0})$

Let us have a look at the properties of this algorithm:

$$time\{searchStern()\} = \binom{k/2}{p/2} pl + \frac{\binom{k/2}{p/2}^2}{2^l} \cdot (p(n-k-l) + \mathcal{O}(1)) \tag{3.19}$$

$$mem\{searchStern()\} = \mathcal{O}\left((l + k/2) \cdot \binom{k/2}{p/2}\right) \tag{3.20}$$

$$\overline{PR}_{Stern}[success = true] = \frac{\binom{k/2}{p/2}^2 \binom{n-k-l}{w-p}}{\binom{n}{w}} \tag{3.21}$$

These equations follow from relatively basic observations: Assuming that k and p are even, we have exactly $\binom{k/2}{p/2}$ many $\vec{\epsilon}_{I_1}$'s and the same number of $\vec{\epsilon}_{I_2}$'s in line 1 and line 2 of algorithm 3.4. According to remark 3.4.1 we need to store at least the list $\mathcal{L}_1$. For each

of its entries we need to save l bits as well as the vectors $\vec{\epsilon}_{I_1}$ themselves, which explains equation (3.20).
The terms $(Q_1\vec{\epsilon}_{I_1})_{[l]}$ and $(Q_2\vec{\epsilon}_{I_2})_{[l]}$ with $\mathrm{wt}\,(\vec{\epsilon}_{I_1}) = \mathrm{wt}\,(\vec{\epsilon}_{I_2}) = \frac{p}{2}$ mean a selection of $\frac{p}{2}$ columns of Q_1 or Q_2 and an addition of the first l bits of these columns. Therefore line 1 and 2 of algorithm 3.4 cost roughly $\binom{k/2}{p/2}pl$ binary additions/operations. Naively computing line 5 of algorithm 3.4 would mean a selection of p columns of Q and adding up those $n-k-l$ bit columns (the first l bits of $\vec{\epsilon}_{I^*}$ are known to be zero). Since we assume the matrices Q_1 and Q_2 to be uniform, the resulting vectors $\vec{\epsilon}_{I_1}$ and $\vec{\epsilon}_{I_2}$ can also be assumed to look uniform. Thus we may expect a l-bit collision between any two of those vectors to occur with probability $\frac{1}{2^l}$. As $\binom{k/2}{p/2}^2$ possible combinations of $\vec{\epsilon}_{I_1}$ and $\vec{\epsilon}_{I_2}$ exist, we obtain equation (3.19).

Remark 3.4.2. *It is an assumption of our runtime model that finding all collisions between the entries of the two lists $\mathcal{L}_1, \mathcal{L}_2$ can be achieved in time $\mathcal{O}(|\mathcal{L}_1| + |\mathcal{L}_2| + C)$ (where $0 \leq C \leq |\mathcal{L}_1||\mathcal{L}_2|$ denotes the expected number of collisions) and therefore does not introduce any logarithmic factors into equation (3.19). Let us outline how this can be done in practice: Instead of using the list $\mathcal{L}_1$, one would usually employ a map $\mathcal{M}_1$ (e.g. a hash map) and store every $\vec{\epsilon}_{I_1}$ at the position indexed by the result of the computation $\vec{\varsigma}_{[l]} + (Q_1\vec{\epsilon}_{I_1})_{[l]}$, i.e. $\mathcal{M}_1[\vec{\varsigma}_{[l]} + (Q_1\vec{\epsilon}_{I_1})_{[l]}] \leftarrow \vec{\epsilon}_{I_1}$. Note the inverse order in comparison to line 1 of algorithm 3.4. So the map is basically a function $\mathcal{M}_1 : \mathbb{F}_2^l \to \mathbb{F}_2^{\lceil k/2 \rceil}$; just imagine storing values $\vec{\epsilon}_{I_1} \in \mathbb{F}_2^{\lceil k/2 \rceil}$ in 2^l many buckets. Since Q is a random matrix, we may expect $\binom{k/2}{p/2} \cdot 2^{-l}$ many $\vec{\epsilon}_{I_1}$'s within each bucket. If we choose $l \approx \log_2\left(\binom{k/2}{p/2}\right)$ (the most common choice), this number is roughly 1, so that lookups within $\mathcal{M}_1$ can be expected to only take constant time, if a hash map is used. Nevertheless it is possible to store multiple $\vec{\epsilon}_{I_1}$'s in the same bucket by allowing the hash map to store (hopefully short) lists in each of its buckets. Respecting remark 3.4.1 (i.e. not using another map or list) we can then iterate over all possible $\vec{\epsilon}_{I_2}$ in line 2 of algorithm 3.4 and check for each of them, whether $\mathcal{M}_1[(Q_2\vec{\epsilon}_{I_2})_{[l]}]$ contains one or multiple $\vec{\epsilon}_{I_1}$'s. If it does, we found a collision between these $\vec{\epsilon}_{I_1}$'s and the current $\vec{\epsilon}_{I_2}$ without requiring any additional overhead.*
More concrete implementation details can be found in [14, Section 6].

Equation (3.21) follows from standard observations regarding the weight distribution of $\vec{\epsilon}$ required by Stern's algorithm to succeed (cf. figure 3.3).
For $p < l$ and $0 < W < 1 - R < 1$ we can once again obtain an explicit bound for the success probability of Stern's algorithm from [15] (also in [23], pp. 106-108). Combining equation (3.1) with this bound and substituting[5] $k = Rn, w = Wn$ we get the following equations for the overall runtime of Stern's algorithm $time\,\{isd()\} = time\,\{\text{Stern}\}$:

$$time\,\{\text{Stern}\} = 2^{\alpha(R,W)n} \cdot \left(\frac{1-R-W}{RW}\right)^p \left(\frac{1-R}{1-R-W}\right)^l \beta(R,W) \cdot err_{ST}^{-1}$$
$$\cdot \frac{((p/2)!)^2}{(n/2)^p} \left(\mathcal{O}(n^3) + \binom{Rn/2}{p/2} pl + \frac{\binom{Rn/2}{p/2}^2}{2^l} \cdot pn(1 - R - \frac{l}{n})\right)$$

$$time\,\{\text{Stern (p = 0)}\} = 2^{\alpha(R,W)n} \left(\frac{1-R}{1-R-W}\right)^l \beta(R,W) \cdot err_{ST}^{-1} \cdot \mathcal{O}(n^3)$$

$$time\,\{\text{Stern (p = 2)}\} = 2^{\alpha(R,W)n} \left(\frac{1-R-W}{RW}\right)^2 \left(\frac{1-R}{1-R-W}\right)^l \beta(R,W) \cdot err_{ST}^{-1}$$

[5] The substitution makes sense as k and w grow with n, whereas R and W can be assumed to be "relatively" constant (cf. remark 2.2.8).

$$\cdot \frac{1}{(n/2)^2}\left(\mathcal{O}(n^3) + Rnl + \frac{(Rn)^2}{2^l}\cdot\frac{1}{2}n(1-R-\frac{l}{n})\right)$$

$$time\,\{\text{Stern (p = 4)}\} = 2^{\alpha(R,W)n}\left(\frac{1-R-W}{RW}\right)^4\left(\frac{1-R}{1-R-W}\right)^l\beta(R,W)\cdot err_{ST}^{-1}$$
$$\cdot \frac{4}{(n/2)^4}\left(\mathcal{O}(n^3) + Rn(\frac{Rn}{2}-1)l + \frac{((\frac{Rn}{2})(\frac{Rn}{2}-1))^2}{2^l}\cdot n(1-R-\frac{l}{n})\right)$$

$$time\,\{\text{Stern (p = 6)}\} = 2^{\alpha(R,W)n}\left(\frac{1-R-W}{RW}\right)^6\left(\frac{1-R}{1-R-W}\right)^l\beta(R,W)\cdot err_{ST}^{-1}$$
$$\cdot\frac{36}{(n/2)^6}\left(\mathcal{O}(n^3)+\frac{Rn}{2}(\frac{Rn}{2}-1)(\frac{Rn}{2}-2)l + \frac{((\frac{Rn}{2})(\frac{Rn}{2}-1)(\frac{Rn}{2}-2))^2}{2^l}\cdot\frac{1}{6}n(1-R-\frac{l}{n})\right)$$

Note that the functions $\alpha(R,W)$ and $\beta(R,W)$ are defined as in definition 2.2.8 and equation (3.14); $err_{ST} := err_{ST}(n,k,w,l,p)$ is an error function with $\lim_{n\to\infty} err_{ST} = 1$ for $p \leq l$, which is more precisely defined in [23, section 5.4.3].
Several observations for the asymptotic runtime of Stern's algorithm follow from these equations:

1. For $p = l = 0$ we obtain Prange's algorithm and its runtime (cf. section 3.2).

2. It seems as if the time spent in the Gaussian elimination step of algorithm 3.1 ($\mathcal{O}(n^3)$) becomes negligible for $p \geq 4$. Thus the optimizations from section 3.1 are asymptotically irrelevant for these p.

3. The factors $\left(\frac{1-R-W}{RW}\right)^p$ and $\left(\frac{1-R}{1-R-W}\right)^l$ are both > 1 and become significantly more important for large p and l. They make it very hard to find asymptotically optimal parameter choices for arbitrary R and W.

4. In order to minimize $\binom{Rn/2}{p/2}pl + \frac{\binom{Rn/2}{p/2}^2}{2^l}\cdot pn(1-R-\frac{l}{n})$, we'll probably want to balance the term $\binom{Rn/2}{p/2}pl$ with $\frac{\binom{Rn/2}{p/2}^2}{2^l}\cdot pn(1-R-\frac{l}{n})$, i.e. choose the parameter l in the range $\log_2(\binom{Rn/2}{p/2}) \leq l \leq \log_2(\binom{Rn/2}{p/2}) + \log_2(n(1-R-\frac{l}{n})) \Rightarrow l = \tilde{\mathcal{O}}\left(\frac{Rn}{2}\cdot H_2(\frac{p}{Rn})\right)$ (cf. remark 2.2.12).
   Note that this choice of the parameter l also allows for a practical application of remark 3.4.2.

In practice it seems to be the best idea to use equation (3.19) in combination with equation (3.1) instead of any of the previous equations to find the optimal parameters p and l for a concrete parameter set (n,k,w).
It is common to set the parameter p to a relatively small integer: For example Chabaud uses $p \in \{4,6\}$ in [13] for practical applications (cf. table 1 in [13] and note that $p = 2p'$, where p' is used to denote Chabaud's parameter p). The table provided by Peters in [24] also indicates choices of $p \in \{4,6\}$ to be optimal (for $q = 2$). Once again note that these are the choices where the Gaussian elimination step from algorithm 3.1 becomes negligible. The rough bounds $2^{0.05564n}$ for the runtime of Stern's algorithm and $2^{0.0135n}$ for the memory consumption of Stern's algorithm for optimal parameters p and l can be obtained from [21] and prove an asymptotically exponential improvement over Lee-Brickell's algorithm.
Generalizations of Stern's algorithm over $\mathbb{F}_q, q \geq 2$ exist (see for example [23]), but are not discussed here.

Optimizations

Optimization 3.4.1 (Multiple choices of Z)**.** *This optimization is another effort at reducing the effect of the Gaussian elimination step (cf. algorithm 3.1) on Stern's algorithm. If you once again look at figure 3.4, it becomes obvious that the position of the $\vec{0}$-vector within $\vec{\epsilon}_{I^*} \in \mathbb{F}_2^{n-k}$ was arbitrarily chosen. Actually we could also put it at the end of $\vec{\epsilon}_{I^*}$, somewhere in the middle or even define an index set $Z \subset I^*$ with $|Z| = l$ and check for collisions there (on $\vec{\epsilon}_Z$). Note that this set does not even need to contain coherent indices. There are $\binom{n-k}{l}$ possibilities to choose such a set Z.*
The basic idea of this optimization is to introduce a new parameter m, choose disjoint sets $Z_1, Z_2, \ldots, Z_m \subset I^*$ and execute searchStern() (algorithm 3.4) for each of these m sets (i.e. look for collisions on $\vec{\epsilon}_{Z_i}$, $1 \leq i \leq m$) before returning to the main loop of algorithm 3.1. At first sight we save the cost of $m-1$ Gaussian elimination processes. At second sight we observe that the iterations over Z_i, $1 \leq i \leq m$ are not independent anymore as they share a common information set I. Nevertheless Bernstein, Lange and Peters claim in [14] that this optimization increases "the chance of finding any particular weight-w word [...] by a factor of nearly m" and that it is worthwhile if the Gaussian elimination "is more than about 5% of the original computation". Whether or not a small value of m might be beneficial for a concrete parameter set can be found out with the program from [24]. For sufficiently large n however this optimization is irrelevant, simply because the complete Gaussian elimination step is asymptotically negligible in Stern's algorithm ($p \geq 4$).

Optimization 3.4.2 (Reusing additions of the l-bit vectors)**.** *We can reuse the idea from optimization 3.3.1 for line 1 and line 2 of Stern's algorithm; however the vectors have only l bits this time. The analysis is essentially the same: If we define time $\{searchStern()[o_{3.4.2}]\} := Z + \binom{k/2}{p/2}^2/2^l \cdot (p(n-k-l) + \mathcal{O}(1))$ (cf. equation (3.19)) we get*

$$\begin{aligned} Z &:= 2l\left(\binom{k/2}{2} + \binom{k/2}{3} + \ldots + \binom{k/2}{p/2}\right) \\ &= 2l\binom{k/2}{p/2}\left(1 + \underbrace{\frac{p/2}{k/2-p/2+1} + \ldots + \frac{(k/2-p/2)!(p/2)!}{(k/2-2)!2!}}_{\delta(k/2,p/2)}\right) \end{aligned}$$

with $\delta(k/2, p/2) \ll 1$ as long as $k \gg p$. Thus we save a factor of $\frac{p}{2(1+\delta(k/2,p/2))}$ for the Z-part of time $\{searchStern()\}$, if we apply this optimization. The memory consumption increases by a constant factor.

Note that it does not seem helpful to use this optimization for line 5 of algorithm 3.4, simply because we may assume that the number of collisions $\binom{k/2}{p/2}^2 \cdot 2^{-l}$ is too small ($\approx \binom{k/2}{p/2} < \binom{k}{p} \cdot p^{-1}$ for $l \approx \log_2(\binom{k/2}{p/2})$) to make it worthwhile.

Optimization 3.4.3 (Early abort)**.** *We can also apply the idea from optimization 3.3.2 to Stern's algorithm, so that we may expect to only need at about $2(w-p+1)$ instead of $(n-k-l)p$ binary operations for line 5 of algorithm 3.4. This results in*

$$time\,\{searchStern()[o_{3.4.3}]\} := \binom{k/2}{p/2}pl + \frac{\binom{k/2}{p/2}^2}{2^l} \cdot (2p(w-p+1) + \mathcal{O}(1))$$

whilst the memory consumption remains the same.

In contrast to optimization 3.4.1 and the optimizations from section 3.1 the previous two optimization techniques improved the asymptotically relevant parts of Stern's algorithm, which underlines their importance. They were all introduced in [14].
The following idea comes from [26, 23, 25]:

Optimization 3.4.4 (Birthday Speedup)**.** *In 2009 Finiasz and Sendrier came up with the idea to modify algorithm 3.4, which resulted in an algorithm that looks similar to algorithm 3.5 (cf. table 2 in [26]). It is also described in [23, 25]. The differences are subtle, but grave:*

- *The error vectors $\vec{\epsilon}_{I_1}$ and $\vec{\epsilon}_{I_2}$ are chosen uniformly at random from the same vector space $\mathbb{F}_2^k$, i.e. figure 3.3 is incorrect for algorithm 3.5 as $\vec{\epsilon}_{I_1}$ and $\vec{\epsilon}_{I_2}$ may contain 1's in every possible position. $\vec{\epsilon}_{I_1}$ and $\vec{\epsilon}_{I_2}$ might even be identical.*
- *Hence the corresponding information sets I_1 and I_2 are not necessarily disjoint as in Stern's original algorithm. Therefore we also need to multiply $\vec{\epsilon}_{I_1}$ and $\vec{\epsilon}_{I_2}$ by Q instead of Q_1 and Q_2 in line 1 and 2 of algorithm 3.5.*
- *$\vec{\epsilon}_I$ is simply the sum of $\vec{\epsilon}_{I_1}$ and $\vec{\epsilon}_{I_2}$ (line 4).*
- *The new parameter N is meant to limit the number of choices for $\vec{\epsilon}_{I_1}$ and $\vec{\epsilon}_{I_2}$. It does not make sense to choose $N > \binom{k}{p/2}$, because otherwise we could also deterministically test all possible combinations of $\frac{p}{2}$ out of k elements.*

Remark 3.4.3. *Actually Table 2 in [26] is not completely identical to algorithm 3.5:*

- *Table 2 in [26] also contains the idea to represent the parity check matrix $\widehat{H}$ differently – we do not use that idea here, but rather introduce it in section 3.6.*
- *The original algorithmic description neither introduces the parameter N, nor does it sample the $\vec{\epsilon}_{I_1}$'s and $\vec{\epsilon}_{I_2}$'s at random. However they require both for their analysis (cf. assumption I1/B1 and the appendices in [26]).*
  However most information set decoding algorithms simply do not sample the $\vec{\epsilon}_{I_1}$'s and $\vec{\epsilon}_{I_2}$'s at random and therefore do not fit into a structure similar to that of algorithm 3.5. Bernstein et al. additionally point out in [19] that their Ball-Collision Decoding algorithm features an inner loop that is faster by a polynomial factor than the one assumed for the algorithm in [26]. Therefore the bounds presented in [26] do not hold for all information set decoding algorithms.

The basic idea is to use the birthday paradox for our purpose and hope for a sufficiently large amount of collisions in line 3 of algorithm 3.5. Unfortunately the algorithm requires an entire analysis on its own. For the sake of brevity we just present the results and shortly comment them:

$$time\left\{searchStern()[o_{3.4.4}]\right\} = Npl + \frac{N^2}{2^l} \cdot \left(p(n-k-l) + \mathcal{O}(1)\right) \tag{3.22}$$

$$mem\left\{searchStern()[o_{3.4.4}]\right\} = \mathcal{O}\left(N \cdot (l+k)\right) \tag{3.23}$$

$$\overline{PR}_{Stern}[success = true][o_{3.4.4}] = \frac{\binom{k}{p}\binom{n-k-l}{w-p}}{\binom{n}{w}} \left[1 - \left(1 - \binom{p}{p/2}\binom{k}{p/2}^{-2}\right)^{N^2}\right] \tag{3.24}$$

Equation (3.22) and equation (3.23) follow from the same observations as for Stern's original algorithm without optimizations (cf. algorithm 3.4). The success probability looks more complicated this time though, because the information sets I_1 and I_2 are not required to be

Algorithm 3.5: searchStern() with the Birthday Speedup

Input: parity check matrix $\widehat{H} = (Q \mid \mathrm{id}^{[n-k]}) \in \mathbb{F}_2^{(n-k)\times n}$, syndrome $\vec{\varsigma} \in \mathbb{F}_2^{n-k}$, $w = \mathrm{wt}(\vec{e})$, algorithmic parameter $0 \leq p \leq w$, algorithmic parameter $0 \leq l \leq n-k-w+p$, algorithmic parameter $0 \leq N < \binom{k}{p/2}$

Output: success indicator (true/false), error vector $\vec{\epsilon} \in \mathbb{F}_2^n$

1 **repeat** N **times:** choose $\vec{\epsilon}_{I_1} \in_r \mathbb{F}_2^k, \mathrm{wt}(\vec{\epsilon}_{I_1}) = \frac{p}{2} \Rightarrow \mathcal{L}_1[\vec{\epsilon}_{I_1}] \leftarrow \vec{\varsigma}_{[l]} + (Q\vec{\epsilon}_{I_1})_{[l]}$
2 **repeat** N **times:** choose $\vec{\epsilon}_{I_2} \in_r \mathbb{F}_2^k, \mathrm{wt}(\vec{\epsilon}_{I_2}) = \frac{p}{2} \Rightarrow \mathcal{L}_2[\vec{\epsilon}_{I_2}] \leftarrow (Q\vec{\epsilon}_{I_2})_{[l]}$
3 **foreach** $\vec{\epsilon}_{I_1}, \vec{\epsilon}_{I_2}, \mathcal{L}_1[\vec{\epsilon}_{I_1}] = \mathcal{L}_2[\vec{\epsilon}_{I_2}]$ **do**
4 $\vec{\epsilon}_I := \vec{\epsilon}_{I_1} + \vec{\epsilon}_{I_2}$
5 $\vec{\epsilon}_{I^*} := \vec{\varsigma} + Q\vec{\epsilon}_I$ // from here on as in algorithm 3.4
6 **if** $wt(\vec{\epsilon}_{I^*}) = w - p$ **then**
7 $\vec{\epsilon} \leftarrow$ `prepend`$(\vec{\epsilon}_I, \vec{\epsilon}_{I^*})$
8 **return** $(true, \vec{\epsilon})$
9 **end**
10 **end**
11 **return** $(false, \vec{0})$

disjoint: If we hope for p-many 1's in the first k positions of $\vec{\epsilon}$ and for zeroes in the l following positions, the probability for the real error vector to have such a weight distribution is $\binom{k}{p}\binom{n-k-l}{w-p}\binom{n}{w}^{-1}$. Furthermore we need to estimate the probability that one of the N^2 combinations of $\vec{\epsilon}_{I_1}$ and $\vec{\epsilon}_{I_2}$ results in the real error vector, if it has the aforementioned weight distribution. This is not guaranteed as $\vec{\epsilon}_{I_1}$ and $\vec{\epsilon}_{I_2}$ are chosen uniformly at random. First note that there are $\binom{p}{p/2}$ ways to split the real $\vec{\epsilon}_I$ into $\vec{\epsilon}_{I_1}$ and $\vec{\epsilon}_{I_2}$, so that $\vec{\epsilon}_I = \vec{\epsilon}_{I_1} + \vec{\epsilon}_{I_2}$ holds.

Remark 3.4.4. *Thereby it is important to see that the sets I_1 and I_2 are required to be disjoint for $\vec{\epsilon}_I = \vec{\epsilon}_{I_1} + \vec{\epsilon}_{I_2}$ to hold (simply because we also demand $wt(\vec{\epsilon}_I) = p$ and $wt(\vec{\epsilon}_{I_1}) = wt(\vec{\epsilon}_{I_2}) = \frac{p}{2}$), even though we allow them not to be. This is somewhat contradictory and is one of the facts that the algorithm presented in section 3.7 improves upon.*

In contrast to the $\binom{p}{p/2}$ ways to split the real $\vec{\epsilon}_I$ into $\vec{\epsilon}_{I_1}$ and $\vec{\epsilon}_{I_2}$ we randomly pick $\frac{p}{2}$ 1's in k positions for $\vec{\epsilon}_{I_1}$ and $\vec{\epsilon}_{I_2}$ in algorithm 3.5, so that the probability for us to "hit" a useful combination of $\vec{\epsilon}_{I_1}$ and $\vec{\epsilon}_{I_2}$ is $\binom{p}{p/2}\binom{k}{p/2}^{-2}$ for each of the N^2 combinations, which explains equation (3.24).
Finiasz and Sendrier state in [26] for the overall runtime of Stern's algorithm with the birthday speedup that

$$time\{Stern[o_{3.4.4}]\} \approx \min_p \left\{ \frac{2l \min\left\{\binom{n}{w}, 2^{n-k}\right\}}{(1-e^{-1})\binom{n-k-l}{w-p}\sqrt{\binom{k+l}{p}}} \right\}$$

$$Adv(Stern[o_{3.4.4}] > Stern) = \Theta(p^{-1/4})$$

They admit that the speedup of $\Theta(p^{-1/4})$ "is rather small in practice". Nevertheless the idea itself is interesting to see and worth remembering.
Regarding the choice of parameters for Stern's algorithm with the birthday speedup the following statements hold:

- *The parameter p is chosen as in Stern's original algorithm (i.e. $p \in \{4, 6\}$ in practice).*

- *Once again similar to Stern's original algorithm, the parameter l is usually chosen in a way to balance out the two summands of equation (3.22), i.e.* $\log_2(N) \leq l \leq \log_2(N) + \log_2(n-k-l)$.
- *The new parameter* $0 \leq N < \binom{k}{p/2}$ *hurts in equation (3.22) and helps in equation (3.24). An example for a sensible choice would be* $N \approx \binom{p}{p/2}^{-1/2}\binom{k}{p/2}$, *so that for* $a(p,k) := N^2 = \binom{p}{p/2}^{-1}\binom{k}{p/2}^2 > 1$ *we have* $1 - \left(1 - \binom{p}{p/2}\binom{k}{p/2}^{-2}\right)^{N^2} = 1 - \left(1 - a(p,k)^{-1}\right)^{a(p,k)}$ *with* $\lim_{n\to\infty} 1 - \left(1 - a(p,k)^{-1}\right)^{a(p,k)} = 1 - \frac{1}{e}$, *which means a chance of* $1 - \frac{1}{e} \approx 63\%$ *to succeed, if the real error vector* $\vec{\epsilon}$ *has the desired weight distribution whilst not performing too bad in comparison to Stern's original algorithm in equation (3.22).*

Apart from optimization 3.4.2 the optimizations presented in this section can be applied to algorithm 3.5 as well. Optimization 3.4.2 would only make sense for $N \approx \binom{k}{p/2}$, *which annihilates the benefits of the birthday speedup.*

3.5 Ball-Collision Decoding

In 2010 Bernstein, Lange and Peters presented a generalized version of Stern's algorithm in [19], which they called "Ball-Collision Decoding". They noticed that it is relatively unlikely for the error vector $\vec{\epsilon}$ to satisfy $(\vec{\epsilon}_{I^*})_{[l]} = \vec{0} \in \mathbb{F}_2^l$ as it is demanded by Stern's algorithm (cf. figure 3.4). Therefore they changed Stern's algorithm in a way that also allows for vectors $(\vec{\epsilon}_{I^*})_{[l]}$ of a certain weight and iterates over all possibilities for $(\vec{\epsilon}_{I^*})_{[l]}$ in a meet-in-the-middle approach. More precisely they decided to introduce a new even parameter $z \geq 0$ (in [19] they actually use a parameter q with $z = 2q$) and allow for $z/2$ 1's in the upper half of $(\vec{\epsilon}_{I^*})_{[l]}$ and the same number of 1's in the lower half of $(\vec{\epsilon}_{I^*})_{[l]}$, which leads to an overall weight distribution of $\vec{\epsilon}$ as depicted in figure 3.5.

