FPGA-based Key Generator for the 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 Institute SIT, Darmstadt, Germany
ruben@polycephaly.org

Abstract. This paper presents a post-quantum secure, efficient, and tunable FPGA implementation of the key-generation algorithm for the Niederreiter cryptosystem using binary Goppa codes. Our key-generator implementation requires as few as 896,052 cycles to produce both public and private portions of a key, and can achieve an estimated frequency Fmax of over 240 MHz when synthesized for Stratix V FPGAs. To the best of our knowledge, this work is the first hardware-based implementation that works with parameters equivalent to, or exceeding, the recommended 128-bit "post-quantum security" level. The key generator can produce a key pair for parameters $m = 13$, $t = 119$, and $n = 6960$ in only 3.7 ms when no systemization failure occurs, and in $3.5 \cdot 3.7$ ms on average. To achieve such performance, we implemented an optimized and parameterized Gaussian systemizer for matrix systemization, which works for any large-sized matrix over any binary field GF(2^m). Our work also presents an FPGA-based implementation of the Gao-Mateer additive FFT, which only takes about 1000 clock cycles to finish the evaluation of a degree-119 polynomial at 2^{13} data points. The Verilog HDL code of our key generator is parameterized and partly code-generated using Python and Sage. It can be synthesized for different parameters, not just the ones shown in this paper. We tested the design using a Sage reference implementation, iVerilog simulation, and on real FPGA hardware.

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

1 Introduction

Once sufficiently large and efficient quantum computers can be built, they will be able to break many cryptosystems used today: Shor's algorithm [22,23] can solve the integer-factorization problem and the discrete-logarithm problem in polynomial time, which fully breaks cryptosystems built upon the hardness of

©IACR 2017. This article is the final version submitted by the authors to the IACR and to Springer-Verlag on June 26, 2017. The version published by Springer-Verlag is available at https://doi.org/10.1007/978-3-319-66787-4_13.
Permanent ID of this document: 503b6c5d84a7a196a4fd4ce7034b06ba.
Date: 2017.07.22.

these problems, e.g., RSA, ECC, and Diffie-Hellman. In addition, Grover's algorithm [10] gives a square-root speedup on search problems and improves brute-force attacks that check every possible key, which threatens, e.g, symmetric key ciphers like AES. However, a "simple" doubling of the key size can be used as mitigation for attacks using Grover's algorithm. In order to provide alternatives for the cryptographic systems that are threatened by Shor's algorithm, the cryptographic community is investigating cryptosystems that are secure against attacks by quantum computers using both Shor's and Grover's algorithm in a field called Post-Quantum Cryptography (PQC).

Currently, there are five popular classes of PQC algorithms: hash-based, code-based, lattice-based, multivariate, and isogeny-based cryptography [3,21]. Most code-based public-key encryption schemes are based on the McEliece cryptosystem [16] or its more efficient dual variant developed by Niederreiter [18]. This work focuses on the Niederreiter variant of the cryptosystem using binary Goppa codes. There is some work based on QC-MDPC codes, which have smaller key sizes compared to binary Goppa codes [12]. However, QC-MDPC codes can have decoding errors, which may be exploited by an attacker [11]. Therefore, binary Goppa codes are still considered the more mature and secure choice despite their disadvantage in the key size. Until now, the best known attacks on the McEliece and Niederreiter cryptosystems using binary Goppa codes are generic decoding attacks which can be warded off by a proper choice of parameters [5].

However, there is a tension between the algorithm's parameters (i.e., the security level) and the practical aspects, e.g., the size of keys and computation speed, resulting from the chosen parameters. The PQCRYPTO project [20] recommends to use a McEliece cryptosystem with binary Goppa codes with binary field of size $m = 13$, adding $t = 119$ errors, code length $n = 6960$, and code rank $k = 5413$ in order to achieve 128-bit post-quantum security for public-key encryption when accounting for the worst-case impact of Grover's algorithm [1]. The classical security level for these parameters is about 266-bit [5]. This recommended parameter set results in a private key of about 13 kB, and a public key of about 1022 kB. These parameters provide maximum security for a public key of at most 1 MB [5]. Our tunable design is able to achieve these parameters, and many others, depending on the user's needs.

The Niederreiter cryptosystem consists of three operations: key generation, encryption, and decryption. In this paper, we are focusing on the implementation of the most expensive operation in the Niederreiter cryptosystem: the key generation. The industry PKCS #11 standard defines a platform-independent API for cryptographic tokens, e.g., hardware security modules (HSM) or smart cards, and explicitly contains functions for public-private key-pair generation [19]. Furthermore, hardware crypto accelerators, e.g., for IBM's z Systems, have dedicated key-generation functions. These examples show that efficient hardware implementations for key generation will also be required for post-quantum schemes. We selected FPGAs as our target platform since they are ideal for hardware development and testing; most parts of the hardware code can also be re-used for developing an ASIC design.

Due to the confidence in the Niederreiter cryptosystem, there are many publications on hardware implementations related to this cryptosystem, e.g., [13,15,24]. We are only aware of one publication [24] that presents a hardware implementation of the key-generation algorithm. The key-generation hardware design in [24], however, uses fixed, non-tunable security and design parameters, which do not meet the currently recommended post-quantum security level, and has a potential security flaw by using a non-uniform permutation, which may lead to practical attacks.

Contributions. This paper presents the first post-quantum secure, efficient, and tunable FPGA-based implementation of the key-generation algorithm for the Niederreiter cryptosystem using binary Goppa codes. The contributions are:

- a key generator with tunable parameters, which uses code-generation to generate vendor-neutral Verilog HDL code,
- a constructive, constant-time approach for generating an irreducible Goppa polynomial,
- an improved hardware implementation of Gaussian systemizer which works for any large-sized matrix over any binary field,
- a new hardware implementation of Gao-Mateer additive FFT for polynomial evaluation,
- a new hardware implementation of Fisher-Yates shuffle for obtaining uniform permutations, and
- design testing using Sage reference code, iVerilog simulation, and output from real FPGA runs.

Source code. The source code is available as Open Source at `http://caslab.csl.yale.edu/code/keygen`.

2 Niederreiter Cryptosystem and Key Generation

The first code-based public-key encryption system was given by McEliece in 1978 [16]. The private key of the McEliece cryptosystem is a randomly chosen irreducible binary Goppa code $\mathcal{G}$ with a generator matrix G that corrects up to t errors. The public key is a randomly permuted generator matrix $G^{\text{pub}} = SGP$ that is computed from G and the secrets P (a permutation matrix) and S (an invertible matrix). For encryption, the sender encodes the message m as a codeword and adds a secret error vector e of weight t to get ciphertext $c = mG^{\text{pub}} \oplus e$. The receiver computes $cP^{-1} = mSG \oplus eP^{-1}$ using the secret P and decodes m using the decoding algorithm of $\mathcal{G}$ and the secret S. Without knowledge of the code G, which is hidden by the secrets S and P, it is computationally hard to decrypt the ciphertext. The McEliece cryptosystem with correct parameters is believed to be secure against quantum-computer attacks.

