How SAGE helps to implement Goppa Codes and McEliece PKCSs

Thomas Risse
DSI GmbH Bremen as well as *Institute of Informatics & Automation, IIA*
Faculty EEE & CS, Hochschule Bremen, University of Applied Sciences
risse@hs-bremen.de

***Abstract*—Common cryptographic methods, especially Public Key Crypto Systems, PKCS based on difficulty to factor large integers or to compute the discrete logarithm, commonly deployed today will not resist attacks using quantum computers once these are operational. However, there are alternatives like hash based digital signature schemes, lattice based cryptography, or multivariate-quadratic-equations public-key cryptography. Here, we present ways to judiciously implement code based cryptography exemplified by the McEliece PKCS based on Goppa Codes. We will show how the open source computer algebra system SAGE can guide the implementation of the PKCS say on FPGAs.**

***Index Terms*—Cryptography, Quantum Computer, Goppa-code, McEliece public key crypto system, FPGA**

I. Introduction

Today's prominent examples of public key crypto systems, PKCSs are RSA[1], Diffie-Hellman(-Merkle)[2], ElGamal[3], elliptic curve cryptography, ECC or the Buchmann-Williams key exchange. Using quantum computers, QCs these common PKCSs all are broken [1] once QCs become operational. The reason is that Shor's seminal algorithm [13] solves the integer factoring problem and the discrete logarithm problem on QCs very efficiently.

However, alternatives like McEliece PKCS, the Goldreich-Goldwasser-Halevi, GGH lattice analogue to McEliece, lattice based crypto systems, hash based digital signature schemes or multivariate PKCSs are believed to be QC resistant because they are not (jet) QC-broken [1].

Hence, it is only provident to replace existing PKCSs by QC resistant PKCSs. Right now we aim at implementing the McEliece PKCS on FPGAs. Here, we propose a dual version programming scheme, i.e. to assess implementation variants first using the open source System for Algebraic and Geometric Experimentation, SAGE [11] and to test the VHDL code by comparing its results with SAGEs results. Right now, we restrict ourselves to the SAGE version.

[1]R.Rivest, A.Shamir, L.Adleman (1978): A Method for Obtaining Digital Signatures and Public-Key Cryptosystems; Communications of the ACM 21 (2): 120 - 126

[2]W.Diffie, M.E.Hellman (1976): New Directions in Cryptography; IEEE Transactions on Information Theory IT-22 (6): 644 654

[3] T. ElGamal (1985): A Public-Key Crypto System and a Signature Scheme Based on Discrete Logarithms; IEEE Transactions Information Theory, IT-31, 4, pp469 – 472

II. Goppa Codes

The McEliece PKCS [6] is based on some error correcting code, e.g. a Goppa Code which can be defined as alternant generalized Reed-Solomon code. Now, let $\mathbb{F} = \mathbb{GF}_2$, $\Phi = \mathbb{GF}(q^m)$ and $g(x) \in \Phi[x]$ be the Goppa polynomial of degree t and let $L = \{\lambda_1, \dots, \lambda_n\} \subset \mathbb{GF}(q^m)$ the code locators with $g(\lambda_i) \neq 0$ for $i = 1, 2, \dots, n$. Then the linear subspace $\mathcal{C}_{\text{Goppa}}(L, g) = \{(c_1, \dots, c_n) \in \mathbb{GF}_2^n : \sum_{i=1}^n \frac{c_i}{x-\lambda_i} = 0 \bmod g(x)\}$ is a (classic, binary, linear) $[n, k, d]$ *Goppa code* with $k \geq n - mt$. The canonical parity matrix of the generalized Reed-Solomon code is $\mathbf{H}_{\text{gRS}} = \left(\lambda_j^{i-1}\right)_{i=1,\ j=1}^{\deg g-1, n} \text{diag}\left(\frac{1}{g(\lambda_1)}, \frac{1}{g(\lambda_2)}, \dots, \frac{1}{g(\lambda_n)}\right)$. Selecting a base of Φ and representing each entry of $\mathbf{H}_{\text{gRS}}$ as a column vector in $\mathbb{F}^m$ results in the parity matrix $\mathbf{H}_{\text{Goppa}}$ of $\mathcal{C}_{\text{Goppa}}$ with some corresponding generator matrix $\mathbf{G}_{\text{Goppa}} = \ker(\mathbf{H}_{\text{Goppa}})$.

