HOW SAGE HELPS TO IMPLEMENT GOPPA CODES AND THE McELIECE PUBLIC KEY CRYPTO SYSTEM

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.

Keywords: Cryptography, Quantum Computer, Goppa-code, McEliece public key crypto system, FPGA

1 INTRODUCTION
Today’s prominent examples of public key crypto systems, PKCSs are RSA, Diffie-Hellman-Merkle, ElGamal, elliptic curve cryptography, ECC or the Buchmann-Williams key exchange. Using quantum computers, QCs these common PKCSs all are broken once QCs become operational. The reason is that Shor’s seminal algorithm 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 (yet) QC-broken.

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 and to test the VHDL code by comparing its results with SAGEs results. Right now, we restrict ourselves to the SAGE version.

2 GOPPA CODES
The McEliece PKCS is based on some error correcting code, e.g. a Goppa Code which can...
be defined as alternating generalized Reed-Solomon code. Now, let $F = \mathbb{GF}_2$, $\Phi = \mathbb{GF}(q^m)$ and $g(x) \in \Phi[x]$ be the Goppa polynomial of degree $t$ and let $L = \{\lambda_1, \ldots, \lambda_n\} \subset \mathbb{GF}(q^m)$ the code locators with $g(\lambda_i) \neq 0$ for $i = 1, 2, \ldots, n$. Then the linear subspace $C_{\text{Goppa}}(L, g) = \{(c_1, \ldots, c_n) \in \mathbb{GF}_2^n : \sum_{i=1}^n c_i \lambda_i^j \equiv 0 \mod g(x)\}$ is a (classic, binary, linear) $[n, k, d]$ Goppa code with code word length $n$, subspace dimension $k \geq n - mt$, and with minimal Hamming distance $d$. The canonical parity matrix of the generalized Reed-Solomon code is $H_{\text{gRS}} = (\lambda_i^{-1})_{i=1}^n \text{diag}(\frac{1}{\lambda_1}, \frac{1}{\lambda_2}, \ldots, \frac{1}{\lambda_n})$. Selecting a base of $\Phi$ and representing each entry of $H_{\text{gRS}}$ as a column vector in $F^m$ results in the parity matrix $H_{\text{Goppa}}$ of $C_{\text{Goppa}}$ with some corresponding generator matrix $G_{\text{Goppa}} = \ker(H_{\text{Goppa}})$.

### 2.1 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 $F = \mathbb{GF}_2$ and $\Phi = \mathbb{GF}(2^m)$,

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

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

```python
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)

```python
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,

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

check that the code locators are not zeroes of Goppa polynomial $g$,

```python
for i in range(N):
    if g(codelocators[i])==Phi(0):
        print 'alarm: g(alpha**i*str(1)+')=0';
```

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

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

From $H_{\text{gRS}}$ we construct $H_{\text{Goppa}}$,

```python
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 $G_{\text{Goppa}}$ as right kernel of $H_{\text{Goppa}}$,

```python
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 $c = uG_{\text{Goppa}}$.

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

### 2.2 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 [11] let us present the Berlekamp-Massey 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 $(\sigma_N(x), \omega_N(x))$ over $\mathbb{F}$

$$
\begin{align*}
\sigma_i(x) &= 0; \sigma_0(x) = 1;
\omega_i(x) &= -x^{i-1}; \omega_0(x) = 0; \\
\mu &= -1; \delta_{-1} = 1;
\end{align*}
$$

for $i = 0; i < N; i + +$)

```python
if g(codelocators[i])==Phi(0):
    print 'alarm: g(alpha**i*str(1)+')=0';
```

$$
\begin{align*}
\delta_i &= \text{coefficient of } x^i \text{ in } \sigma_i(x)b(x); \\
\sigma_{i+1}(x) &= \sigma_i(x) - (\delta_i/\delta_j)x^{i-j}\sigma_j(x); \\
\omega_{i+1}(x) &= \omega_i(x) - (\delta_i/\delta_j)x^{i-j}\omega_j(x); \\
\text{if } \delta_i \neq 0 \& \& \text{2ord}(\sigma_i, \omega_i) \leq 1 \text{ then } \mu = i;
\end{align*}
$$

---

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

For given input \( b \), the algorithm computes the recurrence \( \sigma_n, \omega_n \) 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 \( \omega_i \) and \( \sigma_i \) instead of maintaining just two triples \( \sigma_{\text{new}}, \sigma_{\text{old}}, \sigma_{\mu} \) and \( \omega_{\text{new}}, \omega_{\text{old}}, \omega_{\mu} \) of polynomials which are updated in each loop.