In 1986, Niederreiter introduced a dual variant of the McEliece cryptosystem by using a parity check matrix H for encryption instead of a generator matrix [18]. For the Niederreiter cryptosystem, the message m is encoded as a

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 $\mathrm{GF}(2^m)$.
2 Choose a random polynomial $g(x)$ 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 $K$.
```

weight-t error vector e of length n; alternatively, the Niederreiter cryptosystem can be used as a key-encapsulation scheme where a random error vector is used to derive a symmetric encryption key. For encryption, e is multiplied with H and the resulting syndrome is sent to the receiver. The receiver decodes the received syndrome, and obtains e. Originally, Niederreiter used Reed-Solomon codes for which the system has been broken [25]. However, the scheme is believed to be secure when using binary Goppa codes. Niederreiter introduced a trick to compress H by computing the systemized form of the public key matrix. This trick can be applied to some variants of the McEliece cryptosystem as well.

We focus on the Niederreiter cryptosystem due to its compact key size and the efficiency of syndrome decoding algorithms. As the most expensive operation in the Niederreiter cryptosystem is key generation, it is often omitted from Niederreiter implementations on FPGAs due to its large memory demand. Therefore, our paper presents a new contribution by implementing the key-generation algorithm efficiently on FPGAs.

2.1 Key Generation Algorithm

Algorithm 1 shows the key-generation algorithm for the Niederreiter cryptosystem. The system parameters are: m, the size of the binary field, t, the number of correctable errors, and n, the code length. Code rank k is determined as $k = n - mt$. We implemented Step 2 of the key-generation algorithm by computing an irreducible Goppa polynomial $g(x)$ of degree t as the minimal polynomial of a random element r from a polynomial ring over $\mathrm{GF}(2^m)$ using a power sequence $1, r, \ldots, r^t$ and Gaussian systemization in $\mathrm{GF}(2^m)$ (see Section 5). Step 3 requires the evaluation of $g(x)$ at points $\{\alpha_0, \alpha_1, \ldots, \alpha_{n-1}\}$. To achieve high efficiency, we decided to follow the approach of [4] which evaluates $g(x)$ at all elements of $\mathrm{GF}(2^m)$ using a highly efficient additive FFT algorithm

Param.	Description	Size (bits)
m	Size of the binary field	13
t	Correctable errors	119
n	Code length	6960
k	Code rank	5413

Config.	Description	Size (bits)
$g(x)$	Goppa polynomial	120×13
P	Permutation indices	8192×13
H	Parity check matrix	1547×6960
K	Public key	1547×5413

Table 1: Parameters and resulting configuration for the key generator.

(see Section 4.2). Therefore, we evaluate $g(x)$ at all $\alpha \in \mathrm{GF}(2^m)$ and then choose the required α_i using Fisher-Yates shuffle by computing a random sequence $(\alpha_0, \alpha_1, \ldots, \alpha_{n-1})$ from a permuted list of indices P. For Step 5, we use the efficient Gaussian systemization module for matrices over GF(2) from [26].

2.2 Structure of the Paper

The following sections introduce the building blocks for our key-generator module in a bottom-up fashion. First, we introduce the basic modules for arithmetic in $\mathrm{GF}(2^m)$ and for polynomials over $\mathrm{GF}(2^m)$ in Section 3. Then we introduce the modules for Gaussian systemization, additive FFT, and Fisher-Yates shuffle in Section 4. Finally, we describe how these modules work together to obtain an efficient design for key generation in Section 5. Validation of the design using Sage, iVerilog, and Stratix V FPGAs is presented in Section 6 and a discussion of the performance is in Section 7.

2.3 Reference Parameters and Reference Platform

We are using the recommended parameters from the PQCRYPTO project [20] shown in Table 1 as reference parameters; however, our design is fully parameterized and can be synthesized for any other valid parameter selection.

Throughout the paper (except for Table 9), performance results are reported from Quartus-synthesis results for the Altera Stratix V FPGA (5SGXEA7N), including Fmax (maximum estimated frequency) in MHz, Logic (logic usage) in Adaptive Logic Modules (ALMs), Mem. (memory usage) in Block RAMs, and Reg. (registers). Cycles are derived from iVerilog simulation. Time is calculated as quotient of Cycles and Fmax. Time×Area is calculated as product of Cycles and Logic.

3 Field Arithmetic

The lowest-level building blocks in our implementation are $\mathrm{GF}(2^m)$ finite field arithmetic and on the next higher level $\mathrm{GF}(2^m)[x]/f$ polynomial arithmetic.

3.1 $\mathrm{GF}(2^m)$ Finite Field Arithmetic

$\mathrm{GF}(2^m)$ represents the basic finite field in the Niederreiter cryptosystem. Our code for all the hardware implementations of $\mathrm{GF}(2^m)$ operations is generated by

Algorithm	Logic	Reg.	Fmax (MHz)
Schoolbook Algorithm	90	78	637
2-split Karatsuba Algorithm	99	78	625
3-split Karatsuba Algorithm	101	78	529
Bernstein	87	78	621

Table 2: Performance of different field multiplication algorithms for $\mathrm{GF}(2^{13})$.

code-generation scripts, which take in m as a parameter and then automatically generate the corresponding Verilog HDL code.

$\mathbf{GF(2^m)}$ Addition. In $\mathrm{GF}(2^m)$, addition corresponds to a simple bitwise xor operation of two m-bit vectors. Therefore, each addition has negligible cost and can often be combined with other logic while still finishing within one clock cycle, e.g., a series of additions or addition followed by multiplication or squaring.

