Design and Implementation of Hardware Verification Modules for Post-Quantum Cryptographic Systems

Dibyatanoy Bhattacharjee

May 2017

Abstract

Most widely used cryptographic systems today rely on one of three mathematically hard problems: the integer factorization problem, the discrete logarithm problem or the elliptic-curve discrete logarithm problem. With the increasing viability of quantum computers, which can be used to run algorithms to solve these problems efficiently, it is becoming increasingly important to look into cryptographic schemes for the post-quantum era. In this project, I created a working software version of the modern Niederreiter cryptosystem (one of the main candidates for post-quantum cryptography or PQC), as well as a library of test modules for each step during the decryption phase of the algorithm, which can be used to verify correctness of a chip dedicated to PQC that Professor Szefer and his team are working on. This library will allow both the final output and the intermediate results from the different hardware modules to be verified for correctness.

1 Introduction

Most modern public-key cryptographic systems rely on one of the following three hardness assumptions: the integer factorization problem, the discrete logarithm problem, or the discrete logarithm problem using elliptic curves. While it is computationally impossible for present-day computers to break these cryptosystems, sufficiently powerful quantum computers running Shor’s algorithm could be used to break these systems. In recent years, attention has thus shifted to cryptographic methods resistant to quantum computing attacks. Two very popular methods with such properties are the McEliece cryptosystem and the Niederreiter cryptosystem, both of which use linear block codes. However, the use of these systems remains largely impractical because of 1) their large key sizes, and 2) their low information rates.

The goal of this project was to combine and adapt the optimizations known for the different steps in these cryptosystems, and to implement a working software version of the cryptosystem, complete with test modules for each step, which can be used to verify hardware (FPGA) implementations of these systems that Professor Szefer’s lab is currently trying to build.

2 Background and Implementation Details

The McEliece and Niederreiter cryptosystems are public-key cryptosystems based on algebraic coding theory. The basic idea of coding theory is that an encoder transforms a message word into a code word by adding redundancy, the goal being to protect against errors on a noisy channel. The decoder uses a decoding algorithm to correct errors which might have occurred during transmission.
2.1 Definitions and Terminology

The McEliece and Niederreiter cryptosystems uses linear block codes - a linear block code $C$ of length $n$ and dimension $k$ is a $k$-dimensional subspace of $\mathbb{F}_q^n$, where $\mathbb{F}_q$ is a finite field with $q$ elements. Such a code is called a $q$-ary code and is denoted as an $[n, k]_q$ code - binary codes are used for this system. In particular, both systems use irreducible Goppa codes.

Given a degree-$t$ polynomial $g(x) \in \text{GF}(p^m)$ (the Galois Field of order $p^m$), and a set of $n$ elements of $\text{GF}(p^m)$, $\{\alpha_0, \ldots, \alpha_{n-1}\}$, which are not zeroes of $g(x)$, a Goppa code able to correct $t$ errors is defined as the set of vectors $\mathbf{c} = \{c_0, \ldots, c_{n-1}\}$, with $c_i \in \text{GF}(p)$, such that

$$\sum_{i=0}^{n-1} \frac{c_i}{x - \alpha_i} \equiv 0 \mod g(x)$$

(1)

The $n$ elements $\{\alpha_0, \ldots, \alpha_{n-1}\}$ are called the “support” or “codelocators”. Because there can be at most $p^m$ elements in the field $\text{GF}(p^m)$, $n \leq p^m$.

The generator matrix for an $[n, k]_q$ code $C$ is a $k \times n$ matrix $G$ whose rows form a basis for $C$. That is,

$$C = \{xG \in \mathbb{F}_q^k\}$$

The parity-check matrix for an $[n, k]_q$ code $C$ is a $(n - k) \times n$ matrix $H$ such that:

$$C = \{c \in \mathbb{F}_q^n | Hc^T = 0\}.$$

Thus, the parity-check matrix $H$ can be used to test if a given vector $c$ is a part of the code $C$.

2.2 Key Generation, Encryption and Decryption

Both the McEliece and the Niederreiter Cryptosystems can be broadly divided into three steps: key generation (generation of the private and public key pair), encryption (conversion of the plaintext message to transmit into the ciphertext), and decryption (decoding the ciphertext to recover the plaintext). A brief outline of the steps is given below for each cryptosystem:

2.2.1 Key Generation

The key generation phase is identical for both the McEliece and the Niederreiter cryptosystems. The following steps are adapted, with some modifications, from [Sho+10].

