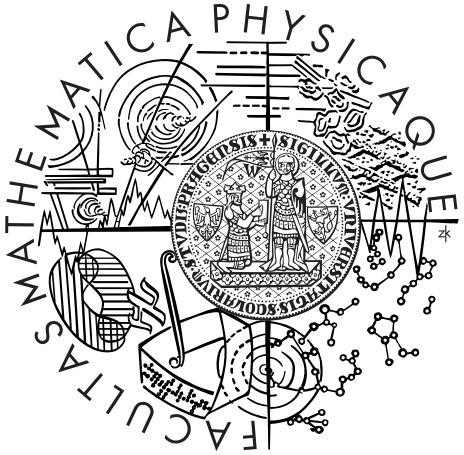
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Miroslav Kratochvíl

Implementation of cryptosystem based on error-correcting codes

Katedra softwarového inženýrství

Supervisor of the bachelor thesis: RNDr. Jakub Yaghob, Ph.D.
Study programme: Informatika
Specialization: Obecná informatika

Prague 2013

Dedicated to everyone who was not afraid to sacrifice late night and early morning to run out and contribute to chaos.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Implementation of cryptosystem based on error-correcting codes

Autor: Miroslav Kratochvíl

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Jakub Yaghob, Ph.D.

Abstrakt: Cílem práce je prozkoumat možnosti implementace uživatelsky přívětivého a praktického kryptosystému založeného na algoritmech neohrozitelných kvantovými počítači. Předpokládá se co nejširší nasazení kryptografie založené na samoopravných kódech (McEliece kryptosystém) a zachování podobnosti s existujícími kryptografickými aplikacemi (GnuPG).

Klíčová slova: šifrování, digitální podpis, samoopravné kódy, postkvantová kryptografie

Title: Implementation of cryptosystem based on error-correcting codes

Author: Miroslav Kratochvíl

Department: Katedra softwarového inženýrství

Supervisor: RNDr. Jakub Yaghob, Ph.D.

Abstract: The goal of this thesis is to present the problem of implementation of user-friendly and practical cryptosystem based on algorithms that are intractable by quantum computing. Resulting software is expected to use code-based cryptography (McEliece-based cryptosystems) to the highest possible extent while maintaining similarity with already-existing cryptographical applications (GnuPG).

Keywords: encryption, digital signatures, error-correcting codes, post-quantum cryptography

Contents

Introduction

Modern cryptographic algorithms and protocols have become an undisposable part of virtually all recent communication technologies. Asymmetric cryptography, widely used in encryption and authentication schemes, is in most cases based on some of the hard RSA-related number-theoretic problems like integer factorization or discrete logarithm computation. This thesis aims to explore and implement asymmetric cryptography schemes that are constructed using different trapdoor problems — main target is the cryptography based on assumptions from *error-correcting codes* theory, namely the intractability of *syndrome decoding* problem. It will be shown that schemes derived from this theory are not only a viable alternative for modern schemes, but outperform them in many interesting ways.

Motivation and goals

As stated above, practically all modern asymmetric cryptography is based on some assumptions from number theory. In 1997, Shor [Sh97] has presented an algorithm that performs integer factorization on quantum computers in polynomial time, and while recent development in the field of quantum computer construction has only allowed mathematicians to run the computation on very small numbers (as of May 2013 the largest number ever factored using Shor's algorithm was 21), there is no reason to believe that the quantum computers will not eventually grow big enough to allow actual RSA modulus factorization.

Main goal of this thesis is therefore to implement a software package that would allow anyone to securely encrypt and authenticate data using algorithms that cannot be broken by possible advances in quantum computing. Software is expected to take an user-friendly form of UNIX shell tool, possibly similar to [GPG] or related packages, and make use of code-based cryptography (based on intractability of syndrome decoding problem, as stated for example in [LGS12]) to the highest practical extent.

Note that there are also several other branches of post-quantum[1] cryptosystems:

- Hash-based cryptography is used in this thesis as a replacement for code-based signature scheme, which is found to be too slow for practical usage. For details, see Chapter 2.

[1] this term broadly encompasses classical-computing-based cryptography without algorithms that could be broken by quantum computers

- Lattice-based cryptography [Lat] is a good candidate for post-quantum cryptosystems, but the algorithm for encryption, called NTRUEncrypt, is patented (see [NPat]) and currently unavailable for general limitless usage. Related digital signature algorithms (notably NTRUSign) are known to contain serious security weaknesses [NSig].

- Multivariate quadratic equation (MQ-based) algorithms were considered as candidates for post-quantum digital signatures, but most of the attempts to produce a secure signature scheme have been broken as well — for details, see [Cz12].

Because of those problems, this thesis does not aim to explore nor implement any possible alternatives with lattice- and MQ-based algorithms.

Second goal of the thesis is to document author's exploration of the topic, especially to present a condensed but practical bottom-up construction of all mathematical necessities needed to work with code-based cryptography.[2]

Due to the nature of cryptographic research needed, *questions of actual security of the software, especially cipher trapdoor usage, security and effectivity of padding schemes, choice of hash algorithms, key sizes, related cryptosystem parameters and communication protocols used, are left open,* because the difficulty of making any serious assumptions in that direction effectively takes them out of the scope of this thesis. From the other side, if there is no significant mistake in the implementation, author is currently not aware of any practical attack or security vulnerability that could cause failure of used security guarantees.

Related and similar work

Similar software

While there are lots of quality scientific publications about the topic, to the best of author's knowledge there is currently no software package that would provide general post-quantum code- or hash-based cryptography in an user-friendly way.

Only current software implementations of McEliece-like cryptosystems known to the author are:

- The "original" archived Usenet message from anonymous `prometheus@uucp` dated 1991, available at [Prom91] that contains implementation of McEliece with fixed parameters, providing (considering also recent decoding improvements) the attack security of around 2^{58}. The source code is not supplied in a very programmer-friendly form[3], and is quite unthinkable to be used in any modern software.

- FlexiProvider library [FP] provides Java implementation of McEliece and Niederreiter cryptosystems including many used padding schemes and conventions for the programs compatible with FlexiProvider API.

[2] Thesis assumes that the reader has some previous experience with linear and abstract algebra.

[3] It is actually obfuscated into a *rectangle* of symbols.

- HyMES [HM], a project of INRIA SECRET team and the usual McEliece scientific testing platform, is available online. It is not meant for any practical use other than testing and benchmarking.

To the best of author's knowledge, there is currently no publicly available implementation of Quasi-Dyadic McEliece.

Although cryptographic schemes based on same ideas as hash-based digital signatures implemented in this thesis are already used in several software protocols (notably BitCoin, DirectConnect file-sharing and Gnutella, see [TTH]), author was not able to find any software that would allow users to directly create actual hash-based digital signatures.

Related research

Due to the simplicity of implementing the hash-based signature schemes, there has not been much presented research on the topic — the paper [NSW05] that describes the digital signature scheme used by this thesis already provides more than sufficient insight into implementation details.

On the other hand, thanks to the challenges involved, implementation of McEliece-based systems has attracted many researchers:

- PhD thesis of Bhaskar Biswas [Bis10] is an excellent resource that is conceptually similar to this thesis — author presents in-depth description of construction of the HyMES software (see above).

- Many authors focus on performance of various parts of McEliece cryptosystems, especially for working on memory-constrained devices. Best examples of those are [Hey09] that implements very efficient McEliece on FPGA-like devices, and [Str10] that primarily addresses small, smartcard-like embedded devices.

- Bachelor thesis of Gerhard Hoffman [Hof11] is the only work that explores implementation of the quasi-dyadic McEliece scheme (also used by this thesis). The implementation is done atop HyMES, but is not publicly accessible.

Author is not aware about any research considering implementations of used padding schemes (Fujisaki-Okamoto padding, see Chapter 1).

Acknowledgments

I am very grateful for help from Rafael Misoczki and Gerhard Hoffmann — the quasi-dyadic variant of McEliece cryptosystem probably could not be implemented without their explanations.

Chapter 1

Code-based cryptography

The whole notion of code-based cryptography was established by Robert J. McEliece in 1978 by developing the original McEliece asymmetric cryptosystem [McE78]. Because of several disadvantages when compared to RSA-like systems, it received quite little attention until early 2000's when Shor's algorithm was developed.

This chapter is meant as an introduction to the mathematical and structural background of the cryptosystem, overview of recent development, common applications, derived cryptosystems and related algorithms.

Where not stated otherwise, the reference for facts and details of this chapter is the excellent book [MWS77].

1.1 Error-correcting codes

Generally speaking, the whole practice of error correction is a method to encode a *word* into a *codeword*, usually by changing structure and *adding redundancy*, and to *decode* the possibly damaged codeword (sometimes called *received word* — transmitting information over error-producing ("noisy") channels is the original purpose of error-correcting codes) back into the original word, even if some parts of the codeword got changed during transfer, or are missing.

The *code* is mathematically defined as a set of all possible codewords, usually a subset of some vector space. Although it is possible to define a code over any vector space, for practical purposes only the binary codes over vector spaces over $\mathbf{GF}(2)$ finite field will be considered and defined here:

Definition 1. $\mathbf{GF}(2)$ *is a finite field of elements* $\{0, 1\}$ *intuitively isomorphic to* $\mathbb{Z}_2$.

c is called a word of length n if $c \in \mathbf{GF}(2)^n$.

Code $\mathcal{C}$ *of length n is a subset of all possible words of length n, e.g.* $\mathcal{C} \subseteq \mathbf{GF}(2)^n$.

c is a codeword *of code* $\mathcal{C}$ *if* $c \in \mathcal{C}$.

When transferring information, the smallest possible (atomic) error flips a single "bit" of the codeword vector. For measuring codeword qualities and error counts it is useful to define the concept of *Hamming distance*.

Definition 2. Hamming weight *of word c is equal to the number of non-zero components of c, and written as* $|c|$.

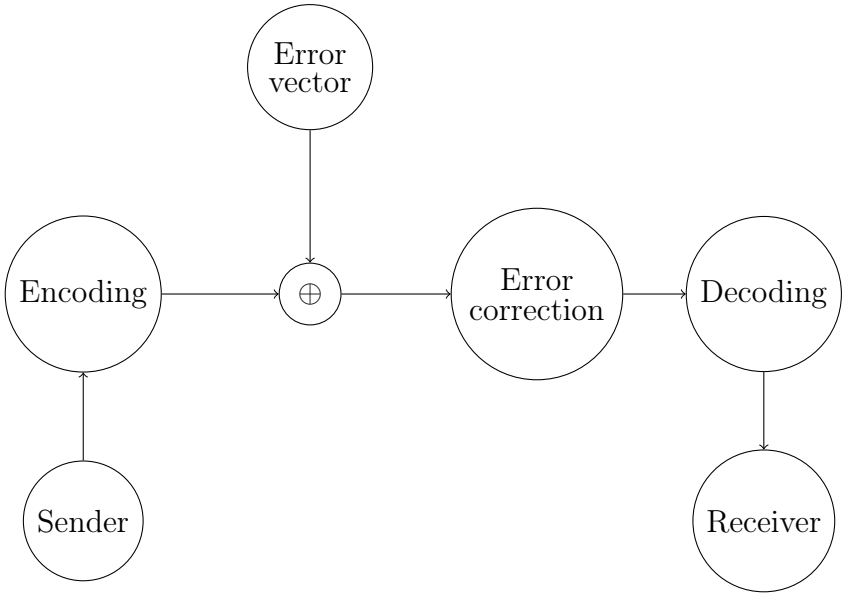


Figure 1.1: Transfer of a word over a noisy channel, with bit errors and error correction

Words a and b of equal length have Hamming distance d if $d = |a - b|$.[1]

Hamming distances provide an useful way to look on codes. Observe that if c is a codeword and r is a received word that was changed by e single-bit-flipping errors, Hamming distance of c and r is at most e.

Assume that there exists some upper limit n on how many bits can get flipped in the codeword during transfer. If a received word is closer to some codeword than n (e.g. they have lower or equal Hamming distance), it is possible that this codeword was also the original transmitted codeword. Moreover, if the distance is larger than n, the codeword could not have been transmitted in the first place, because it would need to receive more than n errors.

The code can be arranged so that for every possible received word there is *only one* codeword with the distance less or equal to n, for the purpose that the original transmitted codeword can be found deterministically (because there is only single possibility). Such code is then called a n-error correcting code. Definition of the structure is, for simplicity, done using codeword distances:

Definition 3. Minimum distance *of code $\mathcal{C}$ is defined as* $\min\{|a - b|; a, b \in \mathcal{C} \wedge a \neq b\}$.

Code is called n-error-correcting *if it has minimum distance at least* $2n + 1$.

This gives a simple method to reliably transfer messages over a noisy channel that creates at most n errors: Define a bijective mapping between all possible messages and the codewords of n-error-correcting code. Map the message to a codeword and transmit it. On a receiving side, find a codeword closer than n to the received word[2] and map it back to the message.

[1] Used notation of distance is intentionally similar to metric distances. In binary case, one can equivalently write $|a + b|$, $|b - a|$, or expressing it most computationally precisely $|a \oplus b|$.

[2] With slight variation, this simple approach is called "Nearest neighbor algorithm". For a received word, decoder evaluates all possible words within given Hamming distance (such words form an imaginary "ball" around the received word), and returns the one which is a codeword.