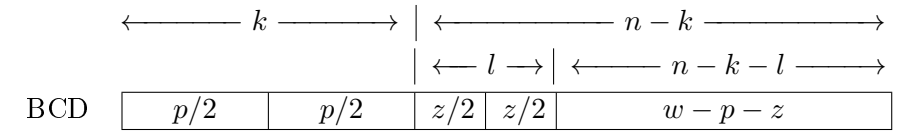


Figure 3.5: weight distribution of $\vec{\epsilon}$ as demanded by the Ball-Collision decoding algorithm

To understand algorithm 3.6 it is sufficient to reconsider equation (3.18) for arbitrary $(\vec{\epsilon}_{I^*})_{[l]}$:

$$\begin{aligned} \vec{\varsigma}_{[l]} + (Q_1\vec{\epsilon}_{I_1})_{[l]} + (Q_2\vec{\epsilon}_{I_2})_{[l]} &= (\vec{\epsilon}_{I^*})_{[l]} \\ \Leftrightarrow \vec{\varsigma}_{[l]} + (Q_1\vec{\epsilon}_{I_1})_{[l]} &= (Q_2\vec{\epsilon}_{I_2})_{[l]} + (\vec{\epsilon}_{I^*})_{[l]} \end{aligned} \tag{3.25}$$

So for example equation (3.25) would provide us with one possibility to look for collisions. Bernstein, Lange and Peters do not use that specific possibility in [19] though: Instead they split $(\vec{\epsilon}_{I^*})_{[l]}$ into two parts of the same size and rewrite $(\vec{\epsilon}_{I^*})_{[l]}$ as the sum of the upper part, padded with zeroes at the bottom, with the lower part, padded with zeroes at the top. Let us denote those two vectors by $\vec{\epsilon}_1, \vec{\epsilon}_2 \in \mathbb{F}_2^l$ with $(\vec{\epsilon}_{I^*})_{[l]} = \vec{\epsilon}_1 + \vec{\epsilon}_2$ and $\mathrm{wt}(\vec{\epsilon}_1) = \mathrm{wt}(\vec{\epsilon}_2) = z/2$, $\mathrm{wt}\left((\vec{\epsilon}_{I^*})_{[l]}\right) = z$. Then equation (3.26) can be used to search for collisions.

$$\vec{\varsigma}_{[l]} + (Q_1\vec{\epsilon}_{I_1})_{[l]} + \vec{\epsilon}_1 = (Q_2\vec{\epsilon}_{I_2})_{[l]} + \vec{\epsilon}_2 \tag{3.26}$$

The advantage of doing it that way lies in the fact that there are equally many possibilities for both sides of equation (3.26) (resulting in lists $\mathcal{L}_1$ and $\mathcal{L}_2$ of equal size), whereas for equation (3.25) the number of combinations on the right-hand side exceeds the number of possibilities on the left by far. As indicated by remark 3.5.1 this would result in a less optimal runtime.

The name "Ball-Collision Decoding" comes from the fact that we basically expand each $\vec{\varsigma}_{[l]} + (Q_1\vec{\epsilon}_{I_1})_{[l]}$ and each $(Q_2\vec{\epsilon}_{I_2})_{[l]}$ from Stern's algorithm by balls of hamming radii $z/2$ in algorithm 3.6. Note that we obtain Stern's algorithm (cf. section 3.4) for the choice $z = 0$.

To understand line 15 of algorithm 3.6, recall that equation (2.3) holds for arbitrary $(\vec{\epsilon}_{I^*})_{[l]}$. To be exact, we do not really need to compute the first l bits of $\vec{\epsilon}_{I^*}$ in line 15 of algorithm 3.6, but this is once again ignored in the algorithmic description for the sake of clarity. We do not ignore it in the following analysis though:

$$\begin{aligned} time\{searchBCD()\} &= pl\binom{k/2}{p/2} + \min\{1,z\} \cdot l\binom{k/2}{p/2}\binom{l/2}{z/2} \\ &\quad + \frac{\binom{k/2}{p/2}^2\binom{l/2}{z/2}^2}{2^l} \cdot p(n-k-l) \end{aligned} \tag{3.27}$$

$$mem\{searchBCD()\} = \mathcal{O}\left(l\binom{k/2}{p/2}\binom{l/2}{z/2} + \min\{1,z\}\frac{l}{2}\binom{l/2}{z/2} + \frac{k}{2}\binom{k/2}{p/2}\right) \tag{3.28}$$

Algorithm 3.6: searchBCD()

Input: parity check matrix $\widehat{H} = (Q \mid \mathrm{id}^{[n-k]}) \in \mathbb{F}_2^{(n-k)\times n}$, syndrome $\vec{\varsigma} \in \mathbb{F}_2^{n-k}$, $w = \mathrm{wt}(\vec{e})$, algorithmic parameter $0 \leq p \leq w$, algorithmic parameter $0 \leq l \leq n-k-w+p+z$, algorithmic parameter $0 \leq z \leq l$ with $0 \leq p+z \leq w$

Output: success indicator (true/false), error vector $\vec{\epsilon} \in \mathbb{F}_2^n$

1 **foreach** $\vec{\epsilon}_{I_1} \in \mathbb{F}_2^{\lceil k/2 \rceil}, wt(\vec{\epsilon}_{I_1}) = \frac{p}{2}$ **do**
2 **foreach** $\vec{u}_1 \in \mathbb{F}_2^{\lceil l/2 \rceil}, wt(\vec{u}_1) = \frac{z}{2}$ **do**
3 $\vec{\epsilon}_1 \leftarrow \mathtt{prepend}\,(\vec{u}_1, \vec{0})$ // $\vec{0} \in \mathbb{F}_2^{\lfloor l/2 \rfloor}$
4 $\mathcal{L}_1[\vec{\epsilon}_{I_1}, \vec{\epsilon}_1] \leftarrow \vec{\varsigma}_{[l]} + (Q_1\vec{\epsilon}_{I_1})_{[l]} + \vec{\epsilon}_1$
5 **end**
6 **end**
7 **foreach** $\vec{\epsilon}_{I_2} \in \mathbb{F}_2^{\lfloor k/2 \rfloor}, wt(\vec{\epsilon}_{I_2}) = \frac{p}{2}$ **do**
8 **foreach** $\vec{u}_2 \in \mathbb{F}_2^{\lfloor l/2 \rfloor}, wt(\vec{u}_2) = \frac{z}{2}$ **do**
9 $\vec{\epsilon}_2 \leftarrow \mathtt{prepend}\,(\vec{0}, \vec{u}_2)$ // $\vec{0} \in \mathbb{F}_2^{\lceil l/2 \rceil}$
10 $\mathcal{L}_2[\vec{\epsilon}_{I_2}, \vec{\epsilon}_2] \leftarrow (Q_2\vec{\epsilon}_{I_2})_{[l]} + \vec{\epsilon}_2$
11 **end**
12 **end**
13 **foreach** $\vec{\epsilon}_{I_1}, \vec{\epsilon}_{I_2}, \vec{\epsilon}_1, \vec{\epsilon}_2, \mathcal{L}_1[\vec{\epsilon}_{I_1}, \vec{\epsilon}_1] = \mathcal{L}_2[\vec{\epsilon}_{I_2}, \vec{\epsilon}_2]$ **do**
14 $\vec{\epsilon}_I \leftarrow \mathtt{prepend}\,(\vec{\epsilon}_{I_1}, \vec{\epsilon}_{I_2})$
15 $\vec{\epsilon}_{I^*} := \vec{\varsigma} + Q\vec{\epsilon}_I$
16 **if** $wt(\vec{\epsilon}_{I^*}) = w - p$ **then**
17 $\vec{\epsilon} \leftarrow \mathtt{prepend}\,(\vec{\epsilon}_I, \vec{\epsilon}_{I^*})$
18 **return** $(true, \vec{\epsilon})$
19 **end**
20 **end**
21 **return** $(false, \vec{0})$

$$\overline{PR}_{BCD}[success = true] = \frac{\binom{k/2}{p/2}^2 \binom{l/2}{z/2}^2 \binom{n-k-l}{w-p-z}}{\binom{n}{w}} \tag{3.29}$$

Once again assuming that all of the parameters are even integers, equation (3.29) directly follows from figure 3.5. The memory consumption only pays regard to the list $\mathcal{L}_1$ as the second list can be computed "on the fly" according to remark 3.4.1: We require at least l bits for each sum on the right side of line 4, $l/2$ bits to save each $\vec{\epsilon}_1$ and $k/2$ bits for each $\vec{\epsilon}_{I_1}$. However it should be sufficient to store each $\vec{\epsilon}_{I_1}$ and $\vec{\epsilon}_1$ only once, whereas we need to store the l-bit sums for each of the $\binom{k/2}{p/2}\binom{l/2}{z/2}$ combinations resulting in equation (3.28). The factor $\min\{1, z\}$ is used to respect the case $z = 0$, i.e. Stern's algorithm. Understanding equation (3.27) is a little trickier: As usual the operations $(Q_1\vec{\epsilon}_{I_1})_{[l]}$ in line 4 and the corresponding operation in line 10 of algorithm 3.6 are very expensive operations during each iteration of the loops started in line 1 and 7. Let us for example take the loop from lines 1-6: Computing $(Q_1\vec{\epsilon}_{I_1})_{[l]}$ for every $\vec{\epsilon}_{I_1}$ means a selection of $p/2$ columns of Q_1 and thus an overall cost of $\frac{p}{2}l\binom{k/2}{p/2}$. The addition of $\vec{\epsilon}_1$ in the inner loop (lines 2-5 of algorithm 3.6) is done $\binom{l/2}{z/2}$ times for every of the $\binom{k/2}{p/2}$ many $\vec{\epsilon}_{I_1}$'s. So additionally we get $\binom{k/2}{p/2}\binom{l/2}{z/2}$ additions of $l/2$-bit vectors (the upper or lower part being all zero).

Remark 3.5.1. *If we had decided to use equation (3.25) over equation (3.26), all of the additions of $\vec{\epsilon}_1$ and $\vec{\epsilon}_2$ would have occurred for list $\mathcal{L}_2$ only. This would have resulted in an additional workload of* $\min\{1, z\} \cdot \frac{l}{2} \cdot \binom{k/2}{p/2}\binom{l/2}{z/2}$ *for $\vec{\epsilon}_1$, but as we have to iterate over all of these entries to add $\vec{\epsilon}_2$, it means a workload of* $\min\{1, z\} \cdot \frac{l}{2} \cdot \binom{k/2}{p/2}\binom{l/2}{z/2}^2$ *for the second vector. This is clearly suboptimal to the way chosen by Bernstein et al. in [19], which is presented here. Nevertheless it is unclear, whether this approach can be made equally efficient by picking unbalanced lists for the outer loops, i.e. by making the $\vec{\epsilon}_{I_1}$'s have a slightly larger weight than the $\vec{\epsilon}_{I_2}$'s.*
(Note that we still split $(\vec{\epsilon}_{I^*})_{[l]}$ into two parts to respect the weight distribution according to figure 3.5; not doing so would result in an even different success probability and complexity.)

The last term in equation (3.27) comes from the fact that we expect $\binom{k/2}{p/2}^2\binom{l/2}{z/2}^2 \cdot 2^{-l}$ collisions in line 13 and thus reach line 15 that many times. We omit the other less expensive operations.

Remark 3.5.2. *Note that we use the term $\binom{k/2}{p/2}^2\binom{l/2}{z/2}^2 \cdot 2^{-l}$ to model the expected number of collisions in line 13 of algorithm 3.6, although the lists $\mathcal{L}_1$ and $\mathcal{L}_2$ do not contain uniformly distributed entries. In fact, only a specific fraction of $\binom{l/2}{z/2}^{-1}$ of their entries can be assumed to be uniform. You can easily see this from line 4 and line 10, where everything apart from $\vec{\epsilon}_1$ and $\vec{\epsilon}_2$ is fixed within the inner loop. Certainly, $\vec{\epsilon}_1$ and $\vec{\epsilon}_2$ are not sampled uniformly at random. Basically $\mathcal{L}_1$ and $\mathcal{L}_2$ contain $\binom{k/2}{p/2}$ uniform entries plus a factor of $\binom{l/2}{z/2}$ variations of each of these entries. So the list entries correlate: For example whenever we find a collision with $\mathcal{L}_1[\vec{\epsilon}_{I_1}, \vec{\epsilon}_1] = \mathcal{L}_2[\vec{\epsilon}_{I_2}, \vec{\epsilon}_2]$ in line 13 of algorithm 3.6, we immediately know that for these specific values of $\vec{\epsilon}_{I_1}, \vec{\epsilon}_{I_2}$ and $\vec{\epsilon}_1$, no other $\vec{\epsilon}_2$ can make that equation hold, because $\vec{\epsilon}_1$ and $\vec{\epsilon}_2$ contain 1's in disjoint positions*[6]*. This fact does not affect the expected number of collisions; however we performed several experiments, which showed that the variance of the number of collisions is slightly affected (cf. appendix D).*
To see that the expected number of collisions remains the same, let us first reformulate the problem: We speak of collisions, whenever the equation $\vec{x} + \vec{\epsilon}_1 = \vec{y} + \vec{\epsilon}_2 \Leftrightarrow \vec{x} + \vec{y} = \vec{\epsilon}_1 + \vec{\epsilon}_2$ for $\vec{x} := \vec{\varsigma}_{[l]} + (Q_1\vec{\epsilon}_{I_1})_{[l]}$ and $\vec{y} := (Q_2\vec{\epsilon}_{I_2})_{[l]}$ holds. Thanks to the uniformity of Q the vectors $\vec{x}, \vec{y} \in \mathbb{F}_2^l$ can be assumed to be fixed uniform values, for which we iterate over many $\vec{\epsilon}_1, \vec{\epsilon}_2$'s, which have a weight distribution as in the l-block of figure 3.5. Basically we obtain a collision, if $\vec{x} + \vec{y} \in M := \{prepend(\vec{u}_1, \vec{u}_2), \vec{u}_1 \in \mathbb{F}_2^{\lceil l/2 \rceil}, \vec{u}_2 \in \mathbb{F}_2^{\lfloor l/2 \rfloor} \mid wt(\vec{u}_1) = wt(\vec{u}_2) = z/2\}$. Clearly, the probability for a collision to occur is $Pr[\vec{x} + \vec{y} \in M] = |M|/2^l$ with $|M| = \binom{l/2}{z/2}^2$ for an even l. We do this for $\binom{k/2}{p/2}$ vectors $\vec{x}$ and $\binom{k/2}{p/2}$ vectors $\vec{y}$.
It is also important to see that the success probability is not affected, if $\mathcal{L}_1$ and $\mathcal{L}_2$ are not entirely uniform lists.

No recommendations regarding the choice of parameters for the Ball-Collision Decoding algorithm are provided in [19]. However Bernstein et al. prove that the choice $z = 0$, i.e. Stern's algorithm, is asymptotically suboptimal. Instead they use $z = 2$ in their examples. The parameter p has the same role as in Stern's algorithm and is therefore probably chosen in a similar way. It once again makes sense to choose the parameter l in a way that balances the terms of equation (3.27), i.e. choose $\log_2(\binom{k/2}{p/2}\binom{l/2}{z/2}) \leq l \leq \log_2(\binom{k/2}{p/2}\binom{l/2}{z/2}) + \log_2(n - k - l)$.
The paper [21] contains a rough bound of $2^{0.05559n}$ for the algorithm.
Bernstein provides a reference implementation at `http://cr.yp.to/ballcoll.html`.

[6] This can be exploited in practice.

Remark 3.5.3. *Note that the original algorithm allows for different choices of k, l, p and z for each of the iterations in line 1-6 and line 7-12 of algorithm 3.6. Balanced choices are asymptotically optimal though.*

Optimizations

We can reuse all of the optimizations from Stern's algorithm (cf. section 3.4) for its generalisation:

Optimization 3.5.1 (Reusing additions of the l-bit vectors)**.** *We can apply this technique four times within algorithm 3.6: During the matrix operations in line 4 and 10 analogously to optimization 3.3.1 as well as for the addition of both $\vec{\epsilon}_1$ and $\vec{\epsilon}_2$. To understand the latter, imagine the $\vec{\epsilon}_1$'s as sums of columns of the matrix $I := (id^{[l/2]} \mid 0^{[(l/2)\times(l/2)]})^T$. The iterations of the loop in line 2 are meant to add all combinations of $z/2$ columns of that matrix to $(Q_1\vec{\epsilon}_{I_1})_{[l]}$. Thus we can first compute the sums of $(Q_1\vec{\epsilon}_{I_1})_{[l]}$ with every single column of I, then with every combination of 2 columns and so on always using the previously computed sums, so that we only require a single binary operation per sum (as each column of I only contains a single "1"). This involves exactly $\binom{l/2}{1} + \binom{l/2}{2} + \binom{l/2}{3} + \ldots + \binom{l/2}{z/2} = l/2 + \binom{l/2}{z/2} \cdot (1 + \delta(l/2, z/2))$ bit flips.*
All in all we get:

$$time\{searchBCD()[o_{3.5.1}]\} = 2l \cdot (1 + \delta(k/2, p/2))\binom{k/2}{p/2} + \min\{1, z\} \cdot 2\binom{k/2}{p/2} \left(\frac{l}{2} + \binom{l/2}{z/2} \cdot (1 + \delta(l/2, z/2))\right) + \frac{\binom{k/2}{p/2}^2 \binom{l/2}{z/2}^2}{2^l} \cdot p(n-k-l)$$

Note that the factor l is missing in the second summand in comparison to equation (3.27). In practice however lines 4 and 10 of algorithm 3.6 could be implemented as two times $z/2$ bit flips anyway (usually just one word operation), which makes the idea to apply this optimization on $\vec{\epsilon}_1$ and $\vec{\epsilon}_2$ a rather theoretical construct, especially for those small z seen in practice (actually we need to define $1 + \delta(l/2, 1) := 0$ for $z = 2$).
The function $\delta(k, p)$ was defined in optimization 3.3.1. The memory consumption increases by a constant factor.

Optimization 3.5.2 (Early abort)**.** *By applying optimization 3.4.3 to line 15 of algorithm 3.6 we obtain*

$$\frac{\binom{k/2}{p/2}^2 \binom{l/2}{z/2}^2}{2^l} \cdot 2p(w-p-z+1)$$

for the collision part of equation (3.27). Note that we can check the last $n-k-l$ bits of $\vec{\epsilon}_{I^*}$ for weight $w-p-z$ in line 16 of algorithm 3.6. The memory consumption with this technique remains the same.

Optimization 3.5.3 (Birthday speedup)**.** *We can even apply optimization 3.4.4 to lines 1 and 7 of algorithm 3.6. Recall that we then have $\vec{\epsilon}_{I_1}, \vec{\epsilon}_{I_2} \in_r \mathbb{F}_2^k$. By defining $N' := \binom{l/2}{z/2}N$ we get:*

$$time\{searchBCD()[o_{3.5.3}]\} = Npl + \min\{1, z\} \cdot N'l + \frac{N'^2}{2^l} \cdot p(n-k-l)$$

$$mem\{searchBCD()[o_{3.5.3}]\} = \mathcal{O}\left(lN' + \min\{1, z\} \cdot \frac{l}{2}\binom{l/2}{z/2} + kN\right)$$

$$\overline{PR}_{BCD}[success = true][o_{3.5.3}] = \frac{\binom{k}{p}\binom{l/2}{z/2}^2\binom{n-k-l}{w-p-z}}{\binom{n}{w}} \cdot \left[1 - \left(1 - \binom{p}{p/2}\binom{k}{p/2}^{-2}\right)^{N^2}\right]$$

Note that the last equation has N^2 rather than N'^2 in the exponent, because it is the number of combinations of $\vec{\epsilon}_{I_1}$ and $\vec{\epsilon}_{I_2}$ that is important for the success probability.
We can choose $\log_2(N\binom{l/2}{z/2}) \leq l \leq \log_2(N\binom{l/2}{z/2}) + \log_2(n-k-l)$ and $N' \approx \binom{p}{p/2}^{-1/2}\binom{k}{p/2}$ as in optimization 3.4.4.
Additionally applying the birthday speedup to lines 2 and 8 is possible, but probably complicated and not very rewarding. In particular a bad choice of $\vec{\epsilon}_{I_1} \in_r \mathbb{F}_2^k$ and $\vec{\epsilon}_{I_2} \in_r \mathbb{F}_2^k$ (e.g. one where both vectors have 1's in common positions) in lines 1 and 7 of algorithm 3.6 (with the birthday speedup applied) can never lead to the overall success of algorithm 3.6 for any choice of $\vec{\epsilon}_1$ and $\vec{\epsilon}_2$, i.e. the probabilities cannot be assumed to be independent.

Remark 3.5.4. *For large N it can be hard to compute the factor $\left[1 - \left(1 - \binom{p}{p/2}\binom{k}{p/2}^{-2}\right)^{N^2}\right]$ in practice due to the large N^2 in the exponent. We can use Bernoulli's inequality to obtain an upper bound and the inequality $1 + x \leq e^x$ for $x \in \mathbb{R}$ to obtain a lower bound though:*

$$1 - e^{-\binom{p}{p/2}\binom{k}{p/2}^{-2}N^2} \leq \left[1 - \left(1 - \binom{p}{p/2}\binom{k}{p/2}^{-2}\right)^{N^2}\right] \leq \frac{N^2 \cdot \binom{p}{p/2}}{\binom{k}{p/2}^2}$$

Note that this lower bound might be easier to compute, because $0 < \binom{p}{p/2}\binom{k}{p/2}^{-2} < 1$.

3.6 FS-ISD

Somewhere hidden between the lines of [26], a paper published by Finiasz and Sendrier in 2009, lies an important observation that enables us to get rid of the size-l zero-block of figure 3.4 completely: First note that we can easily rewrite figure 2.1 to respect the parameter l from Stern's algorithm (cf. section 3.4), which results in figure 3.6. Thereby we define the matrix $L \in \mathbb{F}_2^{(n-k)\times l}$ as those l columns of the parity check matrix $\widehat{H}$, which correspond to the entries of $(\vec{\epsilon}_{I^*})_{[l]}$. By the vector $\underline{\vec{\epsilon}_{I^*}} \in \mathbb{F}_2^{n-k-l}$ we mean the last $n-k-l$ entries of the error vector $\vec{\epsilon}_{I^*}$ (i.e. $\vec{\epsilon}_{I^*} = prepend((\vec{\epsilon}_{I^*})_{[l]}, \underline{\vec{\epsilon}_{I^*}})$ should hold). The columns of the matrix $S \in \mathbb{F}_2^{(n-k)\times(n-k-l)}$ correspond to these entries. The first l rows of S are all zero; afterwards follow $n-k-l$ rows that form the $(n-k-l)$-identity matrix id. A parity check matrix $\widehat{H}$ that looks as in figure 3.6 is said to be in *quasi-systematic form*.

$$\left(\begin{array}{c|c|c} & & O \\ \hline Q & L & \underbrace{id}_{S} \end{array}\right) \left(\begin{array}{c} \vec{\epsilon}_I \\ \hline (\vec{\epsilon}_{I^*})_{[l]} \\ \hline \underline{\vec{\epsilon}_{I^*}} \end{array}\right) = \vec{\varsigma}$$

Figure 3.6: Structure of $\widehat{H}\vec{\epsilon} = \vec{\varsigma}$ with $id := \mathrm{id}^{[n-k-l]}$, $O := 0^{[l\times(n-k-l)]}$, $Q \in \mathbb{F}_2^{(n-k)\times k}$, $L \in \mathbb{F}_2^{(n-k)\times l}$, $S \in \mathbb{F}_2^{(n-k)\times(n-k-l)}$, $\vec{\epsilon}_I \in \mathbb{F}_2^k$, $(\vec{\epsilon}_{I^*})_{[l]} \in \mathbb{F}_2^l$, $\underline{\vec{\epsilon}_{I^*}} \in \mathbb{F}_2^{n-k-l}$, $\widehat{H} \in \mathbb{F}_2^{(n-k)\times n}$, $\vec{\epsilon} \in \mathbb{F}_2^n$, $\vec{\varsigma} \in \mathbb{F}_2^{n-k}$

Let us define $Q' := (Q \mid L)$ and $\underline{\vec{\epsilon}_I} := prepend(\vec{\epsilon}_I, (\vec{\epsilon}_{I^*})_{[l]}) \in \mathbb{F}_2^{k+l}$. Then we have

$$Q'\underline{\vec{\epsilon}_I} + S\underline{\vec{\epsilon}_{I^*}} = \vec{\varsigma} \tag{3.30}$$

Now the basic idea is to guess $\underline{\vec{\epsilon}_I} \in \mathbb{F}_2^{k+l}$ instead of $\vec{\epsilon}_I$ in a meet-in-the-middle approach and then compute $\underline{\vec{\epsilon}_{I^*}}$ from it. Guessing $\underline{\vec{\epsilon}_I}$ implies a guess for $(\vec{\epsilon}_{I^*})_{[l]}$, so that we do not need to do this explicitly anymore (as e.g. with the Ball-Collision algorithm from section 3.5). Note that $S\underline{\vec{\epsilon}_{I^*}}$ is just the vector $\underline{\vec{\epsilon}_{I^*}}$ with l zeroes padded at the top (i.e. $S\underline{\vec{\epsilon}_{I^*}} = prepend(\vec{0}, \underline{\vec{\epsilon}_{I^*}})$, $\vec{0} \in \mathbb{F}_2^l$), so that

$$(Q'\underline{\vec{\epsilon}_I})_{[l]} + \vec{\varsigma}_{[l]} = \vec{0} \tag{3.31}$$

holds for *any* $(\vec{\epsilon}_{I^*})_{[l]}$, whereas for Stern's algorithm the equivalent equation (3.18) did only hold for $(\vec{\epsilon}_{I^*})_{[l]} = \vec{0}$ (simply because $(\vec{\epsilon}_{I^*})_{[l]}$ is part of $\underline{\vec{\epsilon}_I}$).
Similar to figure 3.3 we can then once again divide Q' into two halves Q'_1 and Q'_2 as well as $\underline{\vec{\epsilon}_I}$ into two halves $\underline{\vec{\epsilon}_{I_1}}$ and $\underline{\vec{\epsilon}_{I_2}}$ to search for collisions and hope for a distribution of the real error vector $\vec{\epsilon}$ as in figure 3.7. Algorithm 3.7 immediately follows. The combination of algorithm 3.1 with algorithm 3.7 is called "Finiasz-Sendrier Information Set Decoding" (or "FS-ISD" for short) by May et al. in [20, 21].