A. encoding

Encoding is the easy part. In order to introduce the Python-based programming in SAGE, here we are rather explicit. We initialize our finite fields $\mathbb{F} = \mathbb{GF}_2$ and $\Phi = \mathbb{GF}(2^m)$,

```
F.<x> = GF(2); Phi.<x> = GF(2∧m));
```

choose the code locators $L = \{\lambda_i : i = 1, \dots, N\} \subset \Phi$, e.g. powers of variable x for some constant N

```
codelocators = [x∧i for i in range(N)];
```

specify a suitable Goppa polynomial g, e.g. by defining a function for some suitable constant K (in Python, indentation replaces begin-end brackets, and # starts a comment)

```
def goppapolynomial(F,z):
    # return a Goppa polynomial in z over field F
    X = PolynomialRing(F,repr(z)).gen();
    return X∧(N-K)+X+1;
```

check whether the Goppa polynomial is irreducible,

```
g = goppapolynomial(Phi,z);
if g.is_irreducible():
    print 'g(z) =',g,'is irreducible';
```

check that the code locators are not zeroes of g,

```
for i in range(N):
    if g(codelocators[i])==Phi(0):
        print 'alarm: g(alpha_'+str(i)+')=0';
```

and set up the parity matrix $\mathbf{H}_{\mathrm{gRS}}$ (Python is a formatting language, so a backslash indicates the continuation of a statement on the next line),

```
H_gRS = matrix([[codelocators[j]^(i) for j in \
                    range(N)] for i in range(N-K)]);
H_gRS = H_gRS*diagonal_matrix([ \
         1/g(codelocators[i]) for i in range(N)]);
```

From $\mathbf{H}_{\mathrm{gRS}}$ we construct $\mathbf{H}_{\mathrm{Goppa}}$

```
H_Goppa = matrix(F,m*H_gRS.nrows(),H_gRS.ncols());
for i in range(H_gRS.nrows()):
    for j in range(H_gRS.ncols()):
        be = bin(eval(H_gRS[i,j].int_repr()))[2:];
        be = '0'*(m-len(be))+be; be = list(be);
        H_Goppa[m*i:m*(i+1),j]=vector(map(int,be));
```

and compute a generator matrix $\mathbf{G}_{\mathrm{Goppa}}$ as right kernel of $\mathbf{H}_{\mathrm{Goppa}}$.

```
Krnl = H_Goppa.right_kernel();
G_Goppa = Krnl.basis_matrix();
```

Now we can specify the function `encode` which encodes information words u to code words $\mathbf{c} = \mathbf{u}\mathbf{G}_{\mathrm{Goppa}}$

```
def encode(u):
    return u*G_Goppa;
```

B. decoding

There are several relevant decoding algorithms, a variant based on the extended Euclidean algorithm, the Berlekamp-Massey and the Patterson algorithm. First, in the representation of [10] let us present the Berlekamp[4]-Massey[5] algorithm which computes a N-recurrence of $b(x)$.

```
Input: polynomial $b(x) \in \mathbb{F}[x]$, $0 < N \in \mathbb{N}$
Output: pair of polynomials $\big(\sigma_N(x), \omega_N(x)\big)$ over $\mathbb{F}$
$\sigma_{-1}(x) = 0; \sigma_o(x) = 1;$
$\omega_{-1}(x) = -x^{-1}; \omega_o(x) = 0;$
$\mu = -1; \delta_{-1} = 1;$
for($i = 0; i < N; i++$){
    $\delta_i$ = coefficient of $x^i$ in $\sigma_i(x)b(x)$;
    $\sigma_{i+1}(x) = \sigma_i(x) - (\delta_i/\delta_\mu)x^{i-\mu}\sigma_\mu(x);$
    $\omega_{i+1}(x) = \omega_i(x) - (\delta_i/\delta_\mu)x^{i-\mu}\omega_\mu(x);$
    if $\delta_i \neq 0$ && $2\,\mathrm{ord}(\sigma_i, \omega_i) \leq 1$ then $\mu = i$;
}
```

For given input b, the algorithm computes the N-recurrence (σ_n, ω_n) over $\mathbb{F}$ with smallest recurrence order. Applied to a syndrome b the algorithm returns the pair of error locator polynomial $\sigma(x)$ and error evaluator polynomial $\omega(x)$. In case of a binary code we need not compute the error evaluator polynomial. We can translate the algorithm more or less literally to SAGE keeping in mind that arrays in SAGE as in C, C++, JAVA etc. are numbered starting with index 0. For the sake of clarity, our implementation uses actual polynomials ω_i and σ_i instead of maintaining just two triples σ_{new}, σ_{old}, σ_{mu} and ω_{new}, ω_{old}, ω_{mu} of polynomials which are updated in each loop.

