FPGA-based Niederreiter Cryptosystem using Binary Goppa Codes

Wen Wang[1], Jakub Szefer[1], and Ruben Niederhagen[2]

[1] Yale University, New Haven, CT, USA
{wen.wang.ww349, jakub.szefer}@yale.edu
[2] Fraunhofer SIT, Darmstadt, Germany
ruben@polycephaly.org

Abstract. This paper presents an FPGA implementation of the Niederreiter cryptosystem using binary Goppa codes, including modules for encryption, decryption, and key generation. We improve over previous implementations in terms of efficiency (time-area product and raw performance) and security level. Our implementation is constant time in order to protect against timing side-channel analysis. The design is fully parameterized, using code-generation scripts, in order to support a wide range of parameter choices for security, including binary field size, the degree of the Goppa polynomial, and the code length. The parameterized design allows us to choose design parameters for time-area trade-offs in order to support a wide variety of applications ranging from smart cards to server accelerators. For parameters that are considered to provide "128-bit post-quantum security", our time-optimized implementation requires 966,400 cycles for the generation of both public and private portions of a key and 14,291 cycles to decrypt a ciphertext. The time-optimized design uses only 121,806 ALMs (52% of the available logic) and 961 RAM blocks (38% of the available memory), and results in a design that runs at about 250 MHz on a medium-size Stratix V FPGA.

Keywords: post-quantum cryptography, code-based cryptography, Niederreiter cryptosystem, FPGA, hardware implementation.

1 Introduction

Arguably today's most wide-spread asymmetric cryptographic algorithms are the Rivest-Shamir-Adleman (RSA) cryptosystem, Diffie-Hellman key exchange (DH), and a variety of primitives from the field of Elliptic-Curve Cryptography (ECC), e.g., ECDSA, EdDSA, ECDH, etc. These cryptosystems are based on the hardness of the integer-factorization problem and the discrete-logarithm problem. Using today's computing systems, no efficient algorithms for solving these problems are known. However, the picture changes drastically if quantum computers are taken into account. In the 1990s, Shor proposed algorithms that can solve both the integer-factorization problem and the discrete-logarithm problem in polynomial time on a quantum computer [23,24]. In order to provide

Permanent ID of this document: 939f29123f6853e858d367a6a143be76.
Date: 2018.05.18.

alternatives to the threatened schemes, the field of Post-Quantum Cryptography (PQC) emerged in the 2000s and has received increased attention recently, most noticeably due to a standardization process for PQC schemes started by NIST in 2017 [7].

Currently, there are five categories of mathematical problems that are under investigation for PQC: code-based systems, lattice-based systems, hash-based systems, systems based on multivariate polynomial equations, and systems based on supersingular isogenies of elliptic curves [4,22]. Each of these categories has advantages and disadvantages. They vary in the performance measures (sizes of public and private keys, sizes of ciphertext and key-exchange messages, computational cost, etc.) and in maturity: some schemes (e.g., some code-based schemes and hash-based signature schemes) are considered well-understood and there is a general agreement on the required security parameters while other schemes are more recent and the exact security that they provide is yet under investigation.

Conservative and well-understood choices for code-based cryptography are the McEliece cryptosystem [18] and its dual variant by Niederreiter [19] using binary Goppa codes. In this paper, we focus on the Niederreiter cryptosystem. This cryptosystem has relatively large public keys of up to 1MB for roughly 256-bit classical security (corresponding to "128-bit post-quantum security" meaning that a quantum computer needs to perform at least 2^{128} "operations" using the best known attacks) using parameters proposed in [2]. There are more efficient PQC schemes than Niederreiter with binary Goppa codes. However, some of these schemes exhibit weaknesses that restrict their application to certain use-cases (e.g., Niederreiter with QC-MDPC codes instead of binary Goppa codes is affected by decoding errors [13] which restricts their use to ephemeral applications without long-term usage of keys) while how to choose security parameters for some schemes is challenging (e.g., for some lattice-based schemes that have a security reduction, parameters need to be chosen either based on best-known attacks or based on the non-tight security reduction, which results in a dilemma of choosing either more efficient or more reliable parameters [1]).

The large public keys of the Niederreiter cryptosystem using binary Goppa codes make it particularly troublesome for use in embedded systems (due to strong restrictions on resource usage) and in server scenarios (given a large number of simultaneous connections). In both cases, hardware acceleration can help to improve the performance — either by providing a low-area, power efficient crypto core in the embedded scenario or by providing a large, latency or throughput optimized crypto accelerator for the server scenario. Therefore, we describe and evaluate an FPGA implementation of this cryptosystem. Our FPGA implementation can be tuned in regard to performance and resource usage for either low-resource usage in embedded systems or high performance as accelerator for servers. Furthermore, we provide a generic implementation that can be used for different performance parameters. This enables us to synthesize our design for the above mentioned 256-bit security parameters and also smaller parameter sets for comparison with prior art. For a given set of parameters, i.e. security level, the design can be further configured to trade-off performance and area, by

changing widths of data paths, memories, and other parameters inside the design, without affecting the security level. All of the parameters can be configured for key generation, encryption, and decryption.

Inspired by the confidence in the code-based cryptosystems, there are a few hardware implementations of different variants of these cryptosystems, e.g., [14,17,26]. Most of the work only focuses on the encryption and decryption parts of the cryptosystem due to the complexity of the key generation module. Moreover, none of the prior designs are fully configurable as ours nor do they support the recommended "128-bit post-quantum security" level. We are aware of only one publication [26] that provides the design of a full McEliece cryptosystem including key generation, encryption and decryption modules. However, their design only provides a 103-bit classical security level, which does not meet the currently recommended security level for defending against quantum computers. More importantly, the design in [26] is not constant-time and has potential security flaws. For example, within their key generation part, they generate non-uniform permutations, and within the decryption part, they implement a non-constant-time decoding algorithm. Note that our work focuses on a design that can defend against timing side-channel attacks due to its constant-time implementation. However, other types of side-channel attacks are out of scope of this work. A detailed comparison with related work is presented in Section 5.

Contributions. This paper presents the first "128-bit post-quantum secure", constant-time, efficient, and tunable FPGA-based implementation of the Niederreiter cryptosystem using binary Goppa codes. The contributions are:

- full cryptosystem with tunable parameters, which uses code-generation to generate vendor-neutral Verilog HDL code,
- new hardware implementation of merge sort for obtaining uniformly distributed permutations,
- new optimization of the Gao-Mateer additive FFT for polynomial evaluation,
- hardware implementation of a constant-time Berlekamp-Massey decoding algorithm, and
- design testing using Sage reference code, iVerilog simulation, and output from real FPGA runs.

2 Niederreiter Cryptosystem

The first public-key encryption scheme based on coding theory was proposed in 1978 by McEliece [18], known as the McEliece public-key cryptosystem. In 1986, Niederreiter proposed a variant of the McEliece cryptosystem that uses a parity check matrix for encryption instead of a generator matrix as used by McEliece. Furthermore, Niederreiter proposed to use Reed-Solomon codes, which were later shown to be insecure [27]. However, the Niederreiter cryptosystem using binary Goppa codes remains secure and the Niederreiter cryptosystem has been shown to be equivalent (using corresponding security parameters) to the McEliece cryptosystem [15].