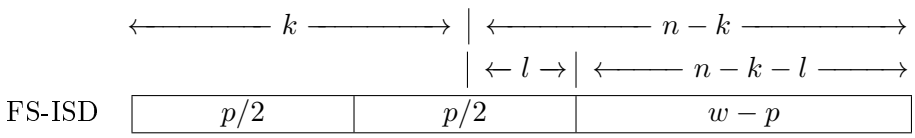


Figure 3.7: weight distribution of $\vec{\epsilon}$ as demanded by algorithm 3.7

Remark 3.6.1. *The original algorithm from table 2 of [26] also contains the idea of using the birthday speedup, which is not applied in algorithm 3.7, but introduced in optimization 3.6.3. So figure 3.7 would be incorrect for the original algorithm.*

Algorithm 3.7: searchFS()

Input: parity check matrix $\widehat{H} = (Q \mid L \mid S) \in \mathbb{F}_2^{(n-k)\times n}$, syndrome $\vec{\varsigma} \in \mathbb{F}_2^{n-k}$, $w = \mathrm{wt}(\vec{e})$, algorithmic parameter $0 \le p \le w$, algorithmic parameter $0 \le l \le n-k-w+p$

Output: success indicator (true/false), error vector $\vec{\epsilon} \in \mathbb{F}_2^n$

1 **foreach** $\underline{\vec{\epsilon}_{I_1}} \in \mathbb{F}_2^{\lceil (k+l)/2 \rceil}, wt(\underline{\vec{\epsilon}_{I_1}}) = \frac{p}{2}$ **do** $\mathcal{L}_1[\underline{\vec{\epsilon}_{I_1}}] \leftarrow \vec{\varsigma}_{[l]} + (Q'_1 \underline{\vec{\epsilon}_{I_1}})_{[l]}$

2 **foreach** $\underline{\vec{\epsilon}_{I_2}} \in \mathbb{F}_2^{\lfloor (k+l)/2 \rfloor}, wt(\underline{\vec{\epsilon}_{I_2}}) = \frac{p}{2}$ **do** $\mathcal{L}_2[\underline{\vec{\epsilon}_{I_2}}] \leftarrow (Q'_2 \underline{\vec{\epsilon}_{I_2}})_{[l]}$

3 **foreach** $\underline{\vec{\epsilon}_{I_1}}, \underline{\vec{\epsilon}_{I_2}}, \mathcal{L}_1[\underline{\vec{\epsilon}_{I_1}}] = \mathcal{L}_2[\underline{\vec{\epsilon}_{I_2}}]$ **do**

4 $\quad \underline{\vec{\epsilon}_I} \leftarrow \mathtt{prepend}\,(\underline{\vec{\epsilon}_{I_1}}, \underline{\vec{\epsilon}_{I_2}})$

5 $\quad S\underline{\vec{\epsilon}_{I^*}} := \vec{\varsigma} + Q' \underline{\vec{\epsilon}_I}$

6 $\quad$ **if** $wt(S\underline{\vec{\epsilon}_{I^*}}) = w - p$ **then**

7 $\quad\quad \underline{\vec{\epsilon}_{I^*}} \leftarrow \mathtt{remove}\,(S\underline{\vec{\epsilon}_{I^*}}, l)$

8 $\quad\quad \vec{\epsilon} \leftarrow \mathtt{prepend}\,(\underline{\vec{\epsilon}_I}, \underline{\vec{\epsilon}_{I^*}})$

9 $\quad\quad$ **return** $(true, \vec{\epsilon})$

10 $\quad$ **end**

11 **end**

12 **return** $(false, \vec{0})$

It is important to see that there are two ways to work with the new matrix $Q' = (Q \mid L) \in \mathbb{F}_2^{(n-k)\times(k+l)}$:

1. We can either do a partial Gaussian elimination to obtain the matrix S in figure 3.6 only. Thus we can save some binary operations during the execution of randomize() (cf. algorithm 3.1). Note however that still Q rather than Q' defines the notion of the information set I. This is the way chosen by Finiasz and Sendrier in [26].

2. Or we can do a complete Gaussian elimination to obtain the matrix $(L \mid S) = \mathrm{id}^{[n-k]}$ and exploit the fact that we know L to be in the form $(\mathrm{id}^{[l]} \mid 0^{[(n-k-l)\times l]})^T$ to save binary operations during the computation of lines 2 and 5 of algorithm 3.7. Obviously Q' cannot be assumed to be uniform in that case, but only Q. However we do not require the assumption of L being uniform; actually it is only used to estimate the number of collisions in line 3 of algorithm 3.7 and in that case remark 3.6.2 applies.

We decided to choose the latter way for our analysis of the algorithm for two reasons: First, we do not need to adapt our model of the randomize() function from section 3.1 to do a partial Gaussian elimination only. Second, the analysis in section 3.4 indicates that the Gaussian elimination process is probably asymptotically irrelevant anyway. Saving binary operations within algorithm 3.7 seems to be more significant. Moreover it can be left to the Gaussian optimizations (e.g. optimization 3.1.3) to reduce the complexity of the Gaussian elimination process.
So we do not compute line 5 of algorithm 3.7 naively, but rather use the fact that $L = (\mathrm{id}^{[l]} \mid 0^{[(n-k-l)\times l]})^T$ and get

$$\begin{aligned} Q'\underline{\vec{\epsilon}_I} &= (Q \mid L) \;\cdot\; \mathrm{prepend}\big(\vec{\epsilon}_I, (\vec{\epsilon}_{I^*})_{[l]}\big) = Q\vec{\epsilon}_I + L(\vec{\epsilon}_{I^*})_{[l]} \\ &= Q\vec{\epsilon}_I \;+\; \mathrm{prepend}\big((\vec{\epsilon}_{I^*})_{[l]}, \vec{0}\big), \quad \vec{0} \in \mathbb{F}_2^{n-k-l} \end{aligned} \tag{3.32}$$

Expecting that $(\vec{\epsilon}_{I^*})_{[l]}$ contains a fraction of $\frac{l}{k+l}$ of the overall number of 1's within $\underline{\vec{\epsilon}_I} \in \mathbb{F}_2^{k+l}$ we need to select and sum up at about $(1-\frac{l}{k+l})p$ columns of Q in line 5 of algorithm 3.7. As line 7 indicates we do not really need to compute the first l bits of $S\underline{\vec{\epsilon}_{I^*}}$ in line 5, so that we can rather compute $\underline{\vec{\epsilon}_{I^*}} = \text{remove}(\vec{\varsigma} + Q'\underline{\vec{\epsilon}_I}, l) = \text{remove}(\vec{\varsigma} + Q\vec{\epsilon}_I, l)$ in line 5. To do so we require roughly $\frac{kp}{k+l}(n-k-l)$ bit operations per computation.
Since Q_2' also contains the matrix L, we get for line 2 of algorithm 3.7:

$$\begin{aligned} Q_2'\underline{\vec{\epsilon}_{I_2}} &= (Q_2'' \mid L) \ \cdot \ \text{prepend}\big(\vec{\gamma}, (\vec{\epsilon}_{I^*})_{[l]}\big) = Q_2''\vec{\gamma} + L(\vec{\epsilon}_{I^*})_{[l]} \\ &= Q_2''\vec{\gamma} \ + \ \text{prepend}\big((\vec{\epsilon}_{I^*})_{[l]}, \vec{0}\big), \quad \vec{0} \in \mathbb{F}_2^{n-k-l} \qquad (3.33) \\ \Rightarrow (Q_2'\underline{\vec{\epsilon}_{I_2}})_{[l]} &= (Q_2''\vec{\gamma})_{[l]} + (\vec{\epsilon}_{I^*})_{[l]} \qquad (3.34) \end{aligned}$$

Thereby the matrix $Q_2'' \in \mathbb{F}_2^{(n-k)\times(k-l)/2}$ and the vector $\vec{\gamma} \in \mathbb{F}_2^{(k-l)/2}$ (assuming k and l are even integers) are implicitly defined by the equations above. Since we iterate over all $\underline{\vec{\epsilon}_{I_2}}$ in line 2 of algorithm 3.7, the vector $(\vec{\epsilon}_{I^*})_{[l]}$ is known to have an average weight of $\psi := \sum_{j=0}^{p/2} \binom{(k-l)/2}{p/2-j}\binom{l}{j} \cdot j \cdot \binom{(k+l)/2}{p/2}^{-1}$ there, so that the operation $(Q_2''\vec{\gamma})_{[l]}$ means a cost of at about $(\frac{p}{2}-\psi)l$ binary operations. In contrast the computation of $(Q_1'\underline{\vec{\epsilon}_{I_1}})_{[l]}$ in line 1 of algorithm 3.7 does not benefit from our knowledge of L, i.e. we obtain the standard number of $\frac{p}{2}l$ binary operations per computation. All in all we get the summand $\frac{p}{2}l\binom{(k+l)/2}{p/2}$ for line 1 of algorithm 3.7, the summand $(\frac{p}{2}-\psi)l\binom{(k+l)/2}{p/2}$ for line 2 and $(1-\frac{l}{k+l})p(n-k-l)\binom{(k+l)/2}{p/2}^2 \cdot 2^{-l}$ for line 5. These observations result in equation (3.35). If we assume $\psi \approx \frac{p}{2} \cdot \frac{2l}{k+l}$, the overall factor that we gain from using the equations 3.32 and 3.33 is roughly $\frac{k}{k+l}$.

Remark 3.6.2. *We still use the term $\binom{(k+l)/2}{p/2}^2 2^{-l}$ to model the expected number of collisions in line 3, even though Q_2' is not entirely uniform. We can understand the labels $(Q_2'\underline{\vec{\epsilon}_{I_2}})_{[l]} = (Q_2''\vec{\gamma})_{[l]} + (\vec{\epsilon}_{I^*})_{[l]}$ (cf. equation (3.34)) of $\mathcal{L}_2$ as uniform list entries $(Q_2''\vec{\gamma})_{[l]}$ expanded by vectors $(\vec{\epsilon}_{I^*})_{[l]}$ of a certain fixed weight $\tau = p/2 - wt(\vec{\gamma})$. Whenever we find a collision in line 3 of algorithm 3.7 for fixed $\underline{\vec{\epsilon}_{I_1}}, \gamma$ and $(\vec{\epsilon}_{I^*})_{[l]}$ with $wt((\vec{\epsilon}_{I^*})_{[l]}) = \tau$, we can immediately ignore all other possibilities for $(\vec{\epsilon}_{I^*})_{[l]}$ with weight τ, which implies a correlation of the list entries with regard to the $(\vec{\epsilon}_{I^*})_{[l]}$'s. However similar to remark 3.5.2 this neither affects the success probability of the algorithm nor does it affect the expected number of collisions in line 3, but only seems to have a small effect on the variance (cf. appendix D).*

$$time\{searchFS()\} = (p-\psi)l\binom{(k+l)/2}{p/2} + \frac{k}{k+l} \cdot \frac{\binom{(k+l)/2}{p/2}^2}{2^l} p(n-k-l) \qquad (3.35)$$

$$mem\{searchFS()\} = \mathcal{O}\left(\frac{1}{2}(3l+k)\binom{(k+l)/2}{p/2}\right) \qquad (3.36)$$

$$\overline{PR}_{FS}[success = true] = \frac{\binom{(k+l)/2}{p/2}^2\binom{n-k-l}{w-p}}{\binom{n}{w}} \qquad (3.37)$$

Regarding the memory consumption we need to store at least the list $\mathcal{L}_1$ (cf. remark 3.4.1), i.e. every $\underline{\vec{\epsilon}_{I_1}} \in \mathbb{F}_2^{\lceil (k+l)/2 \rceil}$ as well as the l-bit results from the matrix operation in line 1 of algorithm 3.7. As we iterate over all of the $\binom{(k+l)/2}{p/2}$ possibilities for $\underline{\vec{\epsilon}_{I_1}}$ and $\underline{\vec{\epsilon}_{I_2}}$, equation (3.36) immediately follows. Equation (3.37) can be obtained from looking at figure 3.6 as usual. Once again note that a uniform choice of the information set I (i.e. a uniform choice of Q) is sufficient to ensure the randomness of the whole error vector $\vec{\epsilon}$, which is required to define the success probability as in equation (3.37).
It is once again reasonable to choose $\log_2\left(\binom{(k+l)/2}{p/2}\right) \leq l \leq \log_2\left(\binom{(k+l)/2}{p/2}\right) + \log_2(n-k-l)$

(note that the lower and upper bound are also functions of l). The parameter p can be chosen as in Stern's algorithm or slightly larger in practice to accommodate the additional l-window in figure 3.6.
It is remarkable that May, Meurer and Thomae prove in [20] that FS-ISD is asymptotically at least as efficient as the Ball-Collision Decoding algorithm from section 3.5. Concretely they state that the cost of Ball-Collision Decoding with the parameters (p', l, z) is asymptotically always equivalent or larger than the cost of FS-ISD with the parameter set $(p = p' + z, l)$. Strangely enough, Finiasz and Sendrier did not see the potential of their algorithm in [26]. Nevertheless Bernstein, Lange and Peters claim in [19][7] that Ball-Collision Decoding is superior to FS-ISD by a polynomial factor. We will test this claim in section 4. It would additionally imply that both algorithms are asymptotically equivalent. This is also indicated by the (asymptotic) rough bound of $2^{0.05559n}$ found in [20], which is the same for both Ball-Collision Decoding and FS-ISD.

Optimizations

Since the algorithm is pretty much the same as Stern's algorithm with different dimensions, we can directly apply the optimizations from section 3.4:

Optimization 3.6.1 (Reusing additions of the l-bit vectors)**.** *We can use this optimization for both line 1 and line 2 of algorithm 3.7. Note that we employ this technique for the whole matrix $Q_2' \in \mathbb{F}_2^{(n-k)\times(k+l)/2}$ instead of for $Q_2'' \in \mathbb{F}_2^{(n-k)\times(k-l)/2}$ only, but still retain the advantage implied by our observation from equation (3.33), because every column selected from L implies a 1-bit addition instead of a l-bit addition – this is also true whilst precomputing the column sums. Hence we get*

$$time\left\{searchFS()[o_{3.6.1}]\right\} = \left(1+\delta\left(\frac{k+l}{2},\frac{p}{2}\right)\right)\left(1-\frac{\psi}{p}\right)2l\binom{(k+l)/2}{p/2}+\frac{k\binom{(k+l)/2}{p/2}^2}{(k+l)2^l}p(n-k-l)$$

The memory consumption increases by a constant factor.
The original technique and the function $\delta(k,p)$ are described in optimization 3.3.1.

Optimization 3.6.2 (Early abort)**.** *The early abort strategy from optimization 3.3.2 leads to a runtime of*

$$time\left\{searchFS()[o_{3.6.2}]\right\} = (p-\psi)l\binom{(k+l)/2}{p/2}+\frac{k}{k+l}\cdot\frac{\binom{(k+l)/2}{p/2}^2}{2^l}2p(w-p+1)$$

whilst the memory consumption remains the same.

Optimization 3.6.3 (Birthday speedup)**.** *We can also use the birthday speedup in lines 1 and 2 of algorithm 3.7. Recall that $\underline{\vec{\epsilon}_{I_1}}, \underline{\vec{\epsilon}_{I_2}}$ are chosen uniformly at random in $\mathbb{F}_2^{k+l}$ rather than $\mathbb{F}_2^{(k+l)/2}$ in that context. All in all we get*

$$\begin{aligned}
time\left\{searchFS()[o_{3.6.3}]\right\} &= (p-\psi)\cdot lN+\frac{k}{k+l}\cdot\frac{N^2}{2^l}p(n-k-l)\\
mem\left\{searchFS()[o_{3.6.3}]\right\} &= \mathcal{O}\left((2l+k)N\right)\\
\overline{PR}_{FS}[success=true][o_{3.6.3}] &= \frac{\binom{k+l}{p}\binom{n-k-l}{w-p}}{\binom{n}{w}}\left[1-\left(1-\binom{p}{p/2}\binom{k+l}{p/2}^{-2}\right)^{N^2}\right]
\end{aligned}$$

Similar to optimization 3.4.4 a sensible choice for N would be $N \approx \binom{p}{p/2}^{-1/2}\binom{k+l}{p/2}$. The parameter l is probably best chosen in the range $\log_2(N) \le l \le \log_2(N)+\log_2(n-k-l)$.

[7] The claim was added to the version from March 2011.

3.7 BJMM

In 2012 Becker, Joux, May and Meurer presented an information set decoding algorithm that uses several new ideas in [21]. They decided to use the Finiasz-Sendrier algorithm and its ideas (cf. figure 3.6) as a starting point for their improvements. Instead of trying to optimize the whole algorithm they concentrated on solving the underlying problem of only the first two lines of algorithm 3.7, i.e. that of finding l-bit collisions, more efficiently. To be precise, the problem can be defined as

Definition 3.7.1 (Submatrix Matching Problem). *Given a uniform matrix $Q'_l \in_r \mathbb{F}_2^{l\times(k+l)}$ and a target vector $\vec{\varsigma}_{[l]} \in \mathbb{F}_2^l$, the submatrix matching problem (SMP) consists in finding a vector $\underline{\vec{\epsilon}_I} \in \mathbb{F}_2^{k+l}$ with weight $wt(\underline{\vec{\epsilon}_I}) = p$, so that $Q'_l\underline{\vec{\epsilon}_I} = \vec{\varsigma}_{[l]}$.*

Remark 3.7.1. *The submatrix matching problem is a binary vectorial version of the subset sum/knapsack problem (see [32]).*

If we define Q'_l as the first l rows of the matrix Q' (cf. section 3.6), the problem is equivalent to solving equation (3.31). Obviously every improvement in solving this problem automatically improves algorithm 3.7. Recall that the collision search was originally introduced by Stern to have more control over the number of times where the costly operation $\vec{\epsilon}_{I^*} = \vec{\varsigma} + Q\vec{\epsilon}_I$ (for example in line 2 of algorithm 3.3) needs to be computed. Therefore the submatrix matching problem is just an instance of the computational syndrome decoding (CSD) problem introduced in section 2.2.3 with smaller parameters. However a solution of the SMP is not directly a solution of the CSD problem.
Nevertheless the SMP is a highly divisible problem: For example the FS-ISD algorithm divides the problem into two halves by trying to find vectors $\underline{\vec{\epsilon}_{I_1}}$ and $\underline{\vec{\epsilon}_{I_2}}$ with $\mathrm{wt}(\underline{\vec{\epsilon}_{I_1}}) = \mathrm{wt}(\underline{\vec{\epsilon}_{I_2}}) = p/2$. In contrast to Lee-Brickell's algorithm this can be understood as a divide & conquer approach of depth 1, which lowers the computational complexity, but comes at the cost of imposing an additional constraint onto the weight distribution of the error vector $\vec{\epsilon}$ (weight $p/2$ in two times $(k+l)/2$ positions instead of just weight p in k positions as with Lee-Brickell's algorithm).
Becker, Joux, May and Meurer even use a divide & conquer approach of depth 3 (^{8}) resulting in constraints on the weight distribution of the error vector $\vec{\epsilon}$ similar to those in figure 3.8.

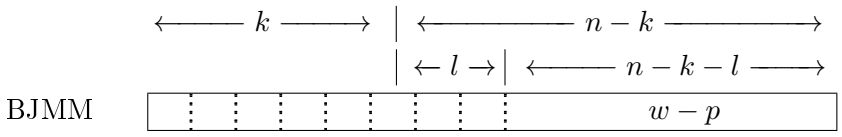


Figure 3.8: rough sketch of the weight distribution of $\vec{\epsilon}$ as demanded by algorithm 3.9

However they do not split the information set into entirely disjoint sets, but allow the sets to intersect, i.e. they use the equation $\underline{\vec{\epsilon}_I} = \underline{\vec{\epsilon}_{I_1}} + \underline{\vec{\epsilon}_{I_2}}$ and allow for $I_1 \cap I_2 \neq \emptyset$. This is somewhat similar to the birthday speedup (cf. optimization 3.4.4). Recall from remark 3.4.4 that it seems contradictory to allow for $I_1 \cap I_2 \neq \emptyset$, but nevertheless try to find a vector $\underline{\vec{\epsilon}_I}$ with $\mathrm{wt}(\underline{\vec{\epsilon}_I}) = p$ consisting of two other vectors each required to have weight $p/2$ and thus requiring $I_1 \cap I_2 = \emptyset$ to succeed. Therefore the BJMM-algorithm allows an error vector $\underline{\vec{\epsilon}_I}$ to split into $\underline{\vec{\epsilon}_{I_1}} + \underline{\vec{\epsilon}_{I_2}}$ with $\mathrm{wt}(\underline{\vec{\epsilon}_{I_1}}) = \mathrm{wt}(\underline{\vec{\epsilon}_{I_2}}) = p/2 + \Delta_1 := p_1$, where Δ_1 is an algorithmic parameter. The basic idea is to accept a certain additional weight Δ_1 on the vectors $\underline{\vec{\epsilon}_{I_1}}$ and $\underline{\vec{\epsilon}_{I_2}}$ in the hope that exactly these Δ_1 additional 1's cancel out in the sum $\underline{\vec{\epsilon}_{I_1}} + \underline{\vec{\epsilon}_{I_2}}$

[8] According to their tests, this particular depth seems optimal in combination with the BJMM-algorithm.

due to the simple equation $1+1=0$ in $\mathbb{F}_2$ (which explains the title of [21]). This idea was inspired by the paper [32].
We can do the same for both $\vec{\epsilon}_{I_1}$ and $\vec{\epsilon}_{I_2}$, i.e. write them as sums of two vectors with weight $p_2 := p_1/2 + \Delta_2$, and so on (until we reach a depth of 3). All in all we hope that the real error vector $\vec{\epsilon}$ can be *represented* as a sum of all of these vectors. Note that the additional weights Δ_1 and Δ_2 provide us with additional possibilities to represent the vectors $\vec{\epsilon}_I$, $\vec{\epsilon}_{I_1}$ and $\vec{\epsilon}_{I_2}$. That is also why figure 3.8 does not contain any specific weight specifications for the real $\vec{\epsilon}_I$ and just indicates the split of $\vec{\epsilon}_I$ into a sum of 8 vectors.

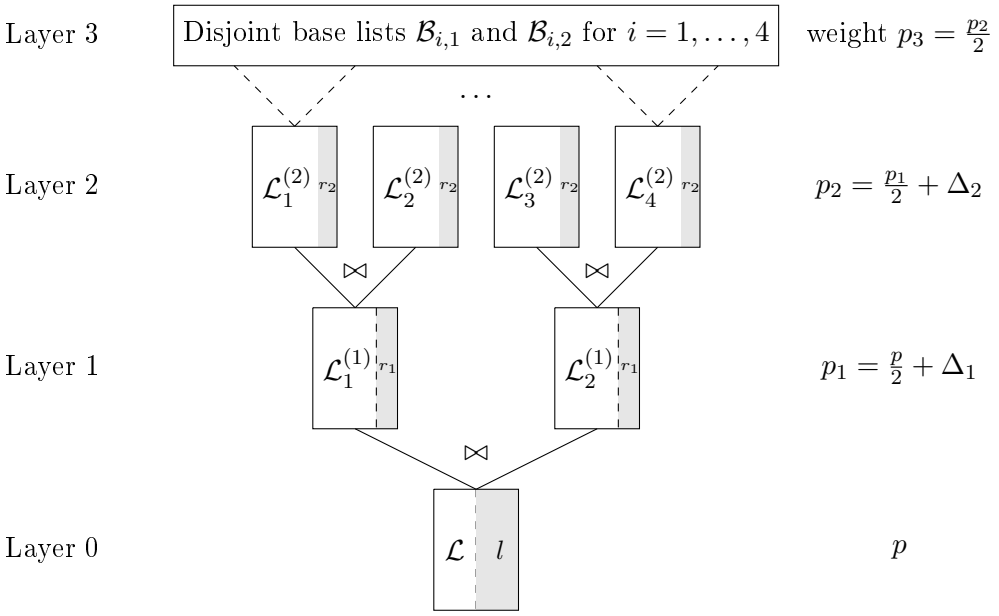


Figure 3.9: Illustration of the divide & conquer structure of algorithm 3.9; the original picture appeared in [21].