1.1.1 Linear codes

Definition 4. *Code is called* linear *if any linear combination of its codewords is also a codeword.*

Linear codes have many useful properties. Because the codewords are closed on linear combinations, the code is a proper vector subspace, and therefore has *dimension* (usually marked k) and can be represented as a span of some *basis* that contains k codewords. Such code can encode 2^k messages, as bijective mapping between messages and codewords is very easy to construct using linear algebra: Basis written as a row-matrix forms a *generator matrix* of k rows and n (code length) columns, and is converted to an echelon form. Messages are represented as vectors of $\mathbf{GF}(2)^k$, and are transformed to codewords using simple matrix-vector multiplication. Conversion back to the message is even easier — because the generator matrix was in the echelon form, first k bits of a (corrected) codeword are the original message.

Linear code of dimension k and length n is usually marked as a (n, k)-code. Similarly, if such code has a minimum distance d, it is labeled as a $[n, k, d]$-code.[3]

For example, Hamming codes are linear — the ubiquitous 7-bit one-error-correcting Hamming code can be represented as linear $[7, 4, 3]$-code with following generator matrix:

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

Notice the echelon form — when computing a codeword from message word m as $c = m^T \cdot G$, first four bits of c will be the same as original message, thus reconstruction of message from received and repaired codeword is trivial.

For example, if one wants to encode message 0101, the codeword after multiplication is 0101110. Left four bits of a codeword are called *data bits*, right three bits are called *redundancy* or *checksum bits*.

Definition 5. *Call H a* parity-check matrix *of (n, k)-code $\mathcal{C}$ if*

$$\forall c : c \in \mathcal{C} \iff H \cdot c = 0^{n-k}$$

e.g. if the null space of H is exactly $\mathcal{C}$.

Observe that the parity-check matrix has exactly $n - k$ rows and n columns. Every code possesses many parity-check matrices related via basis change, as changing the basis does not modify null space. Code generated by the parity-check matrix is usually called *dual code*.

Using some linear algebra, one can easily construct a convenient echelon-form parity-check matrix from generator matrix and vice versa:

Theorem 1. *$G = (I_k|A)$ is a generator matrix of a (n, k)-code if and only if $H = (A^T|I_{n-k})$ is a parity-check matrix of the same code.*

While quite fast for single-error correction, for correcting t errors in a code of length n it runs in time $\mathcal{O}(\binom{n}{t})$, which is too slow for any practical values of t.

[3] Some authors even use $[n, k, t]$-code labeling, where t is the number of errors the code can correct.

Proof. Any codeword generated from G must belong to null space of H, therefore $H(uG)^T = HG^T u^T = o^T$, for all u, equivalently $HG^T = O$.

Substitute $G = (I_k|A)$ and $H = (B|I_{n-k})$:

$$(B|I_{n-k})(I_k|A^T) = B + A^T = O$$

Clearly, $B = A^T$. □

For example, above Hamming $[7, 4, 3]$-code possesses following parity-check matrix:

$$H = \begin{pmatrix} 1\ 0\ 1\ 1\ 1\ 0\ 0 \\ 1\ 1\ 1\ 0\ 0\ 1\ 0 \\ 0\ 1\ 1\ 1\ 0\ 0\ 1 \end{pmatrix}$$

One can easily check that multiplying codeword 0101110 from the example with H results in zero vector.

Moreover, almost any parity-check matrix, even in non-echelon form can be used to construct a workable echelon-form code generator:

Theorem 2. *If $H = (L|R)$ is a parity-check matrix of linear code $\mathcal{C}$ and R is invertible, $\mathcal{C}$ possesses a generator matrix in form $G = (I|L^T(R^T)^{-1})$.*

Proof. Similarly as in previous theorem, substitute into $HG^T = O$, getting

$$(L|R)(I_k|X^T) = L + R \cdot X^T = O$$

Solution of that equation is $X^T = R^{-1}L$, therefore $G = (I|X) = (I|L^T(R^T)^{-1})$. □

Note that if R is not invertible, one can permute columns of H to get an invertible R. That is always possible, because H generates $(n-k)$-dimensional space (which is a null space of the k-dimensional space generated by G), and therefore must have $n-k$ linearly independent columns.

Linearity also gives us an useful tool to find a minimum distance of the code:

Theorem 3. *Linear code has a minimum distance at least d if and only if it has no non-zero codeword with Hamming weight less than d.*

Proof. Forward implication is trivial — as zero vector is a codeword, there must be no codeword closer than d to it: $|c - 0| \geq d \implies |c| \geq d$.

Backward implication is proven indirectly: If linear code has minimum distance less than d, there exist two distinct codewords a and b which form the minimum distance, so that $0 < |a-b| < d$. Because the code is linear, it must also contains codeword $c = a-b$, which has Hamming weight $0 < |c| = |a-b| < d$. □

1.1.2 Algebraic codes

The basic trouble of the construction of linear codes is to find a code that has the largest possible minimum distance, but still possesses some good systematic structure so that decoding can be performed by some better algorithm than exhaustively searching for the nearest neighbor.

There exist many algebraic approaches to the problem, most interesting for the purpose of this thesis belong to the subclass of *alternant codes* called *Goppa codes*.

For definition and work on alternant codes, one must use several mathematical structures — larger finite fields $\mathbf{GF}(2^m)$, polynomials, and several decoding algorithms.

This section defines the necessary primitives to show practicality of alternant codes, notably the simplicity of constructing alternant code for target error correction capability. Best reference with more details about all types of codes can be found in [MWS77].

$\mathbf{GF}(2^m)$ finite fields

Large finite fields needed for algebraic code construction can be created as splitting fields from smaller and simpler finite fields. While the construction described here is applicable for construction of any $\mathbf{GF}(q)$ where q is a *prime power* (e.g. p is prime, $m \in \mathbb{N}$ and $q = p^m$), for simplicity this thesis only considers binary case.

Suppose a finite field of 2^m elements is needed. Let $\mathbf{GF}(2^m)$ be a set of all polynomials of degree less than m over $\mathbf{GF}(2)$:

$$\mathbf{GF}(2^m) = \left\{ \sum_{i=0}^{m-1} a_i x^i \mid a \in \mathbf{GF}(2)^m \right\}$$

Observe that such elements are trivial to convert to m-element vectors over $\mathbf{GF}(2)$. They now certainly form an additive group — addition of polynomials does a simple "xor" on their coefficients, and one can easily see that every element is additively invertible (moreover, every element is an additive inversion of itself, because $a \oplus a = 0$).

Forming a multiplication group is a little harder: the constructed field is not closed on simple polynomial multiplication. For this reason, every multiplication of the polynomials is done *modulo some fixed irreducible polynomial over* $\mathbf{GF}(2)$ *of degree* m.

Definition 6. *Polynomial is irreducible over a field if it is <u>not</u> a product of two polynomials of lower degree in the field.*

Choosing an irreducible polynomial of given degree over $\mathbf{GF}(2)$ is not very hard. Simplest method is just trying all possible polynomials and running some irreducibility test until one is found (see [PBGV92]).

Theorem 4. *If $p(x)$ is an irreducible polynomial of degree m, then every nonzero polynomial $a(x)$ of degree less than m has an unique inverse $a(x)^{-1}$ that satisfies*

$$a(x) \cdot a(x)^{-1} \equiv 1 \mod p(x)$$

Proof. Start with the uniqueness — observe the products $a(x) \cdot b(x)$ where $a(x)$ runs through all non-zero polynomials with degree smaller than m, and $b(x)$ is some fixed polynomial also with degree smaller than m. All these products are distinct modulo $p(x)$. Consider, for contradiction, that some product is not unique:

$$a_1(x)b(x) \equiv a_2(x)b(x) \mod p(x)$$

or equivalently,

$$(a_1(x) - a_2(x))b(x) \equiv 0 \mod p(x)$$

which would mean that $p(x)$ divides $(a_1(x) - a_2(x))$ or $b(x)$. Because $a_1(x)$, $a_2(x)$ and $b(x)$ are non-zero and of smaller degrees than $p(x)$, the only possibility is to set $a_1(x) = a_2(x)$.

Because all products are distinct, they are just a "permuted" set of all original polynomials $a(x)$. Specially, for chosen $b(x)$ exactly one $a(x)$ satisfies that $a(x)b(x) \equiv 1$ and $a(x) = b(x)^{-1}$. □

As with all finite fields, multiplicative group of the elements is *cyclic*. Basically, that means that all elements except zero can be expressed as $x^n \mod p(x)$ where $n \in \{0, 1, \ldots, 2^m - 2\}$, and zero can be defined as $x^{-\infty}$. This has a practical value for implementation of multiplication arithmetic on computers: while multiplying the polynomials "by definition" is comparatively slow (it needs $\mathcal{O}(m)$ bit shift and xor operations), precomputing simple logarithm and exponentiation table for all field elements gives a quick way for multiplication, as for all non-zero a and b

$$a \cdot b = \exp(\log(a) + \log(b) \mod 2^m - 1)$$

which can be performed in $\mathcal{O}(1)$.

Inversion of nonzero element can be performed in similar manner, because $1 = x^0 = x^{2^m-1} = x^a \cdot x^{2^m-1-a}$ for all $a \in \{0, \ldots, 2^m - 2\}$:

$$a^{-1} = \exp(2^m - 1 - \log(a))$$

For decoding of Goppa codes, one must also be able to compute square roots of the elements:

$$\sqrt{x^n} = \begin{cases} x^{\frac{n}{2}} & \text{if } n \text{ is even} \\ x^{\frac{n+2^m-1}{2}} & \text{otherwise} \end{cases}$$

Polynomial operations and irreducibility

For creation of splitting fields (both $\mathbf{GF}(2^m)$ from previous section, and larger polynomial fields for construction of Goppa codes), one has to be able to determine the irreducibility of polynomial fast. Factoring by brute force is not very fast, therefore following result is used in a more sophisticated algorithm:

Theorem 5. *Every irreducible polynomial of degree n from polynomial ring $T[x]$ where T is a finite commutative field divides the polynomial $x^{|T|^n} - x$.*

Proof is a result of finite field algebra, and because it is somewhat out of scope of this thesis, it is omitted and can be seen in [GaPa97].

This is used in Ben-Or algorithm that tries to find irreducible factor of $a(x)$ by trying to compute the greatest common divisor (using standard euclidean algorithm) with all $x^{|T|^i} - x$ for all $i \in \{1, \ldots, \deg(a(x))/2\}$. The algorithm is very efficient for generating of irreducible polynomials — the basic idea is that when testing a random polynomial for irreducibility, there is higher probability that it will have small-degree factor. Ben-Or algorithm filters out small-degree factors first, which makes the irreducible polynomial construction provably faster (for explanation see [GaPa97]).

Definition 7. *Polynomial $p(x)$ is called* monic *if its highest-order nonzero element coefficient is 1, e.g. $p_{\deg(p)-1} = 1$.*

Algorithm 1 The Ben-Or irreducibility test.

Input: Monic polynomial $a \in \mathbf{GF}(q)[x]$
Output: Either "a is irreducible" or "a is reducible"

```
p ← 1
for i = 1 → deg(a)/2 do
    p ← p · x^q  mod a
    if gcd(a, p) ≠ 1 then
     return "a is reducible"
    end if
end for
     return "a is irreducible"
```

Using the same method as with creation of $\mathbf{GF}(2^m)$ from previous section, it is also possible to construct a polynomial finite field $\mathbf{GF}(2^m)[x] \mod p(x)$ exactly if polynomial $p(x)$ is irreducible.

While some computations in this field are trivial to do by hand (addition and additive inversion is done element-wise on base field), log and exp value tables tend to be too large for practical parameters (for construction of Goppa codes, degree of $p(x)$ will usually get to hundreds). Multiplication is therefore carried out "by hand" by traditional polynomial multiplication algorithm modulo $p(x)$ (in time $\mathcal{O}(\deg(a)^2)$). Division (and inversion) and square roots must have separate implementations.

Inversion in the field is done by the extended euclidean algorithm (which can be slightly modified to also provide full-scale division without the need to invert in the first step and multiply the result later — it basically saves half of the execution time.)

Algorithm 2 Division of polynomials in $\mathbf{GF}(2^m)[x] \mod p(x)$

Input: Polynomials $p \in \mathbf{GF}(2^m)[x]$ and $a, b \in \mathbf{GF}(2^m)[x] \mod p(x)$.
Output: Polynomial c so that $b(x)c(x) = a(x) \mod p(x)$

```
r_0 ← p
r_1 ← b  mod p                          ▷ Euclidean algorithm sequence
s_0 ← 0
s_1 ← a  mod p                ▷ Modified Bézout coefficient that results in c
while deg(r_1) ≥ 0 do
    q_0, q_1 ← DivMod(r_0, r_1)
    r_0 ← r_1
    r_1 ← q_1
    t ← s_1
    s_1 ← s_0 − (q_0 · s_1)  mod p
    s_0 ← t
end while▷ r_0 is now scalar and is invertible only using GF(2^m) ops. return
s_0 · (r_0)^−1
```

Vandermonde matrix

Definition 8. *Matrix constructed from sequence a in form*

$$\mathrm{Vdm}(a) = \begin{pmatrix} 1 & a_1 & a_1^2 & a_1^3 & \dots & a_1^{n-1} \\ 1 & a_2 & a_2^2 & a_2^3 & \dots & a_2^{n-1} \\ 1 & a_3 & a_3^2 & a_3^3 & \dots & a_3^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & a_n & a_n^2 & a_n^3 & \dots & a_n^{n-1} \end{pmatrix}$$

is called a Vandermonde matrix.