The private key of the Niederreiter cryptosystem is a binary Goppa code $\mathcal{G}$ that is able to correct up to t errors. It consists of two parts: a generator, which is a monic irreducible polynomial $g(x)$ of degree t over $\mathrm{GF}(2^m)$, and a support, which is a random sequence of n distinct elements from $\mathrm{GF}(2^m)$. The public key is a binary parity check matrix $H \in \mathrm{GF}(2)^{mt \times n}$, which is uniquely defined by the binary Goppa code. To reduce the size of the public key, the matrix H of size $mt \times n$ can be compressed to a matrix $K \in \mathrm{GF}(2)^{mt \times k}$ of size $mt \times (n - mt)$ with $k = (n - mt)$ by computing its systematic form. This is often called "modern Niederreiter" and can also be used for the McEliece cryptosystem. For encryption, the sender encodes the message as a weight-t error vector e of length n. Then e is multiplied with the public parity check matrix H and the resulting syndrome is sent to the receiver as the ciphertext c. For decryption, the receiver uses the secret support and the generator to decrypt the ciphertext in polynomial time using an efficient syndrome decoding algorithm of $\mathcal{G}$. If neither the support nor the generator is known, it is computationally hard to decrypt the ciphertext, given only the public key H. The Niederreiter cryptosystem has performance advantages over the McEliece system if it is used as a key-encapsulation scheme, where a symmetric key is derived from the weight-t error vector e. The Niederreiter cryptosystem with properly chosen parameters is believed to be secure against attacks using quantum computers.

Security Parameters. The PQCRYPTO project [21] gives "initial recommendations" for several PQC schemes. For McEliece and Niederreiter using binary Goppa codes, they recommend to use a binary field of size $m = 13$, adding $t = 119$ errors, code length $n = 6960$, and code rank $k = n - mt = 6960 - 13 \cdot 119 = 5413$ for "128-bit post-quantum security" [2]. More precisely, these parameters give a classical security level of 266-bit (slightly overshooting 256-bit security); they were chosen to provide maximum security for a public key size of at most 1 MB [6]. We use these recommended parameters as primary target for our implementation. However, since our design is fully parameterized, we can synthesize our implementation for any meaningful choice of m, t, n, and k for comparison with prior art (see Section 5).

2.1 Algorithms

There are three main operations within the Niederreiter cryptosystem: key generation, encryption and decryption. Key generation is the most expensive operation; it is described in Algorithm 1. The implementation of the key generator has been described in detail in [28]. To generate a random sequence of distinct field elements, [28] presents a low-cost Fisher-Yates shuffle module which generates a uniform permutation. However, the runtime of the permutation module in [28] depends on the generated secret random numbers. This non-constant-time design of the permutation module might have vulnerabilities which enable timing side-channel analysis. In our work, we present a merge sort module, which generates a uniform permutation within constant time, as described in Section 3.1.

Within the Niederreiter cryptosystem, the ciphertext is defined as a syndrome, which is the product between the parity check matrix and the plaintext.

Algorithm 1: Key-generation algorithm for the Niederreiter cryptosystem.

Input : System parameters: m, t, and n.
Output: Private key $(g(x), (\alpha_0, \alpha_1, \ldots, \alpha_{n-1}))$ and public key K.

1 Choose a random sequence $(\alpha_0, \alpha_1, \ldots, \alpha_{n-1})$ of n distinct elements in $\mathrm{GF}(2^m)$ as support.
2 Choose a random polynomial $g(x)$ as generator such that $g(\alpha) \neq 0$ for all $\alpha \in (\alpha_0, \ldots, \alpha_{n-1})$.
3 Compute the $t \times n$ parity check matrix

$$H = \begin{bmatrix} 1/g(\alpha_0) & 1/g(\alpha_1) & \cdots & 1/g(\alpha_{n-1}) \\ \alpha_0/g(\alpha_0) & \alpha_1/g(\alpha_1) & \cdots & \alpha_{n-1}/g(\alpha_{n-1}) \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_0^{t-1}/g(\alpha_0) & \alpha_1^{t-1}/g(\alpha_1) & \cdots & \alpha_{n-1}^{t-1}/g(\alpha_{n-1}) \end{bmatrix}.$$

4 Transform H to a $mt \times n$ binary parity check matrix H' by replacing each entry with a column of m bits.
5 Transform H' into its systematic form $[\mathbb{I}_{mt}|K]$.
6 Return the private key $(g(x), (\alpha_0, \alpha_1, \ldots, \alpha_{n-1}))$ and the public key K.

Algorithm 2: Encryption algorithm for the Niederreiter cryptosystem.

Input : Plaintext e, public key K.
Output: Ciphertext c.

1 Compute $c = [\mathbb{I}_{mt}|K] \times e$.
2 Return the ciphertext c.

Algorithm 3: Decryption algorithm for the Niederreiter cryptosystem.

Input : Ciphertext c, secret key $(g(x), (\alpha_0, \alpha_1, \ldots, \alpha_{n-1}))$.
Output: Plaintext e.

1 Compute the double-size $2t \times n$ parity check matrix

$$H^{(2)} = \begin{bmatrix} 1/g^2(\alpha_0) & 1/g^2(\alpha_1) & \cdots & 1/g^2(\alpha_{n-1}) \\ \alpha_0/g^2(\alpha_0) & \alpha_1/g^2(\alpha_1) & \cdots & \alpha_{n-1}/g^2(\alpha_{n-1}) \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_0^{2t-1}/g^2(\alpha_0) & \alpha_1^{2t-1}/g^2(\alpha_1) & \cdots & \alpha_{n-1}^{2t-1}/g^2(\alpha_{n-1}) \end{bmatrix}.$$

2 Transform $H^{(2)}$ to a $2mt \times n$ binary parity check matrix $H'^{(2)}$ by replacing each entry with a column of m bits.
3 Compute the double-size syndrome: $S^{(2)} = H'^{(2)} \times (c|0)$.
4 Compute the error-locator polynomial $\sigma(x)$ by use of the decoding algorithm given $S^{(2)}$.
5 Evaluate the error-locator polynomial $\sigma(x)$ at $(\alpha_0, \alpha_1, \ldots, \alpha_{n-1})$ and determine the plaintext bit values.
6 Return the plaintext e.

As shown in Algorithm 2, the encryption operation is very simple and maps to the multiplication between the extended public key $[\mathbb{I}_{mt}|K]$ and the plaintext e. In our work, we only focus on the core functionalities of the Niederreiter cryptosystem, therefore we assume that the input plaintext e is an n-bit error message of weight t.

As shown in Algorithm 3, the decryption operation starts from extracting the error locator polynomial out of the ciphertext using a decoding algorithm. We use the Berlekamp-Massey's (BM) algorithm in our design and develop a dedicated BM module for decoding, as described in Section 3.2. One problem within BM-decoding is that it can only recover $\frac{t}{2}$ errors. To solve this issue, we use the trick proposed by Nicolas Sendrier [14]. We first compute the double-size parity check matrix $H^{(2)}$ corresponding to $g^2(x)$, then we append $(n - mt)$ zeros to c. Based on the fact that e and $(c|0)$ belong to the same coset given $H^{(2)} \times (c|0) = H \times e$, computing the new double-size syndrome $S^{(2)}$ enables the BM algorithm to recover t errors. Once the error locator polynomial is computed, it is evaluated at the secret random sequence $(\alpha_0, \alpha_1, \ldots, \alpha_{n-1})$, and finally the plaintext e is recovered.

2.2 Structure of the Paper

The following sections introduce the building blocks for our cryptosystem in a bottom-up fashion. Details of the GF(2^m) finite field arithmetic and of the higher-level GF(2^m)$[x]/f$ polynomial arithmetic can be found in [28]. Leveraging the arithmetic operations are modules that are used in key generation, encryption, and decryption. For key generation, the description of the Gaussian systemization and additive FFT module has been provided in [28] and in this paper we will focus on the introduction of the new merge sort module and the optimization of the additive FFT module, as described in Section 3. For encryption, a simple matrix-vector multiplication is needed. For decryption, additive FFT is used as well, and a new Berlekamp-Massey decoding module is introduced and described in Section 3. Then we describe how these modules work together to obtain an efficient design for the full cryptosystem in Section 4. Validation of the design using Sage, iVerilog, and Stratix V FPGAs is presented in Section 5 together with a discussion and comparison with related work.