Figure 3.9 illustrates the overall idea, which we explain from bottom to top as an act of list/problem splitting, although the actual algorithm works from top to bottom and happens to merge the lists: On layer 0 we have our usual list of candidates for the real $\vec{\epsilon}_I$ with weight p, that can be written as a sum of the two vectors $\vec{\epsilon}_{I_1}$ and $\vec{\epsilon}_{I_2}$ both with weight $p_1 = p/2 + \Delta_1$ and that happen to match the syndrome on its first l entries $((Q'\vec{\epsilon}_I)_{[l]} + \vec{\varsigma}_{[l]} = \vec{0} \Leftrightarrow \vec{\varsigma}_{[l]} + (Q'\vec{\epsilon}_{I_1})_{[l]} = (Q'\vec{\epsilon}_{I_2})_{[l]})$. We can define

$$\mathcal{L} := \left\{ \vec{\epsilon}_I \in \mathbb{F}_2^{k+l} \mid \mathrm{wt}(\vec{\epsilon}_I) = p \text{ and } (Q'\vec{\epsilon}_I)_{[l]} + \vec{\varsigma}_{[l]} = \vec{0} \right\} \tag{3.38}$$

We split this list into two lists $\mathcal{L}_1^{(1)}$ and $\mathcal{L}_2^{(2)}$ containing candidates for $\vec{\epsilon}_{I_1}$ and $\vec{\epsilon}_{I_2}$. It is possible to write both $\vec{\epsilon}_{I_1}$ and $\vec{\epsilon}_{I_2}$ as sums of two more vectors each, let's say $\vec{\epsilon}_{I_1} = \vec{e}_1^{(2)} + \vec{e}_2^{(2)} \in \mathbb{F}_2^{k+l}$ and $\vec{\epsilon}_{I_2} = \vec{e}_3^{(2)} + \vec{e}_4^{(2)} \in \mathbb{F}_2^{k+l}$. The superscript index is meant to indicate the layer. These vectors $\vec{e}_i^{(2)}$ all have weight $\mathrm{wt}(\vec{e}_i^{(2)}) = p_2 := p_1/2 + \Delta_2$ using the same representation trick as before. To keep the number of elements in $\mathcal{L}_1^{(1)}$ and $\mathcal{L}_2^{(1)}$ relatively small, it is additionally required that candidates for $\vec{\epsilon}_{I_1}$ and $\vec{\epsilon}_{I_2}$ from the lists of layer 2 match some *pseudo-syndromes* $\vec{u}_1^{(1)} \in \mathbb{F}_2^{r_1}$ and $\vec{u}_2^{(1)} \in \mathbb{F}_2^{r_1}$ on their first $r_1 \leq l$ entries. Thus we get

$$\mathcal{L}_i^{(1)} := \left\{ \vec{\epsilon}_{I_i} \in \mathbb{F}_2^{k+l} \mid \mathrm{wt}(\vec{\epsilon}_{I_i}) = p_1 \text{ and } (Q'\vec{\epsilon}_{I_i})_{[r_1]} + \vec{u}_i^{(1)} = \vec{0} \right\} \quad \text{for } i = 1, 2 \tag{3.39}$$

However May et al. choose the pseudo-syndromes, so that $(Q'\underline{\vec{\epsilon}_I})_{[r_1]} = (Q'(\underline{\vec{\epsilon}_{I_1}} + \underline{\vec{\epsilon}_{I_2}}))_{[r_1]} \stackrel{!}{=} \vec{\varsigma}_{[r_1]}$ already holds for layer 0. To be more precise, they choose $\vec{u}_1^{(1)} \in_r \mathbb{F}_2^{r_1}$ and set $\vec{u}_2^{(1)} := \vec{\varsigma}_{[r_1]} + \vec{u}_1^{(1)}$ to make that equation hold. Note that $\vec{u}_1^{(1)}$ is chosen uniformly at random, so that layer 1 can be assumed to be independent of layer 0 with regard to the choice of the "syndrome".
Exactly the same is done on layer 2, just with a different parameter $r_2 \leq r_1 \leq l$:

$$\mathcal{L}_i^{(2)} := \left\{ \vec{e}_i^{(2)} \in \mathbb{F}_2^{k+l} \mid \mathrm{wt}(\vec{e}_i^{(2)}) = p_2 \text{ and } (Q'\vec{e}_i^{(2)})_{[r_2]} + \vec{u}_i^{(2)} = \vec{0} \right\} \quad \text{for } i = 1, \ldots, 4 \quad (3.40)$$

This time we choose $\vec{u}_1^{(2)}, \vec{u}_3^{(2)} \in_r \mathbb{F}_2^{r_2}$ and set $\vec{u}_{2j}^{(2)} := (\vec{u}_j^{(1)})_{[r_2]} + \vec{u}_{2j-1}^{(2)}$ for $j = 1, 2$, which guarantees that $(Q'\underline{\vec{\epsilon}_{I_i}})_{[r_2]} = (Q'(\vec{e}_{2i-1}^{(2)} + \vec{e}_{2i}^{(2)}))_{[r_2]} \stackrel{!}{=} (\vec{u}_i^{(1)})_{[r_2]}$ for $i = 1, 2$ already holds for layer 1.
The question how to create those lists on layer 2 remains: May et al. exemplary explain how to create the list $\mathcal{L}_1^{(2)}$. The other lists can be constructed analogously (cf. algorithm 3.9). They once again write the vector $\vec{e}_1^{(2)}$ as $\vec{e}_1^{(2)} = \vec{y} + \vec{z}$, but this time they demand that the positions of the 1's in $\vec{y}$ and $\vec{z}$ do not overlap. Recall that this is the classical approach already seen in Stern's algorithm (cf. section 3.4). More precisely they choose an index set $P_{1,1} \subset \{1, 2, \ldots, k+l\}$ at random with $|P_{1,1}| = \lceil (k+l)/2 \rceil$, that indexes the positions where $\vec{y}$ may distribute $p_3 = p_2/2$ many 1's. The remaining entries of $\vec{y}$ are demanded to be zero. Similarly $P_{1,2} := \{1, \ldots, k+l\} \setminus P_{1,1}$ indexes the positions where $\vec{z}$ may distribute p_3 many 1's. Then they simply iterate over all possible vectors $\vec{y}$ and $\vec{z}$ exposing the demanded weight distribution in a brute-force approach and add those vectors to the initial base lists $B_{1,1}$ and $B_{1,2}$, which are used to create the list $\mathcal{L}_1^{(2)}$.
Note that the index sets are chosen uniformly at random this time[9] to ensure that the choice of $\mathcal{L}_1^{(2)}$ is independent of the choice of $\mathcal{L}_2^{(2)}$, which would not be the case if both lists were created from the same base lists.
It is also very important to see that *not* every vector $\vec{e}_1^{(2)}$ can be written as $\vec{e}_1^{(2)} = \vec{y} + \vec{z}$ with one half of its $1's$ at positions indexed by the uniformly chosen index set $P_{1,1}$ and the other half at positions indexed by the disjoint set $P_{1,2} = \{1, 2, \ldots, k+l\} \setminus P_{1,1}$. In fact the probability that such a split is possible for an element $\vec{l}_1 \in \mathcal{L}_1^{(2)}$ as defined in equation (3.40) is

$$\mathcal{P}_s := \frac{\binom{(k+l)/2}{p_2/2}^2}{\binom{k+l}{p_2}} \quad (3.41)$$

Remark 3.7.2. *Therefore the list definitions in equation (3.38), (3.39) and (3.40) are not entirely precise: Since we require list entries on layer 2 (cf. figure 3.9) to have a certain structure (it must be possible to split them into elements of the base lists), this structure must already exist on the other layers (layer 1 and 0). So basically the list definitions would have to include the additional requirement that the split on layer 2 is possible. As they are defined in equation (3.38), (3.39) and (3.40) they give a superset of the real lists occurring in algorithm 3.9; their sizes can be used as upper bounds on the list sizes occurring in algorithm 3.9. We prefer to use the more exact equations though.*
Also note that the splits on layer 1 and layer 2 are always possible.

Since the choice of $\mathcal{L}_1^{(2)}$ is independent of the choice of $\mathcal{L}_2^{(2)}$, we can use $(\mathcal{P}_s)^2$ to model the probability that both an element $\vec{l}_1 \in \mathcal{L}_1^{(2)}$ and an element $\vec{l}_2 \in \mathcal{L}_2^{(2)}$ split as desired (if we don't use the adapted list definition from remark 3.7.2).

[9] In Stern's algorithm we fixed the index sets as $I_1 := \{1, \ldots, \lceil \frac{k}{2} \rceil\}$ and $I_2 := \{\lceil \frac{k}{2} \rceil + 1, \ldots, k\}$.

As already mentioned, the actual BJMM-algorithm starts the other way around with the 8 base lists $B_{i,1}$ and $B_{i,2}$ for $i = 1, \ldots, 4$ and merges $B_{i,1}$ and $B_{i,2}$ to reach layer 2, then merges the lists from layer 2 to reach layer 1 and so on. This is indicated by the $\bowtie$ symbol in figure 3.9. Thereby algorithm 3.8 realizes the merge-operation. Basically the function findColl() just encapsulates the search for collisions between the entries of two lists. In line 5 of algorithm 3.8 we check whether the collisions have the desired weight for the next layer and were not found before (duplicates may occur, if $\vec{l}_1$ and $\vec{l}_2$ are not required to have their 1's in disjoint positions). For example we can rewrite the FS-ISD algorithm to use the findColl() function resulting in a searchFS() function as displayed in algorithm C.1 in appendix C. In contrast the function searchBJMM(), which is the essential part of the BJMM-algorithm as part of the isd()-algorithm from section 3.1, is provided in pseudocode-notation as algorithm 3.9.

Algorithm 3.8: findColl($\mathcal{L}_1, \mathcal{L}_2, Q', \vec{u}, r, p^*$)

Input: lists $\mathcal{L}_1$ and $\mathcal{L}_2$ with entries from $\mathbb{F}_2^{k+l}$, matrix $Q' \in \mathbb{F}_2^{(n-k)\times(k+l)}$, vector $\vec{u} \in \mathbb{F}_2^r$, integer $0 \leq r \leq n-k$, integer $0 \leq p^* \leq w$

Output: list $\mathcal{L}$ containing all entries $\vec{l}_1 \in \mathcal{L}_1$, $\vec{l}_2 \in \mathcal{L}_2$ with $\vec{u}_{[r]} + (Q'\vec{l}_1)_{[r]} = (Q'\vec{l}_2)_{[r]}$

1 **foreach** $\vec{l}_1 \in \mathcal{L}_1$ **do** $\mathcal{L}_1'[\vec{l}_1] \leftarrow \vec{u}_{[r]} + (Q'\vec{l}_1)_{[r]}$
2 **foreach** $\vec{l}_2 \in \mathcal{L}_2$ **do** $\mathcal{L}_2'[\vec{l}_2] \leftarrow (Q'\vec{l}_2)_{[r]}$
3 **foreach** $\vec{l}_1, \vec{l}_2, \mathcal{L}_1'[\vec{l}_1] = \mathcal{L}_2'[\vec{l}_2]$ **do**
4 $\vec{l} := \vec{l}_1 + \vec{l}_2$
5 **if** $wt(\vec{l}) = p^*$ **and** $\vec{l} \notin \mathcal{L}$ **then** $\mathcal{L} \leftarrow \vec{l}$
6 **end**
7 **return** $\mathcal{L}$

In order to properly estimate the runtime of algorithm 3.9, we first need to analyse the runtime of the function findColl() from algorithm 3.8. Assuming that all list elements share a common weight $a := \mathrm{wt}(\vec{l}_1) = \mathrm{wt}(\vec{l}_2)\ \forall\ \vec{l}_1 \in \mathcal{L}_1, \vec{l}_2 \in \mathcal{L}_2$ ($\mathcal{L}_1$ and $\mathcal{L}_2$ are the input lists of findColl()), we know that lines 1 and 2 of algorithm 3.8 mean a selection and r-bit vector addition of a columns of Q' for each of the elements of both lists. This should explain the first part of equation (3.42). Keep in mind that $a \neq p^*$. Also note that we do *not* use the trick from section 3.6 to do a full Gaussian elimination in algorithm 3.1 and use the fact that we know the L-matrix in $Q' = (Q \mid L)$. This time we rather do a partial Gaussian elimination in algorithm 3.1 only, because we are unsure whether various theorems from [21] still hold, if Q' is not entirely uniform. The rather small advantage of doing a *partial* Gaussian elimination is quantified at a later point in this section.

$$time\{findColl()\} = (|\mathcal{L}_1| + |\mathcal{L}_2|) \cdot ar + C \cdot (k+l) + t_{sort}(|\mathcal{L}_1|) + t_{sort}(|\mathcal{L}_2|) \tag{3.42}$$

$$mem\{findColl()\} = \min\{|\mathcal{L}_1|, |\mathcal{L}_2|\} \cdot (r+k+l) + |\mathcal{L}| \cdot (k+l) \tag{3.43}$$

We use C to denote the number of collisions in line 3 of algorithm 3.8, because we do not know whether the lists $\mathcal{L}_1$ and $\mathcal{L}_2$ are entirely uniform (indeed, this is not the case for the BJMM-algorithm due to the way the lists are constructed).

For each collision we need to do a $k+l$-bit addition in line 4 (especially if $\vec{l}_1$ and $\vec{l}_2$ intersect in some positions). We neglect the weight check as well as the duplicate check ($\vec{l} \notin \mathcal{L}$) in line 5. Neglecting the complexity of the duplicate check is reasonable, if we replace the list $\mathcal{L}$ with a hash map that only allows exactly one entry in each of its buckets[10].

[10] The size of the hash map needs to be chosen large enough. If we should still fear unwanted hash collisions within the map, we could also allow for multiple entries in each bucket and check whether the entries are really duplicates whenever a hash collision occurs.

Remark 3.7.3. *For both $\mathcal{L}'_1$ and $\mathcal{L}'_2$ we propose to use (hash) maps with 2^r buckets that allow for multiple entries within each of their buckets similar to remark 3.4.2; actually it is sufficient to just use one hash map and check for collisions on the fly. Thereby it is desirable to ensure that the number of entries within each bucket does not grow too large in order not to loose the advantage of using (hash) maps. Basically we desire an equation such as $\frac{\min\{|\mathcal{L}_1|,|\mathcal{L}_2|\}}{2^r} = \mathcal{O}(1)$ to hold. Unfortunately this imposes relatively strict constraints on the choices of the parameters Δ_1 ($\Rightarrow r_1$) and Δ_2 ($\Rightarrow r_2$) for algorithm 3.9; it is unclear, whether exactly those choices are optimal.*
Therefore May et al. propose a different way to implement algorithm 3.8 (cf. [21, section 3]), which does not introduce any additional constraints, allows for a more proper analysis and is probably more memory-efficient, but also happens to perform worse than this proposal by at least a logarithmic factor: Basically it lexicographically sorts the input lists according to the labels $\vec{u}_{[r]} + (Q'\vec{l}_1)_{[r]}$ and $(Q'\vec{l}_2)_{[r]}$ and uses the sorted lists to find collisions in just one iteration over all elements of the lists. It is well known that sorting an input lists $\mathcal{L}_1$ can be achieved in $\mathcal{O}(|\mathcal{L}_1|\log_2(|\mathcal{L}_1|))$ iterations. During each iteration we need to compare at least r bits resulting in an overall complexity of $\mathcal{O}(|\mathcal{L}_1|\log_2(|\mathcal{L}_1|)\cdot r)$.
To model the possibility that using hash maps results in suboptimal parameters we introduce two summands of the following kind into equation (3.42):

$$t_{sort}(|\mathcal{L}_1|) := \begin{cases} 0 & \text{if } \frac{\min\{|\mathcal{L}_1|,|\mathcal{L}_2|\}}{2^r} = \mathcal{O}(1) \quad \textit{(hash maps usable)} \\ \mathcal{O}(|\mathcal{L}_1|\log_2(|\mathcal{L}_1|)\cdot r) & \text{else} \end{cases} \tag{3.44}$$

We propose to use $\frac{\min\{|\mathcal{L}_1|,|\mathcal{L}_2|\}}{2^r} < \log_2(\min\{|\mathcal{L}_1|,|\mathcal{L}_2|\})$ as a more practical condition to check, whether or not hash maps are usable.
Since it is not that important, we do not adapt the memory consumption modeled by equation (3.43).

All in all we obtain equation (3.42). With regard to the memory consumption of findColl() we only model the *additional* memory required by executing findColl() (i.e. we do not include the memory required by the input parameters): The lists $\mathcal{L}'_1$ and $\mathcal{L}'_2$ store r-bit values at $k+l$-bit indices and the returned list $\mathcal{L}$ contains $|\mathcal{L}|$-many $k+l$-bit entries. We only need to store one of the lists $\mathcal{L}'_1$ and $\mathcal{L}'_2$ (the smaller one) in practice and can check for collisions with entries of the other list directly after computing the r-bit label[11]. This results in equation (3.43). Note that the size $|\mathcal{L}|$ remains unknown, if we cannot specify how many collisions passed the weight and duplicate check in line 5 of algorithm 3.8. It is worth mentioning that at the end of every execution of findColl() we can not only free the memory used by the lists $\mathcal{L}'_1$ and $\mathcal{L}'_2$, but also the memory used by the input lists $\mathcal{L}_1$ and $\mathcal{L}_2$ as they are not required anymore in algorithm 3.9.
Let us define $|\mathcal{L}^{(j)}| := \max_i\{|\mathcal{L}_i^{(j)}|\}$ as well as $|\mathcal{B}| := |\mathcal{B}_{i,1}| = |\mathcal{B}_{i,2}| = \binom{(k+l)/2}{p_3}$ (for an even $k+l$) and assume that the lists of algorithm 3.9 are balanced on each layer. Then we can write equation (3.42) and equation (3.43) more concretely in combination with algorithm 3.9, where findColl() always gets two lists of size $|\mathcal{L}^{(j+1)}|$ as input and outputs a list of size $|\mathcal{L}^{(j)}|$. In that case we have

$$time\{findColl()\} = 2|\mathcal{L}^{(j+1)}|p_{j+1}(r_j - r_{j+1}) + \frac{|\mathcal{L}^{(j+1)}|^2}{2^{r_j - r_{j+1}}}(k+l) + 2t_{sort}(|\mathcal{L}^{(j+1)}|) \tag{3.45}$$

$$mem\{findColl()\} = |\mathcal{L}^{(j+1)}|\cdot(r_j - r_{j+1} + k + l) + |\mathcal{L}^{(j)}|\cdot(k+l) \tag{3.46}$$

$$\text{for } j = 0,1,2 \quad (\text{with } |\mathcal{L}^{(3)}| := |\mathcal{B}|,\ |\mathcal{L}^{(0)}| := |\mathcal{L}|,\ r_3 := 0,\ r_0 := l)$$

[11] This only works with hash maps, i.e. when no list sorting is required.

Algorithm 3.9: searchBJMM()

Input: parity check matrix $\widehat{H} = (Q \mid L \mid S) \in \mathbb{F}_2^{(n-k)\times n}$, syndrome $\vec{\varsigma} \in \mathbb{F}_2^{n-k}$, $w = \mathrm{wt}(\vec{e}) \leq k + l$, algorithmic parameter $0 \leq p \leq w$, algorithmic parameters $0 \leq r_2 \leq r_1 \leq l \leq n - k - w + p$, algorithmic parameters $0 \leq \Delta_1 \leq k + l - p$, $0 \leq \Delta_2 \leq k + l - p_1$

Output: success indicator (true/false), error vector $\vec{\epsilon} \in \mathbb{F}_2^n$

/* choose and set parameters */

1 $p_1 := p/2 + \Delta_1$; $p_2 := p_1/2 + \Delta_2$; $p_3 = p_2/2$

2 choose $\vec{u}_1^{(1)} \in_r \mathbb{F}_2^{r_1}$

3 $\vec{u}_2^{(1)} := \vec{\varsigma}_{[r_1]} + \vec{u}_1^{(1)}$

4 choose $\vec{u}_1^{(2)}, \vec{u}_3^{(2)} \in_r \mathbb{F}_2^{r_2}$

5 $\vec{u}_2^{(2)} := (\vec{u}_1^{(1)})_{[r_2]} + \vec{u}_1^{(2)}$; $\vec{u}_4^{(2)} := (\vec{u}_2^{(1)})_{[r_2]} + \vec{u}_3^{(2)}$

/* create base lists */

6 **for** $i = 1 \ldots 4$ **do**

7 choose the index set $P_{i,1} \subset \{1, \ldots, k + l\}$, $|P_{i,1}| = \lceil (k + l)/2 \rceil$ at random

8 $P_{i,2} := \{1, 2, \ldots, k + l\} \setminus P_{i,1}$

9 $\mathcal{B}_{i,1} := \{\vec{y} \in \mathbb{F}_2^{k+l} \mid \mathrm{wt}(\vec{y}) = p_3,\ \vec{y}_{P_{i,2}} = \vec{0}\}$

10 $\mathcal{B}_{i,2} := \{\vec{z} \in \mathbb{F}_2^{k+l} \mid \mathrm{wt}(\vec{z}) = p_3,\ \vec{z}_{P_{i,1}} = \vec{0}\}$

11 **end**

/* find collisions */

12 **for** $i = 1 \ldots 4$ **do** $\mathcal{L}_i^{(2)} \leftarrow$ `findColl` $(\mathcal{B}_{i,1}, \mathcal{B}_{i,2}, Q', \vec{u}_i^{(2)}, r_2, p_2)$

13 **for** $i = 1 \ldots 2$ **do** $\mathcal{L}_i^{(1)} \leftarrow$ `findColl` $(\mathcal{L}_{2i-1}^{(2)}, \mathcal{L}_{2i}^{(2)}, Q', \vec{u}_i^{(1)}, r_1, p_1)$

14 $\mathcal{L} \leftarrow$ `findColl` $(\mathcal{L}_1^{(1)}, \mathcal{L}_2^{(1)}, Q', \vec{\varsigma}_{[l]}, l, p)$

/* use collisions */

15 **foreach** $\underline{\vec{\epsilon}_I} \in \mathcal{L}$ **do**

16 $S\underline{\vec{\epsilon}_{I^*}} := \vec{\varsigma} + Q'\underline{\vec{\epsilon}_I}$

17 **if** $wt(S\underline{\vec{\epsilon}_{I^*}}) = w - p$ **then**

18 $\underline{\vec{\epsilon}_{I^*}} \leftarrow$ `remove` $(S\underline{\vec{\epsilon}_{I^*}}, l)$

19 $\vec{\epsilon} \leftarrow$ `prepend` $(\underline{\vec{\epsilon}_I}, \underline{\vec{\epsilon}_{I^*}})$

20 **return** $(true, \vec{\epsilon})$

21 **end**

22 **end**

23 **return** $(false, \vec{0})$

There are several subtle differences to the previously mentioned equations: The expected number of collisions C between lists on layer $j + 1$ is given by $C = |\mathcal{L}^{(j+1)}|^2/2^{r_j - r_{j+1}}$, because the entries from the lists of layer $j+1$ already collide on r_{j+1} bits by construction, i.e. only $r_j - r_{j+1}$ bits are significant for the number of collisions. Similarly we only need to compute the labels for these significant $r_j - r_{j+1}$ bits in line 1 and 2 of algorithm 3.8. Thereby we would like to stress once more that the expected size of the output list $|\mathcal{L}^{(j)}|$ is *not* identical to the expected number of collisions between the two lists of layer $j + 1$, because collisions have to pass the weight and duplicate check in line 5 of algorithm 3.8. To estimate the expected size of the output list $|\mathcal{L}^{(j)}|$, we can rather observe from the list definitions (cf. equations (3.38), (3.39) and (3.40)) in combination with remark 3.7.2 that equation (3.47) holds for $p_0 := p$ and $f(j) = 2^{|2-j|}$. The factor $(\mathcal{P}_s)^{f(j)}$ accommodates the fact that only a fraction of $\mathcal{P}_s$ many entries of the lists $\mathcal{L}_i^{(2)}$ as defined in equation (3.40)

can be expected to split into the base lists $\mathcal{B}_{i,1}$ and $\mathcal{B}_{i,2}$. On layer 1 this is only a fraction of $(\mathcal{P}_s)^2$ entries and on layer 0 a fraction of $(\mathcal{P}_s)^4$ entries.

$$|\mathcal{L}^{(j)}| = \underbrace{\binom{k+l}{p_j}}_{b_j} \cdot 2^{-r_j} \cdot (\mathcal{P}_s)^{f(j)} \quad \text{for } j = 0, 1, 2 \tag{3.47}$$
$$|\mathcal{L}^{(3)}| = |\mathcal{B}| = \binom{(k+l)/2}{p_3}$$

Let us define $b_j := \binom{k+l}{p_j}$ and assume that equation (3.47) is always correct apart from constant factors. If equation (3.47) does not hold, we could actually stop the computation of algorithm 3.9 and just restart algorithm 3.9 in the hope that we choose "better" vectors $\vec{u}_i^{(j)}$ this time. May et al. actually prove in [21, appendix A] that this strategy works for almost all randomly chosen matrices Q' at the cost of a constant runtime factor.
To estimate the runtime and memory consumption of algorithm 3.9 we refer to lemma 3.7.1, lemma 3.7.2 and equation (3.51). All in all we obtain the following set of equations:

$$\begin{aligned} time\{searchBJMM()\} &= 8|\mathcal{B}| \cdot p_3 r_2 + \frac{4|\mathcal{B}|^2}{2^{r_2}}(k+l+p_2(r_1-r_2)) \\ &+ \frac{2}{2^{r_1}}\left(\frac{|\mathcal{B}|^4}{2^{r_2}}(k+l) + b_1(\mathcal{P}_s)^2 p_1(l-r_1)\right) \\ &+ \frac{(\mathcal{P}_s)^4}{2^l}\left(\frac{b_1^2}{2^{r_1}}(k+l) + b_0 p(n-k-l)\right) \\ &+ \sum_{i=1}^{3} 2^i \cdot t_{sort}(|\mathcal{L}^{(i)}|) \end{aligned} \tag{3.48}$$

$$\begin{aligned} mem\{searchBJMM()\} &= M(3) + \mathcal{O}\Bigg(\max\Bigg\{|\mathcal{B}| \cdot (r_2+k+l), \frac{|\mathcal{B}|^2}{2^{r_2}} \cdot (r_1-r_2+k+l), \\ &\frac{b_1(\mathcal{P}_s)^2}{2^{r_1}} \cdot (2l-r_1+k)\Bigg\}\Bigg) \end{aligned} \tag{3.49}$$

$$\overline{PR}_{BJMM}[success = true] = \frac{\binom{k+l}{p}\binom{n-k-l}{w-p}}{\binom{n}{w}} \cdot (\mathcal{P}_s)^4 \cdot c(3) \tag{3.50}$$

Lemma 3.7.1. *Under the following assumptions equation (3.48) models the number of binary operations required by algorithm 3.9 apart from constant factors:*

1. *Only the lines 12,13,14 and 16 are relevant for the overall runtime of algorithm 3.9.*
2. *Apart from constant factors, equation (3.47) holds.*

Proof. We use the notation t_i to denote the (expected) number of binary operations required in line i.
Then we simply use equation (3.45) in combination with equation (3.47) to get

$$\begin{aligned} t_{12} &= 4 \cdot \left(2|\mathcal{B}| \cdot p_3 r_2 + \frac{|\mathcal{B}|^2}{2^{r_2}}(k+l) + 2t_{sort}(|\mathcal{B}|)\right) \\ t_{13} &= 2 \cdot \left(2|\mathcal{L}^{(2)}| \cdot p_2(r_1-r_2) + \frac{|\mathcal{L}^{(2)}|^2}{2^{r_1-r_2}}(k+l) + 2t_{sort}(|\mathcal{L}^{(2)}|)\right) \\ &= 2 \cdot \left(2\frac{b_2\mathcal{P}_s}{2^{r_2}} \cdot p_2(r_1-r_2) + \frac{(b_2\mathcal{P}_s)^2}{2^{r_1+r_2}}(k+l) + 2t_{sort}(|\mathcal{L}^{(2)}|)\right) \end{aligned}$$

$$= 2 \cdot \left(2\frac{|\mathcal{B}|^2}{2^{r_2}} \cdot p_2(r_1 - r_2) + \frac{|\mathcal{B}|^4}{2^{r_1+r_2}}(k+l) + 2t_{sort}(|\mathcal{L}^{(2)}|) \right)$$

$$t_{14} = 2|\mathcal{L}^{(1)}| \cdot p_1(l - r_1) + \frac{|\mathcal{L}^{(1)}|^2}{2^{l-r_1}} \cdot (k+l) + 2t_{sort}(|\mathcal{L}^{(1)}|)$$

$$= 2\frac{b_1(\mathcal{P}_s)^2}{2^{r_1}} \cdot p_1(l - r_1) + \frac{b_1^2(\mathcal{P}_s)^4}{2^{l+r_1}} \cdot (k+l) + 2t_{sort}(|\mathcal{L}^{(1)}|)$$

$$t_{16} = p(n-k-l) \cdot |\mathcal{L}| = p(n-k-l) \cdot b_0 \cdot 2^{-l} \cdot (\mathcal{P}_s)^4$$