Theorem 6. *Vandermonde matrix has determinant* $\prod_{j=1}^{n-1} \prod_{i=j-1}^{n} (a_i - a_j)$.

Proof. By row matrix operations, the Vandermonde matrix can be converted to smaller form without changing the determinant:

$$\mathrm{Vdm}(a) = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & (a_2 - a_1) & (a_2 - a_1)a_2^2 & \dots & (a_2 - a_1)a_2^{n-2} \\ 0 & (a_3 - a_1) & (a_3 - a_1)a_3^2 & \dots & (a_3 - a_1)a_3^{n-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & (a_n - a_1) & (a_n - a_1)a_n^2 & \dots & (a_n - a_1)a_n^{n-2} \end{pmatrix}$$

After removal of top-left border and application of determinant multiplicativity rules, one gets

$$|\mathrm{Vdm}(a)| = |\mathrm{Vdm}(a_{2\dots n})| \cdot \prod_{i=2}^{n} (a_i - a_1)$$

Final formula is then obtained by induction. □

Vandermonde matrices are a very useful tool for code construction. For more details about them, see [MWS77].

Alternant codes

Alternant codes are typically defined as a restriction of GRS[4] codes, but since there is no intention to work with GRS codes alone in this thesis, definition is done directly:

Definition 9. *For parameters r, n, $\alpha \in \mathbf{GF}(2^m)^n$ where α_i are all distinct and $y \in GF(2^m)^n$ where all y_i are non-zero,* alternant code *is defined as*

$$\mathcal{A}(\alpha, y) = \left\{ c \in \mathbf{GF}(2^m)^n \mid Hc^T = 0 \right\}$$

where H is a parity check matrix of form

$$H = \mathrm{Vdm}^T(\alpha)\,\mathrm{diag}(y) = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ \alpha_1 & \alpha_2 & \alpha_3 & \dots & \alpha_n \\ \alpha_1^2 & \alpha_2^2 & \alpha_3^2 & \dots & \alpha_n^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha_1^{r-1} & \alpha_2^{r-1} & \alpha_3^{r-1} & \dots & \alpha_n^{r-1} \end{pmatrix} \begin{pmatrix} y_1 & & & & 0 \\ & y_2 & & & \\ & & y_3 & & \\ & & & \ddots & \\ 0 & & & & y_n \end{pmatrix}$$

[4] Generalized Reed-Solomon

One can easily convert an alternant code to equivalent binary code by constructing a binary parity-check matrix: every element from $\mathbf{GF}(2^m)$ can be split into m binary elements which are written on m successive rows of the binary matrix. The resulting matrix is called *trace matrix*.[5]

Note that the order of how the rows are expanded or permuted after expansion does not matter for any actual computation or code structure. For various purposes it is viable to exploit this to some extent — best example is the *co-trace construction* that preserves quasi-dyadic matrix structure: binary matrix is constructed as a stack of partial binary matrices, n-th of which contains n-th bits of binary representations of $\mathbf{GF}(2^m)$ elements.

Theorem 7. *Minimum distance of alternant code is* $r + 1$.

Proof. Since alternant codes are linear, all nonzero codewords of the code must have hamming weight at least $r + 1$.

Suppose there is a nonzero codeword c with $|c| \leq r$. Since it is a codeword, $Hc = \mathrm{Vdm}^T(\alpha)\,\mathrm{diag}(y)c = 0$. Because $\mathrm{diag}(y)$ is diagonal and invertible, c and $b = \mathrm{diag}(y)c$ have the same hamming weights. Because any r columns of $\mathrm{Vdm}^T(\alpha)$ are linearly independent (because of Vandermonde determinant is nonzero), equation $\mathrm{Vdm}^T(\alpha)b = 0$ has no nonzero solution for $|b| \leq r$. □

1.1.3 Binary Goppa codes

Goppa codes were first introduced in 1970 by V. D. Goppa in [Gop70]. Following section defines the binary variant of the code, and shows proof of minimum distance and an example of *syndrome decoding* algorithm used to correct the errors.

Definition 10. *Given a polynomial* g *over* $\mathbf{GF}(2^m)$ *of degree* t *and sequence* L *of* n *distinct elements of* $\mathbf{GF}(2^m)$ *where* $\forall g(L_i) \neq 0$, binary Goppa code $\Gamma(L, g)$ *is a code of length* n *that satisfies*

$$c \in \Gamma(L, g) \iff \sum_{i=1}^{n} \frac{c_i}{x - L_i} \equiv 0 \mod g$$

. If g *is irreducible, the code is called* irreducible binary Goppa code.

The sum from definition is traditionally called *syndrome function* and written as $S(c)$, the sequence L is *code support*. Notice that if the code is irreducible, L can contain all elements of $\mathbf{GF}(2^m)$ since for all $g(L_i) \neq 0$.

There are several methods to construct a parity check matrix of Goppa code. A matrix over polynomials over $\mathbf{GF}(2^m)$ can be constructed as

$$H = \left(\frac{1}{x - L_0} \mod g, \frac{1}{x - L_1} \mod g, \ldots, \frac{1}{x - L_{n-1}} \mod g \right)$$

H satisfies the condition for being a parity-check matrix, since $(\forall c)c \in \Gamma(L, g) \iff Hc = 0$ from definition. Thanks to the polynomial coefficient

[5]Note that the vectors of the original matrix columns and traced columns behave absolutely same under addition, so there is no need to compute or multiply $\mathbf{GF}(2^m)$ elements when, for instance, computing the codeword syndromes.

additivity, it can then be expanded to a matrix over $\mathbf{GF}(2^m)$ by writing polynomial coefficients on separate rows, and finally traced to a binary parity-check matrix.

Second method to construct the matrix is more complicated and shown completely for example in [EOS06]. The result is interesting though — it clearly shows that Goppa codes are indeed a subclass of alternant codes, have minimum distance at least $t+1$, and also that alternant decoding algorithms can be used to correct $\lfloor t/2 \rfloor$ errors.

$$H' = \begin{pmatrix} g_t & 0 & 0 & \cdots & 0 \\ g_{t-1} & g_t & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ g_1 & g_2 & g_3 & \cdots & g_t \end{pmatrix} \cdot \mathrm{Vdm}^T(L) \cdot \begin{pmatrix} \frac{1}{g(L_0)} & & & \\ & \frac{1}{g(L_1)} & & \\ & & \vdots & \\ & & & \frac{1}{g(L_n)} \end{pmatrix}$$

Better and very important result about actual minimum distance of Goppa codes can be obtained as follows:

Theorem 8. *Let $\Gamma(L, g)$ be an irreducible binary Goppa code. The minimum distance of such code is at least $2\deg(g)+1$.*

Proof. For codeword c, define an *error locator polynomial* $\sigma_c \in \mathbf{GF}(2^m)[x]$:

$$\sigma_c = \prod_{c_i=1} (x - L_i)$$

Its formal derivative is

$$\sigma_c' = \sum_{c_i=1} \prod_{c_j=1 \wedge j \neq i} (x - L_j)$$

After substitution into the syndrome function one obtains

$$\sigma_c S(c) \equiv \sigma_c' \mod g$$

σ_c is invertible modulo g, because for all elements of L holds $g(L_i) \neq 0$ and therefore $\gcd(g, \sigma_c) = 1$. Using that, the equation can be rewritten as

$$\frac{\sigma_c'}{\sigma_c} \equiv S(c) \mod g$$

and again from definition of Goppa codes

$$c \in \Gamma(L, g) \iff \sigma_c' \equiv 0 \mod g$$

Observe a function

$$\mathbf{GF}(2^m)[x] \to \mathbf{GF}(2^m)[x], f(x) = \sum_{i=0}^{n} f_i x^i \mapsto (f(x))^2 = \sum_{i=0}^{n} f_i^2 x^{2i}$$

The function is a ring homomorphism, and its image, $\mathbf{GF}(2^m)[x^2]$ is a set of polynomials called *perfect squares* of the ring $\mathbf{GF}(2^m)[x]$. σ' is a perfect square, because from definition of formal derivation

$$\sigma' = \sum_{i=1}^{n} i\sigma_i x^{i-1}$$

and thanks to properties of $\mathbf{GF}(2^m)$, $i\sigma_i = 0$ for each even i.

Now, because g is irreducible[6], the equation can be rewritten as

$$c \in \Gamma(L, g) \iff \sigma'_c \equiv 0 \mod g^2$$

From this, one can directly bound the Hamming weight of any nonzero codeword:

$$|c| = \deg(\sigma_c) \geq 1 + \deg(\sigma'_c) \geq 2\deg(g) + 1 = 2t + 1$$

□

Syndrome decoding

Decoding algorithm for Goppa code syndromes can basically be constructed from above proof, simply by solving the equation system to get σ_c from $S(c)$ and $g(x)$. The algorithm is as follows:

1. Compute $\nu = \sqrt{S^{-1}(c) - x}$. This works for all nonzero syndromes (in which case there are no errors and algorithm can easily return correct result).

2. Compute a and b so that $a = b\nu \mod g$, but still $\deg(a) \leq \lfloor \frac{\deg(g)}{2} \rfloor$ and $\deg(b) \leq \lfloor \frac{\deg(g)-1}{2} \rfloor$. This can be achieved by using extended euclidean algorithm on ν and g and stopping it "in the middle". Detailed explanation can be found in [EOS06].

3. Error locator polynomial can be now constructed as $\sigma = a^2 + x \cdot b^2$.

4. Factoring the error locator polynomial will bring linear factors $(x - L_i)$ where L_i are the code support values corresponding to error positions. If a non-linear or squared linear factor is found, the decoding failed and the original word was damaged by more than t errors.

Note that factoring σ may be the most time-consuming step, especially if using the "dumb" trial-zero-finding algorithm that just evaluates σ for all possible L_i, and especially if the code length is far greater than t (which is the usual case). Berlekamp trace algorithm for polynomial factorization, as defined in [Ber70], is usually the simplest and fastest choice for most setups.

1.2 McEliece cryptosystem

McEliece algorithm is an asymmetric encryption algorithm developed by Robert J. McEliece in 1978 [McE78]. Its operation is specified for parameters n, m, k, t as follows:

Key pair generation

- Generate a random Goppa $[n, k]$-code over $\mathbf{GF}(2^m)$ capable of correcting t errors, with generating polynomial g, support L and generator matrix G.

[6] it actually suffices that g is separable, e.g. consisting only of linear factors

- Generate a random permutation over n elements represented by matrix P.
- Generate a random invertible binary matrix S of size k.
- Compute $G' = SGP$. Publish tuple (G', t) as a public key.
- Securely store (g, L, S, P) as a private key.

Encryption
- Generate random binary vector e of size n with hamming weight t.
- For plaintext message m represented as a binary vector, compute ciphertext as $c = m^T G' + e$ and send it to the owner of private key.

Decryption
- Compute $c' = cP^{-1}$.
- Apply error correction of stored code on vector c', obtaining c''.
- Retrieve the plaintext message as $m = c''S^{-1}$.

To understand the principle of McEliece, observe how the message would be consecutively modified by being multiplied by S, G and P matrices:

First, multiplication by S "scrambles" the message to the form that is, if S was random-enough[7], indistinguishable from random vector without knowledge of S. Notice that if the scrambled message is modified by bit errors, simple multiplication by S^{-1} yields a random vector (e.g. for error vector e, the result is $m \oplus \sum_{e_i=1} S^{-1}_{i*}$, to retrieve message in such case, one would need to guess the position of all error bits).

Next, scrambled message is multiplied by G, thus adding redundancy bits and converting it to codeword of the Goppa code. After adding t errors to this form of the matrix, the scrambled message is "protected" by the bit errors from being unscrambled, and the errors can be easily removed only by decoding the Goppa code, which requires the knowledge of g and L.

Matrix P has the purpose of hiding the structure of SG, which would otherwise easily reveal the S and G matrices (if G was in the echelon form, S would be the left square of the SG and G could be easily retrieved by inversion). If the columns are permuted, there is no way for attacker to guess which of them would belong to the echelon part of G, thus making the Gaussian elimination approach effectively impossible).

Basic properties of the cryptosystem are now apparent:

- Plaintext length is exactly k bits, ciphertext is n bits long.
- Key generation takes $\mathcal{O}(n^3)$ binary operations. Most of the time is usually consumed by the inversion of matrix S and multiplication of matrices.
- Encryption has complexity $\mathcal{O}(n^2)$ and is algorithmically simple, using only matrix-vector multiplication and some kind of generator of random bit vector.
- Decryption has complexity $\mathcal{O}(ntm^2)$ which is, for practical parameters, roughly equivalent to $\mathcal{O}(n^2)$.

[7]Didn't have any apparent structure or passed some good randomness test, anyone can choose.

- Public key has size $n \cdot k$ bits.

First parameters suggested by McEliece, $n = 1024, k = 644, m = 10, t = 38$, were designed for attack complexity around 2^{80} (although now broken by a more effective attack to around 2^{58}) and provide a scheme with public key size around 80kBytes.[8]

1.2.1 Modern variants

Research of the cryptosystem brought several interesting conclusions.