3 Modules

The main building blocks within our Niederreiter cryptosystem (as shown in Figure 2) are: two Gaussian systemizers for matrix systemization over GF(2^m) and GF(2) respectively, Gao-Mateer additive FFT for polynomial evaluations, a merge-sort module for generating uniformly distributed permutations, and a Berlekamp-Massey module for decoding. The Gaussian systemizer and the original version of additive FFT have been described in detail in [28]. We will focus on the merge-sort module, the Berlekamp-Massey module and our optimizations for the additive-FFT module in this section.

Algorithm 4: Fisher-Yates shuffle

Output: Shuffled array A
Initalize: $A = \{0, 1, \ldots, n-1\}$
1 **for** i from $n-1$ downto 0 **do**
2 Generate j uniformly from range $[0, i)$
3 Swap $A[i]$ and $A[j]$

Algorithm 5: Merge sort

Input: Random list A, of length 2^k
Output: Sorted list A
1 Split A into 2^k sublists.
2 **for** i from 0 to $k-1$ **do**
3 Merge adjacent sublists.

3.1 Random Permutation

An important step in the key-generation process is to compute a random permutation of selected field elements, which is part of the private key and therefore must be kept secret. In [28], the random permutation was computed by performing Fisher-Yates shuffle [11] on the ordered list $(0, 1, \ldots, 2^m - 1)$. Algorithm 4 shows the operation of the Fisher-Yates shuffle. This algorithm computes a permutation efficiently and requires only a small amount of computational logic. As shown in in Algorithm 4, in each iteration step i (in decrementing order), this module generates a random integer $0 \leq j < i$ (Alg. 4, line 2), and then swaps the data in array position i and j. In [28], a PRNG is used, which keeps generating random numbers until the output is in the required range. Therefore, this implementation of Fisher-Yates shuffle produces a non-biased permutation (under the condition that the PRNG has no bias) but it is not constant-time because different seeds for the PRNG will lead to different cycle counts for the Fisher-Yates shuffle. This causes a potential risk of timing side-channel attacks, which is hard to eliminate even if a larger PRNG is used.

To fully eliminate potential timing attacks using the Fisher-Yates shuffle approach from [28], in this work, we implemented a constant-time sorting module for permutation based on the merge-sort algorithm. Sorting a random list can be regarded as the reverse operation of a permutation: Sorting a randomly permuted list can be seen as applying swapping operations on the elements until a sorted list is achieved. Applying the same swapping operations in reverse order to a sorted list results in a randomly permuted list. Therefore, given a constant-time sort algorithm, a constant-time algorithm for generating a random permutation can easily be derived.

Merge Sort. Merge sort is a comparison-based sorting algorithm which produces a stable sort. Algorithm 5 shows the merge sort algorithm. For example, a given random list $A = (92, 34, 18, 78, 91, 65, 80, 99)$ can be sorted by using merge sort within three steps: Initially, list A is divided into eight sublists $(92), (34), (18), (78), (91), (65), (80)$, and (99) with granularity of one. Since there

is only one element in each sublist, these sublists are sorted. In the first step, all the adjacent sublists are merged and sorted, into four sublists $(34, 92), (18, 78)$, $(65, 91)$, and $(80, 99)$ of size two. Merging of two sorted lists is simple: Iteratively, first elements of the lists are compared and the smaller one is removed from its list and appended to the merged list, until both lists are empty. In the second step, these lists are merged into two sublists $(18, 34, 78, 92)$ and $(65, 80, 91, 99)$ of size four. Finally, these two sublists are merged to the final sorted list $A_{\text{sorted}} = (18, 34, 65, 78, 80, 91, 92, 99)$.

In general, to sort a random list of n elements, merge sort needs $\log_2(n)$ iterations, where each step involves $O(n)$ comparison-based merging operations. Therefore, merge sort has an asymptotic complexity of $O(n \log_2(n))$.

Random Permutation. As mentioned above, sorting a random list can be regarded as the reverse operation of permutation. When given a random list A, before the merge sort process begins, we attach an index to each element in the list. Each element then has two parts: value and index, where the value is used for comparison-based sorting, and the index labels the original position of the element in list A. For the above example, to achieve a permutation for list $P = (0, 1, ..., 7)$, we first attach an index to each of the elements in A, which gives us a new list $A' = ((92, 0), (34, 1), (18, 2), (78, 3), (91, 4), (65, 5), (80, 6), (99, 7))$. Then the merge sort process begins, which merges elements based on their value part, while the index part remains unchanged. Finally, we get $A'_{\text{sorted}} = ((18, 2), (34, 1), (65, 5), (78, 3), (80, 6), (91, 4), (92, 0), (99, 7))$. By extracting the index part of the final result, we get a random permutation of P, which is $(2, 1, 5, 3, 6, 4, 0, 7)$. In general, to compute a random permutation, we generate 2^m random numbers and append each of them with an index. The sorting result of these random numbers will uniquely determine the permutation.

In case there is a collision among the random values, the resulting permutation might be slightly biased. Therefore, the bit-width of the randomly generated numbers needs to be selected carefully to reduce the collision rate and thusly the bias. If the width of the random numbers is b, then the probability that there are one or more collisions in 2^m randomly generated numbers is $1 - \prod_{i=1}^{2^m-1} \frac{(2^b-i)}{2^b}$ due to the birthday paradox. Therefore, for a given m, the collision rate can be reduced by using a larger b. However, increasing b also increases the required logic and memory. Both m and b are parameters which can be chosen at compile time in our implementation. The value for b can easily be chosen to fit to the required m. For the parameters $m = 13$ and $b = 32$ the collision rate is 0.0078. We further reduce the collision rate and thus the bias within merge sort by incorporating the following trick in our design at low logic cost: In case the two random to-be-merged values are equal, we do a conditional swap based on the least significant bit of the random value. Since the least significant bit of the random value is random, this trick will make sure that if some random numbers are generated twice, we can still get a non-biased permutation. There still is going to be a bias in the permutation if some random values appear more than two times. This case could be detected and the merge sort module could

Design	Algorithm	Const.	Cycles	Logic	Time×Area	Mem.	Reg.	Fmax
[28]	FY-shuffle	×	23,635	149	$3.52 \cdot 10^6$	7	111	334 MHz
Our	merge-sort	✓	147,505	448	$6.61 \cdot 10^7$	46	615	365 MHz

Table 1: Performance of computing a permutation on $2^{13} = 8192$ elements with $m = 13$ and $b = 32$; Const. = Constant Time.

be restarted repeatedly until no bias occurs. However, the probability of this is very low (prob $\approx 2^{-27.58}$ according to [10]) for $m = 13$ and $b = 32$.

Fully Pipelined Hardware Implementation. We implemented a parameterized merge sort module using two dual-port memory blocks P and P' of depth 2^m and width $(b+m)$. First, a PRNG is used, which generates 2^m random b-bit strings, each cell of memory block P then gets initialized with one of the random b-bit strings concatenated with an m-bit index string (corresponding to the memory address in this case). Once the initialization of P finishes, the merge sort process starts. In our design, the merge sort algorithm is implemented in a pipelined way. The basic three operations in the merge-sort module are: read values from two sublists, compare the two values, and write down the smaller one to a new list. In our design, there are four pipeline stages: issue reads, fetch outputs from memory, compare the outputs, and write back to the other memory. We built separate logic for these four stages and time-multiplex these four stages by working on independent sublists in parallel whenever possible. By having the four-stage pipelines, we achieve a high-performance merge-sort design with a small logic overhead.

Table 1 shows a comparison between our new, constant time, sort-based permutation module with the non-constant time Fisher-Yates shuffle approach in [28]. Clearly, the constant-time permutation implementation requires more time, area, and particularly memory. Therefore, a trade-off needs to be made between the need for increased security due to the constant-time implementation and resource utilization. In scenarios where timing side-channel protection is not needed, the cheaper Fisher-Yates shuffle version might be sufficient.