Note that we do not need to compute the first l bits in line 16, because we discard them in line 18 anyway.
Equation (3.48) follows as $t_{12} + t_{13} + t_{14} + t_{16}$. □

Lemma 3.7.2. *Under the following assumptions equation (3.49) models the memory consumption of algorithm 3.9:*

1. *Only the lines 9,10,12,13,14 are relevant for the memory required by algorithm 3.9.*
2. *Apart from constant factors, equation (3.47) holds.*

Proof. To save memory, the following strategy can be applied: Instead of executing lines 12 to 14 in the order as displayed in algorithm 3.9, we can first compute the lists $\mathcal{L}_1^{(2)}$ and $\mathcal{L}_2^{(2)}$ from their base lists, free the memory used by the base lists after computing each of them, then merge those two lists to $\mathcal{L}_1^{(1)}$, free the memory occupied by $\mathcal{L}_1^{(2)}$ and $\mathcal{L}_2^{(2)}$ and do the same for the right half of the tree in figure 3.9. This way there are always at most two lists of the same layer and one list per layer below them (except for layer 0) in memory at the same time. Assuming that there are $\phi + 1$ layers (ϕ layers above layer 0), the maximum memory consumption imposed by saving lists at any point in time is thus given by

$$M(\phi) := \mathcal{O}\left(\max_{i=0,\dots,\phi} \left\{ \min\left\{2^i, 2\right\} \cdot |\mathcal{L}^{(i)}| + \sum_{j=1}^{i-1} |\mathcal{L}^{(j)}| \right\} \cdot (k+l) \right) \tag{3.51}$$

The factor $\min\left\{2^i, 2\right\}$ is meant to cope with the possibility that the list $\mathcal{L}$ might consume more memory than any other combination.
In contrast the naive way as presented in algorithm 3.9 implies a maximum memory consumption of $M_n(\phi) := \mathcal{O}\left(\max_{i=0,\dots,\phi}\left\{2^i \cdot |\mathcal{L}^{(i)}|\right\} \cdot (k+l)\right)$ from storing the lists, which is clearly suboptimal to $M(\phi)$.
We still need to add the memory used in lines 12, 13 and 14 of algorithm 3.9 that can be freed after a single execution of findColl(), namely the memory occupied by the lists $\mathcal{L}_1'$ and $\mathcal{L}_2'$ during each execution of algorithm 3.8. We use the notation m_i to denote the additional memory required and discarded after a single call of findColl() in line i.
By combining the first part of equation (3.46) with equation (3.47) we obtain:

$$m_{12} = \mathcal{O}\left(|\mathcal{B}| \cdot (r_2 + k + l)\right)$$

$$m_{13} = \mathcal{O}\left(|\mathcal{L}^{(2)}| \cdot (r_1 - r_2 + k + l)\right) = \mathcal{O}\left(\frac{|\mathcal{B}|^2}{2^{r_2}} \cdot (r_1 - r_2 + k + l)\right)$$

$$m_{14} = \mathcal{O}\left(|\mathcal{L}^{(1)}| \cdot (l - r_1 + k + l)\right) = \mathcal{O}\left(\frac{b_1(\mathcal{P}_s)^2}{2^{r_1}} \cdot (2l - r_1 + k)\right)$$

Equation (3.49) then follows as $M(3) + \max\{m_{12}, m_{13}, m_{14}\}$, because the consumption implied by m_{12}, m_{13} and m_{14} never occurs at the same time.
Regarding line 14 we could also decide not to store the list $\mathcal{L}$, but rather use collisions directly in line 15 ff. at the cost of omitting the duplicate check in line 5 of algorithm 3.8. □

To understand equation (3.50), it is necessary to consider the criteria that make algorithm 3.9 succeed:

1. We require that the real error vector $\vec{\epsilon}$ has a weight of p on its first $k+l$ bits (also cf. figure 3.8). The probability to see such an $\vec{\epsilon}$ is $\binom{k+l}{p}\binom{n-k-l}{w-p}\binom{n}{w}^{-1}$.

2. The l-bit collision requirement in line 14 of algorithm 3.9 must hold. This is the case for the real $\underline{\vec{\epsilon}_I}$ according to equation (3.31). It just filters out many (though not all) incorrect choices of $\underline{\vec{\epsilon}_I}$.

3. It must be possible to split the first $k+l$ bits of the real error vector, i.e. $\underline{\vec{\epsilon}_I}$, into $\underline{\vec{\epsilon}_{I_1}}$ and $\underline{\vec{\epsilon}_{I_2}}$, so that $\underline{\vec{\epsilon}_I} = \underline{\vec{\epsilon}_{I_1}} + \underline{\vec{\epsilon}_{I_2}}$ with $\mathrm{wt}(\underline{\vec{\epsilon}_{I_1}}) = \mathrm{wt}(\underline{\vec{\epsilon}_{I_2}}) = p_1 := p/2 + \Delta_1$ (cf. figure 3.9). Due to the choice of p_1 and since $\mathrm{wt}(\underline{\vec{\epsilon}_I}) = p$ this is *always* possible. Actually there are even $R_1 := \binom{p}{p/2}\binom{k+l-p}{\Delta_1}$ ways to split the vector $\underline{\vec{\epsilon}_I}$ into the vectors $\underline{\vec{\epsilon}_{I_1}}$ and $\underline{\vec{\epsilon}_{I_2}}$. However just one of these *representations* in the lists $\mathcal{L}_i^{(1)}$, $i = 1, 2$ suffices to make algorithm 3.9 succeed. That's why May et al. introduced the artificial requirement that $(Q'\underline{\vec{\epsilon}_{I_i}})_{[r_1]} + \vec{u}_i^{(1)} = \vec{0}$ (cf. equation (3.39)) for $i = 1, 2$. Imagine splitting the list $\mathcal{L}$ into two lists without this requirement: Clearly, their size would be larger than necessary by a factor of R_1. To counteract, we can fix r_1 as $r_1 \approx \log_2(R_1)$. Note that this choice does *not* depend on r_2, because we are looking at figure 3.9 from bottom to top.
   Also note that for any $\underline{\vec{\epsilon}_{I_1}} \in \mathcal{L}_1^{(1)}$ that makes the additional requirement $(Q'\underline{\vec{\epsilon}_{I_i}})_{[r_1]} + \vec{u}_i^{(1)} = \vec{0}$ hold, the corresponding $\underline{\vec{\epsilon}_{I_2}} \in \mathcal{L}_2^{(2)}$ automatically fulfils the additional requirement by the construction of the $\vec{u}_i^{(1)}$'s.
   Nevertheless this way of randomly discarding valid representations of the solution vector $\vec{\epsilon}$ on layer 1 introduces a probability to fail. More precisely we define the random variable X_1, which shall equal one, if at least one representation of the real error vector $\vec{\epsilon}$ survives the split between layer 0 and layer 1 (cf. figure 3.9) and zero otherwise. Clearly the probability that none of the R_1 representations survive that split is given by $Pr[X_1 = 0] = (1 - 2^{-r_1})^{R_1}$. Note that $Pr[X_1 = 0] \sim e^{-1}$ is constant for the choice $r_1 \approx \log_2(R_1)$.

4. It must be possible to split the vectors $\underline{\vec{\epsilon}_{I_1}}$ and $\underline{\vec{\epsilon}_{I_2}}$ from the first layer into vectors from the second layer (cf. figure 3.9), i.e. it must be possible to write $\underline{\vec{\epsilon}_{I_i}} = \vec{e}_{2i-1}^{(2)} + \vec{e}_{2i}^{(2)}$ with $\mathrm{wt}(\vec{e}_j^{(2)}) = p_2 := p_1/2 + \Delta_2$ for $j = 1, \ldots, 4$. For a correct choice of $\underline{\vec{\epsilon}_{I_i}}$ with $\mathrm{wt}(\underline{\vec{\epsilon}_{I_i}}) = p_1$, $i = 1, 2$ this is once again *always* possible. This time we use the additional requirement $(Q'\vec{e}_j^{(2)})_{[r_2]} + \vec{u}_j^{(2)} = \vec{0}$ to make only a fraction of 2^{-r_2} of the $R_2 := \binom{p_1}{p_1/2}\binom{k+l-p_1}{\Delta_2}$ many representations of $\underline{\vec{\epsilon}_{I_i}}$ survive. Once more the construction of the $\vec{u}_j^{(2)}$'s ensures a pairwise "survival" of the $\vec{e}_j^{(2)}$'s. A sensible choice of the parameter r_2 immediately follows as $r_2 \approx \log_2(R_2)$.
   Analogously to item 3 we define the random variables $X_{2,1} \in \{0, 1\}$ and $X_{2,2} \in \{0, 1\}$ to model the possible event that none or at least one of the representations on layer 1 survives the splits between layer 1 and 2. We know that $Pr[X_{2,1} = 0] = Pr[X_{2,2} = 0] = (1 - 2^{-r_2})^{R_2}$.

5. The vectors $\vec{e}_j^{(2)}$ on the second layer must be in a form that allows for a disjoint split with $p_3 := p_2/2$ 1's in positions indexed by the set $P_{j,1}$ and another p_3 1's in positions indexed by the set $P_{j,2}$ with $|P_{j,1}| = |P_{j,2}| = (k+l)/2$ (assuming that

$k+l$ is even). Clearly, for a specific vector $\vec{e}_j^{(2)}$ this is only possible with probability $\mathcal{P}_s := \binom{(k+l)/2}{p_2/2}^2 \binom{k+l}{p_2}^{-1}$. Since we choose the index sets $P_{j,1}, P_{j,2}$ independently for each $j = 1, \ldots, 4$, we may assume that the splitting probabilities for each $\vec{e}_j^{(2)}$, $j = 1, \ldots, 4$ are independent as well. Thus we can use $\mathcal{P} := (\mathcal{P}_s)^4$ as the probability that all splits on the second layer are possible for the *real* error vector $\vec{\epsilon}$.
May et al. point out that $\mathcal{P}$ is asymptotically only inverse-polynomial in n and therefore does not affect the asymptotic advantage of using algorithm 3.9 over previously discussed algorithms.

As only item 1 and 5 contain non-constant success probabilities, the overall success probability can be computed as $\binom{k+l}{p}\binom{n-k-l}{w-p}\binom{n}{w}^{-1} \cdot \mathcal{P}$ multiplied by a constant factor $c(3)$; this results in equation (3.50).
Thereby the constant factor $c(\phi)$ is defined in accordance to item 3 and 4 as

$$c(\phi) := Pr\left[\prod_{i,j} X_{i,j} = 1\right] \tag{3.52}$$

It models the probability that the j-many splits between the layers $i-1$ and the layers $1 \leq i \leq \phi - 1$ work as intended, i.e. do not discard all valid representations of the error vector $\vec{\epsilon}$. Unfortunately $Pr[X_{i,j} \cdot X_{i+1,j} = 1] \neq Pr[X_{i,j} = 1] \cdot Pr[X_{i+1,j} = 1]$, i.e. the probabilities are *not* independent, because the probability to have a valid representation of $\vec{\epsilon}$ on layer 2 ($\phi = 3$) becomes zero, if layer 1 is known to contain no valid representation at all. However we can use the *union bound* to obtain the upper bound $c(\phi) = 1 - Pr[\prod_{i,j} X_{i,j} = 0] \geq 1 - \sum_{i,j} Pr[X_{i,j} = 0]$.

Remark 3.7.4. *Using the union bound is a rather theoretical way to solve the problem of dependent probabilities. For practical purposes it seems plausible to use* $c(\phi) = Pr[\prod_{i,j} X_{i,j} = 1] \approx \prod_{i,j} Pr[X_{i,j} = 1]$ *even though the probabilities are dependent, because the dependence can have a positive effect as well (if more representations than expected survive a split, even more representations exist for the next one). This is especially required for more layers ($\phi > 3$, cf. appendix E), where the union bound becomes extremely bad ($c(\phi) \geq 0$). Therefore we used this more practical approach in section 4 (table 4.3).*
Other than that, the parameters r_1 and r_2 could be chosen slightly smaller at the cost of increasing the list sizes. In any case they should be rounded down.

To optimize the runtime of algorithm 3.9, we need to optimize the parameter set $(p, l, \Delta_1, \Delta_2)$ with regard to the runtime of the function isd() (algorithm 3.1) from section 3.1. Note that choosing Δ_i immediately fixes r_i according to the previously mentioned equations $r_1 \approx \log_2\left(\binom{p}{p/2}\binom{k+l-p}{\Delta_1}\right)$ and $r_2 \approx \log_2\left(\binom{p_1}{p_1/2}\binom{k+l-p_1}{\Delta_2}\right)$ with $p_1 := p/2 + \Delta_1$. It seems almost impossible to see anything general from the equations 3.48 and 3.50. A brute-force strategy to find the optimal parameters within certain ranges is feasible though. The results of such an implementation are presented in section 4. Also recall that we only need to do a partial Gaussian elimination in algorithm 3.1, which lowers the complexity of the function randomize() to that of doing a Gaussian elimination on a matrix $T' \in \mathbb{F}_2^{(n-k-l)\times(n-k-l)}$ only. By substituting $(n-k) \to (n-k-l)$ we obtain a complexity of $(n-k-l)^2 \cdot (n - \frac{1}{2}(n-k-l+1))$ binary operations for doing a naive partial Gaussian elimination. The same substitution also works for the optimizations of the Gaussian elimination process (we ensured that it works at least for the formulas in table 4.1); only the analysis of the overall success probability of algorithm 3.1 in combination with optimization 3.1.3 needs to be adapted to cope with the l additional columns (also cf. remark B.0.9).

Remark 3.7.5. *Note that the substitution is not entirely correct, because it only respects the complexity of obtaining an identity matrix on the $n-k-l$ lower rows of the matrix $S \in \mathbb{F}_2^{(n-k)\times(n-k-l)}$ and disrespects the complexity of creating an all-zero matrix in the first l rows (cf. figure 3.6). For example a more exact formula for the complexity of a naive partial Gaussian elimination would be $(n-k)(n-k-l)\cdot(n-\frac{1}{2}(n-k-l+1))$.*
Due to the utter insignificance of the Gaussian elimination step for most information set decoding algorithms and practically relevant parameter sets one can usually live with that error though.

For asymptotic observations we can find a rough bound of $2^{0.04934n}$ for the runtime of algorithm 3.1 in combination with algorithm 3.9 in [21]. So at least from an asymptotic point of view, the BJMM-algorithm is the current state-of-the-art information set decoding algorithm. May et al. also show that their algorithm is not a mere time-memory trade-off in comparison to the previously discussed algorithms as well as in comparison to the MMT-algorithm, which is presented in [20]. We could verify this in practice (cf. table 4.3). Formulas to model the runtime and memory complexity of the BJMM algorithm for an arbitrary number of layers can be found in appendix E.

Optimizations

Optimization 3.7.1 (Reusing vector additions)**.** *Clearly, we can apply optimization 3.3.1 to algorithm 3.9 (lines 1 and 2 of algorithm 3.8). However there are several possibilities how to exactly apply the technique:*

1. *We can either apply it for every one of the four calls of the findColl() function in line 12 of algorithm 3.9 and precompute eight times*[12] $\binom{(k+l)/2}{p_3}$ *column sums or directly precompute all of the $\binom{k+l}{p_3}$ possible column sums and therefore only require to do the precomputations once.*

2. *We could try to partially apply the technique to lines 13 and 14 as well. For example it is possible to compute the $\binom{(k+l)/2}{p_3}$ column sums on l rather than r_2 bits. Whenever we need to compute something such as $(Q'\vec{l}\,)_{[r_j]}$ to obtain a list of layer j, we can then use the fact that the vector $\vec{l}$ can be rewritten as the sum of $f(j) = 2^{|2-j|}$ elements of base lists, for which we precomputed the column sums. Thus the computation $(Q'\vec{l}\,)_{[r_j]}$ would cost $2^{|2-j|}\cdot(r_j - r_{j+1})$ instead of $p_{j+1}(r_j - r_{j+1})$ binary operations after performing the precomputations at a cost of $l\cdot\binom{(k+l)/2}{p_3}(1+\delta((k+l)/2, p_3))$ once (for $\delta(k,p)$ cf. optimization 3.3.1).*
   Note that it does not make any sense to compute significantly more than $\binom{(k+l)/2}{p_3}$ column sums: If we computed e.g. all $\binom{k+l}{p_2}$ column sums, we could also directly create the lists on layer 2.

Regarding item 1 it is useful to know that $\binom{k+l}{p_3}\binom{(k+l)/2}{p_3}^{-1} > 2^{p_3}$ (cf. lemma 3.7.3), i.e. computing eight times $\binom{(k+l)/2}{p_3}$ column sums is certainly the better choice, if $p_3 \geq 3$. For $p_3 = 2$ it becomes questionable, whether this optimization technique makes any sense at all; we do not need to precompute anything, but can rather save the sums of two columns ($p_3 = 2$) whenever we need to compute them and reuse the saved sums whenever possible. For $p_3 = 1$ we do not need to compute any sums at all – we just select columns.
To estimate whether the second proposal to compute the $\binom{(k+l)/2}{p_3}$ column sums on e.g. l

[12] We might be able to reuse a part of the precomputations of the column sums of Q' for non-disjoint index sets $P_{i,b}$, $P_{j,b}$, $j \neq i$ though.

rather than r_2 bits is worthwhile, let us first review the naive approach: Using the naive approach of precomputing eight times $\binom{(k+l)/2}{p_3}$ column sums we get for the runtime t_{12} in line 12 of algorithm 3.9:

$$t_{12} = 4 \cdot \left(2|\mathcal{B}| \cdot r_2 \cdot (1 + \delta((k+l)/2, p_3)) + \frac{|\mathcal{B}|^2}{2^{r_2}}(k+l) + 2t_{sort}(|\mathcal{B}|) \right)$$

The runtime in the other lines remains the same. According to lemma 3.7.1 this results in

$$time\{searchBJMM()[o_{3.7.1}]\} = 8|\mathcal{B}| \cdot (1 + \delta((k+l)/2, p_3))r_2 + \ldots$$

whilst the memory consumption increases by a constant factor.
In contrast the second proposal implies the following runtimes (cf. lemma 3.7.1), if we decide to compute the $\binom{(k+l)/2}{p_3}$ column sums on l rather than r_2 bits:

$$\begin{aligned} t_{12} &= 4 \cdot \Big(2|\mathcal{B}| \cdot l \cdot (1 + \delta((k+l)/2, p_3)) + \ldots \\ t_{13} &= 2 \cdot \Big(2|\mathcal{L}^{(2)}| \cdot f(1) \cdot (r_1 - r_2) + \ldots \\ t_{14} &= 2 \cdot |\mathcal{L}^{(1)}| \cdot f(0) \cdot (l - r_1) + \ldots \end{aligned}$$

So in comparison to the naive approach we get an additional workload of $A := 8|\mathcal{B}| \cdot (l - r_2)(1 + \delta((k+l)/2, p_3))$ in line 12 and gain an advantage of $G := 4|\mathcal{L}^{(2)}|(p_2 - f(1))(r_1 - r_2) + 2|\mathcal{L}^{(1)}| \cdot (p_1 - f(0))(l - r_1)$ in lines 13 and 14. Thus the second proposal is worthwhile, if $G > A$. Certainly this is not the case for all parameter sets: The inequality does not hold, if $|\mathcal{L}^{(1)}|$ or $|\mathcal{L}^{(2)}|$ happen to be significantly smaller than $|\mathcal{B}|$ or if the p_i's are rather small. However this does not seem to be the usual case in practice (cf. table 4.3 on page 70). Nevertheless we implemented both options for section 4. Whenever this optimization technique was applied, table 4.3 contains the runtime complexities for the better of the two variants. Thereby "1" indicates that the naive variant was used, whereas "2" means that the precomputation of the column sums on l bits was more optimal.
The formulas in table 4.1 refer to the naive variant.

Lemma 3.7.3. $\binom{k+l}{p_3}\binom{(k+l)/2}{p_3}^{-1} > 2^{p_3}$

Proof.

$$\begin{aligned} \binom{k+l}{p_3} &= \frac{(k+l)!}{p_3!(k+l-p_3)!} = \binom{(k+l)/2}{p_3} \cdot \frac{(k+l) \cdot \ldots \cdot ((k+l)/2+1)}{(k+l-p_3) \cdot \ldots \cdot ((k+l)/2-p_3+1)} \\ &= \binom{(k+l)/2}{p_3} \cdot \frac{(k+l) \cdot \ldots \cdot (k+l-p_3+1)}{((k+l)/2) \cdot \ldots \cdot ((k+l)/2-p_3+1)} > \binom{(k+l)/2}{p_3} \cdot 2^{p_3} \end{aligned}$$

Actually we even have $\binom{k+l}{p_3} \approx \binom{(k+l)/2}{p_3} \cdot 2^{p_3}$ as long as $k+l \gg p_3$. Lemma 3.7.3 immediately follows. □

Optimization 3.7.2 (Early abort)**.** *The usual early abort strategy introduced in optimization 3.3.2 also works in line 4 of algorithm 3.8 as well as in line 16 of algorithm 3.9. Applying this technique to those two lines reduces the runtime of the findColl() function to*

$$time\{findColl()[o_{3.7.2}]\} = 2|\mathcal{L}^{(j+1)}|p_{j+1}(r_j - r_{j+1}) + \frac{|\mathcal{L}^{(j+1)}|^2}{2^{r_j - r_{j+1}}}2(p_j + 1) + 2t_{sort}(|\mathcal{L}^{(j+1)}|)$$

and the overall runtime of algorithm 3.9 becomes

$$\begin{aligned} time\{searchBJMM()[o_{3.7.2}]\} = 8|\mathcal{B}| \cdot p_3 r_2 + \frac{4|\mathcal{B}|^2}{2^{r_2}}(2(p_2+1) + p_2(r_1 - r_2)) \\ + \frac{2}{2^{r_1}}\left(\frac{|\mathcal{B}|^4}{2^{r_2}}2(p_1+1) + b_1(\mathcal{P}_s)^2 p_1(l-r_1)\right) \\ + \frac{(\mathcal{P}_s)^4}{2^l}\left(\frac{b_1^2}{2^{r_1}}2(p+1) + b_0 2p(w-p+1)\right) \\ + \sum_{i=1}^{3} 2^i \cdot t_{sort}(|\mathcal{L}^{(i)}|) \end{aligned}$$

whilst the memory consumption remains the same.

Optimization 3.7.3 (Birthday Speedup)**.** *To understand whether the birthday speedup (cf. optimization 3.4.4) can be applied to algorithm 3.9 or not, let us first review how the BJMM algorithm works (also cf. figure 3.9): On layer 0 we hope to have the list $\mathcal{L}$ as defined in equation (3.38). Note that the additional constraint $(Q'\vec{e}_{\underline{I}})_{[l]} + \vec{\varsigma}_{[l]} = \vec{0}$ does not remove any valid solutions of the computational syndrome decoding problem. We split all of the list items of $\mathcal{L}$ into two summands that can be found in the lists $\mathcal{L}_1^{(1)}$ and $\mathcal{L}_2^{(1)}$. Per list item on layer 0 there exist R_1 many representations of these items on layer 1, so that we introduce an additional constraint (collisions on $r_1 \approx \log_2(R_1)$ bits) to control the number of representations, because we just need one representation of each of the list items on layer 0 on average. Basically the same is done between layer 1 and layer 2. However something strange happens between layer 2 and 3: We split each list item on layer 2 into two vectors, that are required to have their 1's in disjoint positions, because we desire to stop the divide-and-conquer approach and thus need to construct the lists on layer 2 in some way. May et al. use a meet-in-the-middle approach to do so, because it is the simplest and fastest way to obtain relevant insights from a theoretical point of view; however this also introduces the rather counter-intuitive and practically relevant inverse polynomial probability $\mathcal{P}_s$, which affects the list definitions on all layers (cf. remark 3.7.2). Alternatively we could use a straightforward approach and split each list element on layer 2 into two vectors as usual (with possibly non-disjoint sets of 1's), which both have a weight of $p_3' := p_2/2 + \Delta_3$. Then we could iterate over all of these $\binom{k+l}{p_3'}$ elements in a brute-force approach. Clearly, this part is more complex than the meet-in-the-middle approach, but also eliminates the split probability $\mathcal{P}_s$, which is beneficial for the success probability (cf. equation (3.50)) and hurts about the same in equation (3.47). It is unclear, whether or not this straightforward approach is more efficient. It is also rather unclear, whether the number of layers is optimal with this approach; even though it seems as if adding more layers could be useful to lower the brute force complexity, if needed. Answers to these questions can be found in section 4.*
Anyway it is important to see that such a split between layer 2 and 3 would once again imply a number of $R_3 := \binom{p_2}{p_2/2}\binom{k+l-p_2}{\Delta_3}$ representations of each list item from layer 2 on layer 3, i.e. the brute-force approach would imply a factor of R_3 useless computations[13]! Note that we cannot use the collision trick this time to reduce that number, because we do not know how to directly generate list items satisfying the collision constraint. This is where the birthday speedup helps: We can sample the required base lists uniformly at random and only generate as many base list items as can be assumed to be necessary to ensure that the base lists contain roughly one representation of every list item on layer 2.
The basic question to analyse this technique is: With what probability does a list item on

[13] Therefore $\Delta_3 = 0$ is always optimal with this approach.

layer 2 have a representation on layer 3, if the base lists are sampled uniformly at random and both have size N? For a fixed vector the chance to "hit" a representation of that vector by two uniform choices in $\mathbb{F}_2^{k+l}$ with weight p_3' is given by $R_3 \cdot \binom{k+l}{p_3'}^{-2}$. Since N^2 combinations exist, the probability that a list item on layer 2 has a representation on layer 3 is[14]

$$\mathcal{P}_N := 1 - \left(1 - R_3 \binom{k+l}{p_3'}^{-2}\right)^{N^2}$$

By using the following two redefinitions in the context of the birthday speedup in combination with the BJMM algorithm we can then easily obtain all relevant information from the equations (3.48), (3.49) and (3.50):

$$\mathcal{P}_s := \mathcal{P}_N \quad \textit{(redefinition)}$$
$$|\mathcal{B}| := N \quad \textit{(redefinition)}$$

Note that we introduced the additional variable Δ_3 this time to apply the representation trick from [21] to the birthday speedup as well.
As usual, one way to choose the new parameter N is to set it to $N \approx R_3^{-1/2}\binom{k+l}{p_3'}$. This way we get $\lim_{n\to\infty} \mathcal{P}_N = 1 - \frac{1}{e} \approx 63\%$ and $\lim_{n\to\infty}(\mathcal{P}_N)^4 = (1-\frac{1}{e})^4 \approx 16\%$. So in comparison to the straightforward brute-force approach our base lists are smaller by a factor of $R_3^{1/2}$ for this choice of N, but the success probability decreases to approximately 16% of its original value. This rather large penalty in comparison to the applications of the birthday speedup to other algorithms comes from the layer structure of algorithm 3.9. It remains the task of section 4 to discuss whether or not this penalty is worth it.