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

```python
def decode(y):
    s = H*gRS*y.transpose();
    if s==matrix(Phi,H*gRS.nrows(),1):
        return y;
    b = PR([s[i,0] for i 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);
        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 i in range(n)]);
            for i in range(N):
                if (ELP(x(i))==Phi(0)): # an error occurred
                    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 \( \sigma(z)s(z) \equiv \omega(z) \mod g(z) \) 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 \( \sigma(z) = a^2(z) + zb^2(z) \), because \( \Phi = \mathbb{GF}(2^m) \) has characteristic 2 so that in general \( (a + \beta)^2 = a^2 + \beta^2 \) for \( a, \beta \in \Phi \). And then the formal derivative gives \( \omega(z) = b^2(z) \) so that the key equation becomes

\[
s(z)(a^2(z) + zb^2(z)) \equiv b^2(z) \mod g(z).
\]
The Patterson algorithm \([8]\) can then be described as follows. Initialize \(w(z)\) such that \(w^2(z) \equiv z \mod g(z)\), cp. \([9]\). Let \(T(z)\) be the \(g(z)\)-inverse of the syndrome represented as the polynomial \(s(z)\) and let \(R(z) \equiv \sqrt{T(z)} + z \mod g(z)\) be the \(g(z)\)-square root of \(T(z) + z\), i.e. \(R^2(z) \equiv (T(z) + z) \mod g(z)\). Then \(\sigma(z) = a^2(z) + z b^2(z)\) with \(a(z) + b(z) R(z) \equiv 0 \mod g(z)\) solves the key equation for (suitable) polynomials \(a(z)\) and \(b(z)\) representing even and odd parts of \(\sigma(z)\). Here is why:

\[
\begin{align*}
  a(z) + b(z) R(z) &\equiv 0 \mod g(z) \\
\Rightarrow a^2(z) + b^2(z) R^2(z) &\equiv 0 \mod g(z) \\
\Rightarrow a^2(z) + b^2(z) T(z) + z &\equiv 0 \mod g(z) \\
\Rightarrow s(z)a^2(z) + z b^2(z) &\equiv s(z)\sigma(z) \equiv b^2(z) \mod g(z).
\end{align*}
\]

Then, Patterson’s algorithm looks like \([4]\):

**Input:** syndrome \(s(z) \in F[z], 0 < N \in \mathbb{N}\)

**Output:** polynomial \(\sigma(z)\) over \(F\)

1. Initialize \(w(z)\) with \(w^2(z) \equiv z \mod g(z)\);
2. Set \(T(z) = 1 / s(z) \mod g(z)\)
3. If \((T(z) = z)\) set \(\sigma(z) = z\)
4. Else\{compute \(R(z) = \sqrt{T(z)} + z\), and to do so:
   \(T(z) + z\) 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 \(R^2(z) \equiv (T(z) + z) \mod g(z)\);
   by the extended Euclidean algorithm compute
   \(a(z)\) and \(b(z)\) with \(a(z) \equiv b(z) R(z) \mod g(z)\);
   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, \(gcd(g(z), z) = 1\) so that there is a polynomial \(w(z)\) with \(w^2(z) \equiv z \mod g(z)\). 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 \(w(z) = g_0(z) g_1^{-1}(z) \mod g(z)\) where \(g_1^{-1}(z)\) is the \(g(z)\)-inverse of \(g_1(z)\). \([5]\). 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 \(p_0(z)\) and odd part \(p_1(z)\) by making the square root of the coefficients of \(p \in \Phi\) the coefficients of \(p_0\) and \(p_1\), alternately. The following function split solves this task generically.

```python
def split(p):
    # split polynomial p over F into even part p0
    # and odd part p1 such that p(z) = p0(z) + z p1(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 \(u = p^{-1} \mod g\) for irreducible \(g\), we provide the function \texttt{g_inverse}.

```python
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 \((g0,g1) = \text{split}(g)\) 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 \(g_1\).

\((g0,g1) = \text{split}(g); u = g0^*\text{g_inverse}(g1);\)

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 \(T_0\) and odd part \(T_1\) such that \(T(z) = T_0^2(z) + z T_1^2(z)\) and set \(R(z) = T_0(z) + w(z) T_1(z)\).

\(T = g^{-1}(s); (T0,T1) = \text{split}(T+z); R = T0+w*T1;\)

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

\(\sigma(z) = a^2 + z b^2;\)

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 \([3]\).
2.3 Testing (and Assessing) the 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 \( u \) to code words \( c \), by adding random errors \( e \) of Hamming weight \( t \), by specifying received words \( y = c + e \), by decoding \( y \) to a code word \( c_\hat{y} \) and by comparing \( c_\hat{y} \) with the original code word \( c \).

```python
u = vector(F,[randint(0,1) for _ in range(k)]); # \( c \) = encode(u); # \( e = \) 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 \( uG_{\text{Goppa}} = c_y \) determines the sent information word \( u \).

3 THE McELIECE PKCS

In order to define the McEliece PKCS choose three binary matrices

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

Then the triple \((S, G, P)\) is a private key with corresponding public key \( G = SP \) with \( t' \leq t \).

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

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

Because decoding of linear codes is NP-complete [14] this PKCS is believed to be secure. Typical parameters in the original paper [2] 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 [4] 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 \( S \) by generating some binary \( k \times k \) matrix and modifying elements until it gets non-singular.

\[
S = \text{matrix}(\text{GF}(2),k, \begin{bmatrix}
\text{random()}<& .5 \text{ for } _i \text{ in range}(k^2))
\end{bmatrix})
\]

\[
\text{while (rank}(S) < k): \quad S[[\text{floor}(k*\text{random()}),\text{floor}(k*\text{random()}))] += 1;
\]

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

For any \( n \), we generate a \( n \times n \) permutation matrix \( P \) at random

\[
\text{rng} = \text{range}(n); \quad P = \text{matrix}(\text{GF}(2),n);
\]

\[
\text{for } i \text{ in range}(n):
\text{p} = \text{floor}(\text{len(rng)}\text{rand()});
P[i,\text{rng}[p]]=1; \quad \text{rng=\text{rng[:p]}+\text{rng[p+1:]}};
\]

Then, the public key is just \( G_{\text{pub}} = SG_{\text{Goppa}}P \).

\[
G_{\text{pub}} = S*G_{\text{Goppa}*P};
\]

4 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 \( \text{GF}(2^n) \) by table look up or by recursion [9] (a SAGE version is presented in [10]).
whether to compute square roots in $\mathbb{GF}(2^m)$ – as well as in $\mathbb{GF}(2^m)[z]/g(z)$ – by an algorithm [5] 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


PKCS; ICCS/ISITA ‘92, Singapore 1992, IEEE 268-269