First, consider the role of matrix S: It actually serves as a kind of a symmetric cipher that converts any plaintext to a vector that is *indistinguishable* from a random vector, so that the added bit errors can have the needed destructive effect on plaintext readability. Note also that not much happens if S somehow leaks from the key — attacker can get the permuted G matrix, which alone is still unusable for finding the original g and L code parameters. Using actual symmetric cipher that could be more effective against simple linear-algebraic analysis could also bring more security. Therefore, it seems reasonable to remove whole S matrix from the whole process:

- It is not needed for protection of key trapdoor,
- it is the source of the general slowness of key generator, and
- if removed, the public key can be effectively stored in $k(n-k)$ bits instead of nk, making it about 3 times smaller for usual parameters.

Modern variant of the McEliece cryptosystem would therefore publish only right part of the echelon form of G. Function of S would be supplied by any symmetric cipher, and function of P (permuting the rows of G) could be replaced by basis change of the H space.

Second conclusion is the simple possibility of attacking the trapdoor. If anyone would ever encrypt the same message m more times using the same public key to obtain ciphertexts c_1, c_2, attacker could easily compute the compound error position vector $e' = c_1 \oplus c_2 = e_1 \oplus e_2$. With high probability, this vector would have around $2t$ ones, giving the attacker a very good hint on guessing the error positions. While this is still not sufficient for practical attack, availability of third such ciphertext makes the error correction trivial in almost linear time.

1.3 Quasi-dyadic Goppa codes

The most annoying problem of McEliece scheme is the need to handle big matrix data — the most efficient system based on general Goppa codes available still takes $\mathcal{O}(n^3)$ time to generate the keys (which is basically a result of Gauss-Jordan elimination that is needed to produce the generator matrix) and key size is $\mathcal{O}(n^2)$.

While the cost of key generating algorithm can be ignored to some extent, storage requirements for a common keyring are quite demanding — public key

[8] At the time, this was just too much compared to RSA.

of a keypair that provides attack complexity around 2^{100} has size around 0.5 megabytes, growing to approximately 4.5 megabytes for 2^{256}.[9]

The best approach for reducing the key size is to make the parity check (and therefore also the generator matrix) that has some non-random structure, so it can be easily compressed. Such property is in direct conflict with McEliece cryptosystem requirements (generator matrix should be indistinguishable from a completely random one), which brought two interesting results:

- Most (nearly all) attempts to produce a lightweight compressible parity-check matrix have been broken. Those include cyclic, quasi-cyclic, LDPC[10], generalized BCH and GRS-based McEliece.

- Separate security reduction proof is required for such cryptosystem.

The first scheme that resisted cryptanalysis and provided the needed security reduction proof was that of Misoczki and Barreto published in [MiBa10], using the so-called quasi-dyadic Goppa codes.

Definition 11. *Every square matrix of size 1 is dyadic.*

If A and B are dyadic matrices, matrix $\begin{pmatrix} A & B \\ B & A \end{pmatrix}$ *is also dyadic.*

Observe that to be able to retrieve the whole dyadic matrix, one only needs to store one of its columns or rows (and remember which one it was), because all other rows and columns are only dyadically permuted always by the same permutations. Effectively, for matrix of size n this gives a nice n-times compression factor. For simplicity, whole dyadic matrix "generated" by first row a is usually written as $\Delta(a)$.

In the paper, Misoczki and Barreto show that it is easily possible to construct a Goppa code with dyadic parity check matrix. The algorithm is simple – basically it fills first row of the matrix with several random $\mathbf{GF}(2^m)$ elements, computes the rest from equations that need to be satisfied for the matrix to be of Goppa code, and then check whether the result is actual Goppa code (by computing g and checking uniqueness of all support elements), possibly retrying until eventually a Goppa code parity check matrix is found. For detailed description, see Algorithm 2 from the article [MiBa10].

Using the *co-trace* construction the matrix is then expanded to binary form. For clarity, following example shows the co-trace construction for $\mathbf{GF}(2^4)$ elements over (4,4)-matrix:

[9] key size for 2^{256} security level is computed for parameters around $n = 8192$, $t = 250$ that have been suggested by several authors

[10] Low Density Parity Check matrix code. It has a low number of 1's per column.

$$\begin{pmatrix} a & b & c & d \\ b & a & d & c \\ c & d & a & b \\ d & c & b & a \end{pmatrix} \mapsto \left(\begin{array}{cccc} a_0 & b_0 & c_0 & d_0 \\ b_0 & a_0 & d_0 & c_0 \\ c_0 & d_0 & a_0 & b_0 \\ d_0 & c_0 & b_0 & a_0 \\ \hline a_1 & b_1 & c_1 & d_1 \\ b_1 & a_1 & d_1 & c_1 \\ c_1 & d_1 & a_1 & b_1 \\ d_1 & c_1 & b_1 & a_1 \\ \hline a_2 & b_2 & c_2 & d_2 \\ b_2 & a_2 & d_2 & c_2 \\ c_2 & d_2 & a_2 & b_2 \\ d_2 & c_2 & b_2 & a_2 \\ \hline a_3 & b_3 & c_3 & d_3 \\ b_3 & a_3 & d_3 & c_3 \\ c_3 & d_3 & a_3 & b_3 \\ d_3 & c_3 & b_3 & a_3 \end{array}\right)$$

Note that for the Goppa codes, the matrices are usually not square, but have height t. The result can still be stored in pretty small space (only first lines of each dyadic block are needed).

Conversion to generator matrix must be again done using only dyadicity-conserving operations on whole blocks. Advantage is that the matrix does actually never need to be expanded into full scale, everything can be computed using only first rows of dyadic blocks.[11]

Decoding is, for simplicity (especially to avoid converting syndromes to canonical Goppa syndrome-function results) done using alternant decoder working with $g^2(x)$ polynomial and the same support. Proof of equivalence of the codes and description of alternant decoding can be found in [Hof11].

Using all those properties to construct QD-McEliece scheme brings very good results:

- Encryption and keypair generation are several orders faster (the improvements are respectively from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$ and from $\mathcal{O}(n^3)$ to $\mathcal{O}(n \log n)$).
- Public key size is $\mathcal{O}(nm)$ or basically $\mathcal{O}(n \log n)$. For a scheme with attack complexity of 2^{256} the size is only around 16 kilobytes, which makes the key storage very practical.

1.3.1 FWHT in QD-McEliece

Dyadic matrices have one more beneficial property: they can be multiplied in $\mathcal{O}(n \log n)$ time using a Fast Walsh-Hadamard Transform algorithm that only operates on first rows of the matrices. This not only speeds up the encryption and keypair generation, but also saves much memory for encryption operations because the matrices do not need to be expanded (possibly making implementation easier on embedded hardware).

[11] Description of how Gauss-Jordan elimination is done using FWHT on quasi-dyadic matrices is largely out of scope, and can be seen in source code of the software part of this thesis

The best available description of FWHT transform with usage for QD-matrix multiplication can be again found in [Hof11]. This thesis only defines FWHT and its usage for matrix multiplication.

Definition 12. *Let's mark $t(x)$ a function that performs FWHT on vector x. If size of x is 1, $t(x) \mapsto x$. If size of x is a multiple of 2 and x_1, x_2 are the first and second half of x, then $t(x_1, x_2) \mapsto t(x_1 + x_2), t(x_1 - x_2)$.*

Notice that the FWHT is basically a divide-and-conquer algorithm working in $\mathcal{O}(n \log n)$.

Multiplication of square dyadic matrices with first rows a and b of size n is done as follows:

$$\Delta(a) \cdot \Delta(b) = \Delta\left(\frac{t(t(a) \circledcirc t(b))}{n}\right)$$

Note that all computation is done *lifted to* $\mathbb{Z}$, $\circledcirc$ denotes element-by-element vector multiplication, and division by n is done as integer division with rounding down.

Observation of dyadicity also shows a really simple way to tell whether a dyadic matrix is invertible or not: If it has an even number of 1's in first row, it is singular. If the number of 1's is odd, then the matrix regular, and its inversion is the same matrix.

1.4 Improving cryptographic properties of the scheme

1.4.1 Improving original McEliece

To solve the problem from described in Section 2.2.1 that compromises the original McEliece scheme when the same message is ever encrypted twice, several padding schemes have surfaced. Most notable of those is a Kobara-Imai scheme described in [KI01].

Another attack can be performed when adversary wants to remove the added errors from any message and is able to determine whether the owner of private key has succeeded decoding the code: He simply flips two random bits of the message at a time and sends the modified message to the decoder. If he determines that decoding failed, he knows that either both bits weren't flipped by errors, or both were and decoder refused to accept message with $t - 2$ errors. If he determines that decoding succeeded (for example when the owner of the private key replies with any kind of reasonable answer), he is sure that one of the flipped bits was an error bit and the second was not. Positions of full t errors can be easily recovered in $n - 1$ attempts, simply trying to flip i-th and $i + 1$-th bit at i-th attempt.

For this purpose, the padding adds two improvements:

- Message text is padded with a random string so that there is only an extremely tiny probability that any message is ever repeated.
- Error positions are not random, but determined from the plaintext using a hash function. Owner of the private key refuses the message whenever the error positions differ from what the hash function says.

Schematically, given classical McEliece encryption function E, decryption D, cryptographically secure hash function h that maps message space to error-vector space, message m and one-time random vector r, encryption is done as $c = E(m+r, h(m+r))$, while decryption is done as $(m+r, e) = D(c)$ and accepted only if $e = h(m + r)$.

It should be noted that Kobara and Imai proved that such padding establishes indistinguishability of the message under adaptive chosen-ciphertext attack[12], which is a strong security property with many interesting implications.

1.4.2 Removing the plaintext indistinguishability requirement from modern McEliece variants

The problem of both the "modern" McEliece scheme that uses only echelon form of G and the QD-McEliece scheme is that if the plaintext known not to be indistinguishable from a completely random vector, it is usually easy to remove the errors by hand and retrieve the plaintext. For illustration, encrypting the word "`this is plaintext`" with QD-McEliece would most probably result in something like "`tbis$iw plbintektG3D6`" — data part of the codeword can be easily "decoded" by average human.

For this purpose, Fujisaki and Okamoto suggested a combination of symmetric cipher and McEliece cryptosystem in a padding scheme published in [FO99]: Asymmetric encryption is only used to encrypt the symmetric key, and the message encrypted by this symmetric key is then appended to the block of the asymmetric cipher. Similar error-guessing protection as in previous case is used as well.

Schematically, similarly as in Kobara-Imai scheme but with functions E_s and D_s that provide symmetric encryption and decryption, the padding can be described as follows:

- Encrypt as $c = E(r, h(r + m)) + E_S(m, r)$.
- Decrypt by splitting the message c into (a, b) blocks with asymmetric and symmetric part. Compute $(r, e) = D(a)$ and $m = D_S(b, r)$. Accept message m if $e = h(r + m)$.

Similarly to Kobara-Imai scheme, Fujisaki-Okamoto scheme provides CCA2-IND property of the cipher. Moreover, it is quite practical — if the symmetric encryption function is good and some simple plaintext padding that prevents leaking information about plaintext length is added, one can pretty easily encrypt arbitrarily long messages (which has been a general problem with block-based McEliece encryption — for example, in case of Kobara-Imai the ciphertext is around two times as big as plaintext, while in Fujisaki-Okamoto scheme, the sizes are (asymptotically) the same).

1.5 Niederreiter cryptosystem

Niederreiter cryptosystem is a "dual" variant of McEliece [Nie86]. The idea is to encode information as an error vector and transfer its *syndrome*, which the

[12] this long name is commonly shortened to CCA2-IND

attacker still cannot decode without knowledge of corresponding code.

Keypair generation
- Generate a random Goppa $[n-k]$-code capable of correcting t errors with parity check matrix H indistinguishable from a random matrix. Generate a random invertible matrix S of $(n-k)$ rows, and a random permutation over n elements represented by matrix P.
- Compute $H' = SHP$, publish (H', t) as a public key.
- Securely store (g, L, S, P) as a private key.

Encryption
- Encode message m as a binary vector e of length n that contains exactly t 1's.
- Compute ciphertext as $c = H'e$.

Decryption
- Compute $c' = cP^{-1}$,
- Run syndrome decoding as if c' was a syndrome of some message, obtaining an error vector c''
- Compute $e = c''S^{-1}$.
- Decode message m from e, using same method as in encryption.

While Niederreiter cryptosystem is very fast and quite efficient, it is not widely used for several reasons:

- There is a known attack that distinguishes a parity-check matrix of Goppa code (effectively the public key of Niederreiter cryptosystem) from a completely random matrix in polynomial time [Fau11], which contradicts the setup of the cryptosystem.
- Although the cryptosystem itself is extremely fast, the need to prepare plaintexts that have exact hamming weight either requires some kind of memory or CPU-intensive ranking function such as colex ranking. Simpler and faster approaches usually do not effectively cover whole possible combination space, thus greatly decreasing information rate of the cipher.
- To the best of author's knowledge, there are no plaintext padding schemes or CCA/CCA2-resistant schemes for Niederreiter that would cope with the requirement of constant-weight plaintext vector.

1.6 Other code-based cryptographic primitives

Three more schemes that use hardness of syndrome decoding problem as an security guarantee exist:

- FSB (Fast Syndrome Hash) is a SHA-3 candidate fast secure hash function with security reduction to syndrome decoding and easily variable parameters. For details see [Aug08].