**Require:** Parameters $n, m$ and $t$ Let $k = n - mt$

**Steps:**

1. Construct $\text{GF}(2^m)$ and select $n$ elements of the field $\{\alpha_0, \ldots, \alpha_{n-1}\}$.
2. Generate a random, monic (coefficient of highest degree term is 1), irreducible polynomial $g(x)$ with $\text{deg}(g) = t$ and coefficients in $\text{GF}(2^m)$.
3. Create matrices $Y$ and $Z$, as shown above.
4. Compute the parity check matrix $H = YZ$.
5. Generate a random $n \times n$ permutation matrix $P$.
6. Calculate a permutated control matrix $\tilde{H} = HP^T$.
7. Transform the $t \times n$ matrix $\tilde{H}$ over $\text{GF}(2^m)$ into an $mt \times n$ matrix $\tilde{H}_2$ over $\text{GF}(2)$. 

2
8. Bring $\tilde{H}_2$ into systematic form $\tilde{G} = [I_{mt}|R]$.

9. Return $R^T(McEliece)/R(Niederreiter)$ as the public key, and $(P, g(x))$ as the private key.

The expanded public key is $G = [R^T|I_k]$ for McEliece, and $H = [I_{mt}|R]$ for Niederreiter, but we only need to transmit the left part of it, reducing the key size. Note that an alternative to generating a permutation matrix, $P$, is to randomly permute the elements of the support $\{\alpha_0, \ldots, \alpha_{n-1}\}$ [HG13]. This would mean that the support would need to be a part of the private key instead of the permutation matrix, so the difference in size of the private key is negligible (since an $n \times n$ permutation matrix can be represented as a list of length $n$).

Step 8 requires bringing matrix $\tilde{H}_2$ into systematic form. However, not all matrices can be systematized - in fact, there is about a 30% chance that a matrix can be systematized. In that case, the key generation procedure is repeated, with a different seed (so that the random matrix $P$ is different).

2.2.2 Encryption

The encryption procedure for the McEliece and Niederreiter cryptosystems are different. Both methods are described below. In our implementation, we chose to use the Niederreiter cryptosystem (the reason for this choice is explained in section 3).

McEliece Encryption

Require: $k$-bit plaintext $m$; public key $R^T$

Steps:

1. Expand the public key to obtain the generator matrix $G = [R^T|I_k]$.
2. Generate a random error vector $e$ of length $n$ and weight $t$ (having exactly $t$ ones).
3. Calculate and return the ciphertext $c = mG \oplus e$.

Niederreiter Encryption

Require: $k$-bit plaintext $m$; public key $R$

Steps:

1. Generate a random length $n$ vector $e$ of weight $t$. This will be used as a private key for symmetric encryption.
2. Use symmetric encryption with private key $e$ to obtain the ciphertext $r$ from message $m$.
3. Expand the public key into the parity check matrix $H = [I_{mt}|R]$.
4. Compute and return the syndrome $c = He^T$ and $r$.

Note that in the Niederreiter cryptosystem, the ciphertext returned is actually a syndrome $c$ [HG13], along with the symmetrically encrypted message $r$ (with $e$ as the symmetric key used). An alternative to this is to encode the message $m$ into a word with length $n$ and constant weight $t$ [HG13]. However, this imposes constraints on the size of the message $m$ and its weight, which can be avoided by using symmetric key encryption as described here.

2.2.3 Decryption

Decryption of the ciphertext for both the Niederreiter and the McEliece cryptosystems involves calculating an error-locator polynomial $\sigma(x)$ from the syndrome, the roots of which give the positions of the erroneous
bits in the error vector (or, in the case of Niederreiter, the position of 1’s in the private key for symmetric encryption).

There are three methods we considered for calculating the error locator polynomial $\sigma(x)$: Patterson’s algorithm [Pat75], the Berlekamp-Massey (BM) algorithm [EVB99], and the Berlekamp-Massey-Sugiyama (BMS) algorithm [HG13]. Out of these three, only the BM algorithm runs for a fixed number of iterations, irrespective of the weight of the error vector. This prevents an attacker from carrying out timing attacks (and other side-channel attacks) on the system by feeding it error vectors of different weights.

One disadvantage of the BM algorithm is that it can only correct up to $t^2$ errors given a goppa polynomial of degree $t$. The way to solve this problem is to use the fact that in $\mathbb{GF}(2)$, both $g(x)$ and $g^2(x)$ produce equivalent Goppa codes, as mentioned in [HG13], so $g^2(x)$ can be used in decryption to work on a syndrome of double the size.

In the Niederreiter cryptosystem, the ciphertext $c$ from the encryption phase is actually a syndrome of length $t$, converted into binary (the $\mathbb{GF}(2)$ field). To convert it into a doubled syndrome, the doubled parity check matrix, $H_2$, computed using $g^2(x)$, must be used. Note that $H(c|0^{n-mt})^T = [I_{mt}|R][c|0^{n-mt})^T = c$, (where $0^{n-mt}$ denotes a string of $(n-mt)$ 0’s since $c$ is of length $mt$). This means that $(c|0^{n-mt})$ and $e$ both belong to the same coset, so they have the solution $e$. Due to the equivalence of Goppa codes over $g(x)$ and $g^2(x)$, this means that $s_2 = H_2(c|0)^T$ has the same solution $e$ as $He^T$. Note that $H_2$ has dimensions $2mt \times n$ and $(c|0^{n-mt})$ has dimensions $n \times 1$, so $s_2$ is a syndrome of length $2t$, corresponding to a syndrome polynomial of degree $(2t - 1)$.