3.2 Berlekamp-Massey Algorithm

Finding a codeword at distance t from a vector v is the key step in the decryption operation. We apply a decoding algorithm to solve this problem. Among different algorithms, the Berlekamp-Massey (BM) algorithm [16] and Patterson's algorithm [20] are the algorithms most commonly used. Patterson's algorithm takes advantage of certain properties present in binary Goppa codes, and is able to correct up to t errors for binary Goppa codes with a designated minimum distance $d_{min} \geq 2t + 1$. On the other hand, general decoding algorithms like the BM algorithm can only correct $\frac{t}{2}$ errors by default, which can be increased to t errors using the trick proposed by Nicolas Sendrier [14]. However, the process of BM algorithm is quite simple compared to Patterson's algorithm. More importantly, it is easier to protect the implementation of BM algorithm against timing

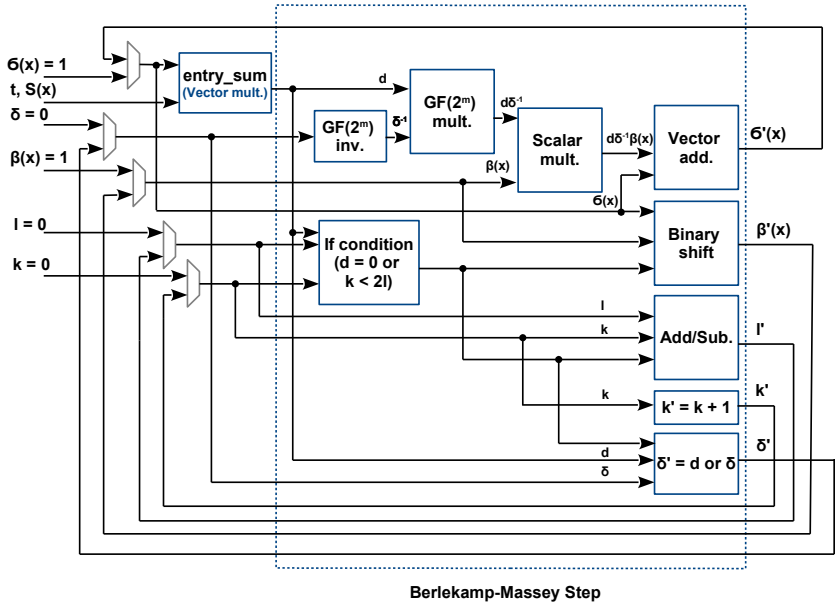


Fig. 1: Dataflow diagram of the Berlekamp-Massey module.

attacks given the simplicity of the decryption steps. Consequently, we use BM algorithm in our decryption module.

Our implementation follows the Berlekamp iterative algorithm as described in [16]. The algorithm begins with initializing polynomials $\sigma(x) = 1 \in \mathrm{GF}(2^m)[x]$, $\beta(x) = x \in \mathrm{GF}(2^m)[x]$, integers $l = 0$ and $\delta = 1 \in \mathrm{GF}(2^m)$. The input syndrome polynomial is denoted as $S(x) = \sum_{i=1}^{2t-1} S_i x^i \in \mathrm{GF}(2^m)[x]$. Then within each iteration step k ($0 \leq k \leq 2t-1$), the variables $\{\sigma(x), \beta(x), l, \delta\}$ are conditionally updated using operations described in Algorithm 6. Note that updating polynomial $\beta(x)$ only involves multiplying a polynomial by x, which can be easily mapped to a binary shifting operation on its coefficients in hardware. Updating integer l and field element δ only involves subtraction/addition operations, and these operations can also be easily implemented in hardware. Therefore the bottleneck of the algorithm lies in computing d and updating $\sigma(x)$.

Hardware Implementation. The first step within each iteration is to calculate d (Alg. 6, line 3). We built an `entry_sum` module (as shown in Figure 1) for this computation, which maps to a vector-multiplication operation as described in [28]. We use two registers $\boldsymbol{\sigma_{vec}}$ and $\boldsymbol{\beta_{vec}}$ of $m \cdot (t+1)$ bits to store the coefficients of polynomials $\sigma(x)$ and $\beta(x)$, where the constant terms σ_0 and β_0 are stored in the lowest m bits of the registers, σ_1 and β_1 are stored in the second lowest m bits, and so on. We also use a register $\boldsymbol{S_{vec}}$ of $m \cdot (t+1)$ bits to store at most $(t+1)$ coefficients of $S(x)$. This register is updated within each iteration, where S_k is stored in the least significant m bits of the register, S_{k-1} is stored in the second least significant m bits, and so on. The computation of d can then be regarded as an entry-wise vector multiplication between register $\boldsymbol{\sigma_{vec}}$ and

Algorithm 6: Berlekamp-Massey algorithm for decryption.

Input : Public security parameter t, syndrome polynomial $S(x)$.
Output: Error locator polynomial $\sigma(x)$.

Initialize: $\sigma(x) = 1$, $\beta(x) = x$, $l = 0$, $\delta = 1$.
for k from 0 to $2t - 1$ **do**
    $d = \sum_{i=0}^{t} \sigma_i S_{k-i}$
    if $d = 0$ or $k < 2l$:
        $\{\sigma(x), \beta(x), l, \delta\} = \{\sigma(x) - d\delta^{-1}\beta(x), x\beta(x), l, \delta\}$.
    else:
        $\{\sigma(x), \beta(x), l, \delta\} = \{\sigma(x) - d\delta^{-1}\beta(x), x\sigma(x), k - l + 1, d\}$.
Return the error locator polynomial $\sigma(x)$.

register $\boldsymbol{S_{vec}} = (0, 0, ..., S_0, S_1, ..., S_{k-1}, S_k)$ for all $0 \leq k \leq 2t-1$. Register $\boldsymbol{\sigma_{vec}}$ is initialized as $(0, 0, ..., 1)$ for the first iteration, and then gets updated with the new coefficients of $\sigma(x)$ for the next iteration. $\boldsymbol{S_{vec}}$ is initialized as all zeroes, and then constructed gradually by reading from a piece of memory which stores coefficient S_i of syndrome polynomial $S(x)$ at address i for $0 \leq i \leq 2t-1$. Within the k-th iteration, a read request for address k of the memory is issued. Once the corresponding coefficient S_k is read out, it is inserted to the lowest m bits of $\boldsymbol{S_{vec}}$. After the computation of d, we start updating variables $\{\sigma(x), \beta(x), l, \delta\}$. To update $\sigma(x)$, one field-element inversion, one field-element multiplication, one scalar multiplication as well as one vector subtraction are needed. At first, field element δ is inverted. As described in [28], the inversion of elements in GF(2^m) can be implemented by use of a pre-computed lookup table. Each entry of the table can be read in one clock cycle. After reading out δ^{-1}, a field-element multiplication between d and δ^{-1} is performed, which makes use of the GF(2^m) multiplication module as described in [28]. Once we get $d\delta^{-1}$, a scalar multiplication between field element $d\delta^{-1}$ and polynomial $\beta(x)$ starts, which can be mapped to an entry-wise vector multiplication between vector $(d\delta^{-1}, d\delta^{-1}, ..., d\delta^{-1})$ and $(\beta_t, \beta_{t-1}, ..., \beta_1, \beta_0)$. The last step for updating $\sigma(x)$ is to subtract $d\delta^{-1}\beta(x)$ from $\sigma(x)$. In a binary field GF(2^m), subtraction and addition operations are equivalent. Therefore, the subtraction between $\sigma(x)$ and $d\delta^{-1}\beta(x)$ can simply be mapped to bit-wise `xor` operations between vector $(\sigma_t, \sigma_{t-1}, ..., \sigma_1, \sigma_0)$ and vector $(d\delta^{-1}\beta_t, d\delta^{-1}\beta_{t-1}, ..., d\delta^{-1}\beta_1, d\delta^{-1}\beta_0)$. Updating polynomial $\beta(x)$ is done by conditionally replacing its coefficient register $\boldsymbol{\beta_{vec}}$ with $\boldsymbol{\delta_{vec}}$, and then shift the resulting value leftwards by m bits. Updating integer l and field element δ only involves simple and cheap hardware operations.