$\mathbf{GF(2^m)}$ Multiplication. Multiplication over $\mathrm{GF}(2^m)$ is one of the most used operations in the Niederreiter cryptosystem. A field multiplication in $\mathrm{GF}(2^m)$ is composed of a multiplication in $\mathrm{GF}(2)[x]$ and a reduction modulo f, where f is a degree-m irreducible polynomial. For the case of $m = 13$, we use the pentanomial $f(x) = x^{13} + x^4 + x^3 + x + 1$ since there is no irreducible trinomial of degree 13. We are using plain schoolbook multiplication, which turns out to deliver good performance. Table 2 shows that the schoolbook version of $\mathrm{GF}(2^{13})$ multiplication achieves a higher Fmax while requiring less logic compared to several of our implementations using Karatsuba multiplication [14,17]. The performance of the schoolbook version is similar to Bernstein's operation-count optimized code [2]. We combine multiplication in $\mathrm{GF}(2)[x]$ and reduction modulo f such that one $\mathrm{GF}(2^m)$ multiplication only takes one clock cycle.

$\mathbf{GF(2^m)}$ Squaring. Squaring over $\mathrm{GF}(2^m)$ can be implemented using less logic than multiplication and therefore an optimized squaring module is valuable for many applications. However, in the case of the key-generation algorithm, we do not require a dedicated squaring module since an idle multiplication module is available in all cases when we require squaring. Squaring using $\mathrm{GF}(2^m)$ multiplication takes one clock cycle.

$\mathbf{GF(2^m)}$ Inversion. Inside the $\mathrm{GF}(2^m)$ Gaussian systemizer, elements over $\mathrm{GF}(2^m)$ need to be inverted. An element $a \in \mathrm{GF}(2^m)$ can be inverted by computing $a^{-1} = a^{|\mathrm{GF}(2^m)|-2}$. This can be done with a logarithmic amount of squarings and multiplications. For example, inversion in $\mathrm{GF}(2^{13})$ can be implemented using twelve squarings and four multiplications. However, this approach requires at least one multiplication circuit (repeatedly used for multiplications and squarings) plus some logic overhead and has a latency of at least several cycles in order to achieve high frequency. Therefore, we decided to use a pre-computed lookup table for the implementation of the inversion module. For inverting an element $\alpha \in \mathrm{GF}(2^m)$, we interpret the bit-representation of α as an integer value and use this value as the address into the lookup table. For convenience, we added an additional bit to each value in the lookup table that is set high in case the input element α can not be inverted, i.e., $\alpha = 0$. This saves additional logic that otherwise would be required to check the input value. Thus, the lookup table has a width of $m+1$ and a depth of 2^m, and each entry can be read in one clock cycle. The lookup table is read-only and therefore can be stored in either RAM or logic resources.

3.2 $\mathbf{GF(2^m)[x]/f}$ Polynomial Arithmetic

Polynomial arithmetic is required for the generation of the secret Goppa polynomial. $\mathrm{GF}(2^m)[x]/f$ is an extension field of $\mathrm{GF}(2^m)$. Elements in this extension field are represented by polynomials with coefficients in $\mathrm{GF}(2^m)$ modulo an irreducible polynomial f. We are using a sparse polynomial for f, e.g., the trinomial $x^{119} + x^8 + 1$, in order to reduce the cost of polynomial reduction.

Polynomial Addition. The addition of two degree-d polynomials with $d+1$ coefficients is equivalent to pair-wise addition of the coefficients in $\mathrm{GF}(2^m)$. Therefore, polynomial addition can be mapped to an xor operation on two $m(d+1)$-bit vectors and finishes in one clock cycle.

Polynomial Multiplication. Due to the relatively high cost of $\mathrm{GF}(2^m)$ multiplication compared to $\mathrm{GF}(2^m)$ addition, for polynomials over $\mathrm{GF}(2^m)$ Karatsuba multiplication [14] is more efficient than classical schoolbook multiplication in terms of logic cost when the size of the polynomial is sufficiently large.

Given two polynomials $A(x) = \sum_{i=0}^{5} a_i x^i$ and $B(x) = \sum_{i=0}^{5} b_i x^i$, schoolbook polynomial multiplication can be implemented in hardware as follows: Calculate $(a_5b_0, a_4b_0, \ldots, a_0b_0)$ and store the result in a register. Then similarly calculate $(a_5b_i, a_4b_i, \ldots, a_0b_i)$, shift the result left by $i \cdot m$ bits, and then add the shifted result to the register contents, repeat for all $i = 1, 2, \ldots, 5$. Finally the result stored in the register is the multiplication result (before polynomial reduction). One can see that within this process, 6×6 $\mathrm{GF}(2^m)$ multiplications are needed.

Karatsuba polynomial multiplication requires less finite-field multiplications compared to schoolbook multiplication. For the above example, Montgomery's six-split Karatsuba multiplication [17] requires only 17 field element multiplications over $\mathrm{GF}(2^m)$ at the cost of additional finite field additions which are cheap for binary field arithmetic. For large polynomial multiplications, usually several levels of Karatsuba are applied recursively and eventually on some low level schoolbook multiplication is used. The goal is to achieve a trade-off between running time and logic overhead.

The multiplication of two polynomials of degree $d = t - 1$ is a key step in the key-generation process for computing the Goppa polynomial $g(x)$. Table 3 shows the results of several versions of polynomial multiplication for $t = 119$, i.e., $d = 118$, using parameterized six-split Karatsuba by adding zero-terms in order to obtain polynomials with 120 and 24 coefficients respectively. On the lowest level, we use parameterized schoolbook multiplication. The most efficient approach for the implementation of degree-118 polynomial multiplication turned out to be one level of six-split Karatsuba followed by schoolbook multiplication, parallelized using twenty $\mathrm{GF}(2^{13})$ multipliers. Attempts using one more level of

Algorithm	Mult.	Cycles	Logic	Time×Area	Fmax (MHz)
1-level Karatsuba $17\times(20\times20)$	20	377	$11,860$	$4.47\cdot10^6$	342