The algorithm for decryption in the Niederreiter cryptosystem is presented below:

**Require:** permutation matrix $P$, goppa polynomial $g(x)$, ciphertext $c$ from encryption

**Steps:**

1. Compute $\tilde{c} = Pc^T$
2. Calculate the doubled parity check matrix $H_2$ using the polynomial $g^2(x)$.
3. Compute the doubled syndrome $s_2 = H_2(c|0^{n-mt})$.
4. Use BM algorithm to compute the error locator polynomial $\sigma(x)$ of degree $t$.
5. Evaluate the $t$ roots of $\sigma(x)$ - the positions of the roots in the support $\{\alpha_0, \ldots, \alpha_{n-1}\}$ gives the positions of the 1’s in the permuted error vector $\tilde{e}$.
6. De-permute the error vector to obtain $e = P^T \tilde{e}^T$.

Once the vector $e$ has been recovered, it can be used as the symmetric key to decrypt $r$ and obtain the original plaintext $m$.

Note that in step 4, the modification to the BM algorithm that does not use inversions was used [You91], thus preventing modular inversion operations in $\mathbb{GF}(2^m)$, which are expensive in hardware.

### 3 Choice of Algorithms

The parameters used for the system are $n = 6960$, $m = 13$, $t = 119$, according to the recommendations in [Aug+15].

In this project, the Niederreiter cryptosystem was chosen over the classical McEliece cryptosystem. In the McEliece cryptosystem, the size of the ciphertext is $n$ bits, while the Niederreiter cryptosystem produces ciphertexts of smaller length - $(n - k)$, where $k = mt$. In addition, the Niederreiter cryptosystem moves the
syndrome calculation part to the encryption stage (which still remains relatively fast), so that the decryption phase is shorter and the times for the encryption and decryption phases are more balanced.

A choice also exists for the syndrome decoding algorithm used in decryption. Out of the 3 choices we considered - Patterson’s algorithm, BMS algorithm, and BM algorithm, only the BM algorithm guarantees a constant number of iterations irrespective of the weight of the error vector, thus preventing timing attacks. In addition, both the Patterson’s and BMS algorithms require the use of the Extended Euclid Algorithm, which involves polynomial inversions. However, the BM-without-inversion algorithm from [You91] used in this project eliminates the need for such polynomial inversions. These properties make the BM-without-inversion algorithm the most suitable algorithm for decoding. Other methods, such as list-decoding algorithms, may be explored in the future for potential advantages over the BM-without-inversion algorithm.

4 Deliverables

The deliverables include a full working software implementation of the Niederreiter cryptosystem, along with test modules for each part of the decryption phase, written in Sage. These modules are intended to be used for verification of the hardware/ FPGA implementations of the cryptosystem, and thus the inputs and outputs are in the form of binary files. A brief description of each module in the decryption phase is provided below (the test modules for encryption already exist and can be provided upon request from Professor Szefer’s lab).

1. **Vector-vector multiplication:** This module is used to check the computation of a dot product between a vector in field $\mathbb{GF}(2^m)$, and another vector in field $\mathbb{GF}(2)$. Such a module is required at many stages of the pipeline.

2. **Systematic matrix - vector multiplication:** This module is used to check computation of the product of a systematic matrix in $\mathbb{GF}(2)$ with a vector in $\mathbb{GF}(2)$. The matrix must have the form $[I|K]$, i.e., a systematic matrix. This module uses the previous vector-vector multiplication module repeatedly to obtain the final result vector.

3. **Doubled syndrome calculation:** This module is used to check the calculation of a doubled syndrome (steps 1-3 of Niederreiter decryption) given the ciphertext from Niederreiter encryption and matrix $P$. The rows of the doubled parity check matrix are calculated on the fly, mimicking the hardware implementation of this module.

4. **BM algorithm iteration:** This module is used to check the result of one iteration of the BM-without-inversion algorithm as described in [You91]. The module returns the intermediate values $\sigma(x), \beta(x), l, \delta(x)$ after one iteration of the algorithm. There is also an input generator file for this module to generate valid input values.

5. **Complete BM algorithm:** This module is used to check the correctness of the error-locator polynomial $\sigma(x)$ returned by the BM-without-inversion algorithm. An input generator file for this module is used to generate valid input values.

The evaluation of the roots of the error-locator polynomial is done using an additive FFT module that is also used in encryption, so this module already has verification code. All code used for this project is available from the project site.
5 Acknowledgements

A special thanks to Professor Jakub Szefer for helping me with my work even with his busy schedule, and to Wen Wang for all her help and support in understanding and implementing the algorithms.

References