[14] Actually there exist no list items on layer 2 without representation in layer 3 in algorithm 3.9. Therefore the sentence only makes sense for a "perfect" list definition as the one in equation (3.40) without remark 3.7.2.

4 Comparison

Over the last fifty years many new information set decoding algorithms were developed and old ones improved upon. A common source of improvement is to generalize existing algorithms. Figure 4.1 shows the relations between the algorithms presented in this thesis. Thereby we may assume that a generalization is usually the better choice than a specialization in practice, simply because it is highly unlikely that fixing a parameter with a certain value for arbitrary parameter sets is always more optimal than being able to choose that parameter freely. In fact, Bernstein et al. proved in [19] that Ball-Collision Decoding (BCD) performs better than its specialization (Stern's algorithm) both asymptotically and in practice. May et al. provide some Mathematica code in combination with [21], which aims to prove that $\phi = 3$, i.e. using four layers for the BJMM algorithm is asymptotically optimal, which indirectly makes the FS-ISD algorithm ($\phi = 1$) asymptotically inferior to the BJMM algorithm. Moreover they proved in [20] that FS-ISD is asymptotically at least as efficient as the BCD algorithm; a statement which effectively makes the BJMM algorithm the best out of the discussed algorithms – at least asymptotically.

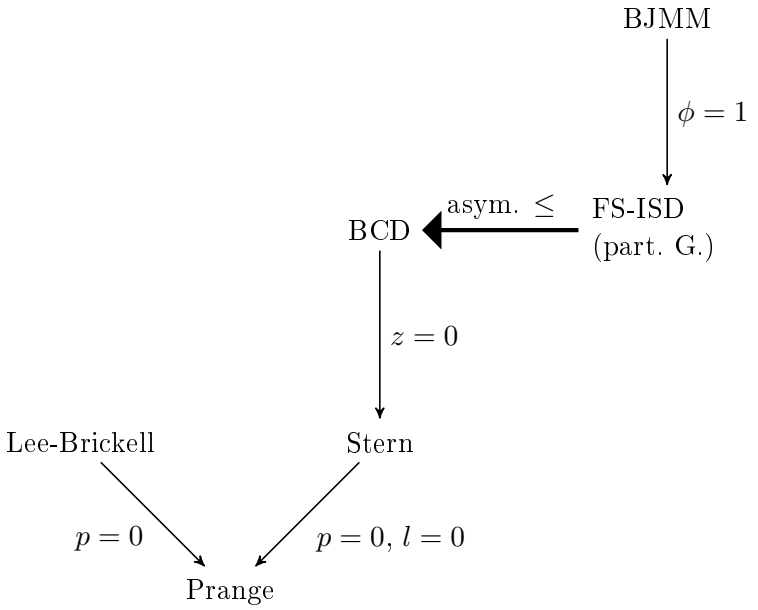


Figure 4.1: Generalizations and specializations of the presented information set decoding algorithms. $\phi + 1$ is the number of layers in the BJMM algorithm; the description in section 3.7 uses $\phi = 3$. Contrary to the FS-ISD algorithm described in section 3.6 we assume that FS-ISD uses a partial Gaussian elimination here.

It remains our task to estimate the practical relevance of these asymptotic statements. To do so, we used the formulas for the runtime and memory complexities of the various algorithms presented in the previous sections and implemented them in C++ with the help of the libraries NTL and GMP [33, 34], which were mostly chosen due to their support of big integers and arbitrary precision floating point operations. We also implemented the formulas for all of the optimizations presented in this thesis as well as arbitrary combinations of them. In particular we did not implement the specialized formulas ($\phi = 3$) for the BJMM algorithm presented in section 3.7, but rather used the formulas for an arbitrary number of layers from appendix E. The goal was to provide an application, which enables the user to find the optimal algorithmic parameters for a specific parameter set (n, k, w). This is possible by iterating over all sets of algorithmic parameters in a brute-force approach,

which can easily be achieved for many algorithms (Stern, Lee-Brickell, FS-ISD), because their parameter spaces are rather small. For some algorithms however (BCD, BJMM) it is necessary to only search in reasonable parameter ranges.
At the heart of the program lie the following two equations already mentioned in section 3.1:

$$time\{isd()\} = \overline{PR}_{ALG}[success]^{-1}(time\{randomize()\} + time\{searchALG()\}) \quad (4.1)$$

$$mem\{isd()\} = \mathcal{O}(n^2) + mem\{searchALG()\} \quad (4.2)$$

Recall that $time\{randomize()\}$ models the time spent in the Gaussian elimination process, $time\{searchALG()\}$ the time spent in the specific search algorithm and $\overline{PR}_{ALG}[success]$ the average success probability of the search algorithm. An overview of the formulas for the Gaussian elimination process in combination with different optimizations as well as an overview of the runtime formulas of the various search algorithms with different optimizations is presented in table 4.1. Table 4.2 shows the formulas for the average success probabilities. These formulas are closely related to the weight distributions of the error vector $\vec{\epsilon}$ that the different algorithms hope for (cf. figure 4.2).
All of these formulas (and some more) were implemented. We also implemented the following two *security bounds*, which are meant to help designers of code-based cryptosystems estimate the complexity of attacking a specific parameter set (n, k, w):

FS bound: This lower bound is the main result of the paper [26]. The bound is known to be beaten by Ball-Collision Decoding (see [19, section 6]) for sufficiently large n and can thus be expected to be beaten by BJMM and other algorithms, which are asymptotically at least as efficient as BCD, as well (cf. [21, table 1]); ironically this would even include the FS-ISD algorithm according to [20]. The bound is beaten for two reasons: First, the structure of [26, table 2] is not necessarily applicable to all information set decoding algorithms (Finiasz and Sendrier do not claim it to be though) and second, some of their assumptions do not seem to hold (cf. remark 3.4.3 and see [19, section 6]).
In any case it is interesting to investigate, whether or not this bound is beaten for practically relevant parameter sets as well.
Due to its complexity we do not display the concrete bound here, but rather refer the reader to [26, proposition 2].

BCD bound: In contrast Bernstein et al. proposed the following rather straightforward bound in 2010:

$$\min_{p \geq 0} \left\{ \frac{1}{2} \cdot \binom{n}{w} \binom{n-k}{w-p}^{-1} \binom{k}{p}^{-1/2} \right\} \quad (4.3)$$

This is basically the inverse success probability of Lee-Brickell's algorithm (cf. table 4.2) multiplied by $\binom{k}{p}^{1/2}$, because Bernstein et al. assume that each iteration of the algorithm has to consider at least $\binom{k}{p}^{1/2}$ error patterns (as with birthday-type attacks) and testing each pattern costs at least one binary operation. For a more detailed description the interested reader is referred to [19, appendix B]. The authors claim: "In any event, it is clear that beating this bound would be an astonishing breakthrough." Its rough bound $2^{0.05265n}$ however indicates that it is at least asymptotically beaten by the BJMM algorithm.

To concretely compare the performance of the information set decoding algorithms we chose the following parameter sets (n, k, w):

(1024,524,50): The parameter set originally proposed by R. J. McEliece in [3] for his cryptosystem (cf. section 2.2.2).

(1632,1269,34): McEliece parameter set proposed for 80-bit security in [14].

(2960,2288,57): McEliece parameter set proposed for 128-bit security in [14].

(6624,5129,117): McEliece parameter set proposed for 256-bit security in [14].

Note that these parameter sets were proposed in 2008 with Stern's algorithm in mind only. Our results are displayed in table 4.3: The time and memory complexities use logarithmic units (base 2) and the memory complexity ignores constant factors (Landau notation). Apart from that, all polynomial and mostly even constant factors are meant to be respected. With regard to the FS-ISD algorithm we looked at both the variant described in section 3.6 as well as the original FS-ISD algorithm, which only does a partial Gaussian elimination (part. G.). The BJMM algorithm was investigated in the three variants discussed in optimization 3.7.3, i.e. once with the meet-in-the-middle approach to create the base lists on the uppermost layer as described in the original paper [21] (BJMM), once with a uniform sampling approach (BJMM (uniform)) and once with a brute force approach for these lists (BJMM (brute f.)).
The parameters seen in table 4.3 are known to be the optimal ones for these specific (n, k, w), given that the parameter set is not marked with a "*" to indicate that we could not search the whole parameter space. The set M in combination with the BJMM algorithm is meant to contain the layers, in which the usage of hash maps was possible and taken into account according to remark 3.7.3 (e.g. $M = \{1, 2\}$ means that hash maps could be used on layers 1 and 2). Since we were interested in the impact of applying hash maps, we also list some parameter sets that are optimal, if hash map usage is avoided (marked with "(no hash maps)").
Note that the union bound mentioned in remark 3.7.4 was *not* applied for the BJMM algorithm, but we rather assumed the probabilities in question to behave as if they were independent. In all other cases where inequalities had to be applied, the runtime and memory complexities were correctly upper bounded.
Table 4.3 enables us to gain various interesting insights:

Gaussian elimination: Only Prange's algorithm always benefits from the Gaussian optimizations, because it basically only consists of performing Gaussian eliminations. That's also why for Prange's algorithm $x = 1$ always seems optimal for optimization 3.1.3. For all of the other algorithms, the Gaussian elimination process becomes utterly irrelevant for relatively large n already. For example Stern's algorithm with the parameter set $n = 1632$, $k = 1269$, $w = 34$ without optimization 3.1.3 has a complexity of 80.53276093 ($\log_2$), i.e. by using optimization 3.1.3 we gain a factor of less than 1.072. For the larger parameter sets and especially the BJMM algorithm this factor is even less significant. Therefore we only used the less powerful optimization 3.1.2 or no Gaussian elimination optimizations at all for these parameter sets.

Partial Gaussian elimination: Whether or not the FS-ISD algorithm uses a partial Gaussian elimination does not really make a difference. Nevertheless it is worth observing that the benefit of using a partial Gaussian elimination instead of a full Gaussian elimination becomes totally insignificant in the presence of techniques such as the Gaussian optimization 3.1.3.

Stern vs. FS-ISD vs. BCD: Stern's algorithm and the FS-ISD algorithm behave quite similarly. The BCD algorithm starts to become better than both for larger n at the cost of an increasing memory usage due to the relatively large choices of the parameter l. According to Bernstein et al. the reason for this observation lies in the fact

that the inner loop of the BCD algorithm (lines 2-5 and lines 8-11 of algorithm 3.6) makes BCD faster than Stern's algorithm and the FS-ISD algorithm by a polynomial factor (see [19, section 4]). In combination with the proof from [20] this indicates that BCD and FS-ISD are asymptotically equivalent, but BCD performs better by the aforementioned polynomial factor in practice.

Birthday Speedup for FS-ISD: At first sight it is rather surprising to see that the birthday speedup is not the best way to optimize the FS-ISD algorithm. In fact using optimization 3.6.1, which is incompatible to the birthday speedup (optimization 3.6.3), is always the better choice: For example for $n = 2960$, $k = 2288$, $w = 57$, FS-ISD with optimizations 3.1.2 and 3.6.2 only has a runtime complexity of 129.4602838 ($\log_2$), which is insignificantly lowered by the birthday speedup, but lowered by a factor of more than 2 by optimization 3.6.1 (cf. table 4.3).

BJMM memory consumption: As long as using hash maps is allowed, the BJMM algorithm performs better than all other algorithms, even for the smaller parameter sets. However the comparatively large choices of l and p result in a larger memory consumption than with the other algorithms. Luckily it seems to be possible to find BJMM parameters that trade a relatively small runtime factor for a relatively large memory consumption factor. We exemplary listed some of these parameter sets in table 4.3 (see e.g. the two values for the parameter set $(2960, 2288, 57)$ in the row "BJMM (uniform)"). Also note that we could find a parameter set for $(n, k, w) = (6624, 5129, 117)$, for which the BJMM algorithm (brute force) performs better than the BCD algorithm by a factor of at about 2^{10} whilst consuming roughly four times less memory.
However recall that the formulas modeling the memory consumption of the BJMM algorithm model the worst case memory consumption at any point in time during the execution of the algorithm. This implies that most of the time the memory consumption will be smaller, but the number of *memory accesses* may even be larger. Since memory access times are usually a bottleneck, this could be a problem in practice. Unfortunately it is not common in literature to model the number of memory accesses as well, because such models tend to be more complicated.

BJMM without hash maps: Without using hash maps, the BJMM algorithm is only relevant for the two larger parameter sets. For the parameter sets in table 4.3 the usual cost of not using hash maps is a factor of $2^2 \dots 2^3$.

BJMM variants: The meet-in-the middle approach for the BJMM base lists on the uppermost layer is inferior to both other variants in practice. Out of the latter two, the brute force approach is usually the better choice in table 4.3. However it must be admitted that we could not compute the success probability for extremely large N in combination with the uniform sampling approach (e.g. for the parameter set $(6624, 5129, 117)$) exactly, but had to use disadvantageous lower bounds instead[15] (similar to remark 3.5.4), so that the exact complexity might actually be better than the one of the brute force approach.

BJMM – optimal number of layers: $\phi = 2$ is usually optimal for BJMM, if the brute force approach is not used. So FS-ISD ($\phi = 1$) is clearly the suboptimal choice, but the asymptotically proven optimal choice $\phi = 3$ is not generally optimal neither. For BJMM in combination with the brute force approach $\phi = 3$ is almost always optimal in table 4.3, because adding an additional layer decreases the size of the base lists

[15] Even libraries such as NTL and GMP do not accept arbitrary length exponents.

and thus the complexity of iterating over all of their items, which seems to be worth the additional list merging operations between layer 3 and 2 (cf. figure 3.9).

BJMM optimizations: Let us exemplary outline the effect of the asymptotically most important optimization techniques, namely optimization 3.7.1 and optimization 3.7.2 on the BJMM algorithm: Without any optimizations the BJMM algorithm (brute f.) has a complexity of 82.62480007 ($\log_2$) for the parameters $n = 1632$, $k = 1269$, $w = 34$, $l = 93$, $p = 16$, $\phi = 3$, $\Delta_1 = 4$, $\Delta_2 = 0$. Applying optimization 3.7.2 impressively decreases the complexity to 78.11274743. Additionally applying optimization 3.7.1 (the more complex variant on l bits as indicated by the "2" in table 4.3) further decreases it to 77.27066725. Note that the naive variant of optimization 3.7.1 would only have had the effect of slightly decreasing the complexity to 78.1127468.
It may differ per parameter set, whether optimization 3.7.1 or optimization 3.7.2 is more significant.

FS bound: The FS bound is beaten for all parameter sets, that might offer an acceptable security level in practice. Therefore it must be considered to be useless in practice as it fails to achieve its goal, namely to "help other designers choose durable parameters with more ease" [26].

BCD bound: The BCD bound is beaten as well by two variants of the BJMM algorithm for the parameter set $(6624, 5129, 117)$. Even applying the union bound in the case of the brute force approach only increases the complexity to only 236.317966 binary operations ($\log_2$) and thus does not affect this statement. The question is: Why was the BCD bound beaten? The answer is relatively straightforward: First, the success probability of the BJMM algorithm is significantly better than the one of Lee-Brickell's algorithm due to the additional parameter l, which enables us to better balance the terms in the numerator as well as the number of iterations and the complexity per iteration in general (cf. table 4.2). Second, the BJMM algorithm considers less than $\binom{k+l}{p}^{1/2}$ and even less than $\binom{k}{p}^{1/2}$ error patterns per iteration by ignoring many patterns at the cost of losing a polynomial factor for the success probability; it requires more than one binary operation per error pattern though.
For instance for the parameter set $(6624, 5129, 117)$ the BCD bound implies $2^{133.0270922}$ iterations at a complexity of $2^{104.4693296}$ binary operations each (for $p = 23$ and without the constant factor). In contrast the BJMM algorithm using the brute-force approach only requires $2^{97.33275571}$ iterations, each at a complexity of $2^{138.0450819}$ bit operations.
We do not want to propose another security bound, but rather ask designers of code-based cryptosystems to consider the runtime complexities of the current state-of-the-art information set decoding algorithms instead. This can be achieved rather easily by using tools such as the one provided with this thesis.

All in all the BJMM algorithm can be expected to significantly lower the complexity of decoding random linear codes for practical applications as well.

Table 4.1: Overview of all algorithms/functions that were analysed in this thesis. The logarithmic base-2 examples (ex.) use $n = 1024, k = 524, w = 50$; other parameters are optimized with regard to the best result for that specific example, if possible.

function	**i**	**# binary operations**		**memory consumption**		**remark**
		formula	ex.	formula	ex.	
$randomize()[o_{3.1.1}]$	-	$= (n-k) \cdot b_r$ $b_r := n(n-k) - \frac{1}{2}(n-k)(n-k+1)$	27.53	$= \mathcal{O}(n^2)$	20.00	-
$randomize()[o_{3.1.1}, o_i]$	3.1.2	$= (n-k)[b_r - b_{opt-3.1.2}]$ $b_{opt-3.1.2} := \frac{(n-k)^2[2n^2-n-(n-k)^2]}{2n^2}$	26.31	$= \mathcal{O}(n^2)$	20.00	-
$randomize()[o_{3.1.1}, o_i]$	3.1.3	$= (n-k)[b_r - b_{opt-3.1.3}]$ $b_{opt-3.1.3} := s'(n-\frac{1}{2}) - \frac{1}{2}(s')^2$ $s' := n-k-x$	18.00 $x = 1$	$= \mathcal{O}(n^2)$	20.00	parameter x; supersedes optimization 3.1.2; increases the overall number of iterations (cf. appendix B)!
$randomize()[o_{3.1.1}, o_i]$	3.1.4	$\approx \frac{1}{2}\left(\frac{2^{r+1}(n-k)^2}{r} + \frac{(n-k)^3}{r}\right)$	25.29 $r = 6$	$= \mathcal{O}(n^2 + r(n-k) + 2^r(n-r))$	20.00	parameter r; might not work well with optimization 3.1.2 or 3.1.3.
$searchPrange()$	-	$= \mathcal{O}(1)$	0.00	$= \mathcal{O}(n)$	10.00	-
$searchLB()$	-	$= \binom{k}{p} \cdot (p(n-k) + \mathcal{O}(1))$	27.03 $p = 2$	$= \mathcal{O}(n)$	10.00	parameter $p = 2$ asymptotically optimal; $p = 1 \Rightarrow$ Prange
$searchLB()[o_i]$	3.3.1, 3.3.2	$= \binom{k}{p} \cdot (2(w-p+1)(1+\delta(k,p)) + \mathcal{O}(1))$	23.68 $p = 2$	$= \mathcal{O}\left(\binom{k}{p}(n-k) + n\right)$	26.03	cf. remark 3.3.2, $\delta(k,p) \ll 1$ for $k \gg p$
$searchStern()$	-	$= \binom{k/2}{p/2} pl + \frac{\binom{k/2}{p/2}^2}{2^l}(p(n-k-l) + \mathcal{O}(1))$	29.64 $p = 6$ $l = 26$	$= \mathcal{O}\left((l + k/2) \cdot \binom{k/2}{p/2}\right)$	29.67	parameter $p \in \{4, 6\}$ in practice, $\log_2(\binom{k/2}{p/2}) \leq l \leq \log_2(\binom{k/2}{p/2}) + \log_2(n-k-l)$
$searchStern()[o_i]$	3.4.2, 3.4.3	$= 2l(1+\delta(k/2, p/2))\binom{k/2}{p/2} + \frac{\binom{k/2}{p/2}^2}{2^l}(2p(w-p+1) + \mathcal{O}(1))$	27.76 $p = 6$ $l = 26$	$= \mathcal{O}\left((l + k/2) \cdot \binom{k/2}{p/2}\right)$	29.67	memory consumption increases by a constant factor
$searchStern()[o_i]$ ("Birthday speedup")	3.4.4	$= Npl + \frac{N^2}{2^l} \cdot (p(n-k-l) + \mathcal{O}(1))$	30.16 $p = 6$ $l = 28$	$= \mathcal{O}(N \cdot (l+k))$	31.45	additional parameter $0 \leq N < \binom{k}{p/2}$, sensible choice: $N = \binom{p}{p/2}^{-1/2}\binom{k}{p/2}$
$searchStern()[o_i]$ ("Birthday speedup")	3.4.4, 3.4.3	$= Npl + \frac{N^2}{2^l} \cdot (2p(w-p+1) + \mathcal{O}(1))$	29.83 $p = 6$ $l = 28$	$= \mathcal{O}(N \cdot (l+k))$	31.45	for sufficiently large N
$searchBCD()$	-	$= pl\binom{k/2}{p/2} + \min\{1, z\} \cdot l\binom{k/2}{p/2}\binom{l/2}{z/2} + \frac{\binom{k/2}{p/2}^2\binom{l/2}{z/2}^2}{2^l} \cdot p(n-k-l)$	31.32 $p = 6$ $l = 36$ $z = 2$	$= \mathcal{O}\left(l\binom{k/2}{p/2}\binom{l/2}{z/2} + \min\{1, z\}\frac{l}{2}\binom{l/2}{z/2} + \frac{k}{2}\binom{k/2}{p/2}\right)$	31.33	parameter $z = 2$ in practice, $z = 0$ (Stern) asymptotically suboptimal, $\log_2(\binom{k/2}{p/2}\binom{l/2}{z/2}) \leq l \leq \log_2(\binom{k/2}{p/2}\binom{l/2}{z/2}) + \log_2(n-k-l)$

Continued...

(continued)

function	i	# binary operations		memory consumption		remark
		formula	ex.	formula	ex.	
$searchBCD()[o_i]$	3.5.1, 3.5.2	$= 2l \cdot (1+\delta(k/2,p/2))\binom{k/2}{p/2} + \min\{1,z\} \cdot 2\binom{k/2}{p/2}\left(\frac{l}{2} + \binom{l/2}{z/2} \cdot (1+\delta(l/2,z/2))\right) + \frac{\binom{k/2}{p/2}^2\binom{l/2}{z/2}^2}{2^l} \cdot 2p(w-p-z+1)$	28.36 $p=6$ $l=36$ $z=2$	$= \mathcal{O}\left(l\binom{k/2}{p/2}\binom{l/2}{z/2} + \min\{1,z\}\frac{l}{2}\binom{l/2}{z/2} + \frac{k}{2}\binom{k/2}{p/2}\right)$	31.33	memory consumption increases by a constant factor
$searchFS()$	-	$= (p-\psi)l\binom{(k+l)/2}{p/2} + \frac{k}{k+l} \cdot \frac{\binom{(k+l)/2}{p/2}^2}{2^l} p(n-k-l)$	29.87 $p=6$ $l=26$	$= \mathcal{O}\left(\frac{1}{2}(3l+k)\binom{(k+l)/2}{p/2}\right)$	29.94	parameter p slightly larger than for Stern's algorithm, $\log_2(\binom{(k+l)/2}{p/2}) \leq l \leq \log_2(\binom{(k+l)/2}{p/2})) + \log_2(n-k-l)$, $\psi := \sum_{j=0}^{p/2}\binom{(k-l)/2}{p/2-j}\binom{l}{j} \cdot j \cdot \binom{(k+l)/2}{p/2}^{-1}$
$searchFS()[o_i]$	3.6.1, 3.6.2	$= (1+\delta\left(\frac{k+l}{2}, \frac{p}{2}\right))\left(1-\frac{\psi}{p}\right) 2l\binom{(k+l)/2}{p/2} + \frac{k}{k+l} \cdot \frac{\binom{(k+l)/2}{p/2}^2}{2^l} 2p(w-p+1)$	27.96 $p=6$ $l=26$	$= \mathcal{O}\left(\frac{1}{2}(3l+k)\binom{(k+l)/2}{p/2}\right)$	29.94	memory consumption increases by a constant factor
$searchBJMM()$	-	$= 8\lvert\mathcal{B}\rvert \cdot p_3 r_2 + \frac{4\lvert\mathcal{B}\rvert^2}{2^{r_2}}(k+l+p_2(r_1-r_2)) + \frac{2}{2^{r_1}}\left(\frac{\lvert\mathcal{B}\rvert^4}{2^{r_2}}(k+l) + b_1(\mathcal{P}_s)^2 p_1(l-r_1)\right) + \frac{(\mathcal{P}_s)^4}{2^l}\left(\frac{b_1^2}{2^{r_1}}(k+l) + b_0 p(n-k-l)\right) + \sum_{i=1}^{3} 2^i \cdot t_{sort}(\lvert\mathcal{L}^{(i)}\rvert)$	32.56 $\phi=2$ $p=8$ $l=42$ $\Delta_1=2$	$= M(3) + \mathcal{O}\left(\max\left\{\lvert\mathcal{B}\rvert \cdot (r_2+k+l), \frac{\lvert\mathcal{B}\rvert^2}{2^{r_2}} \cdot (r_1-r_2+k+l), \frac{b_1(\mathcal{P}_s)^2}{2^{r_1}} \cdot (2l-r_1+k)\right\}\right)$	32.78	BJMM only requires a partial gaussian elimination; for the defintions of the various parameters see section 3.7; generalizations and variants: appendix E
$searchBJMM()[o_i]$	3.7.1, 3.7.2	$= 8\lvert\mathcal{B}\rvert \cdot (1+\delta((k+l)/2,p_3))r_2 + \frac{4\lvert\mathcal{B}\rvert^2}{2^{r_2}}(2(p_2+1)+p_2(r_1-r_2)) + \frac{2}{2^{r_1}}\left(\frac{\lvert\mathcal{B}\rvert^4}{2^{r_2}}2(p_1+1) + b_1(\mathcal{P}_s)^2 p_1(l-r_1)\right) + \frac{(\mathcal{P}_s)^4}{2^l}\left(\frac{b_1^2}{2^{r_1}}2(p+1) + b_0 2p(w-p+1)\right) + \sum_{i=1}^{3} 2^i \cdot t_{sort}(\lvert\mathcal{L}^{(i)}\rvert)$	29.70 $\phi=2$ $p=8$ $l=42$ $\Delta_1=2$	$= M(3) + \mathcal{O}\left(\max\left\{\lvert\mathcal{B}\rvert \cdot (r_2+k+l), \frac{\lvert\mathcal{B}\rvert^2}{2^{r_2}} \cdot (r_1-r_2+k+l), \frac{b_1(\mathcal{P}_s)^2}{2^{r_1}} \cdot (2l-r_1+k)\right\}\right)$	32.78	memory consumption increases by a constant factor

Remark 4.0.6. *At first sight it is irritating to see that the memory examples in table 4.1 exhibit greater exponents than the runtime examples. However the memory formulas mostly describe the memory consumption in bits, whereas we assumed for the runtime model that it is possible to generate certain bit vectors of fixed length within a single binary operation. So basically the units for both example types are fundamentally different ones. That's also why the Landau notation is used for the memory formulas.*

Remark 4.0.7. *The values for the FS bound in table 4.3 are usually slightly below the values found in literature (e.g. in [26, 19]). The difference to the examples found in [26, table 3] can be explained by the fact that we did not discard small p's as suggested by Finiasz and Sendrier. Other than that the variable K_{w-p} was not fixed in [26], which makes comparisons even more difficult. We used $K_{w-p} := 2(w-p+1)$ with the early abort optimization in mind.*

algorithm	$\overline{PR}[success = true]$	**ex.**	**reference**
Prange	$\binom{n-k}{w}\binom{n}{w}^{-1}$	53.61	sec. 3.2
Lee-Brickell	$\binom{k}{p}\binom{n-k}{w-p}\binom{n}{w}^{-1}$	42.93	sec. 3.3
Stern	$\binom{k/2}{p/2}^2\binom{n-k-l}{w-p}\binom{n}{w}^{-1}$	33.70	sec. 3.4
Stern ("Birthday speedup")	$\frac{\binom{k}{p}\binom{n-k-l}{w-p}}{\binom{n}{w}}\left[1-\left(1-\binom{p}{p/2}\binom{k}{p/2}^{-2}\right)^{N^2}\right]$	32.97	opt. 3.4.4
Ball-Collision Decoding	$\binom{k/2}{p/2}^2\binom{l/2}{z/2}^2\binom{n-k-l}{w-p-z}\binom{n}{w}^{-1}$	33.33	sec. 3.5
FS-ISD	$\binom{(k+l)/2}{p/2}^2\binom{n-k-l}{w-p}\binom{n}{w}^{-1}$	33.28	sec. 3.6
BJMM	$\binom{k+l}{p}\binom{n-k-l}{w-p}\binom{n}{w}^{-1}\cdot(\mathcal{P}_s)^4\cdot c(3)$	31.51	sec. 3.7

Table 4.2: Overview of the average success probabilities per iteration of the algorithms that were analysed in this thesis. The logarithmic base-2 examples (ex.) refer to the inverse success probability $\overline{PR}[success = true]^{-1}$ and use $n = 1024$, $k = 524$, $w = 50$; the other parameters were chosen as in table 4.1. $\mathcal{P}_s := \binom{(k+l)/2}{p_2/2}^2\binom{k+l}{p_2}^{-1}$.