2-level Karatsuba $17\times17\times(4\times4)$	16	632	$12,706$	$8.03\cdot10^6$	151
2-level Karatsuba $17\times17\times(4\times4)$	4	1788	$11,584$	$2.07\cdot10^7$	254

Table 3: Performance of different multiplication algorithms for degree-118 polynomials.

six-split Karatsuba did not notably improve area consumption (or even worsened it) and resulted in both more cycles and lower frequency. Other configurations, e.g., five-split Karatsuba on the second level or seven-split Karatsuba on the first level, might improve performance, but our experiments do not indicate that performance can be improved significantly.

In the final design, we implemented a one-level six-split Karatsuba multiplication approach, which uses a size-$\lceil\frac{d+1}{6}\rceil$ schoolbook polynomial multiplication module as its building block. It only requires 377 cycles to perform one multiplication of two degree-118 polynomials.

4 Key Generator Modules

The arithmetic modules are used as building blocks for the units inside the key generator, shown later in Figure 2. The main components are: two Gaussian systemizers for matrix systemization over $\mathrm{GF}(2^m)$ and $\mathrm{GF}(2)$ respectively, Gao-Mateer additive FFT for polynomial evaluation, and Fisher-Yates shuffle for generating uniformly distributed permutations.

4.1 Gaussian Systemizer

Matrix systemization is needed for generating both the private Goppa polynomial $g(x)$ and the public key K. Therefore, we require one module for Gaussian systemization of matrices over $\mathrm{GF}(2^{13})$ and one module for matrices over $\mathrm{GF}(2)$. We use a modified version of the highly efficient Gaussian systemizer from [26] and adapted it to meet the specific needs for Niederreiter key generation. As in [26], we are using an $N \times N$ square processor array to compute on column blocks of the matrix. The size of this processor array is parameterized and can be chosen to either optimize for performance or for resource usage.

The design from [26] only supports systemization of matrices over $\mathrm{GF}(2)$. An important modification that we applied to the design is the support of arbitrary binary fields — we added a binary-field inverter to the diagonal "pivoting" elements of the processor array and binary-field multipliers to all the processors. This results in a larger resource requirement compared to the GF(2) version but the longest path still remains within the memory module and not within the computational logic for computations on large matrices.

4.2 Gao-Mateer Additive FFT

Evaluating a polynomial $g(x) = \sum_{i=0}^{t} g_i x^i$ at n data points over $\mathrm{GF}(2^m)$ is an essential step for generating the parity check matrix H. Applying Horner's rule is a common approach for polynomial evaluation. For example, a polynomial $f(x) = \sum_{i=0}^{7} f_i x^i$ of degree 7 can be evaluated at a point $\alpha \in \mathrm{GF}(2^m)$ using Horner's rule as

$$\begin{aligned} f(\alpha) &= f_7\alpha^7 + f_6\alpha^6 + \cdots + f_1\alpha + f_0 \\ &= (((f_7\alpha + f_6)\alpha + f_5)\alpha + f_4)\ldots)\alpha + f_0 \end{aligned}$$

using 7 field additions and 7 field multiplications by α. More generically speaking, one evaluation of a polynomial of degree d requires d additions and d multiplications. Evaluating several points scales linearly and is easy to parallelize. The asymptotic time complexity of polynomial evaluation of a degree-d polynomial at n points using Horner's rule is $\mathrm{O}(n \cdot d)$.

In order to reduce this cost, we use a characteristic-2 additive FFT algorithm introduced in 2010 by Gao and Mateer [9], which was used for multipoint polynomial evaluation by Chou in 2013 [4]. This algorithm evaluates a polynomial at *all* elements in the field $\mathrm{GF}(2^m)$ using a number of operations logarithmic in the length of the polynomial. Most of these operations are additions, which makes this algorithm particularly suitable for hardware implementations. The asymptotic time complexity of additive FFT is $\mathrm{O}\big(2^m \cdot \log_2(d)\big)$.

The basic idea of this algorithm is to write f in the form $f(x) = f^{(0)}(x^2 + x) + xf^{(1)}(x^2 + x)$, where $f^{(0)}(x)$ and $f^{(1)}(x)$ are two half-degree polynomials, using *radix conversion*. The form of f shows a large overlap between evaluating $f(\alpha)$ and $f(\alpha+1)$. Since $(\alpha+1)^2 + (\alpha+1) = \alpha^2 + \alpha$ for $\alpha \in \mathrm{GF}(2^m)$, we have:

$$\begin{aligned} f(\alpha) &= f^{(0)}(\alpha^2 + \alpha) + \alpha f^{(1)}(\alpha^2 + \alpha) \\ f(\alpha + 1) &= f^{(0)}(\alpha^2 + \alpha) + (\alpha + 1) f^{(1)}(\alpha^2 + \alpha). \end{aligned}$$

Once $f^{(0)}$ and $f^{(1)}$ are evaluated at $\alpha^2 + \alpha$, it is easy to get $f(\alpha)$ by performing one field multiplication and one field addition. Now, $f(\alpha + 1)$ can be easily computed using one extra field addition as $f(\alpha + 1) = f(\alpha) + f^{(1)}(\alpha^2 + \alpha)$. Additive FFT applies this idea recursively until the resulting polynomials $f^{(0)}$ and $f^{(1)}$ are 1-coefficient polynomials (or in another word, constants). During the recursive operations, in order to use the α and $\alpha + 1$ trick, a *twisting* operation is needed for all the subspaces, which is determined by the new basis of $f^{(0)}$ and $f^{(1)}$. Finally, the 1-coefficient polynomials of the last recursion step are used to recursively evaluate the polynomial at all the 2^m data points over $\mathrm{GF}(2^m)$ in a concluding *reduction* operation.

Radix Conversion. Radix conversion converts a polynomial $f(x)$ of coefficients in $\mathrm{GF}(2^m)$ into the form of $f(x) = f^{(0)}(x^2 + x) + xf^{(1)}(x^2 + x)$. As a basic example, consider a polynomial $f(x) = f_0 + f_1x + f_2x^2 + f_3x^3$ of 4 coefficients with basis $\{1, x, x^2, x^3\}$. We compute the radix conversion as follows: Write the coefficients as a list $[f_0, f_1, f_2, f_3]$. Add the 4th element to the 3rd element and add the new 3rd element to the 2nd element to obtain $[f_0, f_1+f_2+f_3, f_2+f_3, f_3]$. This transforms the basis to $\{1, x, (x^2+x), x(x^2+x)\}$, we have