We use `F` $= \mathbb{GF}_2$, `Phi` $= \Phi = \mathbb{GF}(2^m)$ and `PR` where `PR` is the ring of polynomials over Φ in some variable, say z.

```
PR = PolynomialRing(Phi,'z');
```

Moreover, here we skip declaring and initializing the scalar variables `bigN`, `mu` and `flag` (the latter in a way representing the rational function $1/z$ so that the exception that ω_{-1} is not a polynomial is covered), the vectors of polynomials `sigma` and `omega` as well as the vector `delta` of elements in $\mathbb{GF}(2^m)$

```
def decode(y):
    s = H_gRS*y.transpose();
    if s==matrix(Phi,H_gRS.nrows(),1):
        return y;
    b = PR([s[_,0] for _ in range(s.nrows())]);
    # init $\sigma_{-1}$ and $\sigma_0$ as well as $\omega_{-1}$ and $\omega_0$
    sigma[-1+1] = PR(0); sigma[0+1] = PR(1);
    flag = 2*bigN; # exponent flags rational 1/z
    omega[-1+1] = z^(flag); omega[0+1] = PR(0);
    # init mu and delta
    mu = -1; delta[-1+1] = 1;
    for i in range(bigN):
        delta[i+1] = (sigma[i+1]*b).coeffs()[i];
        sigma[i+1+1] = sigma[i+1](z)-z^(i-mu)* \
          (delta[i+1]/delta[mu+1])*sigma[mu+1](z);
        if (omega[mu+1].degree()==flag):
            omega[i+1+1] = omega[i+1](z)- \
                (delta[i+1]/delta[mu+1])*z^(i-mu-1);
        else:
            omega[i+1+1] =omega[i+1](z)-z^(i-mu)*\
            (delta[i+1]/delta[mu+1])*omega[mu+1](z);
        ord = max(sigma[i+1].degree(), \
                1+omega[i+1].degree());
        if (delta[i+1]<>0)and(2*rord<=i):
            mu = i;
    ELP = sigma[bigN+1]; # ErrorLocatorPolynomial
    ee = vector(F,[0 for _ in range(n)]);
    for i in range(N):
        if (ELP(x^i)==Phi(0)): # an error occured
            ee[mod(N-i,N)] += 1; # in position N-i
    c_y = y+ee; return c_y;
```

The Berlekamp-Massey algorithm decodes any generalized Reed-Solomon code over some $\mathbb{GF}(p^m)$, not necessarily binary.

In contrast, the Patterson algorithm decodes only binary Goppa codes. It computes the syndrome $s(z)$ of a received word and then solves the key equation $\sigma(z)s(z) \equiv \omega(z) \bmod g(z)$ with $\omega(z) = \sigma'(z)$ by heavily exploiting the requirement that the code is binary. Then, the error locator polynomial can be split in even and odd powers of z such that $\sigma(z) = a^2(z) + z\,b^2(z)$, because $\Phi = \mathbb{GF}(2^m)$ has characteristic 2 so that in general $(\alpha+\beta)^2 = \alpha^2 + \beta^2$ for $\alpha, \beta \in \Phi$. And then the formal derivative gives $\omega(z) = b^2(z)$ so that the key equation becomes

$$s(z)\big(a^2(z) + z\,b^2(z)\big) \equiv b^2(z) \bmod g(z).$$

The Patterson algorithm [7] can then be described as follows, [3]. Initialize $w(z)$ such that $w^2(z) \equiv z \bmod g(z)$, cp. [4].

[4] Elwyn R. Berlekamp: Algebraic Coding Theory; Aegan Park Press 1984, Section 7.4

[5] James L. Massey: Shift-Register Synthesis and BCH Decoding; IEEE Trans. Inform. Theory. Vol. IT-15, Nr. 1, 1969, 122-127