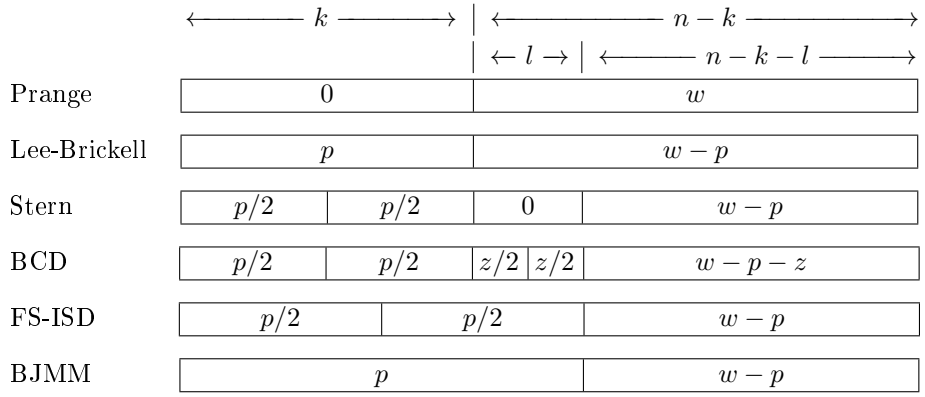


Figure 4.2: weight distribution of $\vec{\epsilon}$ for different information set decoding algorithms

Table 4.3: Runtime and memory complexities of the algorithms discussed in this thesis for concrete parameter examples.

(n,k,w)	algorithm	optimizations	time	mem	parameters
(1024,524,50)	FS bound	-	59.03068672	-	-
	BCD bound	-	49.69260406	-	-
	Prange	3.1.3	74.96105237	20.00140819	$x = 1$
	Lee-Brickell	3.1.3, 3.3.1, 3.3.2	67.31974294	26.05180852	$p = 2$, $x = 21$
	Stern	3.1.3, 3.4.2, 3.4.3	61.49962849	29.67043085	$l = 26$, $p = 6$, $x = 16$
	BCD	3.1.3, 3.5.1, 3.5.2	61.49962849	29.67043085	$l = 26$, $p = 6$, $z = 0$, $x = 16$ *
	FS-ISD	3.1.3, 3.6.1, 3.6.2	61.32666184	29.94419939	$l = 26$, $p = 6$, $x = 41$
		3.1.3, 3.6.3, 3.6.2	62.47230937	31.72874353	$l = 26$, $p = 6$, $N = 6176651$, $x = 73$ *
	FS-ISD (part. G.)	3.1.3, 3.6.1, 3.6.2	61.39336128	29.94419939	$l = 26$, $p = 6$, $x = 41$
	BJMM	3.1.3, 3.7.1[1], 3.7.2	61.23874904	32.78018819	$\phi = 2$, $l = 42$, $p = 8$, $\Delta_1 = 2$, $x = 39$, $M = \{1, 2\}$ *
	(no hashmaps)	3.1.3, 3.7.1[2], 3.7.2	63.05486635	28.10745039	$\phi = 2$, $l = 26$, $p = 6$, $\Delta_1 = 1$, $x = 29$, $M = \{\}$ *
	BJMM (uniform)	3.1.3, 3.7.2	59.93327887	29.51981192	$\phi = 2$, $l = 30$, $p = 6$, $\Delta_1 = 1$, $\Delta_2 = 0$, $N = 115000$, $x = 44$, $M = \{1, 2\}$ *
	(no hashmaps)	3.1.3, 3.7.2	62.08378628	29.50707214	$\phi = 2$, $l = 28$, $p = 6$, $\Delta_1 = 1$, $\Delta_2 = 0$, $N = 120000$, $x = 69$, $M = \{\}$ *
	BJMM (brute f.)	3.1.3, 3.7.1[1], 3.7.2	60.19616469	29.56963429	$\phi = 3$, $l = 30$, $p = 6$, $\Delta_1 = 1$, $\Delta_2 = 0$, $x = 38$, $M = \{1, 2\}$ *
	(no hashmaps)	3.1.3, 3.7.1[2], 3.7.2	62.09057953	29.54185414	$\phi = 2$, $l = 28$, $p = 6$, $\Delta_1 = 1$, $x = 71$, $M = \{\}$ *
(1632,1269,34)	FS bound	-	79.89972879	-	-
	BCD bound	-	68.60114018	-	-
	Prange	3.1.3	97.77578034	21.34573442	$x = 1$
	Lee-Brickell	3.1.3, 3.3.1, 3.3.2	88.61387167	28.13474649	$p = 2$, $x = 29$
	Stern	3.1.3, 3.4.2, 3.4.3	80.45794519	34.70839529	$l = 30$, $p = 6$, $x = 36$
	BCD	3.1.3, 3.5.1, 3.5.2	80.45794519	34.70839529	$l = 30$, $p = 6$, $z = 0$, $x = 36$ *
	FS-ISD	3.1.3, 3.6.1, 3.6.2	80.36424594	34.85289291	$l = 31$, $p = 6$, $x = 64$
		3.1.3, 3.6.3, 3.6.2	81.63695977	36.59671547	$l = 29$, $p = 6$, $N = 78311771$, $x = 95$ *
	FS-ISD (part. G.)	3.1.3, 3.6.1, 3.6.2	80.39786015	34.85289291	$l = 31$, $p = 6$, $x = 65$
	BJMM	3.1.4, 3.7.1[2], 3.7.2	78.76389955	44.97813375	$\phi = 2$, $l = 51$, $p = 10$, $\Delta_1 = 1$, $r = 6$, $M = \{1\}$ *
		3.1.4, 3.7.1[2], 3.7.2	78.51821136	52.47056773	$\phi = 2$, $l = 77$, $p = 14$, $\Delta_1 = 3$, $r = 6$, $M = \{1, 2\}$ *
	(no hashmaps)	3.7.1[2], 3.7.2	82.15989874	48.79537723	$\phi = 2$, $l = 61$, $p = 12$, $\Delta_1 = 2$, $M = \{\}$ *
	BJMM (uniform)	3.1.2, 3.7.2	78.70995636	46.57088464	$\phi = 2$, $l = 51$, $p = 10$, $\Delta_1 = 1$, $\Delta_2 = 0$, $N = 145520039$, $M = \{1\}$ *
		3.1.2, 3.7.2	78.1223687	54.17856675	$\phi = 2$, $l = 79$, $p = 14$, $\Delta_1 = 3$, $\Delta_2 = 0$, $N = 3319223401112$, $M = \{1, 2\}$ *
	BJMM (brute f.)	3.1.4, 3.7.1[2], 3.7.2	79.38055986	34.74485393	$\phi = 3$, $l = 35$, $p = 6$, $\Delta_1 = 1$, $\Delta_2 = 0$, $r = 6$, $M = \{1, 2\}$ *
		3.7.1[2], 3.7.2	77.27066725	58.0441454	$\phi = 3$, $l = 93$, $p = 16$, $\Delta_1 = 4$, $\Delta_2 = 0$, $M = \{1, 2\}$ *
	(no hashmaps)	3.7.1[2], 3.7.2	80.62077901	58.01580343	$\phi = 3$, $l = 91$, $p = 16$, $\Delta_1 = 4$, $\Delta_2 = 0$, $M = \{\}$ *

Continued...

(continued)

(n,k,w)	algorithm	optimizations	time	mem	parameters
(2960,2288,57)	FS bound	-	127.7918	-	-
	BCD bound	-	114.3058936	-	-
	Prange	3.1.3	148.7890564	23.06325024	$x = 1$
	Lee-Brickell	3.1.3, 3.3.1, 3.3.2	138.6928159	30.71860344	$p = 2$, $x = 35$
	Stern	3.1.2, 3.4.2, 3.4.3	128.1205798	38.0930324	$l = 34$, $p = 6$
	BCD	3.1.2, 3.5.1, 3.5.2	127.90263	47.33235083	$l = 52$, $p = 8$, $z = 2$ *
	FS-ISD	3.1.2, 3.6.1, 3.6.2	128.0407971	38.17759926	$l = 34$, $p = 6$
		3.1.2, 3.6.3, 3.6.2	129.3178735	39.89535415	$l = 32$, $p = 6$, $N = 434767677$ *
	FS-ISD (part. G.)	3.1.2, 3.6.1, 3.6.2	128.0565165	38.17759926	$l = 34$, $p = 6$
	BJMM	3.7.1[2], 3.7.2	122.7315584	79.59046103	$\phi = 2$, $l = 130$, $p = 22$, $\Delta_1 = 5$, $M = \{1, 2\}$ *
	BJMM (uniform)	3.7.2	123.9918446	56.03955674	$\phi = 2$, $l = 70$, $p = 12$, $\Delta_1 = 2$, $\Delta_2 = 0$, $N = 27 \cdot 10^{10}$, $M = \{1\}$ *
		3.7.2	121.9116707	76.64491978	$\phi = 2$, $l = 118$, $p = 20$, $\Delta_1 = 4$, $\Delta_2 = 0$, $N = 2967 \cdot 10^{15}$, $M = \{1, 2\}$ *
	BJMM (brute f.)	3.7.1[2], 3.7.2	120.229462	80.379599	$\phi = 3$, $l = 133$, $p = 22$, $\Delta_1 = 5$, $\Delta_2 = 0$, $M = \{1, 2\}$ *
	(no hashmaps)	3.7.1[2], 3.7.2	123.6816913	83.70987285	$\phi = 3$, $l = 144$, $p = 24$, $\Delta_1 = 6$, $\Delta_2 = 1$, $M = \{\}$ *
(6624,5129,117)	FS bound	-	255.2602161	-	-
	BCD bound	-	236.4964218	-	-
	Prange	3.1.4	287.5399269	25.3871917	$r = 7$
	Lee-Brickell	3.1.4, 3.3.1, 3.3.2	272.2767406	34.19778883	$p = 2$, $r = 7$
	Stern	3.4.2, 3.4.3	255.8669247	78.32345851	$l = 74$, $p = 14$
	BCD	3.5.1, 3.5.2	254.1518892	88.04789075	$l = 94$, $p = 16$, $z = 2$ *
	FS-ISD	3.6.1, 3.6.2	255.7395016	78.49337684	$l = 75$, $p = 14$
	FS-ISD (part. G.)	3.6.1, 3.6.2	255.760445	78.49337684	$l = 75$, $p = 14$
	BJMM	3.7.1[2], 3.7.2	242.5861657	99.18163169	$\phi = 2$, $l = 149$, $p = 24$, $\Delta_1 = 4$, $M = \{1\}$ *
		3.7.1[2], 3.7.2	237.420653	144.9364252	$\phi = 2$, $l = 257$, $p = 42$, $\Delta_1 = 9$, $M = \{1, 2\}$ *
	BJMM (uniform)	3.7.2	236.0680295	139.0313538	$\phi = 3$, $l = 259$, $p = 40$, $\Delta_1 = 10$, $\Delta_2 = 1$, $\Delta_3 = 0$, $N = 5 \cdot 10^{23}$, $M = \{1, 2\}$ *
	BJMM (brute f.)	3.7.1[2], 3.7.2	235.3778377	139.0398432	$\phi = 3$, $l = 260$, $p = 40$, $\Delta_1 = 10$, $\Delta_2 = 1$, $M = \{1, 2\}$ *
		3.7.1[2], 3.7.2	244.3565428	85.54302293	$\phi = 3$, $l = 149$, $p = 20$, $\Delta_1 = 6$, $\Delta_2 = 0$, $M = \{1, 2\}$ *
	(no hashmaps)	3.7.1[2], 3.7.2	238.3939671	147.7473114	$\phi = 3$, $l = 286$, $p = 44$, $\Delta_1 = 12$, $\Delta_2 = 1$, $M = \{\}$ *

5 Conclusion

We discussed several information set decoding algorithms and optimized them with regard to the runtime complexity. Out of these, two variants of the BJMM algorithm set new speed records in decoding random linear codes for several parameter sets used in practice. Future work might include a generalization of the BJMM algorithm to $\mathbb{F}_q$ and the development of a model for information set decoding algorithms that respects the number of memory accesses as well.

Appendices

A Markov Chains

This section is a brief introduction into Markov chains, which are used in appendix B. Readers familiar with the concept of Markov chains can easily skip this section.
Markov chains can be used to model experiments, where the result of an experiment can affect the outcome of the next experiment with a certain probability.
More formally one defines a set of *states* $S := \{s_1, s_2, \dots, s_t\}$ corresponding to the possible results of an experiment. A Markov Chain starts in some state s_i and can move to state s_j with probability $p_{i,j}$, whenever an experiment is done. The moves are called *steps* and the probabilities $p_{i,j}$ are called *transition probabilities*. To summarize all of the transition probabilities, we define a transition matrix as follows:

Definition A.0.2 (Transition Matrix)**.** *Denote by $p_{i,j}$ $(1 \leq i, j \leq t)$ the transition probabilities of a Markov chain with a set of states $S := \{s_1, s_2, \dots, s_t\}$. Then the matrix*

$$P := \begin{pmatrix} p_{1,1} & p_{1,2} & \cdots & p_{1,t} \\ \vdots & \vdots & \ddots & \vdots \\ p_{t,1} & p_{t,2} & \cdots & p_{t,t} \end{pmatrix}$$

is called the transition matrix $P \in \mathbb{R}^{t \times t}$ of the Markov chain.

We use a probability vector to define the possible starting states of a Markov chain.

Definition A.0.3 (Probability Vector)**.** *A vector $\vec{u} = (u_1, \dots, u_t)^T \in \mathbb{R}^t$ is called a probability vector, if and only if $u_i \geq 0 \; \forall \, i$ and $\sum_{i=1}^{t} u_i = 1$.*

So we can use $\vec{u} = (u_1, \dots, u_t)^T \in \mathbb{R}^t$ to denote a Markov chain starting in state s_1 with probability u_1, in state s_2 with probability u_2 and so on. Then $\vec{u}^T P$ are the probabilities, with which the Markov chain is in the corresponding states after one step/experiment. More generally we observe:

Theorem A.0.1. *Let $P \in \mathbb{R}^{t \times t}$ be the transition matrix of a Markov chain and denote by $\vec{u} = (u_1, \dots, u_t)^T \in \mathbb{R}^t$ the probability vector representing the starting distribution. Then the probability vector $\vec{u}_n, \vec{u}_n^T := \vec{u}^T P^n$ represents the probability distribution of the Markov chain after n steps.*

Proof. By induction:
$n = 1$: We start in state s_i with probability u_i and move to state s_j with probability $p_{i,j}$. Thus we get a probability of $u_i \cdot p_{i,j}$ to move to state s_j, if we previously were in state s_i and after one step of the Markov chain. Iterating over all possible starting states we get an overall probability of $\sum_{i=1}^{t} u_i \cdot p_{i,j}$ to be in state s_j after one step. This is exactly the operation $\vec{u}^T P$ for column j of the transition matrix P.
$n \to n+1$: $\vec{u}^T P^{n+1} = (\vec{u}^T P^n) P = \vec{v}^T P$, where $\vec{v}^T = \vec{u}^T P^n$ represents the probability distribution of the Markov chain after n steps according to theorem A.0.1. The rest of the reasoning is the same as for $n = 1$. □

This allows us to compute the probability, with which a Markov chain is in a certain state after n steps.

B Analysis of Optimization 3.1.3

Optimization 3.1.3 "Force more existing pivots" in section 3.1 describes a technique to optimize the Gaussian elimination process of algorithm 3.1: Instead of permuting the columns of H uniformly at random in line 13 of algorithm 3.1, we reuse the matrix $\widehat{H} = (Q \mid \mathrm{id}^{[n-k]})$ of a previous iteration (i.e. include the comment between lines 3 and 4 of algorithm 3.1) and swap $1 \leq x < \frac{k(n-k)}{n}$ columns from the identity matrix $\mathrm{id}^{[n-k]}$ in $\widehat{H}$ with x random columns from $Q \in \mathbb{F}_2^{(n-k)\times k}$. The resulting matrix is then used as input of the Gaussian elimination process. Even though this speeds up a single iteration of algorithm 3.1, we get a dependence between the iterations and cannot use $\overline{PR}[success = true]^{-1}$ anymore to model the average number of iterations. The average number of iterations will even increase and possibly make the speedup gained during each iteration useless. Therefore the parameter x needs to be optimized with regard to the concrete algorithm searchALG() and the average number of iterations. This section shows a way to do so.
First we need to properly define the notion "swap $1 \leq x < \frac{k(n-k)}{n}$ columns from the identity matrix $\mathrm{id}^{[n-k]}$ in $\widehat{H}$ with x random columns from $Q \in \mathbb{F}_2^{(n-k)\times k}$": We uniformly select x distinct columns from $\mathrm{id}^{[n-k]}$ and x distinct columns from Q. Then we swap the first selected column from $\mathrm{id}^{[n-k]}$ with the first selected column from Q, the second with the second and so on. It is important to see that once a column from $\mathrm{id}^{[n-k]}$ is swapped out, it cannot be swapped in again. Intuitively this is probably the best and easiest to analyze method as it introduces exactly x new columns into the former identity matrix[16].
To analyse the success probability of an algorithm searchALG(), recall from section 2.2.3 that all information set decoding algorithms guess the entries of the *real* error vector $\vec{\epsilon}$ indexed by the information set I or at least assume a certain distribution of these entries (choosing an information set I is not done by searchALG()). More specifically let us assume that implementations of searchALG() succeed with a certain probability, if $\mathrm{wt}\,(\vec{\epsilon}_I)$ is identical to one or multiple constants defined by searchALG(). Note that the entries of $\vec{\epsilon}_I$ define exactly those columns of Q that add up to a non-zero syndrome in equation (2.2) together with the columns of $\mathrm{id}^{[n-k]}$ defined by $\vec{\epsilon}_{I^*}$. Therefore we call those columns the *significant columns* of $\widehat{H}$. Assuming a certain $\mathrm{wt}\,(\vec{\epsilon}_I) = b$ is then identical to assuming b significant columns within the matrix Q and $w - b$ significant columns within the matrix $\mathrm{id}^{[n-k]}$.
So we assume that the number of significant columns within Q defines the success probability of searchALG() (if the order of the columns is important, we can uniformly permute the columns of Q). This number may change with each iteration/swap operation with a certain probability. This probability only depends on the number of significant columns currently found within Q. Hence we can use a Markov chain to model the probability to have a certain number of significant columns within Q:
We define a set $S := \{s_0, s_1, \ldots, s_w, s_{suc}\}$ of $w + 2$ states. The states of the chain are defined as follows:

- s_i: There are i significant columns within Q, $0 \leq i \leq w$.
- s_{suc}: searchALG() returns $success = true$.

The Markov chain may change its state once per iteration of algorithm 3.1 (corresponding to the swap operation, which happens once per iteration). Figure B.1 shows the structure of the transition matrix of this Markov chain.
Note that only the probabilities $p_{i,suc}$, $0 \leq i \leq w$ depend on the concrete implementation of searchALG(). The probability $p_{i,suc}$ denotes the chance of algorithm searchALG() to

[16] This way of doing column swaps is called "Type 3" in [14].

$$\begin{pmatrix} p_{0,0} & p_{0,1} & \cdots & p_{0,w} & p_{0,suc} \\ p_{1,0} & p_{1,1} & \cdots & p_{1,w} & p_{1,suc} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ p_{w,0} & p_{w,1} & \cdots & p_{w,w} & p_{w,suc} \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure B.1: Transition Matrix

succeed, if exactly i significant columns are part of Q ($\Leftrightarrow \mathrm{wt}\,(\vec{\epsilon}_I) = i$). Table B.1 provides an overview of the probabilities $p_{i,suc}$ for the algorithms presented in this thesis.
To compute the remaining probabilities $p_{i,j}$, $0 \leq i, j \leq w$ let us first have a look at the figures B.2 and B.3: Starting in state s_i with a matrix $\widehat{H}_i$ as in figure B.2 swapping x columns may result in state s_j, $j = i + d$ with a matrix $\widehat{H}_j$ as in figure B.3.

$$\widehat{H}_i = (\overbrace{\underbrace{Q}_{k}}^{\substack{k-i \text{ insignificant,} \\ i \text{ significant}}} | \overbrace{\underbrace{\mathrm{id}^{[n-k]}}_{n-k}}^{\substack{n-k-(w-i) \text{ insign.,} \\ w-i \text{ significant}}}) \in \mathbb{F}_2^{(n-k)\times n}$$

Figure B.2: Parity Check Matrix $\widehat{H}$ in state s_i before swapping x columns from $\mathrm{id}^{[n-k]}$ with x random columns from $Q \in \mathbb{F}_2^{(n-k)\times k}$

$$\widehat{H}_j = (\overbrace{\underbrace{Q'}_{k}}^{\substack{k-(i+d) \text{ insignificant,} \\ i+d \text{ significant}}} | \overbrace{\underbrace{\mathrm{id}'}_{n-k}}^{\substack{n-k-(w-i-d) \text{ insign.,} \\ w-(i+d) \text{ significant}}}) \in \mathbb{F}_2^{(n-k)\times n}$$

Figure B.3: Parity Check Matrix $\widehat{H}$ in state s_j after swapping x columns from $\mathrm{id}^{[n-k]}$ with x random columns from $Q \in \mathbb{F}_2^{(n-k)\times k}$

These figures enable us to see that for $d := j - i$ and $0 \leq i, j \leq w$ the transition probabilities $p_{i,j}$ can be computed as

$$p_{i,j} = (1 - p_{i,suc}) \cdot \binom{n-k}{x}^{-1} \binom{k}{x}^{-1} \sum_{v=0}^{w-i} \binom{w-i}{v} \binom{n-k-w+i}{x-v} \binom{i}{v-d} \binom{k-i}{x-v+d} \tag{B.1}$$

Thereby we define the binomial coefficient $\binom{a}{b}$ to be 0 wherever it was previously undefined ($b > a$, $a < 0$ or $b < 0$).
The following remarks should help to understand equation (B.1):

- $(1 - p_{i,suc})$: We only change from state s_i to state s_j, if we did not already succeed in state s_i. This factor takes account of that fact.
- $\binom{n-k}{x}\binom{k}{x}$: Overall number of column combinations for the swap operation. x columns from $\mathrm{id}^{[n-k]}$ swapped with x columns from $Q \in \mathbb{F}_2^{(n-k)\times k}$.
- v: Number of significant columns that we choose from $\mathrm{id}^{[n-k]} \Rightarrow v = 0 \ldots w - i$ (cf. figure B.2) for the swap operation. Note that we assume $n - k - w \geq x$, so that $\mathrm{id}^{[n-k]}$ always has at least x insignificant columns.
- $\binom{w-i}{v}\binom{n-k-w+i}{x-v}$: We choose v significant columns and $x - v$ insignificant columns from $\mathrm{id}^{[n-k]}$ for the swap operation.

- $\binom{i}{v-d}\binom{k-i}{x-v+d}$: We choose $v-d$ out of the i significant columns of Q and the rest from the insignificant columns of Q for the swap operation. So Q' has $i+v-(v-d) = i+d = j$ significant columns as expected.