$$\begin{aligned} f(x) &= f_0 + (f_1+f_2+f_3)x + (f_2+f_3)(x^2+x) + f_3x(x^2+x) \\ &= \big(f_0 + (f_2+f_3)(x^2+x)\big) + x\big((f_1+f_2+f_3) + f_3(x^2+x)\big) \\ &= f^{(0)}(x^2+x) + xf^{(1)}(x^2+x) \end{aligned}$$

with $f^{(0)}(x) = f_0 + (f_2+f_3)x$ and $f^{(1)}(x) = (f_1+f_2+f_3) + f_3x$.

For polynomials of larger degrees, this approach can be applied recursively: Consider a polynomial $g(x) = g_0 + g_1x + g_2x^2 + g_3x^3 + g_4x^4 + g_5x^5 + g_6x^6 + g_7x^7$ of 8 coefficients. Write $g(x)$ as a polynomial with 4 coefficients, i.e.,

$$g(x) = (g_0 + g_1x) + (g_2 + g_3x)x^2 + (g_4 + g_5x)x^4 + (g_6 + g_7x)x^6.$$

Perform the same operations as above (hint: substitute x^2 with y and re-substitute back in the end) to obtain

$$\begin{aligned} g(x) &= (g_0 + g_1x) + \big((g_2 + g_3x) + (g_4 + g_5x) + (g_6 + g_7x)\big)x^2 \\ &\quad + \big((g_4 + g_5x) + (g_6 + g_7x)\big)(x^2+x)^2 + (g_6 + g_7x)x^2(x^2+x)^2 \\ &= (g_0 + g_1x) + \big((g_2 + g_4 + g_6) + (g_3 + g_5 + g_7)x\big)x^2 \\ &\quad + \big((g_4 + g_6) + (g_5 + g_7)x\big)(x^2+x)^2 + (g_6 + g_7x)x^2(x^2+x)^2 \end{aligned}$$

with basis $\{1, x, x^2, x^3, (x^2+x)^2, x(x^2+x)^2, x^2(x^2+x)^2, x^3(x^2+x)^2\}$.

Now, recursively apply the same process to the 4-coefficient polynomials $g^{(\mathrm{L})}(x) = g_0 + g_1x + (g_2+g_4+g_6)x^2 + (g_3+g_5+g_7)x^3$ and $g^{(\mathrm{R})}(x) = (g_4+g_6) + (g_5+g_7)x + g_6x^2 + g_7x^3$. This results in

$$\begin{aligned} g^{(\mathrm{L})}(x) &= g_0 + (g_1 + g_2 + g_3 + g_4 + g_5 + g_6 + g_7)x \\ &\qquad + (g_2 + g_3 + g_4 + g_5 + g_6 + g_7)(x^2+x) + (g_3 + g_5 + g_7)x(x^2+x), \text{ and} \\ g^{(\mathrm{R})}(x) &= (g_4 + g_6) + (g_5 + g_6)x + (g_6 + g_7)(x^2+x) + g_7x(x^2+x). \end{aligned}$$

Substituting $g^{(\mathrm{L})}(x)$ and $g^{(\mathrm{R})}(x)$ back into $g(x)$, we get

$$\begin{aligned} g(x) = g_0 & \\ &+ (g_1 + g_2 + g_3 + g_4 + g_5 + g_6 + g_7)x \\ &+ (g_2 + g_3 + g_4 + g_5 + g_6 + g_7)(x^2+x) \\ &+ (g_3 + g_5 + g_7)x(x^2+x) \\ &+ (g_4 + g_6)(x^2+x)^2 \\ &+ (g_5 + g_6)x(x^2+x)^2 \\ &+ (g_6 + g_7)(x^2+x)^3 \\ &+ (g_7)x(x^2+x)^3. \end{aligned}$$

with basis $\{1, x, (x^2+x)^1, x(x^2+x)^1, \ldots, (x^2+x)^3, x(x^2+x)^3\}$. This representation can be easily transformed into the form of $g(x) = g^{(0)}(x^2+x) + xg^{(1)}(x^2+x)$.

In general, to transform a polynomial $f(x)$ of 2^k coefficients into the form of $f = f^{(0)}(x^2+x) + xf^{(1)}(x^2+x)$, we need 2^i size-$2^{k-i}, i = 0, 1, .., k$ radix conversion operations. We will regard the whole process of transforming $f(x)$ into the form of $f^{(0)}(x^2+x) + xf^{(1)}(x^2+x)$ as one complete radix conversion operation for later discussion.

Twisting. As mentioned above, additive FFT applies Gao and Mateer's idea recursively. Consider the problem of evaluating an 8-coefficient polynomial $f(x)$ for all elements in $\mathrm{GF}(2^4)$. The field $\mathrm{GF}(2^4)$ can be defined as: $\mathrm{GF}(2^4) = \{0, a, \ldots, a^3+a^2+a, 1, a+1, \ldots, (a^3+a^2+a)+1\}$ with basis $\{1, a, a^2, a^3\}$. After applying the radix conversion process, we get $f(x) = f^{(0)}(x^2+x) + xf^{(1)}(x^2+x)$. As described earlier, the evaluation on the second half of the elements ("$\ldots+1$") can be easily computed from the evaluation results of the first half by using the α and $\alpha+1$ trick (for $\alpha \in \{0, a, \ldots, a^3+a^2+a\}$). Now, the problem turns into the evaluation of $f^{(0)}(x)$ and $f^{(1)}(x)$ at points $\{0, a^2+a, \ldots, (a^3+a^2+a)^2 + (a^3+a^2+a)\}$. In order to apply Gao and Mateer's idea again, we first need to *twist* the basis: By computing $f^{(0')}(x) = f^{(0)}((a^2+a)x)$, evaluating $f^{(0)}(x)$ at $\{0, a^2+a, \ldots, (a^3+a^2+a)^2+(a^3+a^2+a)\}$ is equivalent to evaluating $f^{(0')}(x)$ at $\{0, a^2+a, a^3+a, a^3+a^2, 1, a^2+a+1, a^3+a+1, a^3+a^2+1\}$. Similarly for $f^{(1)}(x)$, we can compute $f^{(1')}(x) = f^{(1)}((a^2+a)x)$. After the twisting operation, $f^{(0')}$ and $f^{(1')}$ have element 1 in their new basis. Therefore, this step equivalently twists the basis that we are working with. Now, we can perform radix conversion and apply the α and $\alpha+1$ trick on $f^{(0')}(x)$ and $f^{(1')}(x)$ recursively again.

The basis twisting for $f^{(0)}(x)$ and $f^{(1)}(x)$ can be mapped to a sequence of field multiplication operations on the coefficients. Let $\beta = \alpha^2+\alpha$. f_i denotes the i-th coefficient of a polynomial $f(x)$. For a degree-7 polynomial $f(x)$, we get

$$\begin{aligned}&[f_3^{(1')}, f_2^{(1')}, f_1^{(1')}, f_0^{(1')}, f_3^{(0')}, f_2^{(0')}, f_1^{(0')}, f_0^{(0')}]\\ =\ &[\beta^3 f_3^{(1)}, \beta^2 f_2^{(1)}, \beta f_1^{(1)}, f_0^{(1)}, \beta^3 f_3^{(0)}, \beta^2 f_2^{(0)}, \beta f_1^{(0)}, f_0^{(0)}].\end{aligned}$$

When mapping to hardware, this step can be easily realized by an entry-wise multiplication between the polynomial coefficients and powers of β, which are all independent and can be performed in parallel. Given a polynomial of 2^k coefficients from $\mathrm{GF}(2^m)$, each twisting step takes 2^k $\mathrm{GF}(2^m)$ multiplication operations. In our implementation, we use a parameterized parallel multiplier module that is composed of multiple $\mathrm{GF}(2^m)$ multipliers. The number of $\mathrm{GF}(2^m)$ multipliers is set as a parameter in this module, which can be easily adjusted to achieve an area and running time trade-off, as shown in Table 4.