The above iterations are repeated for a fixed number of $2t$ times, where t is the public security parameter. After $2t$ iterations, the final output is determined as the error locator polynomial $\sigma(x)$. It is easy to see that within each iteration, the sequence of instructions is fixed, as long as we make sure that the conditional updates of variables $\{\sigma(x), \beta(x), l, \delta\}$ are constant time (which is easy to achieve due to its fixed computational mapping in hardware), the run time of the whole design is fixed given the fixed iteration times. Therefore our BM implementation is fully protected against existing timing side-channel attacks, e.g., [3,25].

mul_{BM}	$\text{mul}_{\text{BM\_step}}$	Cycles	Logic	Time×Area	Mem.	Reg.	Fmax
10	10	7379	6285	$4.64 \cdot 10^7$	7	13,089	364 MHz
20	20	4523	7052	$3.19 \cdot 10^7$	7	13,031	353 MHz
30	30	3571	7889	$2.82 \cdot 10^7$	7	12,956	361 MHz
40	40	3095	9047	$2.8 \cdot 10^7$	7	13,079	356 MHz
60	60	2619	11,400	$2.99 \cdot 10^7$	7	13,274	354 MHz

Table 2: Performance of the Berlekamp-Massey module for $m = 13$, $t = 119$, and $\deg(S(x)) = 237$.

We built a two-level design. The lower level is a `BM_step` module, which maps to one iteration, shown as "Berlekamp-Massey Step" in Figure 1. The higher-level `BM` module then iteratively applies `BM_step` and `entry_sum` modules. Table 2 shows performance for the BM module. A time-area trade-off can be achieved by adjusting the design parameters mul_{BM} and $\text{mul}_{\text{BM\_step}}$, which are the number of multipliers used in the `BM` and `BM_step` modules. mul_{BM} and $\text{mul}_{\text{BM\_step}}$ can be freely chosen as integers between 1 and $t + 1$.

3.3 Optimizations for Additive FFT

Evaluating a polynomial at multiple data points over $\text{GF}(2^m)$ is an essential step in both the key generation and the decryption processes. In key generation, an evaluation of the Goppa polynomial $g(x)$ is needed for computing the parity check matrix H, while for decryption, it is required by the computation of the double-size parity check matrix $H^{(2)}$ as well as the evaluation of the error locator polynomial $\sigma(x)$. Therefore, having an efficient polynomial-evaluation module is very important for ensuring the performance of the overall design. We use a characteristic-2 additive FFT algorithm introduced in 2010 by Gao and Mateer [12], which was used for multipoint polynomial evaluation by Bernstein et al. in 2013 [5]. Additive FFT consists of two parts. First, radix conversion and twist is performed on the input polynomial. Given a polynomial $g(x)$ of 2^k coefficients, the recursive twist-then-radix-conversion process returns 2^k 1-coefficient polynomials. Then, these 1-coefficient polynomials are used to iteratively evaluate the input points by use of the reduction process.

We applied some modifications and improvements to both parts of the additive FFT design from [28]:

Optimizing Radix Conversion and Twisting. The radix-conversion step, which includes both radix conversion and twist, consists of several rounds that iteratively compute the new output coefficients of the converted input polynomial. The number of rounds is the base-2 logarithm of the degree of the input polynomial. In each round, new temporary coefficients are computed as the sum of some of the previous coefficients followed by a twist operation, i.e., a multiplication of each coefficient with a pre-computed constant to obtain a new basis for the respective round.

Design	Coeffs.	Mult.	Cycles	Logic	Time×Area	Reg.	Mem.	Fmax
Our	120	2	385	1893	$7.3 \cdot 10^5$	3541	6	305 MHz
Our	120	4	205	2679	$5.5 \cdot 10^5$	3622	10	273 MHz
[28]	128	4	211	5702	$1.2 \cdot 10^6$	7752	0	407 MHz
Our	120	8	115	4302	$4.9 \cdot 10^5$	3633	17	279 MHz
[28]	128	8	115	5916	$6.8 \cdot 10^5$	7717	0	400 MHz

Table 3: Performance of our radix-conversion module compared to [28] for GF(2^{13}).

The radix-conversion module in [28] is using dedicated logic for each round for summing up the required coefficients, computing all coefficients within one cycle. Computing all coefficients with dedicated logic for each round requires a significant amount of area although radix conversion only requires a very small amount of cycles compared to the overall additive FFT process. Therefore, this results in a relatively high time-area product and a poor usage of resources.

We improve the area-time product at the cost of additional cycles and additional memory requirements by using the same logic block for different coefficients and rounds. An additional code-generation parameter is used to specify how many coefficients should be computed in parallel, which equals to the number of multipliers ($1 \leq$ Mult. $\leq t + 1$) used in twist when mapping to hardware implementations. Each round then requires several cycles depending on the selected parameter. The computation of the new coefficients requires to sum up some of the previous coefficients. The logic therefore must be able to add up any selection of coefficients depending on the target coefficient. We are using round- and coefficient-dependent masks to define which coefficients to sum up in each specific case. These masks are stored in additional RAM modules.

Furthermore, in the design of [28], the length of the input polynomial is constrained to be a power of 2. For shorter polynomials, zero-coefficients need to be added, which brings quite some logic overhead especially on some extreme cases. For example, for a polynomial of 129 coefficients ($t = 128$), a size-256 radix conversion module will be needed. Instead, our improved design eliminates this constraint and allows an arbitrary input length with low overhead and therefore is able to further reduce cycle count and area requirements.

Table 3 shows the performance improvements of the current radix-conversion module compared to the design in [28]. The numbers for our new design are given for a polynomial of length 120. The design in [28] requires the next larger power of 2 as input length. Therefore, we give numbers for input length 128 for comparison. For a processing width of four coefficients (multipliers), our new implementation gives a substantial improvement in regard to the time-area product over the old implementation at the cost of a few memory blocks.

Parameterizing Reduction. In the previous design of the additive FFT in [28], the configuration of the reduction module is fixed and uniquely determined by the polynomial size and the binary field size. Before the actual computation begins, the data memory is initialized with the 2^k 1-coefficient polynomials from

Mult.	Cycles	Logic	Time×Area	Mem. Bits	Mem.	Reg.	Fmax
32	968	4707	$4.56 \cdot 10^6$	212,160	63	10,851	421 MHz
64	488	9814	$4.79 \cdot 10^6$	212,992	126	22,128	395 MHz

Table 4: Performance of our parameterized size-128 reduction module for GF(2^{13}).

the output of the last radix-conversion round. The data memory `D` within the reduction module is configured as follows: The depth of the memory equals to 2^k, based on this, the width of the memory is determined as $m \times 2^{m-k}$ since in total $m \times 2^m$ memory bits are needed to store the evaluation results for all the elements in GF(2^m). Each row of memory `D` is initialized with 2^{m-k} identical 1-coefficient polynomials. The other piece of memory within the reduction module is the constants memory `C`. It has the same configuration as the data memory and it stores all the elements for evaluation of different reduction rounds. Once the initialization of data memory and constants memory is finished, the actual computation starts, which consists of the same amount of rounds as needed in the radix conversion process. Within each round, two rows of values (f_0 and f_1) are read from the data memory and the corresponding evaluation points from the constants memory, processed, and then the results are written back to the data memory. Each round of the reduction takes 2^k cycles to finish. In total, the reduction process takes $k \times 2^k$ cycles plus overhead for memory initialization.