- SYND is a stream cipher or random stream generator that uses syndrome decoding to ensure randomness of the generated stream. SYND is very interesting from the point of generator speed (the only faster widely used stream cipher is RC4) and provable security (breaking the keystream can be reduced to decoding a syndrome without knowledge of the underlying code). See [GPLS07].

- Stern identification scheme is used as a zero-knowledge proof protocol for ensuring identity of communication partners. See for example [Ste94].

1.7 Colex ranking

In several places, most notably for the Fujisaki-Okamoto padding scheme a *ranking* function is needed to convert vectors of exact weight to numbers and back. Most authors suggest [PQCSlides] to use the computationally simplest ranking function: the colex ranking.

Thorough investigation of algorithm for colex ranking and unranking can be found for example in [Rus], for this purpose only a simple description will be sufficient:

Definition 13. Ranking function *assigns any k-element subset of the set of n elements an unique number* $0 \leq a < \binom{n}{k}$.

Unranking function *is inverse of ranking function.*

Two k-element subsets A and B of an set with strict linear ordering $<$ *are said to be in* co-lexicographical order *if, for the sequences a and b of their elements sorted in ascending order, there exists index i that* $a_i < b_i$ *and* $(\forall j) j > i \implies a_j = b_j$.

For example, subsets of size 3 from set $\mathbb{N}$ are in following colex order: $\{1,2,3\}$, $\{1,2,4\}$, $\{1,3,4\}$, $\{2,3,4\}$, $\{1,2,5\}$, $\{1,3,5\}$, ...

The ranking function r is pretty easy to construct:

$$r(a_1, a_2, \ldots, a_k) = \sum_{i=1}^{k} \binom{a_i - 1}{i}$$

Unranking requires an algorithm:

- For input rank r cycle with i from k to 1.

- In each step find smallest p so that $p \geq i$ and $\binom{p}{i} \geq r$. Update $r = r - \binom{p-1}{i}$, set $a_i = p$.

- Output $(a_1, a_2, \ldots, a_k)$.

Problematic part of colex ranking and unranking is the need to compute tremendous amounts of combination numbers. Most authors recommend precomputing a table of binomial coefficients that moves the algorithm to (almost) $\mathcal{O}(n)$ time complexity, but the cache consumes enormous amount of memory even when caching only the numbers with reasonable usage probability.

A little slower, but significantly more space-efficient approach exploits the fact that both algorithms only use combination numbers that form a "path" in

two-dimensional space for dimensions n and k (nice image of the situation can be seen in [Rus], Figure 4.2). Because "neighbor" combination numbers are easy to compute (using only one long multiplication and division), computing the whole path in correct order takes only $2(n+k)$ long operations, which is acceptable for most situations.

Chapter 2

Post-quantum digital signatures

2.1 Code-based digital signatures

To produce a full code-based cryptosystem similar to GnuPG, one needs to effectively compute digital signatures. This section briefly introduces available concepts of code-based signatures, observes their properties and suggests a viable signature scheme for the implemented cryptosystem.

2.1.1 Short history

The main difference of code-based signature schemes from RSA-like cryptosystems is that the McEliece encryption function is generally *not invertible* – given a random word not generated by actual encoding process, there is an overwhelmingly low probability that it can actually be decoded. Observe, for example, the very rough estimate of probability of successful decoding a random word by Goppa code with commonly used parameters $[1024, 694, 33]$:

$$\begin{aligned} P_{\text{decodable}} &= \frac{2^{694} \cdot \sum_{i=1}^{33} \binom{694}{i}}{2^{1024}} \\ &\doteq \frac{2^{694} \cdot 694^{33}}{2^{1024}} \\ &\doteq 2^{-315} \end{aligned}$$

Because there are no "interesting functions" other than encoding and decoding that can be run on the actual Goppa codes, most derived signature schemes target some way to workaround the decoding improbability.

Some notable attempts are following:

- First publicly-available attempt, the Xinmei scheme [Xin90] was considered broken one year after [AW92].
- Hamdi signature scheme [Ha06] is based on constructing a large error-correcting code from smaller blocks (with much denser codeword spaces), which are in turn permuted and scrambled like matrix $G' = SGP$ in the original McEliece cryptosystem. Using some knowledge of hidden block structure, the scheme is susceptible to simple diagonalization attack that reveals small G matrices [Ha09].

Some attempts tried to emulate the signature scheme using identification protocols based on Stern identification scheme (most notably the [CGG07]). Such schemes, while provably secure, generally suffer from the signature size: The signature is basically a transcript of multiple Stern identification attempts with random choices of the parties based on the bits of signed message, but because Stern scheme consumes quite a bit of bandwidth and in one turn it provides only $\frac{2}{3}$ assurance that the identified party is indeed the one with private key, the transcript is usually as big as megabytes, which is just too much for practical signatures.

2.1.2 CFS signature scheme

CFS is named for authors Courtois, Finiasz and Sendrier [CFS01]. The authors chose not to differentiate from the original McEliece trapdoor and use decoding inversion as signature, thus easily guaranteeing security. The low probability of success of decoding is solved as follows:

- If the decoding fails, the message to be signed is modified in some simple way (e.g. there is a counter appended to the message, or several random bits are flipped) so that the verifier can easily see that the modification was really done to the original message.

- Used Goppa code parameters are modified to preserve security but *maximize the density of decodable syndromes*. That is most easily done by reducing the minimum distance of the code to the lowest possible value. (Instead, for conserving security, one must greatly increase code dimension.)

The scheme can be applied both to McEliece and Niederreiter cryptosystems – authors suggest using the Niederreiter variant for the fact of smaller public keys and much smaller signature size (in the article, signature of scheme that provides 2^{80} attack complexity based on a Goppa code with $m = 16$ and $t = 10$ can be packed using a ranking function to around 120 *bits*). The obvious problem of CFS is larger size of public keys (resulting from the increased m parameter) and the time needed to sign a message (the time complexity is $\mathcal{O}(t!)$)

For illustration, public key size of above described scheme is around 10MB and signature takes tens of seconds. Unfortunately, CFS security was greatly compromised by subsequent development in the field, and current parameters are several times worse.

Although there have been attempts to reduce the size and cost of CFS, most notably the quasi-dyadic CFS variant from [BCMN11][1] that, with some modifications, achieves signatures in under a second with still-workable public key size (tens of megabytes), none has provided a signature scheme thinkable for general usage in current environments.

[1] QD compression is not as great for CFS as for general McEliece, the obvious problem is that the "size reduction factor" based on t can not ever get high enough. Moreover, for QD to work, t needs to be a power of 2, which either greatly reduces parameter choice, or calls for sub-dividing the t-sized blocks which reduces compression to greatest power of 2 dividing t — for example, with $t = 12$ the compression factor is only 4.

2.1.3 Practicality of CFS

If there is no major breakthrough in non-deterministic decoding, CFS will not be usable on generally available computers for quite a some time, especially not for the parameters that would resist quantum computer attacks using Grover search:

Generally, the biggest problem with CFS is that to increase security (which depends exponentially on parameters t and m) one must also exponentially increase some part of the cryptosystem, be it the signature time (which is $\mathcal{O}(m^2 t!)$) or key size (which is $\mathcal{O}(2^m mt)$).

For this inability to efficiently increase cryptosystem parameters, this thesis states that CFS in current state is impractical and seeks to replace it with some other post-quantum digital signature scheme.[2]

2.2 Hash-based digital signatures

For the purpose of building the cryptosystem software, hash-based signatures have been selected instead of code-based cryptography. Although it breaks the nice "code-based" pattern of the whole software, limiting users with unworkable key sizes or signature times was perceived as more serious software flaw than addition of relatively simple FMTSeq scheme.

All hash-based signature schemes are based on *one-way* functions instead of *trapdoor* ones known from other signature schemes, traditionally using a one-time signature scheme based on provably uninvertible hash function[3].

FMTSeq signature scheme, as defined in [NSW05], is based on Merkle-tree extension of one-time signature scheme to a signature scheme with higher, but still limited number of signatures. This section describes all levels of the structure.

2.2.1 Lamport one-time signature scheme

Security of Lamport one-time signature scheme is based on invertibility of hash functions — general idea is that the private key is some set of random strings, public key is some set of random hashes, and the signer proves the knowledge of private key by revealing some subset of the private key that is chosen by the signed message. Verifier can then easily see that the revealed subset in the message hashes piece-by-piece to the public key, and compares that the subset that signer made public is indeed the subset that was chosen by the message, thus making sure that it was indeed the message that the signer wanted to sign.

In original Lamport scheme, the subsets are generated from the binary representation of the message (or message hash); and the published secrets are chosen from a set of secrets that correspond respectively to all possible 0's and 1's in the message. The scheme is then defined as follows:

Setup For signing a n-bit message, signer generates two sets of random strings A and B of elements $a_1, \ldots, a_n, b_1, \ldots, b_n$ and chooses a cryptographically

[2]Actually, the choice is simple — hash-based digital signatures are the only post-quantum signatures known to the author that seem to be both efficient and not broken.

[3]The greatest advantage of this fact is that instead of trapdoor functions that are relatively scarce and easily breakable, there is plenty of readily available and cryptographically secure hash functions!

secure hash function h. Using some authenticated channel, he publishes $h(a_1), \ldots, h(a_n), h(b_1), \ldots, h(b_n)$.

Signature For each i-th bit in the message, if the bit is zero, append a_i to the signature, otherwise append b_i. Publish the secrets as a signature.

Verification Split signature into $s_1, \ldots, s_n$. For each i-th bit of the message, verify from public key that if the bit is zero that $h(s_i) = h(a_i)$, or otherwise that $h(s_i) = h(b_i)$. If all bits can be verified, the signature is valid.

After signature, private key should never be used to sign any other message again: Seeing both signatures would very probably give the attacker an information about both a_i and b_i for some i, allowing him to easily derive signatures with any chosen bit value on i-th position. (Note that in real situations when messages themselves are hashes, only around $\log_2(n)$ produced signatures will completely reveal the secret key, allowing the attacker to sign anything.)

Attacking the scheme requires inversion of underlying hash function:

- If the attacker wanted to derive a signature from public key, he would need to invert one hash function for every zero and one in the message.
- To produce "related" signature, e.g. to produce a signature of message m from a valid signature of message m' where m and m' have exactly one bit flipped on i-th position, attacker would need to remove i-th secret of original bit value, and compute a new secret s for the other bit value only from the knowledge of $h(s)$. This again requires hash inversion.

2.2.2 Merkle trees

Definition 14. *Merkle tree for sequence of strings $S = (s_1, s_2, \ldots, s_{2^n})$ is defined as a binary tree where all leafs have depth n, i-th leaf from "left" has an associated value $h(s_i)$, and all other nodes have associated value $h(l + r)$ where l and r are values of left and right child.*

Merkle trees can be used to extend any one-time signature scheme to some fixed number of signatures:

Setup Generate 2^n one-time signature scheme secret and public keys into sequences S and P. Store S for later usage. Construct a Merkle tree from sequence P and publish the hash value associated with the root node. Set $i = 0$.

Signature Increase i by 1. Take i-th one-time signature secret and produce a one-time signature of the message and publish it along with i-th public key. To prove that one-time signature keys were already generated before the Merkle root was published in Setup phase, also publish i and *authentication path neighbors* (see below) in the Merkle tree.

Verification Verify one time signature. Hash the public key successively with all authentication path neighbors and check that result is the same as published root node value. If both checks passed, accept the signature.

Concept of authentication path and neighbors is simple, and, among other features of Merkle trees, by far most easily explained using an image.

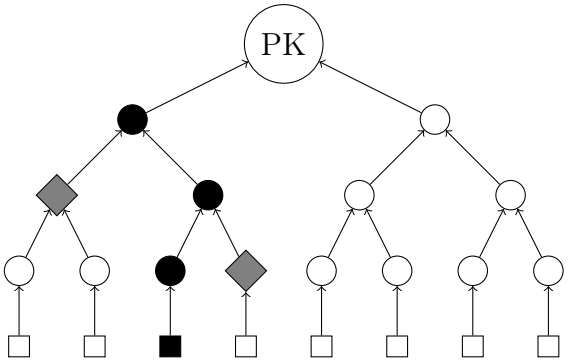


Figure 2.1: Merkle tree path illustration. Squares are public keys of one-time signatures, PK is public key of the whole scheme. Arrow denotes hash function application. Black nodes are authentication path, diamond nodes are authentication neighbors needed for verifier to reconstruct hash path to verify that he gets exactly PK from black-square one-time public key.

Signature size improvement

Because original Lamport signatures tend to be a little (but not restrictively) long, there is also a natural need to shorten them. Simple and very effective approach that effectively reduces signature size by around 60% in common case is described in the FMTSeq paper:

First, do not produce Merkle signature from separate Lamport signature, public key and authentication path. Instead, squash the Lamport signature and public key together into one string — where either a_i or b_i is to be revealed, there is no need to also publish $h(a_i)$ or $h(b_i)$. This removes one third of original signature size.