Reduction. Evaluating a polynomial $f(x) \in \mathrm{GF}(2^m)[x]$ of 2^k coefficients at all elements in $\mathrm{GF}(2^m)$ requires k twisting and k radix conversion operations. The last radix conversion operation operates on 2^{k-1} polynomials of 2 coefficients of the form $g(x) = a+bx$. We easily write $g(x)$ as $g(x) = g^{(0)}(x^2+x) + xg^{(1)}(x^2+x)$ using $g^{(0)}(x) = a, g^{(1)}(x) = b$. At this point, we finish the recursive twist-then-radix-conversion process, and we get 2^k polynomials with only one coefficient. Now we are ready to perform the reduction step. Evaluation of these 1-coefficient polynomials simply returns the constant values. Then by using $g(\alpha) = g^{(0)}(\alpha^2 +$

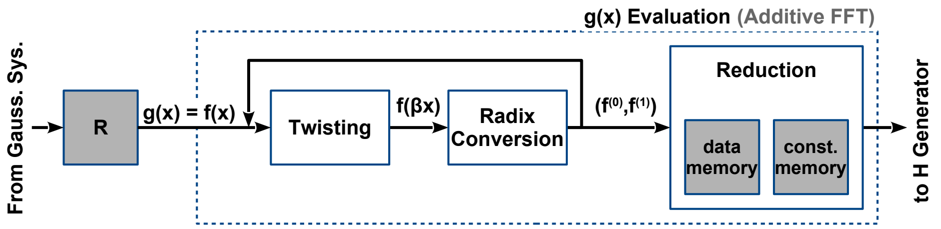

Fig. 1: Dataflow diagram of our hardware version of Gao-Mateer additive FFT. Functional units are represented as white boxes and memory blocks are represented as grey boxes.

Multipliers							
Twist	Reduction	Cycles	Logic	Time×Area	Mem.	Reg.	Fmax (MHz)
4	32	1188	$11,731$	$1.39 \cdot 10^7$	63	$27,450$	399
8	32	1092	$12,095$	$1.32 \cdot 10^7$	63	$27,470$	386
16	32	1044	$12,653$	$1.32 \cdot 10^7$	63	$27,366$	373
32	32	1020	$14,049$	$1.43 \cdot 10^7$	63	$26,864$	322

Table 4: Performance of additive FFT using different numbers of multipliers for twist.

$\alpha) + \alpha g^{(1)}(\alpha^2 + \alpha)$ and $g(\alpha + 1) = g(\alpha) + g^{(1)}(\alpha^2 + \alpha)$, we can recursively finish the evaluation of the polynomial f at all the 2^m points using $\lceil \log_2(t) \rceil$ recursion steps and 2^{m-1} multiplications in GF(2^m) in each step.

Non-recursive Hardware Implementation. We mapped the recursive algorithm to a non-recursive hardware implementation shown in Figure 1. Given a polynomial of 2^k coefficients, the twist-then-radix-conversion process is repeated for k times, and an array containing the coefficients of the resulting 1-coefficient polynomials is fed into the reduction module. Inside the reduction module, there are two memory blocks: A *data memory* and a *constants memory*. The data memory is initialized with the 1-coefficient polynomials and gets updated with intermediate reduction data during the reduction process. The constants memory is initialized with elements in the subspace of $f^{(0)}$ and $f^{(1)}$, which are pre-generated via Sage code. Intermediate reduction data is read from the data memory while subspace elements are read from the constants memory. Then the reduction step is performed using addition and multiplication submodules. The computed intermediate reduction results are then written back to the data memory. The reduction step is repeated until the evaluation process is finished and the final evaluation results are stored in the data memory.

Performance. Table 4 shows performance and resource-usage for our additive FFT implementation. For evaluating a degree-119 Goppa polynomial $g(x)$ at all the data points in GF(2^{13}), 32 finite filed multipliers are used in the reduction step of our additive FFT design in order to achieve a small cycle count while maintaining a low logic overhead. The twisting module is generated by a Sage script such that the number of multipliers can be chosen as needed. Radix con-

Algorithm 2: Fisher-Yates shuffle

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

m	Size ($= 2^m$)	Cycles (avg.)	Logic	Time×Area	Mem.	Reg.	Fmax (MHz)
13	8192	$23,635$	149	$3.52 \cdot 10^6$	7	111	335

Table 5: Performance of the Fisher-Yates shuffle module for 2^{13} elements.

version and twisting have only a small impact in the total cycle count; therefore, using only 4 binary filed multipliers for twisting results in good performance, with best Fmax. The memory required for additive FFT is only a small fraction of the overall memory consumption of the key generator.

4.3 Random Permutation: Fisher-Yates Shuffle

Computing a random list of indices $P = [\pi(0), \pi(1), \ldots, \pi(2^m - 1)]$ for a permutation $\pi \in S_{2^m}$ (here, S_i denotes the symmetric group on $\{0, 1, \ldots, i-1\}$), is an important step in the key-generation process. We compute P by performing Fisher-Yates shuffle [8] on the list $[0, 1, \ldots, 2^m - 1]$ and then using the first n elements of the resulting permutation. We choose Fisher-Yates shuffle to perform the permutation, because it requires only a small amount of computational logic. Algorithm 2 shows the Fisher-Yates shuffle algorithm.

We implemented a parameterized permutation module using a dual-port memory block of depth 2^m and width m. First, the memory block is initialized with contents $[0, 1, \ldots, 2^m - 1]$. Then, the address of port A decrements from $2^m - 1$ to 0. For each address A, a PRNG keeps generating new random numbers as long as the output is larger than address A. Therefore, our implementation produces a non-biased permutation (under the condition that the PRNG has no bias) but it is not constant-time. Once the PRNG output is smaller than address A, this output is used as the address for port B. Then the contents of the cells addressed by A and B are swapped. We improve the probability of finding a random index smaller than address A by using only $\lceil \log_2(A) \rceil$ bits of the PRNG output. Therefore, the probability of finding a suitable B always is at least 50%.

Since we are using a dual-port memory in our implementation, the memory initialization takes 2^{m-1} cycles. For the memory swapping operation, for each address A, first a valid address B is generated and data stored in address A and B is read from the memory in one clock cycle, then one more clock cycle is required for updating the memory contents. On average, $2^{m-1} + \sum_{i=1}^{m} \sum_{j=0}^{2^{i-1}-1} (\frac{2^i}{2^i - j} + 1)$ cycles are needed for our Fisher-Yates shuffle implementation. Table 5 shows performance data for the Fisher-Yates shuffle module.

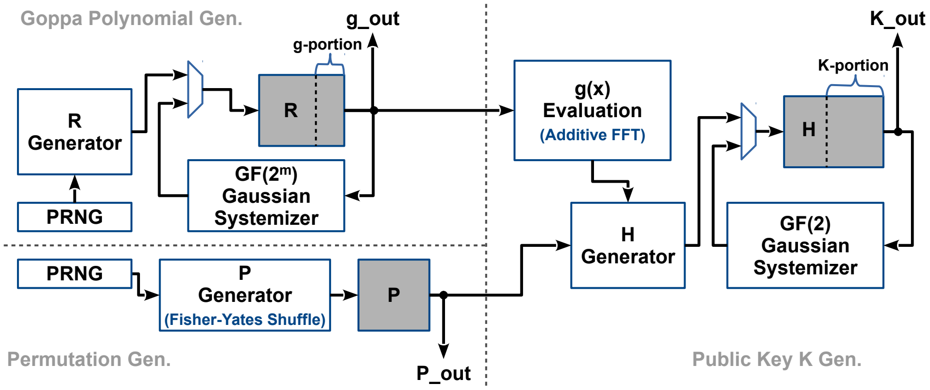