In our current design, we made the reduction module parameterized by introducing a flexible memory configuration. The width of memories `D` and `C` can be adjusted to achieve a trade-off between logic and cycles. The algorithmic pattern for reduction remains the same, while the computational pattern changes due to the flexible data reorganization within the memories. Instead of fixing the memory width as $m \times 2^{m-k}$, it can be configured as a wider memory of width $m \times 2^{m-k+i}, 0 \leq i \leq k$. In this way, we can store multiple 1-coefficient polynomials at one memory address. The organization of the constants memory needs to be adapted accordingly. Therefore, within each cycle, we can either fetch, do computation on, or write back more data and therefore finish the whole reduction process within much fewer cycles ($k \times 2^{k-i}$ plus overhead of few initialization cycles). However, the speedup of the running time is achieved at the price of increasing the logic overhead, e.g., each time the width of the memory doubles, the number of multipliers needed for computation also doubles.

Table 4 shows the performance of our parameterized reduction module. We can see that doubling the memory width halves the cycles needed for the reduction process, but at the same time approximately doubles the logic utilization. We can see that although the memory bits needed for reduction remain similar for different design configurations, the number of required memory blocks doubles in order to achieve the increased memory width. Users can easily achieve a trade-off between performance and logic by tuning the memory configurations within the reduction module.

Table 5 shows performance of the current optimized additive FFT module. By tuning the design parameters in the radix conversion and reduction parts,

Design	Multipliers Rad.	Red.	Cycles	Logic	Time×Area	Mem.	Reg.	Fmax
Our	4	32	1173	7344	$8.61 \cdot 10^6$	73	14,092	274 MHz
[28]	4	32	1179	10,430	$1.23 \cdot 10^7$	63	18,413	382 MHz
Our	8	64	603	13,950	$8.41 \cdot 10^6$	143	25,603	279 MHz
[28]	8	32	1083	10,710	$1.16 \cdot 10^7$	63	18,363	362 MHz

Table 5: Performance of our optimized additive-FFT module compared to [28] for $m = 13$, $\deg(g(x)) = 119$. Rad. and Red. are the number of multipliers used in radix conversion and twist (reduction) separately.

we are able to achieve a 28% smaller time-area product compared to [28] when Rad. = 4 and Red. = 64.

4 Key Generation, Encryption and Decryption

We designed the Niederreiter cryptosystem by using the main building blocks shown in Figure 2. Note that we are using two simple 64-bit Xorshift PRNGs in our design to enable deterministic testing. For real deployment, these PRNGs must be replaced with a cryptographically secure random-number generator, e.g., [8]. We require at most b random bits per clock cycle per PRNG.

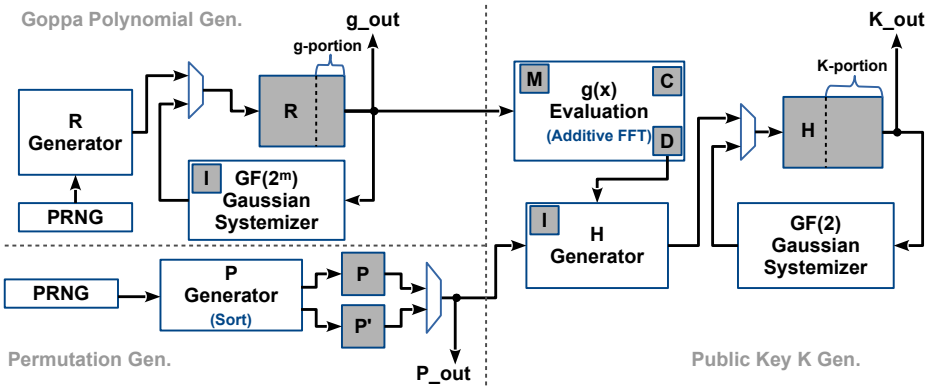
4.1 Key Generation

The overall design of our key-generation module is identical to the design in [28]. The dataflow diagram is shown in Figure 2a. However, we improve the security of private-key generation by substituting the Fisher-Yates Shuffle module with a merge-sort module in order to generate a uniform and random permutation in constant time (see Section 3.1). The generation of the public key is improved by several optimizations applied to the additive FFT module (see Section 3.3).

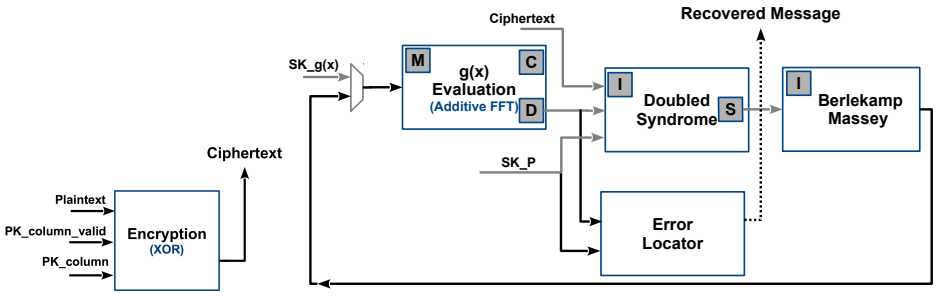
Table 6 shows a comparison of the performance of the old implementation in [28] with our new, improved implementation. Despite the higher cost for the constant-time permutation module, overall, we achieve an improvement in regard to area requirements and therefore to the time-area product at roughly the same frequency on the price of a higher memory demand. However, the overall memory increase is less than 10% which we believe is justified by the increased side-channel resistance due to the use of a constant-time permutation.

4.2 Encryption

Figure 2b shows the interface of the encryption module. The encryption module assumes that the public key K is fed in column by column. The matrix-vector multiplication $[\mathbb{I}_{mt}|K] \times e$ is mapped to serial `xor` operations. Once the `PK_column_valid` signal is high, indicating that a new public-key column



(a) Key generation, with optimizations over [28]



(b) Encryption (c) Decryption

Fig. 2: Dataflow diagrams of the three parts of the full cryptosystem: (a) key generation, (b) encryption, and (c) decryption. Dark gray boxes represent block memories, while white boxes represent major logic modules.

(`PK_column`) is available at the input port, the module checks if the corresponding bit of plaintext e is 1 or 0. If the bit value is 1, then an `xor` operation between the current output register (initialized as 0) and the new public-key column is carried out. Otherwise, no operation is performed. After the `xor` operation between K and the last $(n - mt)$ bits of e is finished, we carry out one more `xor` operation between the output register and the first mt bits of e. Then the updated value of the output register will be sent out as the cipheretxt c. Table 7 shows performance of the encryption module. The encryption module is able to handle one column of the public key in each cycle and therefore requires a fixed number of $(n - mt)$ cycles independent of the secret input vector e.

4.3 Decryption

Within the decryption module, as described in Figure 2c, first the evaluation of the Goppa polynomial $g(x)$ is carried out by use of the optimized additive FFT module, which was described in Section 3.3. In our implementation, instead of first computing the double-size parity-check matrix $H^{(2)}$ and then comput-

Case	N_H	N_R	Cycles	Logic	Time×Area	Mem.	Fmax	Time
				Prior work [28]				
logic	40	1	11,121,220	29,711	$3.30 \cdot 10^{11}$	756	240 MHz	46.43 ms
bal.	80	2	3,062,942	48,354	$1.48 \cdot 10^{11}$	764	248 MHz	12.37 ms
time	160	4	896,052	101,508	$9.10 \cdot 10^{10}$	803	244 MHz	3.68 ms
				Our work				
logic	40	1	11,121,214	22,716	$2.53 \cdot 10^{11}$	819	237 MHz	46.83 ms
bal.	80	2	3,062,936	39,122	$1.20 \cdot 10^{11}$	827	230 MHz	13.34 ms
time.	160	4	966,400	88,715	$8.57 \cdot 10^{10}$	873	251 MHz	3.85 ms

Table 6: Performance of the key-generation module for parameters $m = 13$, $t = 119$, and $n = 6960$. All the numbers in the table come from compilation reports of the Altera tool chain for Stratix V FPGAs.

m	t	n	Cycles	Logic	Time×Area	Mem.	Reg.	Fmax
13	119	6960	5413	4276	$2.31 \cdot 10^7$	0	6977	448 MHz