The overall way to compute the average number of iterations of algorithm 3.1 in combination with optimization 3.1.3 for a specific parameter x can be described as follows:

1. Define the probabilities $p_{i,suc}$, $0 \leq i \leq w$ according to the implementation of searchALG().

2. Compute the probabilities $p_{i,j}$ using equation (B.1).

3. Assuming a uniform parity check matrix to start with, we can define the probabilities by which we are in state s_i at the beginning of the Markov chain as $Pr[s_i] := \binom{n}{w}^{-1}\binom{k}{i}\binom{n-k}{w-i}$ (recall that we have i significant columns within Q in state s_i). Thus we get a probability vector $\vec{u}^T = (u_0, u_1, \ldots, u_w, u_{suc})^T$ with $u_i := Pr[s_i]$, $0 \leq i \leq w$ and $u_{suc} = 0$ to start with.

4. Then we can use theorem A.0.1 to compute the number of iterations m, for which the probability vector $\vec{u}_m^T = (u_{m,0}, u_{m,1}, \ldots, u_{m,w}, u_{m,suc})^T = \vec{u}^T P^m$ exposes the entry $u_{m,suc} \approx 0.5$. This is the average number of iterations that we may expect when we execute algorithm 3.1 in combination with optimization 3.1.3.
   In practice it is not wise to use theorem A.0.1 directly, but one would rather observe that figure B.1 describes a so-called *absorbing Markov chain* and use various theorems related to that special case.

To find the optimal x, we can iterate over all $1 \leq x < \frac{k(n-k)}{n}$ and compute the average number of iterations m for each of them. Using the average number of iterations, we can then compute the number of bit operations of algorithm 3.1 for each possible x according to equation (3.1) using $time\{randomize()[o_{3.1.1}, o_{3.1.3}]\} = (n-k)\cdot(b_r - b_{opt-3.1.3})$ (cf. equation (3.3)) and $time\{searchALG()\}$ as defined by the concrete algorithm. This allows us to choose the parameter x that results in the fewest binary operations for algorithm 3.1.
Bernstein, Lange and Peters describe this analysis for Stern's algorithm (cf. section 3.4) in [14, 25]. They also provide an implementation of the analysis of this optimization technique for Stern's algorithm, which can be found in [24]. Note however that equation (B.1) is slightly different from the one provided on page 10 of [14].
Our implementation of this optimization technique works for all algorithms discussed in this thesis (cf. section 4).

algorithm	$p_{i,suc}$	**section**
Prange	$= \begin{cases} 1 & \text{for } i = 0 \\ 0 & \text{else} \end{cases}$	3.2
Lee-Brickell	$= \begin{cases} 1 & \text{for } i = p \\ 0 & \text{else} \end{cases}$	3.3
Stern	$= \begin{cases} \frac{\binom{k/2}{p/2}^2 \binom{n-k-l}{w-p}}{\binom{k}{p}\binom{n-k}{w-p}} & \text{for } i = p \\ 0 & \text{else} \end{cases}$	3.4
Stern ("Birthday Speedpup")	$= \begin{cases} \frac{\binom{n-k-l}{w-p}}{\binom{n-k}{w-p}} \left[1 - \left(1 - \binom{p}{p/2}\binom{k}{p/2}^{-2}\right)^{N^2}\right] & \text{for } i = p \\ 0 & \text{else} \end{cases}$	3.4
Ball-Collision Decoding	$= \begin{cases} \frac{\binom{k/2}{p/2}^2 \binom{l/2}{z/2}^2 \binom{n-k-l}{w-p-z}}{\binom{k}{p}\binom{n-k}{w-p}} & \text{for } i = p \\ 0 & \text{else} \end{cases}$	3.5
FS-ISD	$= \begin{cases} \frac{\binom{(k+l)/2}{p/2}\binom{(k-l)/2}{j}\binom{l}{(p/2)-j}\binom{n-k-l}{w-p}}{\binom{k}{i}\binom{n-k}{w-i}} & \text{for } i = j + \frac{p}{2} \\ & (0 \leq j \leq \frac{p}{2}) \\ 0 & \text{else} \end{cases}$	3.6
BJMM	$= \begin{cases} (\mathcal{P}_s)^4 \cdot c(3) & \text{for } i = p \\ 0 & \text{else} \end{cases}$	3.7

Table B.1: Probabilites $p_{i,suc}$ for several algorithms. Recall that $p_{i,suc}$ is a conditional probability (if we are in state i, we already have an error vector $\vec{\epsilon}$ with $\text{wt}(\vec{\epsilon}_I) = i$).

Remark B.0.8 ($p_{i,suc}$ of FS-ISD)**.** *If we have $wt(\vec{\epsilon}_I) = i = j + p/2$, $0 \leq j \leq p/2$ in state s_i, there exist exactly $\binom{k}{i}\binom{n-k}{w-i}$ combinations for such an error vector $\vec{\epsilon}$. To succeed with the FS-ISD algorithm, we need a distribution of the error vector as in figure 3.7: We always need $p/2$ 1's in the upper $(k+l)/2$ bits. j more 1's may be distributed among the following $(k-l)/2$ bits resulting in exactly $i = (p/2) + j$ 1's in the first k bits (as we are in state s_i). However according to figure 3.7 we also need an overall weight of $p/2$ on the second block of size $(k+l)/2$, so that we need exactly $(p/2) - j$ 1's in the l-block. The last factor of the equation in table B.1 is self-explanatory.*

Remark B.0.9 ($p_{i,suc}$ of BJMM)**.** *As described in section 3.7 the BJMM algorithm involves a partial gaussian elimination (exactly l less columns) rather than a complete gaussian elimination. Therefore the entire model presented in this section needs to be adapted to work with a matrix $Q' \in \mathbb{F}_2^{(n-k)\times(k+l)}$ instead of $Q \in \mathbb{F}_2^{(n-k)\times k}$ and the matrix $S := (0^{[(n-k-l)\times l]} \mid id^{[n-k-l]})^T$ instead of $id^{[n-k]}$. To define the notion of significant columns we can use equation (3.30) rather than equation (2.2).*
The formula for the probabilities $p_{i,suc}$ of the BJMM algorithm mentioned in table B.1 only holds in that model, whereas the others use the model described in this section. However in the case of FS-ISD it would make sense to use the adapted model as well.

C searchFS() using findColl()

The following algorithm is an example on how to apply the function findColl() (algorithm 3.8) from section 3.7 in the context of the FS-ISD algorithm. The original searchFS() function is defined as algorithm 3.7 in section 3.6.

Algorithm C.1: searchFS() using findColl()

Input: parity check matrix $\widehat{H} = (Q \mid L \mid S) \in \mathbb{F}_2^{(n-k)\times n}$, syndrome $\vec{\varsigma} \in \mathbb{F}_2^{n-k}$, $w = \mathrm{wt}(\vec{e})$, algorithmic parameter $0 \leq p \leq w$, algorithmic parameter $0 \leq l \leq n-k-w+p$

Output: success indicator (true/false), error vector $\vec{\epsilon} \in \mathbb{F}_2^n$

/* create base lists */

1 $\mathcal{L}_1 := \{\vec{l}_1 = \texttt{prepend}\ (\underline{\vec{\epsilon}_{I_1}}, \vec{0}),\ \underline{\vec{\epsilon}_{I_1}} \in \mathbb{F}_2^{\lceil (k+l)/2 \rceil}, \vec{0} \in \mathbb{F}_2^{\lfloor (k+l)/2 \rfloor} \mid \mathrm{wt}(\underline{\vec{\epsilon}_{I_1}}) = \frac{p}{2}\}$

2 $\mathcal{L}_2 := \{\vec{l}_2 = \texttt{prepend}\ (\vec{0}, \underline{\vec{\epsilon}_{I_2}}),\ \underline{\vec{\epsilon}_{I_2}} \in \mathbb{F}_2^{\lfloor (k+l)/2 \rfloor}, \vec{0} \in \mathbb{F}_2^{\lceil (k+l)/2 \rceil} \mid \mathrm{wt}(\underline{\vec{\epsilon}_{I_2}}) = \frac{p}{2}\}$

/* find collisions */

3 $\mathcal{L} \leftarrow \texttt{findColl}\ (\mathcal{L}_1, \mathcal{L}_2, Q', \vec{\varsigma}_{[l]}, l, p)$

/* use collisions */

4 **foreach** $\underline{\vec{\epsilon}_I} \in \mathcal{L}$ **do**

5 $\quad S\underline{\vec{\epsilon}_{I^*}} := \vec{\varsigma} + Q'\underline{\vec{\epsilon}_I}$

6 $\quad$ **if** $wt(S\underline{\vec{\epsilon}_{I^*}}) = w - p$ **then**

7 $\quad\quad \underline{\vec{\epsilon}_{I^*}} \leftarrow \texttt{remove}\,(S\underline{\vec{\epsilon}_{I^*}}, l)$

8 $\quad\quad \vec{\epsilon} \leftarrow \texttt{prepend}\,(\underline{\vec{\epsilon}_I}, \underline{\vec{\epsilon}_{I^*}})$

9 $\quad\quad$ **return** $(true, \vec{\epsilon})$

10 $\quad$ **end**

11 **end**

12 **return** $(false, \vec{0})$

Thereby the list $\mathcal{L}_1$ contains all possible vectors $\underline{\vec{\epsilon}_{I_1}}$ as defined in section 3.6, just with zeroes appended, whereas the list $\mathcal{L}_2$ contains the vectors $\underline{\vec{\epsilon}_{I_2}}$ from section 3.6 with zeroes prepended. The padding with zeroes is necessary as we do not use $\underline{\vec{\epsilon}_I} = \mathrm{prepend}(\underline{\vec{\epsilon}_{I_1}}, \underline{\vec{\epsilon}_{I_2}})$ anymore, but more generally write $\underline{\vec{\epsilon}_I}$ as a sum of two vectors within algorithm 3.8.

D List Experiments

Remark 3.5.2 and 3.6.2 explain the problem that the lists used in the context of the Ball-Collision Decoding (section 3.5) and the FS-ISD algorithm (section 3.6) are not entirely uniform lists, but yet we assume the number of collisions between those lists to behave as if they were. As proven in remark 3.5.2 this is reasonable with regard to the expected number of collisions. However we were unsure with regard to the variance of the number of collisions. Therefore we performed the following two experiments:

BCD lists: We first sampled $\binom{k/2}{p/2}$ elements with length l uniformly at random (arbitrary weight) for two lists $\mathcal{L}_1$ and $\mathcal{L}_2$. This is meant to be equivalent to the "choices" of $\vec{\varsigma}_{[l]} + (Q_1\vec{\epsilon}_{I_1})_{[l]}$ and $(Q_2\vec{\epsilon}_{I_2})_{[l]}$ in line 4 and 10 of algorithm 3.6. Then we iterated over all $\binom{l/2}{z/2}$ possibilities of length $l/2$ elements with weight $z/2$ and added *each* of those possibilities to the upper half of *each* entry of $\mathcal{L}_1$ and the lower half of *each* entry of $\mathcal{L}_2$, i.e. each of the former $\binom{k/2}{p/2}$ many list elements was expanded by a factor of $\binom{l/2}{z/2}$. So basically we simulated the $\binom{l/2}{z/2}$ many additions of $\vec{\epsilon}_1$ and $\vec{\epsilon}_2$ in line 4 and 10 of algorithm 3.6. Afterwards we counted the number of collisions between the two lists.

FS-ISD lists: To simulate the lists $\mathcal{L}_1$ and $\mathcal{L}_2$ occurring during the execution of the FS-ISD algorithm (cf. algorithm 3.7) we sampled $\binom{(k+l)/2}{p/2}$ elements with length l uniformly at random (arbitrary weight) for list $\mathcal{L}_1$. The creation of list $\mathcal{L}_2$ is slightly more complicated: We needed to generate all $\binom{(k+l)/2}{p/2}$ combinations of vectors with weight $p/2$ and length $(k+l)/2$, i.e. iterate over all $\underline{\vec{\epsilon}_{I_2}}$ as in line 2 of algorithm 3.7. For each combination where the first $(k-l)/2$ bits were different, we sampled a uniform element $\vec{a} \in_r \mathbb{F}_2^l$; for the combinations where the first bits were the same, the same sample $\vec{a}$ was used. Note that the first $(k-l)/2$ bits correspond to the vector $\vec{\gamma} \in \mathbb{F}_2^{(k-l)/2}$ in equation (3.34) and the sample $\vec{a}$ is meant to correspond to the result of the computation $(Q_2''\vec{\gamma})_{[l]}$. Then we were able to simulate the entries of $\mathcal{L}_2$ as $\vec{a} + (\vec{\epsilon}_{I^*})_{[l]}$. Recall that the vector $(\vec{\epsilon}_{I^*})_{[l]}$ consists of the last l bits of $\underline{\vec{\epsilon}_{I_2}}$. Finally we could count the number of collisions between the two lists.

Repeating these processes allowed us to compare the expected number of collisions and the variance of this number determined in the experiments with our theoretical ideas (entirely uniform lists).
Note that the variance of the number of collisions between entries of length l of two *uniform* lists $\mathcal{L}_1$ and $\mathcal{L}_2$ with $|\mathcal{L}_1| = |\mathcal{L}_2| =: N$ can be defined as

$$\begin{aligned}\sigma^2 &:= \sum_{X=0}^{N^2} (X-\mu)^2 \cdot B_{N^2,2^{-l}}(X) \\ &= N^2 \cdot 2^{-l}(1-2^{-l}) \end{aligned} \tag{D.1}$$

Thereby we use the random variable X to denote the number of collisions, $\mu := E[X] = N^2 \cdot 2^{-l}$ as the expected number of collisions and $B_{a,b}(x) := \binom{a}{x} \cdot (1-b)^{a-x} \cdot b^x$ as the binomial distribution function. The formula can be verified by observing the fact that $Pr[X = c] = B_{N^2,2^{-l}}(c)$ for every possible number of collisions $0 \leq c \leq N^2$. It is well known that the variance of a binomial distribution can be simplified as in equation (D.1). In contrast we compute the variance observed in the experiments as

$$\sigma_{exp}^2 := \frac{1}{v} \cdot \sum_{i=1}^{v} (x(i) - \mu_{exp})^2$$

In this context v denotes the number of experiments performed and $x(i)$ is the number of collisions seen in experiment i. μ_{exp} is the average number of collisions seen in the experiments.
The implementation of the experiments in C++ in combination with NTL and GMP [33, 34] was part of the work for this thesis.
The results of our experiments for $v = 1000$ are displayed in table D.1: The experimentally determined average number of collisions μ_{exp} behaves as anticipated in theory (cf. μ). The variance σ^2_{exp} is sometimes smaller or larger than the corresponding theoretical result though (cf. σ^2). However the difference is rather small.
All in all we may conclude that it seems reasonable to model the number of collisions between the list elements occurring in the Ball-Collision Decoding (BCD) and the FS-ISD algorithm as the number of collisions occurring between uniform lists. Nevertheless the non-uniformity of the real lists should be kept in mind.

algorithm	k	l	p	z	μ_{exp}	μ	σ^2_{exp}	σ^2
BCD	100	10	2	2	61	61	60	60
BCD	100	10	4	2	36631	36636	34708	36600
BCD	100	28	4	2	1	1	1	1
BCD	100	38	4	2	0	0	0	0
BCD	50	10	4	2	2195	2197	1905	2195
BCD	50	16	6	2	5164	5166	5359	5165
BCD	524	10	2	2	1676	1675	1724	1674
BCD	524	10	4	2	28540455	28540636	28740334	28512765
FS-ISD	100	10	2	-	2	2	3	2
FS-ISD	100	10	4	-	2151	2153	2096	2151
FS-ISD	200	20	4	-	33	34	33	34
FS-ISD	50	10	4	-	185	184	176	184
FS-ISD	50	12	6	-	4929	4932	4975	4931
FS-ISD	524	10	2	-	69	69	71	69

Table D.1: Results of the list experiments.

E BJMM with an arbitrary number of layers

This section shortly describes the formulas required to model the runtime, memory consumption and success probability of the BJMM algorithm (cf. section 3.7) for an arbitrary number of layers. We use ϕ to denote the number of layers above layer 0 (cf. figure 3.9), i.e. $\phi + 1$ is the overall number of layers.
We also take the following possibilities into account: The 2^ϕ base lists on layer ϕ of size $|\mathcal{B}|$ can be generated

1. in a brute-force approach.
2. in a meet-in-the-middle approach.
3. by uniform sampling.

The first and third method are described in optimization 3.7.3; the second method is the one originally found in [21] (also cf. section 3.7).
Let us first define the variables p_j and r_j for an arbitrary number of layers ($p_0 := p$, $r_0 = l$):

$$\begin{aligned} p_j &:= \frac{p_{j-1}}{2} + \Delta_j \quad (1 \le j \le \phi - 1) \\ r_j &:\approx \log_2(R_j) \qquad (1 \le j \le \phi - 1) \\ R_j &:= \binom{p_{j-1}}{p_{j-1}/2}\binom{k+l-p_{j-1}}{\Delta_j} \quad (1 \le j \le \phi) \end{aligned}$$

For the base lists ($j = \phi$) we have $r_\phi := 0$ and the elements have a weight of

$$p_\phi := \begin{cases} \frac{p_{\phi-1}}{2} + \Delta_\phi & \text{(uniform sampling)} \\ \frac{p_{\phi-1}}{2} & \text{(brute force, meet-in-the-middle)} \end{cases}$$

Other important equations that need to be generalized include the probability $\mathcal{P}_s$ (cf. equation (3.41))

$$\mathcal{P}_s := \begin{cases} 1 & \text{(brute force)} \\ \binom{(k+l)/2}{p_\phi}^2 \binom{k+l}{p_{\phi-1}}^{-1} & \text{(meet-in-the-middle)} \\ \mathcal{P}_N := 1 - \left(1 - R_\phi \binom{k+l}{p_\phi}^{-2}\right)^{N^2} & \text{(uniform sampling)} \end{cases}$$

as well as the base list sizes

$$|\mathcal{B}| := \begin{cases} \binom{k+l}{p_\phi} & \text{(brute force)} \\ \binom{(k+l)/2}{p_\phi} & \text{(meet-in-the-middle)} \\ N & \text{(uniform sampling)} \end{cases}$$

Equation (3.47) can still be used to describe the list sizes on layer j, if the function $f(j)$ is more generally defined as $f(j) := 2^{|\phi-1-j|}$.
All in all we get

$$time\{searchBJMM()\} = \left(\sum_{j=0}^{\phi-1} 2^j \cdot time\left\{findColl(|\mathcal{L}^{(j+1)}|)\right\}\right) + t_{16} \tag{E.1}$$

$$mem\{searchBJMM()\} = M(\phi) + \mathcal{O}\left(\max_{j=0,\dots,\phi-1}\left\{|\mathcal{L}^{(j+1)}| \cdot (r_j - r_{j+1} + k + l)\right\}\right) \tag{E.2}$$

$$\overline{PR}_{BJMM}[success = true] = \frac{\binom{k+l}{p}\binom{n-k-l}{w-p}}{\binom{n}{w}} \cdot (\mathcal{P}_s)^{f(0)} \cdot c(\phi) \tag{E.3}$$

The factor 2^j in equation (E.1) comes from the fact that we have 2^j lists on layer j and thus need to call the function findColl() 2^j many times with lists of the layer above (size $|\mathcal{L}^{(j+1)}|$) as input. t_{16} models the time spent with computations on layer 0 (line 16 of algorithm 3.9, cf. lemma 3.7.1); $M(\phi)$ is defined in equation (3.51), $c(\phi)$ in equation (3.52). The rest should be self-explanatory in the context of the explanations from section 3.7.
Note that the FS-ISD algorithm with a partial gaussian elimination (contrary to section 3.6) is the special case $\phi = 1$ of the generalized BJMM algorithm; the runtime formula of FS-ISD (cf. equation (3.35)) is just slightly better than equation (E.1) for $\phi = 1$, because for FS-ISD the complexity of the binary addition in line 4 of algorithm 3.8 can be ignored as it is just a simple prepend-operation. Equation (E.2) for $\phi = 1$ is worse than equation (3.36) as well, partially because the original FS-ISD algorithm does not need to store the list $\mathcal{L}$ on layer 0. Anyway equation (E.2) is more of an upper bound for the overall memory consumption of the BJMM algorithm.

References

[1] Claude E. Shannon. A mathematical theory of communication. Bell Systems Technical Journal, 27:379-423, 623-656, 1948.

[2] F.J. MacWilliams, N.J.A. Sloane. The Theory of Error Correcting Codes. North-Holland, 1977.

[3] R. J. McEliece. A public-key cryptosystem based on algebraic coding theory. In Jet Propulsion Laboratory DSN Progress Report 42-44, pages 114-116, 1978.

[4] E. Berlekamp, R. McEliece, H. van Tilborg. On the inherent intractability of certain coding problems. IEEE Transactions on Information Theory, 24(3):384-386, 1978.

[5] J.-C. Faugère, V. Gauthier, A. Otmani, L. Perret, J.-P. Tillich. A distinguisher for high rate McEliece cryptosystems. Cryptology ePrint Archive, Report 2010/331. 2010.

[6] H. Dinh, C. Moore, A. Russell. McEliece and Niederreiter cryptosystems that resist quantum fourier sampling attacks. In Proceedings of CRYPTO'11, pages 761-779. Springer, 2011.

[7] Daniel J. Bernstein. Grover vs. McEliece. In Proceedings of PQCrypto'10, pages 73-81. Springer, 2010.

[8] Valery D. Goppa. A new class of linear error correcting codes (Russian). Problemy Peredachi Informatsii, 6(3):24-30, 1970.

[9] V. Z. Arlazarov, E. A. Dinic, M. A. Kronrod, I. A. Faradžev. On economical construction of the transitive closure of a directed graph. Soviet Mathematics Doklady, 11(5):1209-1210. 1970.

[10] Nicholas J. Patterson. The algebraic decoding of Goppa codes. IEEE Transactions on Information Theory, IT-21:203-207, 1975.

[11] A. Canteaut, N. Sendrier. Cryptanalysis of the original McEliece cryptosystem. ASIACRYPT'98, volume 1514 of Lecture Notes in Computer Science, pages 187-199. Springer, 1998.

[12] A. Canteaut, F. Chabaud. A new algorithm for finding minimum-weight words in a linear code: application to McEliece's cryptosystem and to narrow-sense BCH codes of length 511. IEEE Transactions on Information Theory, 44(1):367-378. 1998.

[13] F. Chabaud. On the security of some cryptosystems based on error-correcting codes. EUROCRYPT'94, LNCS 950. Springer, 1995.

[14] Daniel J. Bernstein, T. Lange, C. Peters. Attacking and defending the McEliece cryptosystem. PQCrypto 2008, volume 5299 of Lecture Notes in Computer Science, pages 31-46. Springer, 2008.

[15] Daniel J. Bernstein, T. Lange, C. Peters, H. C. A. van Tilborg. Explicit bounds for generic decoding algorithms for code-based cryptography. In Pre-proceedings of WCC'09, pages 168-180. Bergen, 2009.

[16] D. Engelbert, R. Overbeck, A. Schmidt. A summary of McEliece-type cryptosystems and their security. Cryptology ePrint Archive: Report 2006/162, 2006.

[17] K. Kobara, H. Imai. Semantically secure McEliece public-key cryptosystems-conversions for McEliece PKC. PKC'01, volume 1992 of Lecture Notes in Computer Science, pages 19-35. Springer, 2001.

[18] Daniel J. Bernstein, T. Lange, C. Peters. Wild McEliece. SAC'10, volume 6544 of Lecture Notes in Computer Science, pages 143-158. Springer, 2011.

[19] Daniel J. Bernstein, T. Lange, C. Peters. Smaller decoding exponents: ball-collision decoding, 2010 (last revised: 2011). http://eprint.iacr.org/2010/585.

[20] A. May, A. Meurer, E. Thomae. Decoding Random Linear Codes in $\tilde{\mathcal{O}}(2^{0.054n})$. ASIACRYPT'11, pages 107-124. Springer, 2011.

[21] A. Becker, A. Joux, A. May, A. Meurer. Decoding Random Binary Linear Codes in $2^{n/20}$: How $1 + 1 = 0$ Improves Information Set Decoding. EUROCRYPT'12, volume 7237 of Lecture Notes in Computer Science, pages 520-536. Springer, 2012. Mathematica code: http://cits.rub.de/imperia/md/content/opt.zip (last seen: 05.05.2012).

[22] Jacques Stern. A method for finding codewords of small weight. Volume 388 of Lecture Notes in Computer Science, pages 106-113. Springer, 1989.

[23] Christiane Peters. Information-set decoding for linear codes over $\mathbb{F}_q$. Volume 6061 of Lecture Notes in Computer Science, pages 81-94. Springer, 2010.

[24] Christiane Peters. Iteration and operation count for information-set decoding over $\mathbb{F}_q$. Overview and program. Homepage of C. Peters, http://www2.mat.dtu.dk/people/C.Peters/isdfq.html (last seen: 05.05.2012). Provided in the context of [23, 14].

[25] Christiane Peters. Curves, Codes, and Cryptography. PhD Thesis, Technische Universiteit Eindhoven. 2011.

[26] M. Finiasz, N. Sendrier. Security bounds for the design of code-based cryptosystems. ASIACRYPT'09, volume 5912 of Lecture Notes in Computer Science, pages 88-105. Springer, 2009.

[27] Eugene Prange. The use of information sets in decoding cyclic codes. IRE Transactions on Information Theory, 8(5):5-9. 1962.

[28] P. J. Lee, E. F. Brickell. An observation on the security of McEliece's public-key cryptosystem. EUROCRYPT'88, volume 330 of Lecture Notes in Computer Science, pages 275-280. Springer, 1988.

[29] V. Guruswami. Introduction to Coding Theory. Lecture Notes, 2010.

[30] Herbert Robbins. A remark on Stirling's formula. American Mathematical Monthly, 62(1):26-29. 1955.

[31] J. S. Leon. A probabilistic algorithm for computing minimum weights of large error-correcting codes. IEEE Transactions on Information Theory, 34(5):1354-1359. 1988.

[32] A. Becker, J.-S. Coron, A. Joux. Improved generic algorithms for hard knapsacks. EUROCRYPT'11, volume 6632 of Lecture Notes in Computer Science, pages 364-385. Springer, 2011.

[33] Victor Shoup. NTL C++ library. Version 5.5.2 (2009.08.14). http://www.shoup.net/ntl/doc/tour.html

[34] GNU Multiple Precision Arithmetic Library. Version 5.0.4 (2012.02.10). http://gmplib.org/