Let $T(z)$ be the $g(z)$-inverse of the syndrome represented as the polynomial $s(z)$ and let $R(z) \equiv \sqrt{T(z)+z} \bmod g(z)$ be the $g(z)$-square root of $T(z)+z$, i.e. $R^2(z) \equiv (T(z)+z) \bmod g(z)$. Then $\sigma(z) = a^2(z) + z\,b^2(z)$ with $a(z) + b(z)\,R(z) \equiv 0 \bmod g(z)$ solves the key equation for (suitable) polynomials $a(z)$ and $b(z)$ representing even and odd parts of $\sigma(z)$. Here is why: (remember $T(z) = 1/s(z) \bmod g(z)$)

$$
\begin{aligned}
& a(z) + b(z)\,R(z) \equiv 0 \bmod g(z)\\
\Rightarrow\quad & a^2(z) + b^2(z)\,R^2(z) \equiv 0 \bmod g(z)\\
\Rightarrow\quad & a^2(z) + b^2(z)(T(z)+z) \equiv 0 \bmod g(z)\\
\Rightarrow\quad & a^2(z) + z\,b^2(z) \equiv b^2(z)\,T(z) \bmod g(z)\\
\Rightarrow\quad & s(z)\big(a^2(z) + z\,b^2(z)\big) = s(z)\sigma(z) \equiv b^2(z) \bmod g(z).
\end{aligned}
$$

Then, Pattersons algorithm looks like [3]:

```
**Input:** syndrome $s(z) \in F[z]$, $0 < N \in \mathbb{N}$
**Output:** polynomial $\sigma(z)$ over $\mathbb{F}$
initialize $w(z)$ with $w^2(z) \equiv z \bmod g(z)$;
set $T(z) = 1/s(z) \bmod g(z)$
if $(T(z) = z)$ set $\sigma(z) = z$
else { compute $R(z) = \sqrt{T(z)+z}$, and to do so:
    split $T(z)+z$ in even and odd parts $T_0$ and $T_1$
    such that $T(z)+z = T_0^2(z) + z\,T_1^2(z)$;
    for $R(z) = T_0(z) + w(z)\,T_1(z)$
    we then have $R^2(z) \equiv (T(z)+z) \bmod g(z)$;
    by the extended Euclidean algorithm compute
    $a(z)$ and $b(z)$ with $a(z) \equiv b(z)\,R(z) \bmod g(z)$;
    set $\sigma(z) = a^2(z) + z\,b^2(z)$;
}
```

Let us now implement each step of the algorithm separately. To initialize $w(z)$ we observe that the constant term of the irreducible Goppa polynomial $g(z)$ does not vanish. Therefore, $\gcd\big(g(z), z\big) = 1$ so that there is a polynomial $w(z)$ with $w^2(z) \equiv z \bmod g(z)$. Just split $g(z) = g_0^2(z) + z\,g_1^2(z)$ into even and odd parts $g_0(z)$ and $g_1(z)$ respectively and take $w(z) = g_0(z)g_1^{-1}(z) \bmod g(z)$ where $g_1^{-1}(z)$ is the $g(z)$-inverse of $g_1(z)$, [4]. As above, we represent the syndrome s as polynomial $s(z) \in \Phi[z]/g(z)$ with $\Phi = \mathbb{GF}(2^m)$.

The SAGE method `list` gives the list of the coefficients of some polynomial $p(z)$. So we can easily compute (the coefficients of) its even part $p_o(z)$ and odd part $p_1(z)$ by making the square root of the coefficients of $p \in \Phi$ the coefficients of p_o and p_1, alternately. The following function `split` solves this task generically.

```
def split(p):
    # split polynomial p over F into even part $p_o$
    # and odd part $p_1$ such that $p(z) = p_0^2(z) + z\,p_1^2(z)$
    Phi = p.parent()
    p0 = Phi([sqrt(c) for c in p.list()[0::2]]);
    p1 = Phi([sqrt(c) for c in p.list()[1::2]]);
    return (p0,p1);
```

Also, for the computation of $g(z)$-inverses of some polynomial p by the extended Euclidean algorithm, i.e. $1 = \gcd(p, g) = u\,p + v\,g$ such that $u = p^{-1} \bmod g$ for irreducible g, we provide the function `g_inverse`.

```
def g_inverse(p):
    # returns the $g(z)$ inverse of polynomial p
    (d,u,v) = xgcd(p,g);
    return u.mod(g);
```

Now, for initialization call `(g0,g1) = split(g)` to get g_0 and g_1 with $g(z) = g_0^2(z) + z\,g_1^2(z)$ and let SAGE compute w as g_0 times the $g(z)$-inverse of g_1.

```
(g0,g1) = split(g); w = g0*g_inverse(g1);
```

For each syndrome s let SAGE compute T as the $g(z)$-inverse of s, of course as long as $s \neq 0$, split $T+z$ as above into even part T_o and odd part T_1 such that $T(z) = T_o^2(z) + z\,T_1^2(z)$ and set $R(z) = T_0(z) + w(z)\,T_1(z)$.

```
T = g_inverse(s);(T0,T1) = split(T+z);R = T0+w*T1;
```