Fig. 2: Dataflow diagram of the key generator. Functional units are represented as white boxes and memory blocks are represented as grey boxes. The ports `g_out` and `P_out` are for the private-key data, and the port `K_out` is for the public-key data.

5 Key Generator for the Niederreiter Cryptosystem

Using two Gaussian systemizers, Gao-Mateer additive FFT, and Fisher-Yates shuffle, we designed the key generator as shown in Figure 2. Note that the design uses two simple PRNGs to enable deterministic testing. For real deployment, these PRNGs must be replaced with a cryptographically secure random number generator, e.g., [6]. We require at most m random bits per clock cycle per PRNG.

5.1 Private Key Generation

The private key consists of an irreducible Goppa polynomial $g(x)$ of degree t and a permuted list of indices P.

Goppa Polynomial $g(x)$. The common way for generating a degree-d irreducible polynomial is to pick a polynomial g of degree d uniformly at random, and then to check whether it is irreducible or not. If it is not, a new polynomial is randomly generated and checked, until an irreducible one is found. The density of irreducible polynomials of degree d is about $1/d$ [16]. When $d = t = 119$, the probability that a randomly generated degree-119 polynomial is irreducible gets quite low. On average, 119 trials are needed to generate a degree-119 irreducible polynomial in this way. Moreover, irreducibility tests for polynomials involve highly complex operations in extension fields, e.g., raising a polynomial to a power and finding the greatest common divisor of two polynomials. In the hardware key generator design in [24], the Goppa polynomial $g(x)$ was generated in this way, which is inefficient in terms of both time and area.

We decided to explicitly generate an irreducible polynomial $g(x)$ by using a deterministic, constructive approach. We compute the minimal (hence irreducible) polynomial of a random element in $\mathrm{GF}(2^m)[x]/h$ with $\deg(h) = \deg(g) = t$: Given a random element r from the extension field $\mathrm{GF}(2^m)[x]/h$, the minimal polynomial $g(x)$ of r is defined as the non-zero monic polynomial of least degree with coefficients in $\mathrm{GF}(2^m)$ having r as a root, i.e., $g(r) = 0$. The minimal polynomial of a degree-$(t-1)$ element from field $\mathrm{GF}(2^m)[x]/h$ is always of degree t and irreducible if it exists.

The process of generating the minimal polynomial $g(x) = g_0 + g_1x + \cdots + g_{t-1}x^{t-1} + x^t$ of a random element $r(x) = \sum_{i=0}^{t-1} r_i x^i$ is as follows: Since $g(r) = 0$, we have $g_0 + g_1r + \cdots + g_{t-1}r^{t-1} + r^t = 0$ which can be equivalently written using vectors as: $(1^T, r^T, \ldots, (r^{t-1})^T, (r^t)^T) \cdot (g_0, g_1, \ldots, g_{t-1}, 1)^T = 0$. Note that since $R = (1^T, r^T, \ldots, (r^{t-1})^T, (r^t)^T)$ is a $t \times (t+1)$ matrix while $g = (g_0, g_1, \ldots, g_{t-1}, 1)^T$ is a size-$(t+1)$ vector, we get

$$R \cdot g = \begin{bmatrix} 0 & r_{t-1} & \cdots & (r^t)_{t-1} \\ 0 & r_{t-2} & \cdots & (r^t)_{t-2} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & r_1 & \cdots & (r^t)_1 \\ 1 & r_0 & \cdots & (r^t)_0 \end{bmatrix} \begin{pmatrix} g_0 \\ g_1 \\ \vdots \\ g_{t-1} \\ 1 \end{pmatrix} = 0.$$

Now, we can find the minimal polynomial of r by treating g as variable and by solving the resulting system of linear equations for g. By expanding this matrix-vector multiplication equation, we get t linear equations which uniquely determine the solution for $(g_0, g_1, \ldots, g_{t-1})$. Solving systems of linear equations can be easily transformed into a matrix systemization problem, which can be handled by performing Gaussian elimination on the coefficient matrix R.

In our hardware implementation, first a PRNG is used, which generates t random m-bit strings for the coefficients of $r(x) = \sum_{i=0}^{t-1} r_i x^i$. Then the coefficient matrix R is calculated by computing the powers of $1, r, \ldots, r^t$, which are stored in the memory of the $\mathrm{GF}(2^m)$ Gaussian systemizer. We repeatedly use the polynomial multiplier described in Section 3.2 to compute the powers of r. After each multiplication, the resulting polynomial of t coefficients is written to the memory of the $\mathrm{GF}(2^m)$ Gaussian systemizer. (Our Gaussian-systemizer module operates on column-blocks of width N_R. Therefore, the memory contents are actually computed block-wise.) This multiply-then-write-to-memory cycle is repeated until R is fully calculated. After this step is done, the memory of the $\mathrm{GF}(2^m)$ Gaussian systemizer has been initialized with the coefficient matrix R.

After the initialization, the Gaussian elimination process begins and the coefficient matrix R is transformed into its reduced echelon form $[\mathbb{I}_t|g]$. Now, the right part of the resulting matrix contains all the unknown coefficients of the minimal polynomial g.

The part of memory which stores the coefficients of the Goppa polynomial $g(x)$ is shown as the "g-portion" in Figure 2. Later the memory contents stored in the g-portion are read out and sent to the $g(x)$ evaluation step, which uses the additive FFT module to evaluate the Goppa polynomial $g(x)$ at every point in field $\mathrm{GF}(2^m)$.

Table 6 shows the impact of different choices for the Gaussian-systemizer parameter N_R for a matrix of size 119×120 in $\mathrm{GF}(2^{13})$. N_R defines the size of the $N_R \times N_R$ processor array of the Gaussian systemizer [26] and implicitly the width of the memory that is used to store the matrix. It has an impact on the

N_R	Cycles	Logic	Time×Area	Mem.	Reg.	Fmax (MHz)
1	$922,123$	2539	$2.34 \cdot 10^9$	14	318	308
2	$238,020$	5164	$1.23 \cdot 10^9$	14	548	281
4	$63,300$	$10,976$	$6.95 \cdot 10^8$	13	1370	285

Table 6: Performance of the GF(2^m) Gaussian systemizer for $m = 13$ and $t = 119$, i.e., for a 119×120 matrix with elements from GF(2^{13}).