Second third is removed by modifying the Lamport scheme: Instead of having two sets of secrets A and B for zeroes and ones respectively, only produce one set of secrets for 1's (call it S). Compute and publish all subsecret hashes as usual. When signing a message, for each i-th bit either append s_i if the bit is 1, or $h(s_i)$ if the bit is 0. Verifier hashes all signature parts where the message is 1 and compares the result with public key, accepting the signature if both are exactly same.

This scheme has a small drawback: From a valid signature of a message that contains at least one bit with value 1, attacker can easily compute a signature for the same message with 0 at corresponding position only by hashing one part of the signature.

The drawback is fixed using the fact that attacker can not "flip" a 0-bit to 1-bit because that would require hash inversion — message is padded in such way that flipping any 1-bit to 0-bit would also require flipping 0 to 1 elsewhere. Simplest method to achieve this is to append a binary representation of the count of zeroes in the original message — whenever someone flips a bit in the message, count of zeroes increases, which in all cases flips at least one 0 to 1.

Resulting signature size is therefore $|h| \cdot (n + \lceil \log_2(n) \rceil)$ instead of original $|h| \cdot 3n$.

2.2.3 Efficient Merkle tree traversal

Main efficiency consideration when working with Merkle signatures is the fact that to provide a signature, one needs to quickly determine the values of the nodes near the authentication path. This can be problematic — caching the whole tree would require enormous amount of memory (basically the count of cached node values would be twice the amount of signatures Merkle tree was designed to produce), and computing them is very slow (in naive case it is almost equivalent of generating the whole hash tree again, which involves at least 2^{n+1} hash function calls and is generally slow). FMTSeq was the first scheme that showed practical method to improve the situation by subdividing the tree to layers and subtrees:

Authors suggest that on each level of the tree, some values next to authentication path are still cached and precomputed, and after each signature is generated, the precomputed structure is updated, while layering concept enables efficient block-like cache management and prediction of what needs to be computed. In resulting scheme, cache updates are performed after each signature is done with a very good bound on how long can the cache update take – resulting update time and consumed cache space is almost negligible when compared to values needed to produce and store the signature. Details about implementation are out of scope of this thesis and can be found in FMTSeq paper.

2.2.4 Practicality of FMTSeq signature scheme

With several other considerations and improvements, FMTSeq scheme can get extremely efficient, requiring only kilobytes of cache storage and being several orders of magnitude faster than other signature schemes:

- Having a properly seeded cryptographically secure random number generator saves one from storing the private keys. Simplest possible method is implemented in the software attached to this thesis: to get i-th secret key, join a secret value (common for all keys) with some representation of i and use the result to seed some standardized random number generator (simple padded RC4 is used for this thesis).

- Signature calculation and verification is extremely fast when compared to currently used signatures. Even asymptotically, for given security level 2^n, FMTSeq signs and verifies signatures in around $\mathcal{O}(n^2)$ time and space. RSA-based cryptosystems usually operate in $\mathcal{O}(n^3)$.

- There is no good method do efficiently decrease the time needed to initially generate the public key, time is bound to be linear with the design count of the signatures produced. Solutions are possible with some advanced key management: For example, a small sacrifice of signature length can be used to produce signature scheme with squared signature count — sign some other FMTSeq public key with a "master" key, and use the lower-level key to produce signatures from a triplet of lower level public key, lower level message signature, and "master" signature of the lower level public key.

2.2.5 Replacement of signature-intensive routines

From all this perspectives, FMTSeq us still not a viable option for interactive and online applications, especially for cases like SSL/TLS server and client authentication.

Given the CCA2-IND-secure and very efficient McEliece scheme that was described in previous chapter, one can easily replace common online authentication and identification routines.

Note that while following algorithms are illustrational, provided only for demonstration and should not be reproduced in practice without proper analysis, to the best of author's knowledge there is no significant security flaw that would completely rule out the possibility to use them for identification purposes.

In all cases, h denotes some suitable cryptographically secure hash function, and E, D respectively denote CCA2-IND scheme encryption and decryption from previous chapter.

Identification of peer using a pre-existing channel

Challenge Challenger sends $a = E(r)$ where r is some random string of sufficient length.

Identification Identifier returns $b = h(D(a))$, thus proving he has a decryption key and at the same time refusing to function as a decryption oracle.

Verification Challenger accepts if $b = h(r)$.

Authenticated question/answer protocol through insecure channel

Question Challenger sends $a = E(r, q, P)$ where r is sufficiently long random string, q is a question, and P is the public key of challenger.

Identification and answer Answering side calculates $(r, q, P) = D(a)$. It can look up whether owner of P is sufficiently privileged to receive the answer and return $b = E_P(h(r), A(q))$ where E_P is encryption for public key P and A is an answer-generating function.

Verification $(v, d) = D(b)$, answer in d is authenticated if $h(r) = v$.

Symmetric key exchange with client/server authentication

Client Use previous protocol to send a question r_1 consisting of random string.

Server Using the same protocol, answer with r_2 which is some other random string.

Both sides Symmetric key is $h(r_1 + r_2)$.

Chapter 3

Implementation of the cryptosystem

This chapter gives an overview of detailed structure and interfaces of the software implementation carried along with this thesis.

3.1 Overall structure

Software is meant to be an UNIX tool, ideologically compatible with the "design preimage" GnuPG [GPG] to the highest possible extent.

The programming language of implementation was chosen to be C++ for several reasons:

- Intensive data-crunching that happens when e.g. generating keys is usually best implemented as close to hardware as possible. C++ gives excellent coverage of both low-level programming needed for performance, and high-level access to structured data which is needed to keep the implementation "clear" and readable.

- Thanks to C++ STL library, memory allocation and similar bothersome programming work could be left for compiler to do correctly. That greatly simplified the implementation details, leaving more space and time for actual software.

- Many algorithms that are largely out of scope of this thesis (namely hash function implementations) but are needed for cryptosystem to actually work have readily available effective implementations in C or C++ available online for free.

- Programs in C/C++ are a kind of "UNIX standard", do not have any dependencies on compilers or large nonstandard run-time libraries, and can be made easily portable to other platforms.

Overall structure of the implementation is shown on the figure.

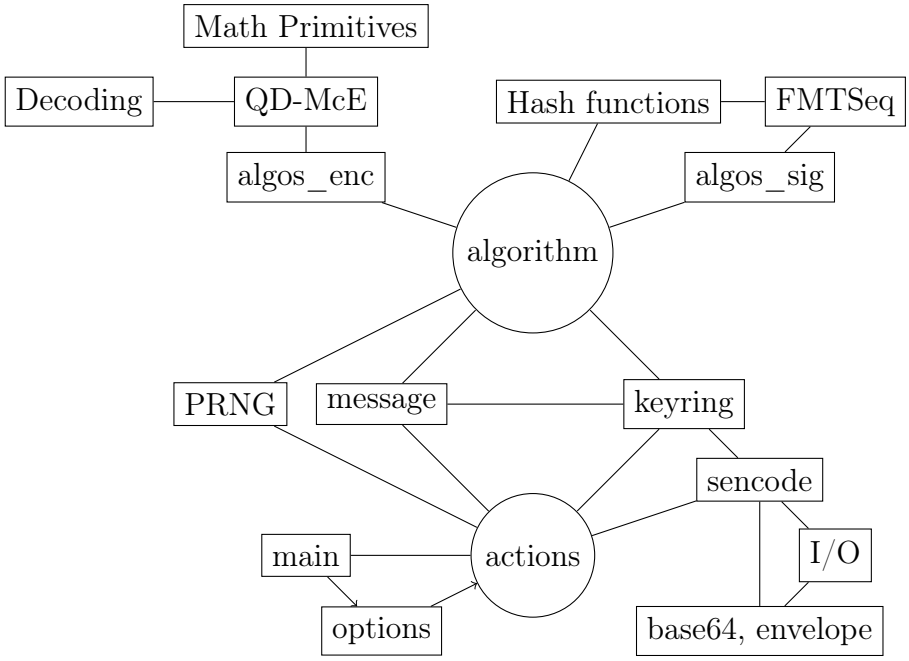


Figure 3.1: Internal organization of `codecrypt`

3.2 Mathematical primitives

3.2.1 Binary data

All binary data (binary vectors and matrices) are stored in a specialization of STL library container `std::vector<bool>` which provides both bit-packing (needed to save memory) and very simple access to individual bits.

Class `bvector` extends `std::vector<bool>` to provide mathematical operations and conversion to other data types. Binary matrices are extended similarly from `std::vector<bvector>`.

Only drawback of using the STL vector container to store binary data is a certain performance penalty for many operations. Single bit operations are needed to unpack and pack underlying words (which can get unnecessarily slow, but cannot be directly avoided), and STL provides no interface for doing operations in any other way. The best example of the problem is function `bvector::add()` that could use platform XOR instructions to add vectors word-by-word instead of bit-by-bit, speeding the process several times.

Because of the fact that this drawback did not bring up any performance bottleneck in final implementation, it was decided not to add redundant complexity at least until this becomes a serious problem.

3.2.2 Finite field calculations

Finite fields are represented as class `gf2m` and their elements are stored as simple integers in `uint`.

`gf2m` has one main purpose — to store (and serialize) the generating polynomial of finite field, and provide fast table-lookup-calculations in them.

While most of the functionality is already described in Chapter 1, there is an interesting observation on the speed of finite field calculations that has been

already made by several authors: As the log/antilog lookup tables are accessed extremely frequently in a random order, one can easily measure a performance drop that happens whenever the finite field is large enough for the lookup tables not to fit into CPU cache.[1] For this purpose, it is suggested not to work with very large finite fields; usually the $m = 18$ that fills around 2MB of CPU cache is the upper practical limit.

3.2.3 Polynomials

Polynomials are represented in class `polynomial`, which is an extension of `std::vector<uint>`, or "a vector of finite field elements". Apart from common mathematical operations, there are also implementations of Ben-Or algorithm and polynomial ring inversion.

3.2.4 McEliece implementations

There are three main McEliece-based cryptosystems implemented:

- The classical, impractical but working McEliece is represented in namespace `mce`. As with all other ciphers, the namespace contains a function `mce::generate()` that fills the contents of supplied `mce::privkey` and `mce::pubkey` with new keys generated according to parameters. Public key structure then provides encryption and signature verification functions, private key provides decryption and signing.

  The common pattern of having a namespace for cipher algorithm that contains `generate()`, `privkey` and `pubkey`, with priv/pub-keys exporting encryption and signature functions and functions that report available sizes of plaintext/ciphertext/hash/signature blocks has proven to be a very useful interface for subsequent implementation of larger structures atop such cryptographic primitives.

  The original McEliece cryptosystem is actually not used anywhere in the program as it is quite impractical when compared to other algorithms. The source code is meant mostly as a working reference for the inner structure of the cryptosystem.

- Niederreiter cryptosystem is implemented in namespace `nd`. Although not used anywhere else in the program, it contains source code that can be used to (very slowly) produce the "original" CFS signatures.

- Quasi-dyadic McEliece is implemented similarly in namespace `mce_qd`.

3.2.5 FMTSeq implementation

FMTSeq (and also several other parts of the software) calls for implementation of many different hash functions. For the purpose of resulting software, those have been imported from various program pieces available online. For simpler usage,

[1] This is only a parameter choice suggestion — actual measurement is not a part of this thesis.

all hash algorithms were wrapped in classes that are derived from functor class `hash_func`:

- RIPEMD-128 is in class `rmd128hash`, providing 128-bit hashes,
- Tiger hash func is `tiger192hash`, for 192-bit hashes,
- SHA-2 variants SHA-256, SHA-384 and SHA-512 reside in `sha256hash`, `sha384hash` and `sha512hash` providing their respective hash sizes.

Hash functions are always used in pairs: one that provides collision and inversion resistance for creating message digest and one that does not need to provide collision resistance for hashing the nodes of Merkle tree[2]. The actual pairs were set as

- RIPEMD-128 for tree hashes and SHA-256 for message digest, for 2^{128} attack complexity,
- Tiger hash and SHA-384 for 2^{192},
- SHA-256 with SHA-512 for 2^{256}.

FMTSeq itself is contained in a similar package as other algorithms, in namespace `fmtseq`. The only notable difference is that the signature algorithm actually *changes* the private key after each signature (which is from the nature of FMTSeq algorithm), so it needs special attention when called from other program parts.

3.2.6 Miscellaneous tools

- Generators of random numbers have been wrapped in virtual class `prng`. This simplifies the injection of generator primitive into various parts of the code, while also opening possibility to quickly change implementations of the actual generator.
- For the purpose of symmetric encryption that is needed for Fujisaki-Okamoto padding scheme, RC4 cipher was implemented in class `arcfour`. Implementation is notable for the fact that underlying RC4 permutation can be extended to any size using a template argument, which, for example, provides an interesting possibility to play with 16-bit-block RC4.[3]
- RC4 implementation is connected with `/dev/[u]random` operating system facilities in class `arcfour_rng` to provide a reliable random data source.

[2] see [NSW05] for explanation of security requirements on the functions

[3] Generally, usage of RC4 cipher is discouraged because its simplicity makes it extremely prone to be used in insecure way [WikiRC4]. For purposes of this software, all problematic requirements that would diminish security of RC4 (especially bad key management and not discarding the beginning of the generated stream) are easily handled correctly, so there is no need to unnecessarily implement anything more complex.

3.3 Keys and algorithms

PGP-like cryptography requires to strongly formalize the concepts of public and private keys, algorithms and user identities. For purposes of this thesis, abstractions of algorithm identifier and public key identifier are used.