At last, a modified version of the extended Euclidean algorithm provides a solution to the equation $a(z) \equiv b(z)\,R(z) \bmod g(z)$. Then, SAGE computes this solution and the error locator polynomial $\sigma(z)$ by

```
(d,u,v) = xgcd(PR(1),PR(R.list()));
a = g*u; b = g*v; sigma = a∧2+z*b∧2;
```

The zeroes of the error locator polynomial $\sigma(z)$ then indicate which positions in the received word are to be corrected by bit flipping. These zeroes can be identified either by exhaustive search or by the more efficient Chien search.

C. testing and assessing our implementation so far

We test our encoding and decoding functions – using the Euclidean or Berlekamp-Massey or Patterson algorithm – by encoding randomly generated information words $\mathbf{u}$ to code words $\mathbf{c}$, by adding random errors $\mathbf{e}$ of Hamming weight t, by specifying received words $\mathbf{y} = \mathbf{c} + \mathbf{e}$, by decoding $\mathbf{y}$ to a code word $\mathbf{c}_y$ and by comparing $\mathbf{c}_y$ with the original $\mathbf{c}$.

```
u = vector(F,[randint(0,1) for _ in range(k)]);
c = encode(u); e = vector(F,n); # e = zero vector
for trial in range(t):
    j = randint(0,n-1); e[j] += 1;
y = c+e; c_y = decode(y);
if (c_y == c):
    print 'corrected received word == sent word';
else:
    print 'alarm: c_y =',c_y,'<>',c,'= c';
```

Solving the linear equation $\mathbf{u}\mathbf{G}_{\text{Goppa}} = \mathbf{c}_y$ determines the sent information word $\mathbf{u}$.

III. The McEliece PKCS

In order to define the McEliece PKCS choose three binary matrices

- $\mathbf{G}$, the $k \times n$ generator matrix of a $[n, k, 2t+1]$-Goppa-Code, i.e. $n = 2^m$, $k = n - mt$
- $\mathbf{S}$, a random non-singular $k \times k$ scrambler matrix
- $\mathbf{P}$, a random $n \times n$ permutation matrix

Then the triple $(\mathbf{S}, \mathbf{G}, \mathbf{P})$ is a private key with corresponding public key $\hat{\mathbf{G}} = \mathbf{SGP}$ with $t' \leq t$.

The plain text is partitioned into bit-blocks $\mathbf{u}$ of length k. Each block is Goppa coded to $\mathbf{c} = \mathbf{u}\hat{\mathbf{G}}$ $\mathbf{c}$ of length n is charged with $t' \leq t$ 1-bit-errors, i.e. modified to $\mathbf{y} = \mathbf{c} + \mathbf{e}$ with error vector $\mathbf{e}$.

Using the private key, the receiver decodes $\mathbf{yP}^{-1} = \mathbf{cP}^{-1} + \mathbf{eP}^{-1} = \mathbf{uSG} + \mathbf{eP}^{-1}$ per fast Goppa decoding to $\mathbf{uS}$ and per $\mathbf{uSS}^{-1}$ to $\mathbf{u}$.

Because decoding of linear codes is NP-complete [12] this PKCS is believed to be secure. Typical parameters in the original paper [6] were $n = 1024$, $k = 524$, $t = 50$; later improvements [2] used e.g. $n = 2960$, $k = 2288$, $t = 56$. The very long keys, e.g. in case of [2] 520047 bits, were at that time a drawback which dwindles away with the progress in computer technology today.

As in the case of say RSA, parameters also for the McEliece PKCS have to be chosen carefully [2].

Reference [3] illustrates exemplary issues when implementing the McEliece PKCS both on a low-cost 8-bit AVR microprocessor and on a Xilinx Spartan-3AN FPGA.

Here, we illustrate how easily SAGE can generate a pair of a private and corresponding public key. For any k, we generate a non-singular binary $k \times k$ scrambler matrix $\mathbf{S}$ by generating some binary $k \times k$ matrix and modifying elements until it gets non-singular.

```
S = matrix(GF(2),k, \
          [random()<.5 for _ in range(k∧2)]);
while (rank(S)<k):
    S[floor(k*random()),floor(k*random())] +=1;
```

The algorithm terminates because the probability of generating a non-singular matrix at random is greater than $\frac{1}{4}$ [5].