Table 7: Performance for the encryption module.

ing the double-size syndrome $S^{(2)}$, we combine these two steps together. The computation of $S^{(2)}$ can be mapped to serial conditional `xor` operations of the columns of $H^{(2)}$. Based on the observation that the last $(n - mt)$ bits of vector $(c|0)$ are all zero, the last $(n - mt)$ columns of $H^{(2)}$ do not need to be computed. Furthermore, the ciphertext c should be a uniformly random bit string. Therefore, for the first mt columns of $H^{(2)}$, roughly only half of the columns need to be computed. Finally, we selectively choose which columns of $H^{(2)}$ we need to compute based on the nonzero bits of the binary vector $(c|0)$. In total, approximately $m \times t^2$ field element multiplications are needed for computing the double-size syndrome. The computation of the corresponding columns of $H^{(2)}$ is performed in a similar column-block-wise method as described in [28]. The size B ($1 \leq \mathrm{B} \leq \frac{mt}{2}$) of the column block is a design parameter that users can pick freely to achieve a trade-off between logic and cycles during computation. After the double-syndrome $S^{(2)}$ is computed, it is fed into the Berlekamp-Massey module described in Section 3.2 and the error-locator polynomial $\sigma(x)$ is determined as the output. Next, the error-locator polynomial $\sigma(x)$ is evaluated using the additive FFT module (see Section 3.3) at all the data points over GF(2^m). Then, the message bits are determined by checking the data memory contents within the additive FFT module that correspond to the secret key-element set $(\alpha_0, \alpha_1, \ldots, \alpha_{n-1})$. If the corresponding evaluation result for α_i, $i = 0, 1, ..., n-1$ equals to zero, then the i-th bit of the plaintext is determined as 1, otherwise is determined as 0. After checking the evaluation results for all the elements in the set $(\alpha_0, \alpha_1, \ldots, \alpha_{n-1})$, the plaintext is determined. Table 8 shows the performance of the decryption module with different design parameters. By tuning design parameters $\mathrm{mul}_{\mathrm{BM\_step}}$, $\mathrm{mul}_{\mathrm{BM}}$, and B, a time-area trade-off can be made.

Case	B	mul_{BM}	Cycles	Logic	Time×Area	Mem.	Reg.	Fmax	Time
area	10	10	34,492	19,377	$6.68 \cdot 10^8$	88	47,749	289 MHz	0.12 ms
bal.	20	20	22,768	20,815	$4.74 \cdot 10^8$	88	48,050	290 MHz	0.08 ms
time	40	40	17,055	23,901	$4.08 \cdot 10^8$	88	49,407	300 MHz	0.06 ms

Table 8: Performance for the decryption module for $m = 13, t = 119$ and $n = 6960$, $\text{mul}_{\text{BM\_step}}$ is set to mul_{BM}.

5 Testing, Evaluation, and Comparison

Our implementation of the Niederreiter cryptosystem is fully parameterized and can be synthesized for any choice of reasonable security parameters. However, the main target of our implementation is the 256-bit (classical) security level, which corresponds to a level at least "128-bit post-quantum security". For testing, we used the parameters suggested in the PQCRYPTO recommendations [2]: $m = 13$, $t = 119$, $n = 6960$ and $k = 5413$ ($k = n - mt$).

Testing. To validate the FPGA implementation, in addition to simulations, we implemented a serial IO interface for communication between the host computer and the FPGA. The interface allows us to send data and simple commands from the host to the FPGA and receive data, e.g., public and private key, ciphertext, and plaintext, from the FPGA. We verified the correct operation of our design by comparing the FPGA outputs with our Sage reference implementation (using the same PRNG and random seeds).

Evaluation. We synthesized our design using Altera Quartus 17.0 for these parameters on a Stratix V FPGA (5SGXEA7N). The results are given in Table 9, with included logic overhead of the IO interface. We provide numbers for three performance parameter sets, one for small area, one for small runtime, and one for balanced time and area. The parameters N_R and N_H control the size of the systolic array in the Gaussian systemizer modules, which are used for computing the private Goppa polynomial and the public key. Parameter B is the matrix-block size used for computing the syndrome. Parameter mul_{BM} determines the number of multipliers used in the high-level `BM` decoding module. The number of multipliers ($\text{mul}_{\text{BM\_step}}$) used in the low-level `BM_step` module is set to mul_{BM} for the evaluation. The memory requirement varies slightly due the differences in the memory word size based on the design parameters. These design parameters can be freely chosen as long as the synthesized result fits on the target FPGA. For security parameter set $m = 13, t = 119, n = 6960$, our experiment shows that the largest design parameter set we can fit on Stratix V FPGA is: $\text{N}_\text{R} = 250$, $\text{N}_\text{H} = 6$, $\text{mul}_{\text{BM}} = 60$, $\text{mul}_{\text{BM\_step}} = 60$, and $\text{B} = 60$.

Comparison. In the following, we compare our work with previous designs.

First, we compare it with a 103-bit classical security-level hardware-design described in [26]. This work is the only previously existing hardware implementation for the whole code-based cryptosystem, including a key generator, that we have found in literature. To compare with their work, we synthesized our design

Case	N_H	N_R	B	mul_{BM}	Logic	Mem.	Reg.	Fmax
area	40	1	10	10	53,447 (23%)	907 (35%)	118,243	245 MHz
bal.	80	2	20	20	70,478 (30%)	915 (36%)	146,648	251 MHz
time	160	4	40	40	121,806 (52%)	961 (38%)	223,232	248 MHz

Table 9: Performance for the entire Niederreiter cryptosystem (i.e., key generation, encryption, and decryption) including the serial IO interface when synthesized for the Stratix V (5SGXEA7N) FPGA; mul_{BM_step} is set to mul_{BM}.

	Cycles			Logic	Freq.	Mem.	Time (ms)		
	Gen.	Dec.	Enc.		(MHz)		Gen.	Dec.	Enc.
$m = 11, t = 50, n = 2048$, Virtex 5 LX110									
[26]	14,670,000	210,300	81,500	14,537	163	75	90.00	1.29	0.50
Our[a]	1,503,927	5864	1498	6660	180	68	8.35	0.03	0.01
$m = 12, t = 66, n = 3307$, Virtex 6 LX240									
[17]	—	28,887	—	3307	162	15	—	0.18	—
Our[b]	—	10,228	—	6571	267	23	—	0.04	—
Our[c]	4,929,400	10,228	2515	17,331	160	142	30.00	0.06	0.02
$m = 13, t = 128, n = 8192$, Haswell vs. StratixV									
[9]	1,236,054,840	343,344	289,152	—	4000	—	309.01	0.09	0.07
Our[d]	1,173,750	17,140	6528	129,059	231	1126	5.08	0.07	0.03

Table 10: Comparison with related work. Logic is given in "Slices" for Xilinx Virtex FPGAs and in "ALMs" for Altera Stratix FPGAs.

[a] (N_H, N_R, B, mul_{BM}, mul_{BM_step}) = (20, 2, 20, 20, 20), entire Niederreiter cryptosystem.
[b] (B, mul_{BM}, mul_{BM_step}) = (20, 20, 20), decryption module.
[c] (N_H, N_R, B, mul_{BM}, mul_{BM_step}) = (20, 2, 20, 20, 20), entire Niederreiter cryptosystem.
[d] (N_H, N_R, B, mul_{BM}, mul_{BM_step}) = (160, 4, 80, 65, 65), entire Niederreiter cryptosystem.

with the Xilinx tool-chain version 14.7 for a Virtex-5 XC5VLX110 FPGA. Note that the performance data of [26] in Table 10 includes a CCA2 conversion for encryption and decryption, which adds some overhead compared to our design. From Table 10, we can see that our design is much faster when comparing cycles and time, and also much cheaper in regard to area and memory consumption.