3.3.1 Algorithm abstraction

In this context, "algorithm" formalized in abstract class `algorithm` is a tool that

- has an associated name, so that it can be easily decided what algorithm is suitable to work with given keys or usable to decrypt/verify a message,
- can report features that it can provide (e.g. encryption chain and/or digital signature chain)
- can, provided with matching keys, eventually *securely* encrypt/decrypt or sign/verify messages in *arbitrary* format.

Emphasized words "securely" and "arbitrary" basically imply that the algorithm must be a finalized version of the cryptosystem resistant to all kinds of attacks (not limited to attacks on underlying cipher security).

At the time of writing this thesis, the software has 6 algorithms available:

- `FMTSEQ128-SHA256-RIPEMD128`, `FMTSEQ192-SHA384-TIGER192` and `FMTSEQ256-SHA512-SHA256` for FMTSeq signatures providing respectively 128-bit, 192-bit and 256-bit security,
- `MCEQD128FO-SHA256-ARCFOUR`, `MCEQD192FO-SHA384-ARCFOUR` and `MCEQD256FO-SHA512-ARCFOUR` for QD-McEliece encryption with Fujisaki-Okamoto padding with the same three levels of security.

Both groups of algorithms provide a simple but effective padding scheme that protects against finding low-weight hash function collisions in case of FMTSeq, and almost all variants of chosen-ciphertext and chosen-plaintext attack in case of QD-McEliece.

3.3.2 Keys and KeyID

KeyID is an 256-bit identifier of each pubkey. It is defined as a result of SHA-256 hash applied to the sencode (see below) representation of the public key and has following purposes:

- It globally identifies given public key (e.g. the user) similarly to RSA Key IDs.[4]
- It can serve as a fingerprint, easily verifying the authenticity of keys.

[4]Hash collision chance, e.g. chance that two randomly generated different keys get the same KeyID, is absolutely minimal, at least when compared to 32-bit RSA Key IDs. For a broader image on what would be needed to produce a collision on 256-bit hash, compare the amount to the number of atoms in all observable universe, which is estimated to be around 2^{270}.

- It is contained in encrypted or signed message to specify the exact decryption or verification key.

For simplicity reasons, the software stores the keys as "keypairs" and "pubkeys" (there is intentionally no private-key-only storage, because some used cryptosystems it may be hard to derive the public key again (namely FMTSeq takes quite a time) and there is no reason to use the private key separately from public — private key owner will always need to have the public key to give it to other users). User can assign a name to any of those to easily remember who it belongs to or what it should be used for.

3.3.3 Message

Messages are abstracted as two separate classes: `encrypted_msg` and `signed_msg`. Those two give access to generalized encryption and signing functions using only algorithms and keys from keyring, and are easily serializable to actual messages meant to be sent and received.

3.4 Interface

3.4.1 User interface

User interface of the software is very closely based on the GnuPG command-line usage — using similar command-line options, users can

- encrypt, decrypt, sign and verify messages, for various users,
- manage the keyring (generate keys, view, export, import and delete them),
- set and use common names of the keys for easier access,
- redirect input and output to files or standard input/output,
- set various formatting flags, like ascii-armoring the input and output.
- view help for all options using `ccr --help`.

Apart from command-line options, user can set the environment variable `CCR_DIR` to the directory in that the software will search public and private keyrings.

3.4.2 sencode

Sencode is a data serialization format used for storage and transfer of structured data in the software. It is a simplified variant of the bencode encoding (see [Bencode]) that is most notably used for encoding .torrent files[5].

Sencode can be used to encode unsigned integers and byte strings, and to join those using lists:

[5] name "sencode" comes from the fact that it is like bencode, but instead of lists and dictionaries the common inner format usually looks like s-expressions

Integer is encoded as string `iXXXXe` where `XXXX` is replaced by decimal representation of the integer.

Byte string is encoded as `N:BBBBB` where `N` is a decimal integer denoting the length of the string, and `BBBBB` are the string data of length exactly `N` bytes.

List is encoded as `s...e` where dots are replaced by concatenated list content.

Sencode has two main advantages that led it for usage with the cryptosystem:

- The mapping between structured data and corresponding sencode string is *bijective*. This eliminates all forms of possible formatting inefficiency (as known from XML and other formats) and simplifies key comparison at storage. Moreover, knowing that the same data will have the exactly same sencode format everywhere allows to securely construct aforementioned KeyID just as a hash of the sencode representation of the key.

- It can store binary data directly as byte strings, making binary matrix or vector storage very efficient.

- Its extreme simplicity removes need for any cumbersome data format specifications, which had a great simplifying impact on the development.

Nearly every data-holding object in the software has `.serialize()` and `.unserialize()` methods that convert it to/from sencode tree representation. Using that, construction of actual communication protocols is extremely easy — messages are just serialized and string-encoded representations of `encrypted_msg` and `signed_msg` classes; disk storage and exported keys are similarly just sencode representations of the `keyring` object.

3.5 Reference guide for the software

For practical reasons, the software produced with this thesis was named "codecrypt" and given UNIX name `ccr`.[6] This section roughly describes how to install and use it on UNIX-like operating system.

3.5.1 Requirements

To compile `codecrypt`, user needs some recent version of a C++ compiler toolchain with standard libraries (or STL-compatible replacement). Any recent (2013) version of [GCC] or [Clang] is almost certainly sufficient.

Because of colex ranking algorithm, `codecrypt` uses the GNU Multiple Precision library [GMP] to compute combination numbers. All GNU-based systems are very likely to have GMP already installed, as it is used by many other software packages.

Packaging scheme used for `codecrypt` requires that some version of GNU Make is present on compiling system (at least for *convenient* compilation).

[6] For it is hard to work with a software piece without a name.

3.5.2 Installation

`codecrypt` is currently distributed only in the form of source code which can be obtained online from GitHub git repository, or from `.tar.gz` package.

To clone `codecrypt` source from GitHub and prepare it as a package, run following commands:

```
$ git clone git://github.com/exaexa/codecrypt.git codecrypt
$ cd codecrypt
$ ./autogen.sh
```

Archives with prepared sources can be found either on optical media attached to this thesis, or online at `http://e-x-a.org/codecrypt/files/`.

After preparing the source, "standard" installation procedure can be used to install the software to any system:

```
$ ./configure
$ make

# make install-strip     # as privileged user
```

Codecrypt should now be available for all users of the system in the form of `ccr` command.

If the user does not have needed software installation privileges, he can also use codecrypt by using the `configure` option `--prefix` and setting the environment variable `PATH` accordingly.

3.5.3 Quick usage tutorial

Users of `codecrypt` are advised to read `ccr --help` thoroughly. Following sequence of commented commands should demonstrate the similarity to GnuPG and give some basic insight into how `codecrypt` works:

```
#list available algorithms
ccr -g help

#create a keypair for signing
ccr -g fmtseq128 --name "John Doe"
#create a keypair for encryption
ccr -g mceqd128 --name "John Doe"

#watch the generated keys
ccr -K
ccr -k

#export own pubkeys for friends
ccr --export -a -o my_pubkeys.asc -F Doe

#see what public keys are in ascii-encoded file
ccr --import -na < friends_pubkeys.asc
```

```
#import the keys from the file and rename them to a better name
ccr -i -R friends_pubkeys.asc --name "Friendly Frank"

#encrypt and sign the file for Frank
ccr -se -r Frank < Document.doc > Message_to_frank.ccr

#decrypt and verify the reply
ccr -dv -o Decrypted_verified_reply.doc < Reply_from_frank.ccr

#explicitly rename public keys
ccr -m Frank -N "Unfriendly Frank"

#delete keys of everyone who is unfriendly
ccr -x Unfriendly
```

3.6 Evaluation of resulting software

3.6.1 Comparison with existing cryptosystems

To determine how the software performs, it was chosen to compare it from several perspectives with GnuPG package that — with some margin and a very simple point of view — does basically the same things: key generation, encryption and signatures.

Features like standard-compatibility or availability of high-level key handling operations (GnuPG provides automated certification, trust management, key expiration, compression and other helper tools) are not compared because those are simply not implemented in `codecrypt`.

Comparison is done in two steps: theoretical side-by-side comparison of used cryptosystem speeds and key sizes, and practical measurement of real time and space needed to run common cryptosystem operations.

For practical measurement, all operations were performed on parameters designed to provide around 2^{128} attack complexity. For `codecrypt`, that means usage of `FMTSEQ128-SHA256-RIPEMD128` algorithm for signing and `MCEQD128FO-SHA256-ARCFOUR` algorithm for encryption. GnuPG is, accordingly to parameter choice guidelines available at [NIST800-57], run with DSA-3072 signatures and RSA-3072 encryption.

For each cryptosystem, following measurements were done:

- time of encryption key generation and resulting key size,
- time of signature key generation and resulting key size
- encryption of small message (1kB of random data) with resulting message size,
- the same for large message (1MB of random data)
- signature of small message (1kB of random data) with resulting signature size,

- again the same for large (1MB) message.

All time measurements have been averaged from multiple attempts, discarding results that were visibly affected by some other variable (the time differed more than 20% from original average value).

Tests were performed on commonly available hardware (Intel Core Duo2 CPU) on a fairly standard Linux distribution, with both GnuPG and `codecrypt` compiled by GCC version 4.7.2, with compile flags `-O2 --fomit-frame-pointer -march=core2`. Last available version of GnuPG (2.0.19) was used for testing.

Theoretical comparison

Following table compares growth rates of all important cryptosystem parameters. Where not indicated, values have been taken from work referenced at algorithm description.

Property		GnuPG	codecrypt
Encryption	keygen time	$\mathcal{O}(n^4)$	$\mathcal{O}(nt)$
	public key size	$\mathcal{O}(n)$	$\mathcal{O}(nm) \doteq \mathcal{O}(n \log n)$
	encryption time	$\mathcal{O}(n^2)$	$\mathcal{O}(n \log n)$
	decryption time	$\mathcal{O}(n^2)$	$\mathcal{O}(nmt) \doteq \mathcal{O}(n \log n)$
	message size	$\mathcal{O}(n)$	$\mathcal{O}(n)$
	attack complexity	$\mathcal{O}(e^{n^{\frac{1}{3}}})$ [NIST800-57]	$\mathcal{O}(2^{\frac{mt}{2}})$
Signatures	keygen time	$\mathcal{O}(n^4)$	$\mathcal{O}(2^h n)$
	public key size	$\mathcal{O}(n)$	$\mathcal{O}(n)$
	encryption time	$\mathcal{O}(n^2)$	$\mathcal{O}(n + h)$
	verification time	$\mathcal{O}(n^2)$	$\mathcal{O}(n + h)$
	message size	$\mathcal{O}(n)$	$\mathcal{O}(n + h)$
	signature count	$\mathcal{O}(2^n)$ (basically ∞)	$\mathcal{O}(2^h)$
	attack complexity	$\mathcal{O}(2^{\frac{n}{2}})$ [Rho]	$\mathcal{O}(2^{\frac{n}{2}})$

Parameters n, m and t in the table have meaning that depends on described cryptosystem:

- In GnuPG column, n is the bit-length of RSA modulus or DSA prime number.
- in codecrypt encryption, n, m and t parameters are the Goppa code parameters respectively of code length, size of $\mathbf{GF}(2^m)$ and error correction capability.

- in codecrypt signatures, n is the count of message bits that are being signed, and h is the depth of FMTSeq hash tree.

Observation of the table clearly shows advantages and disadvantages of used cryptosystems: Biggest advantage of codecrypt algorithms is the speed of all operations except key generation time for FMTSeq, which is linearly dependent on the design signature count. Slight disadvantage is the encryption key size, which is several times larger than its RSA counterpart, and limited but easily expandable count of possible signatures.

Biggest disadvantage of GnuPG algorithms is their asymptotic complexity when one needs to scale security parameters. The worst is the key generation time — while quite reasonable for standard security parameters, to achieve 2^{256} attack complexity, [NIST800-57] implies that 15360-bit RSA keys are needed. Generating keys for such parameters takes a prohibitively big amount of time.

Practical comparison results

Table of practical measurement comparison is organized to lines with single tests. Each of them contains the exact UNIX shell commands used to do the measurement, and average value of the measurement.

`time` utility is used for measuring time, and only reported `user` time is counted — `system` time is not taken into measurement as it heavily depends on underlying operating system implementation, and `real` time is influenced by too much events (especially speed of I/O operations) to provide satisfactory measurements. Moreover, in case of interactive input, `real` time is completely unusable as a metric, because it mostly depends on speed of the user.

Key and message sizes were measured by piping the command output into `|wc -c`, which reports the size in bytes.