For any n, we generate a $n \times n$ permutation matrix $\mathbf{P}$ at random.

```
rng = range(n); P = matrix(GF(2),n);
for i in range(n):
    p = floor(len(rng)*random());
    P[i,rng[p]]=1; rng=rng[:p]+rng[p+1:];
```

Then, the public key is just $\mathbf{G}_{\text{pub}} = \mathbf{SG}_{\text{Goppa}}\mathbf{P}$.

```
G_pub = S*G_Goppa*P;
```

IV. Conclusion and Outlook

We illustrated how easily the McEliece PKCS can be implemented using SAGE. So it is near at hand to use the SAGE implementation for testing the VHDL implementation in the obvious two version test approach.

Moreover, the SAGE implementation allows us to assess different decoding algorithms – algorithms which solve the key equations – by their employability in VHDL, i.e. space and time requirements. Right now, we investigate

- whether to do inversion in $\mathbb{GF}(2^m)$ by table look up or by recursion [8] (a SAGE version is presented in [9])
- whether to compute square roots in $\mathbb{GF}(2^m)$ – as well as in $\mathbb{GF}(2^m)[z]/g(z)$ – by an algorithm [4] or by table look up
- whether to transform to and from normal bases in order to profit from the simple arithmetic using normal bases is more beneficial in terms of space and/or in terms of time.

According to expectation, each alternative is easily implemented in SAGE.

If the blocks are not chained then block-wise data parallelism can be exploited. Linear feedback shift registers are suitable for the matrix operations. A (very) deep pipeline for the decoding algorithm will increase latency but speed up the decryption accordingly, i.e. proportional to the number of stages, given enough hardware to prevent structure hazards. Of course, the SAGE implementation cannot answer this type of design questions which are inherent to the target FPGA which puts the SAGEs help to implement the McEliece PKCS into perspective.

References

[1] Daniel J. Bernstein, Johannes Buchmann, Erik Dahmen (Eds.): Post-Quantum Cryptography; Springer 2009

[2] Daniel J. Bernstein, Tanja Lange, Christiane Peters: Attacking and defending the McEliece cryptosystem; PQCrypto 2008, LNCS 5299, pp. 31–46, 2008 http://eprint.iacr.org/2008/318.pdf

[3] Thomas Eisenbarth, Tim Güneysu, Stefan Heyse, Christof Paar: MicroEliece: McEliece for Embedded Devices; CHES 2009: 49-64 www.crypto.rub.de/imperia/md/content/texte/publications/conferences/ches2009_microeliece.pdf

[4] K. Huber: Note on Decoding Binary Goppa Codes; Electronics Letters, 32:102-103, 1996

[5] Yuan-Xing Li, Da-Xing Li, Chuan-Kun Wu: How to Generate a Random Nonsingular Matrix in McEliece's PKCS; ICCS/ISITA '92, Singapore 1992, IEEE 268-269

[6] Robert McEliece: Public Key Cryptosystem based on Algebraic Coding; DSN Progress Report 42-44, January/February 1978, 114–116 www.cs.colorado.edu/∼jrblack/class/csci7000/f03/papers/mceliece.pdf

[7] N. J. Patterson: The Algebraic Decoding of Goppa Codes; IEEE Trans. on Information Theory, Vol IT-21, No 2, March 1975 203-207

[8] Vincent Rijmen: Effcient Implementation of the Rijndael S-box; was available www.esat.kuleuven.ac.be/∼rijmen/rijndael/sbox.pdf, alternatively www.comms.scitech.sussex.ac.uk/fft/crypto/rijndael-sbox.pdf or [9]

[9] Thomas Risse: SAGE, ein CAS auch und besonders für diskrete Mathematik; 8. Workshop *Mathematik für Ingenieure*, HS Wismar, 23.6.2010 www.weblearn.hs-bremen.de/risse/papers/MathEng8

[10] Ron M. Roth: Introduction to Coding Theory; Cambridge University Press 2006 www.cs.technion.ac.il/∼ronny/

[11] NN: System for Algebraic and Geometric Experimentation, SAGE www.SAGEmath.org together with servers like www.SAGEnb.org or e.g. http://SAGE.informatik.hs-bremen.de

[12] Arto Salomaa: Public Key Cryptography; Springer EATCS Monographs on Theoretical Computer Science Vol 23, 1990

[13] Peter W. Shor: Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer; Proc. 35^{th} Ann. Symposium on Foundations of Computer Science, Santa Fe, NM, Nov. 20–22, 1994 http://arxiv.org/pdf/quant-ph/9508027v2