Second, we compare our work with a hardware design from [17], which presents the previously fastest decryption module for a McEliece cryptosystem. Therefore the comparison of our work with design [17] focuses on the decryption part. We synthesized our decryption module with the parameters they used, which correspond to a 128-bit classical security level, for a Virtex-6 XC6VLX240T FPGA. From Table 10, we can see that the time-area product of our decryption module is $10228 \cdot 6571 = 67,208,188$, which is 30% smaller than the time-area product of their design of $28887 \cdot 3307 = 95,529,309$ when comparing only the decryption module. Moreover, our design is able to achieve a much higher frequency and a smaller cycle counts compared to their design. Overall we are more than 4x faster than [17]. Apart from this, we also provide the performance of the entire Niederreiter cryptosystem corresponding to secu-

rity parameter set $m = 12, t = 66, n = 3307$ when synthesized for a Virtex 6 XC6VLX240T FPGA.

Finally, we also compare the performance of our hardware design with the to-date fastest CPU implementation of the Niederreiter cryptosystem [9]. In this case, we ran our implementation on our Altera Stratix V FPGA and compare it to a Haswell CPU running at 4 GHz. Our implementation competes very well with the CPU implementation, despite the over 10x slower clock of the FPGA.

6 Conclusion

This paper presented a complete hardware implementation of Niederreiters's code-based cryptosystem based on binary Goppa codes, including key generation, encryption and decryption. The presented design can be configured with tunable parameters, and uses code-generation to generate vendor-neutral Verilog HDL code for any set of reasonable parameters. This work presented hardware implementations of an optimization of the Gao-Mateer additive FFT for polynomial evaluation, of merge sort used for obtaining uniformly distributed permutations, and of a constant-time Berlekamp-Massey algorithm.

Open-Source Code. The source code for this project is available under an open-source license at http://caslab.csl.yale.edu/code/niederreiter/.

Acknowledgments. This work was supported in part by United States' National Science Foundation grant 1716541. We would like to acknowledge FPGA hardware donations form Altera (now part of Intel). We also want to thank Tung (Tony) Chou for his invaluable help. This paper has been greatly improved thanks to feedback from our shepherds Lajla Batina and Pedro Maat Costa Massolino and the anonymous reviewers.

References

1. Alkadri, N.A., Buchmann, J., Bansarkhani, R.E., Krämer, J.: A framework to select parameters for lattice-based cryptography. Cryptology ePrint Archive, Report 2017/615 (2017), https://eprint.iacr.org/2017/615
2. Augot, D., Batina, L., Bernstein, D.J., Bos, J., Buchmann, J., Castryck, W., Dunkelman, O., Güneysu, T., Gueron, S., Hülsing, A., Lange, T., Mohamed, M.S.E., Rechberger, C., Schwabe, P., Sendrier, N., Vercauteren, F., Yang, B.Y.: Initial recommendations of long-term secure post-quantum systems. Tech. rep., PQCRYPTO ICT-645622 (2015), https://pqcrypto.eu.org/docs/initial-recommendations.pdf.
3. Avanzi, R., Hoerder, S., Page, D., Tunstall, M.: Side-channel attacks on the McEliece and Niederreiter public-key cryptosystems. JCEN 1(4), 271–281 (2011)
4. Bernstein, D.J., Buchmann, J., Dahmen, E. (eds.): Post-Quantum Cryptography. Springer, Heidelberg (2009)
5. Bernstein, D.J., Chou, T., Schwabe, P.: McBits: fast constant-time code-based cryptography. In: Bertoni, G., Coron, J.S. (eds.) CHES 2013. LNCS, vol. 8086, pp. 250–272. Springer, Heidelberg (2013)

6. Bernstein, D.J., Lange, T., Peters, C.: Attacking and defending the McEliece cryptosystem. In: Buchmann, J., Ding, J. (eds.) PQCrypto 2008. LNCS, vol. 5299, pp. 31–46. Springer, Heidelberg (2008)
7. Chen, L., Moody, D., Liu, Y.K.: NIST post-quantum cryptography standardization, https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/.
8. Cherkaoui, A., Fischer, V., Fesquet, L., Aubert, A.: A very high speed true random number generator with entropy assessment. In: Bertoni, G., Coron, J.S. (eds.) CHES 2013. LNCS, vol. 8086, pp. 179–196. Springer, Heidelberg (2013)
9. Chou, T.: McBits revisited. In: Fischer, W., Homma, N. (eds.) CHES 2017. LNCS, vol. 10529, pp. 213–231. Springer, Heidelberg (2017)
10. DasGupta, A.: The matching, birthday and the strong birthday problem: a contemporary review. J. Stat. Plan. Inference 130(1), 377–389 (2005)
11. Fisher, R.A., Yates, F.: Statistical tables for biological, agricultural and medical research. Oliver and Boyd (1948)
12. Gao, S., Mateer, T.: Additive fast Fourier transforms over finite fields. IEEE Transactions on Information Theory 56(12), 6265–6272 (2010)
13. Guo, Q., Johansson, T., Stankovski, P.: A key recovery attack on MDPC with CCA security using decoding errors. In: Cheon, J.H., Takagi, T. (eds.) ASIACRYPT 2016. LNCS, vol. 10031, pp. 789–815. Springer, Heidelberg (2016)
14. Heyse, S., Güneysu, T.: Code-based cryptography on reconfigurable hardware: tweaking Niederreiter encryption for performance. JCEN 3(1), 29–43 (2013)
15. Li, Y.X., Deng, R.H., Wang, X.M.: On the equivalence of McEliece's and Niederreiter's public-key cryptosystems. IEEE Transactions on Information Theory 40(1), 271–273 (1994)
16. Massey, J.: Shift-register synthesis and BCH decoding. IEEE transactions on Information Theory 15(1), 122–127 (1969)
17. Massolino, P.M.C., Barreto, P.S.L.M., Ruggiero, W.V.: Optimized and scalable coprocessor for McEliece with binary Goppa codes. ACM Transactions on Embedded Computing Systems 14(3), 45 (2015)
18. McEliece, R.J.: A public-key cryptosystem based on algebraic coding theory. DSN Progress Report 42–44, 114—116 (1978)
19. Niederreiter, H.: Knapsack-type cryptosystems and algebraic coding theory. Problems of Control and Information Theory 15, 19–34 (1986)
20. Patterson, N.: The algebraic decoding of Goppa codes. IEEE Transactions on Information Theory 21(2), 203–207 (1975)
21. Post-quantum cryptography for long-term security, PQCRYPTO, ICT-645622, https://pqcrypto.eu.org/.
22. Rostovtsev, A., Stolbunov, A.: Public-key cryptosystem based on isogenies. Cryptology ePrint Archive, Report 2006/145 (2006)
23. Shor, P.W.: Algorithms for quantum computation: Discrete logarithms and factoring. In: Foundations of Computer Science – FOCS '94. pp. 124–134. IEEE (1994)
24. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM review 41(2), 303–332 (1999)
25. Shoufan, A., Strenzke, F., Molter, H.G., Stöttinger, M.: A timing attack against Patterson Algorithm in the McEliece PKC. In: ICISC. vol. 5984, pp. 161–175. Springer, Heidelberg (2009)
26. Shoufan, A., Wink, T., Molter, G., Huss, S., Strentzke, F.: A novel processor architecture for McEliece cryptosystem and FPGA platforms. IEEE Transactions on Computers 59(11), 1533–1546 (2010)

27. Sidelnikov, V.M., Shestakov, S.O.: On insecurity of cryptosystems based on generalized Reed-Solomon codes. Discrete Math. Appl. 2(4), 439–444 (1992)
28. Wang, W., Szefer, J., Niederhagen, R.: FPGA-based key generator for the Niederreiter cryptosystem using binary Goppa codes. In: Fischer, W., Homma, N. (eds.) CHES 2017. LNCS, vol. 10529, pp. 253–274. Springer, Heidelberg (2017)