Random messages for encryption and signing were generated from standard UNIX random data device, using `head -c $SIZE /dev/urandom` and piping the output into tested software.

	codecrypt		GnuPG	
Encryption				
Key generator	`ccr --gen-key mceqd128 -N enc`	260ms	1951ms	`gpg -gen-key`
Public key size	`ccr -p -F enc`	4196B	1693B	`gpg -export enc`
1kB encrypted size	`ccr -e -r enc`	1826B	1489B	`gpg -e -r enc`
1kB encryption time	-"-	13ms	10ms	-"-
1MB encrypted size	-"-	1049kB	1049kB	-"-
1MB encryption time	-"-	335ms	101ms	-"-
1kB decryption time	`ccr -d`	52ms	130ms	`gpg -d`
1MB decryption time	-"-	372ms	200ms	-"-
Signatures				
Key generator	`ccr --gen-key fmtseq128 -N sig`	13930ms	3813ms	`gpg -gen-key`
Public key size	`ccr -p -F sig`	107B	1352B	`gpg -export sig`
signature size	`ccr -s -b signature`	4653B	96B	`gpg -s`
1kB signature time	-"-	10ms	140ms	-"-
1MB signature time	-"-	165ms	173ms	-"-
1kB verification time	`ccr -v -b signature`	4ms	14ms	`gpg -v`
1MB verification time	-"-	231ms	27ms	-"-

As seen in the table, codecrypt is only unnoticeably worse for encryption than GnuPG. Speed problems with encryption and decryption of 1MB messages are a result of quite ineffective implementation of raw-data-processing primitives, profiling showed that much of the time needed for encryption and decryption was spent in conversions among various internal data formats and sencode.

For simplicity, detached signatures were used to measure the digital signature performance – signature size is then easily determined from the detached file. Apart from the disadvantages resulting from aforementioned inefficiencies of FMTSeq (most notably the key creation time and signature size) codecrypt signatures seem to be extremely time-efficient when signing (which is supported by observations of asymptotic complexities), only suffering from the same implementation inefficiencies as encryption when working with signatures of large files.

3.6.2 Possible improvements

`codecrypt`, although working correctly, is far from being an optimized software package. There are several parts of the program that could benefit from extra research:

- Decryption of McEliece cryptosystem is, as seen in previous comparison, noticeably slower than encryption. Profiling determined that the slowness is caused mostly by the volume of computation needed for determining the syndrome from the codeword. Although some constant speedup could probably be achieved (most probably by finding a method to *effectively* cache the parity check matrix, saving $\mathcal{O}(nt)$ multiplications in $\mathbf{GF}(2^m)$), the decryption speed can be considered sufficient (it is still faster than RSA decryption) and this thesis does not aim to optimize any further.

- `bvector` class could be based on something that supports much faster word-by-word binary operations instead of manually unpacking boolean values from words that are hidden by STL implementation.

- Time needed to generate the FMTSeq keys is the most objectionable part of software performance. Profiling has revealed that much of it is spent only by computing hash functions and moving too much data around. Choosing some faster suitable hash functions could be very helpful for some cases. Unnecessary large data moves should be correctable by using smarter design and allocation of data objects.

- There are many small inefficiencies that result from attempts to create easily comprehensible code from limited amount of programmer time, most notably unnecessary data moves, copies and allocations when manipulating with STL objects. These are the only major cause of slowness of manipulating with larger data chunks from comparison above, and should be addressed before the software ever reaches some "production" stage.

Conclusion

This thesis has explored the possibilities of implementing post-quantum code-based cryptosystem. Resulting software, called `codecrypt`, connects all theoretical and practical knowledge gained, and brings following results and conclusions:

- By implementing all requirements that were stated in Introduction, it is the first generally available software package known to author that allows users to use quantum-computing-resistant algorithms for encryption and digital signatures. Chapter 3 presents general instructions on how to work with the software, potential users of the software can also easily adapt to usage of codecrypt if they have any previous knowledge of how very similar GnuPG software works.

  `codecrypt` is, in current state, equipped with several algorithms that provide encryption and digital signatures with attack complexity ranging from 2^{128} to 2^{256}. There is, to the best of author's knowledge, no quantum-computing algorithm that would diminish the security of the schemes except Grover search [Lom02]. Low asymptotic complexities (summarized in chapter 3) guarantee that all used cryptographic primitives can be very easily expanded to double the bit security, thus removing the effect of Grover search.

- As stated in introduction, a condensed but complete and easily readable description of internal algorithms used as encryption and signature trapdoors of `codecrypt` was presented in chapters 1 and 2 of this thesis.

- Using the software implementation, this thesis proves that post-quantum code-based encryption implementation is possible and highly practical. In chapter 3, it was shown that used encryption primitive easily outperforms currently used algorithms in several ways (notably encryption speed and scalability of design attack complexity), while staying comparable or only irrelevantly worse in other aspects.

- Post-quantum code-based signatures are possible and the experimental implementation of CFS cryptosystem works[1], but, as stated in Chapter 2, is highly impractical in any current form.

- From the software implementation, thesis finds that general post-quantum digital signatures are possible and highly practical using the FMTSeq

[1] although not directly accessible from the software, CFS implementation can be found in namespaces of `mce` and `nd` encryption primitives

signature scheme. Only really significant drawback of the scheme is limited number of signatures that can be performed using one generated private key. In chapter 2, this thesis suggests several methods to overcome this drawback.

- As can be seen in chapter 3, in comparison to GnuPG, `codecrypt` greatly simplifies most of the concepts used, which makes it easily modifiable for experimental purposes. Given the non-standardized but coherent and easily comprehensible structure, adding or testing possible new cryptographic algorithms in full-scale user-friendly environment is made very easy.

Further development and open questions

Development of the software package showed several points that might be viable as starting points for future research and work. Basically, there are three main possibilities of expanding the `codecrypt` software further:

- Standardization. Interoperability with ASN.1, OpenPGP, or other similar protocols could greatly increase the usability of the software.
- Creating a socket-security library based on assumptions from `codecrypt`. SSL and TLS protocols today provide a very significant building block of Internet security, therefore building a post-quantum alternative of those is important for the same reasons that motivated this thesis.
- In a recently published paper [Mis12], implementation of McEliece encryption atop MDPC[2] codes is described. Involved algorithms are much less complex and relatively faster than those for QD Goppa codes, while maintaining small public key size. If not proven insecure, MDPC is a good candidate for implementation of another possible encryption trapdoor.

Disclaimer

Author of the thesis has not received any significant formal training regarding cryptography, it is therefore strongly advised not to rely on `codecrypt` for any application where security would matter without carrying out own verification of assumptions used to build it.

[2] codes with Medium-Density Parity-Check matrix

Bibliography

[Sh97] Shor, Peter W. *Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer*. SIAM J. on Computing, 1997. Pages 1484–1509.

[LGS12] Landais, Gregory, and Sendrier. *CFS Software Implementation*. Cryptology ePrint Archive, Report 2012/132, 2012.

[GPG] `http://www.gnupg.org/`

[Lat] `http://en.wikipedia.org/wiki/Lattice-based_cryptography`

[NPat] `http://grouper.ieee.org/groups/802/15/pub/Patent_Letters/15.3/ntru%2015.3.pdf`

[NSig] `http://en.wikipedia.org/wiki/NTRUSign`

[Cz12] Czypek, Peter. *Implementing Multivariate Quadratic Public Key Signature Schemes on Embedded Devices*. Diploma Thesis, Chair for Embedded Security, Ruhr-Universität Bochum, 2012.

[Prom91] `http://www.eccpage.com/goppa_code.c`

[FP] `http://www.flexiprovider.de/#PQCProvider`

[HM] `https://www.rocq.inria.fr/secret/CBCrypto/index.php?pg=hymes`

[TTH] `http://en.wikipedia.org/wiki/Merkle_tree#Tiger_tree_hash`

[Bis10] Biswas, Bhaskar. *Implementational aspects of code-based cryptography*. Diss. PhD thesis, École Polytechnique, Paris, France, 2010.

[Hof11] Hofmann, Gerhard. *Implementation of McEliece using quasi-dyadic Goppa codes*. Bachelor thesis, 2011.

[Hey09] Stefan Heyse. *Code-based cryptography: Implementing the McEliece scheme in reconfigurable hardware*. Diploma thesis, Ruhr Universität Bochum, 2009.

[Str10] Falko Strenzke. *How to implement the public key operations in code-based cryptography on memory-constrained devices*. Cryptology ePrint Archive, Report 2010/465, 2010.

[McE78] R. J. McEliece, *A Public-Key Cryptosystem Based On Algebraic Coding Theory*. DSN Progress Report 42-44, 1978.

[MWS77] MacWilliams, Sloane. *The Theory of Error-Correcting Codes.* North-Holland mathematical library, 1977.

[PBGV92] Preneel, Bosselaers, Govaerts, Vandewalle. *A software implementation of the McEliece public-key cryptosystem.* Proceedings of the 13th Symposium on Information Theory in the Benelux, Werkgemeenschap voor Informatie- en Communicatietheorie, 1992. Pages 119–126.

[GaPa97] Gao, Panario. *Tests and constructions of irreducible polynomials over finite fields.* Foundations of Computational Mathematics. Springer Berlin Heidelberg, 1997. Pages 346–361.

[Gop70] Goppa, V. D. *A New Class of Linear Correcting Codes.* Probl. Peredachi Inf., 6:3, 1970. Pages 24–30.

[EOS06] Engelbert, Overbeck, Schmidt. *A Summary of McEliece-type Cryptosystems and their Security.* TU-Darmstadt, Department of Computer Science, Cryptography and Computer Algebra group, 2006.

[Ber70] Berlekamp, Elwyn R. *Factoring polynomials over large finite fields.* Mathematics of Computation 24.111, 1970. Pages 713–735.

[MiBa10] Misoczki, Barreto. *Compact McEliece Keys from Goppa Codes.* Escola Politéctnica, Universidade de Sao Paulo, Brazil, 2010.

[KI01] Kobara, Imai. *Semantically secure McEliece public-key cryptosystems-conversions for McEliece PKC.* Public Key Cryptography. Springer Berlin Heidelberg, 2001.

[FO99] Fujisaki, Okamoto. *Secure integration of asymmetric and symmetric encryption schemes.* Advances in Cryptology–CRYPTO'99. Springer Berlin Heidelberg, 1999.

[Fau11] Faugère, J-C., et al. *A distinguisher for high rate McEliece cryptosystems.* Information Theory Workshop (ITW), 2011 IEEE. IEEE, 2011.

[Aug08] Augot, Daniel, et al. *Sha-3 proposal: FSB.* Submission to NIST 2008, pages 81–85.

[GPLS07] Gaborit, Philippe, Lauradoux, Sendrier. *Synd: a fast code-based stream cipher with a security reduction.* Information Theory, 2007. ISIT 2007. IEEE International Symposium on. IEEE, 2007. Pages 186–190.

[Ste94] Stern, Jacques. *A new identification scheme based on syndrome decoding.* Advances in Cryptology–CRYPTO'93. Springer Berlin Heidelberg, 1994.

[PQCSlides] Slides on post-quantum cryptography by Paulo S. L. M. Barreto available from `http://www.larc.usp.br/~pbarreto/PQC-4.pdf`

[Rus] Ruskey, Frank. *Combinatorial generation.* Working Version (1j-CSC 425/520) (2003). Pages 66–68.

[Nie86] Niederreiter, Harald. *Knapsack-type cryptosystems and algebraic coding theory.* Problems of control and information theory 15.2, 1986. Pages 159–166.

[WikiRC4] Wikipedia article on RC4 cipher, `http://en.wikipedia.org/wiki/Rc4`

[Bencode] Wikipedia article on bencode, `http://en.wikipedia.org/wiki/Bencode`

[GCC] `http://gcc.gnu.org/`

[Clang] `http://clang.llvm.org/`

[GMP] `http://gmplib.org/`

[Xin90] Xinmei Wang. *Digital signature scheme based on error-correcting codes.* Electronics Letters 26, 1990. Pages 898–899.

[AW92] Alabbadi, Wicker. *Cryptanalysis of the Harn and Wang modification of the Xinmei digital signature scheme.* Electronics Letters 28, 1992. Pages 1756–1758.

[Ha06] Hamdi, Harari, Bouallegue. *Secure and fast digital signatures using BCH codes.* IJCSNS International Journal of Computer Science and Network Security 6(10), 2006. Pages 220–226.

[Ha09] Hamdi, Bouallegue, Harari. *Weakness on Cryptographic Schemes based on Chained Codes.* 2009 Third International Conference on Network and System Security, IEEE, 2009. Pages 574–581.

[CGG07] Cayrel, Gaborit, Girault. *Identity-based identification and signature schemes using correcting codes.* WCC. Vol. 7. 2007. Pages 69–78.

[CFS01] Courtois, Finiasz, Sendrier. *How to achieve a McEliece-based digital signature scheme.* Advances in Cryptology—ASIACRYPT 2001. Springer Berlin Heidelberg, 2001. Pages 157–174.

[BCMN11] Barreto, Paulo SLM, et al. *Quasi-dyadic CFS signatures.* Information Security and Cryptology. Springer Berlin Heidelberg, 2011.

[NSW05] Naor, Shenhav, Wool. *One-time signatures revisited: Have they become practical?* Cryptology ePrint Archive, Report 2005/442, 2005.

[NIST800-57] Barker, Barker, Burr, Polk, Smid. *Recommendation for Key Management — Part 1: General.* NIST Special publication 800-57, 2007.

[Rho] `http://en.wikipedia.org/wiki/Pollard%27s_rho_algorithm_for_logarithms`

[Lom02] Lomonaco, S. J. *Grover's quantum search algorithm.* Proceedings of Symposia in Applied Mathematics. Vol. 58, 2002.

[Mis12] Misoczki, et al. *MDPC-McEliece: New McEliece variants from moderate density parity-check codes.* IACR Cryptology ePrint Archive, 409, 2012.