number of required memory blocks, because the synthesis tools usually require more memory blocks for wider memory words to achieve good performance. Furthermore, have to add zero-columns to the matrix to make the number of columns a multiple of N_R. However, for these parameters, the memory is used most efficiently for $N_R = 4$. When doubling N_R, the number of required cycles should roughly be quartered and the amount of logic should roughly be quadrupled. However, the synthesis results show a doubling pattern for the logic when $N_R = 1$, 2 and 4, which is probably due to some logic overhead that would vanish for larger N_R.

Random Permutation *P*. In our design, a randomly permuted list of indices of size 2^{13} is generated by the Fisher-Yates shuffle module and the permutation list is stored in the memory P in Figure 2 as part of the private key. Later memory P is read by the H generator which generates a permuted binary form the parity check matrix. In our design, since $n \leq 2^m$, only the contents of the first n memory cells need to be fetched.

5.2 Public Key Generation

As mentioned in Section 2, the public key K is the systemized form of the binary version of the parity check matrix H. In [24], the generation of the binary version of H is divided into two steps: first compute the non-permuted parity check matrix and store it in a memory block A, then apply the permutation and write the binary form of the permuted parity-check matrix to a new memory block B, which is of the same size as memory block A. This approach requires simple logic but needs two large memory blocks A and B.

In order to achieve better memory efficiency, we omit the first step, and instead generate a permuted binary form H' of the parity check matrix in one step. We start the generation of the public key K by evaluating the Goppa polynomial $g(x)$ at all $\alpha \in$ GF(2^m) using the Gao-Mateer additive FFT module. After the evaluation finishes, the results are stored in the data memory of the additive FFT module.

Now, we generate the permuted binary parity check matrix H' and store it in the memory of the GF(2) Gaussian systemizer. Suppose the permutation indices

N_H	Cycles	Logic	Time×Area	Mem.	Reg.	Fmax (MHz)
10	$150,070,801$	826	$1.24 \cdot 10^{11}$	663	678	257
20	$38,311,767$	1325	$5.08 \cdot 10^{10}$	666	1402	276
40	$9,853,350$	3367	$3.32 \cdot 10^{10}$	672	4642	297
80	$2,647,400$	$10,983$	$2.91 \cdot 10^{10}$	680	$14,975$	296
160	$737,860$	$40,530$	$2.99 \cdot 10^{10}$	720	$55,675$	290
320	$208,345$	$156,493$	$3.26 \cdot 10^{10}$	848	$213,865$	253

Table 7: Performance of the GF(2) Gaussian systemizer for a 1547 × 6960 matrix.

stored in memory P are $[p_0, p_1, \dots, p_{n-1}, \dots, p_{2^m-1}]$, then

$$H' = \begin{bmatrix} 1/g(\alpha_{p_0}) & 1/g(\alpha_{p_1}) & \cdots & 1/g(\alpha_{p_{n-1}}) \\ \alpha_{p_0}/g(\alpha_{p_0}) & \alpha_{p_1}/g(\alpha_{p_1}) & \cdots & \alpha_{p_{n-1}}/g(\alpha_{p_{n-1}}) \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{p_0}^{t-1}/g(\alpha_{p_0}) & \alpha_{p_1}^{t-1}/g(\alpha_{p_1}) & \cdots & \alpha_{p_{n-1}}^{t-1}/g(\alpha_{p_{n-1}}) \end{bmatrix}.$$

To generate the first column of H', the first element p_0 from P is fetched and stored in a register. Then, the corresponding polynomial evaluation value $g(\alpha_{p_0})$ is read out from the data memory of the additive FFT module. This value is then inverted using a GF(2^m) inverter. After inversion, we get $1/g(\alpha_{p_0})$ which is the first entry of the column. The second entry is calculated by a multiplication of the first entry row and α_{p_0}, the third entry again is calculated by a multiplication of the previous row and α_{p_0} and so on. Each time a new entry is generated, it is written to the memory of the GF(2) Gaussian systemizer (bit-wise, one bit per row). This computation pattern is repeated for all $p_0, p_1, \dots, p_{n-1}$ until H' is fully calculated. After this step, the memory of the GF(2) Gaussian systemizer contains H' and the Gaussian systemization process is started. (Again, this process is actually performed on column-blocks of width N_H due to the architecture of the Gaussian systemizer.)

If a fail signal from the GF(2) Gaussian systemizer is detected, i.e., the matrix cannot be systemized, key generation needs to be restarted. Otherwise, the left part of the matrix has been transformed into a $mt \times mt$ identity matrix and the right side is the $mt \times k$ public key matrix K labeled as "K-portion" in Figure 2.

Success Probability. The number of invertible $mt \times mt$ matrices over GF(2) is the order of the general linear group GL(mt, GF(2)), i.e., $\prod_{j=0}^{mt-1} 2^{mt} - 2^j$. The total number of $mt \times mt$ matrices over GF(2) is $2^{(mt)^2}$. Therefore, the probability of a random $mt \times mt$ matrix over GF(2) being invertible is $(\prod_{j=0}^{mt-1} 2^{mt} - 2^j)/2^{(mt)^2}$. For $mt = 13 \cdot 119 = 1547$, the probability is about 29%. Thus, on average we need about 3.5 attempts to successfully generate a key pair.

Performance. Table 7 shows the effect of different choices for parameter N_H on a matrix of size 1547×6960 in GF(2). Similar to the GF(2^m) Gaussian systemizer, N_H has an impact on the number of required memory blocks. When doubling N_H, the number of required cycles should roughly be quartered (which is the case for small N_H) and the amount of logic should roughly be quadrupled (which is the case for large N_H). The best time-area product is achieved for $N_H = 80$, because for smaller values the non-computational logic overhead is significant and for larger values the computational logic is used less efficiently. Fmax is mainly limited by the paths within the memory.

6 Design Testing

We tested our hardware implementation using a Sage reference implementation, iVerilog, and an Altera Stratix V FPGA (5SGXEA7N) on a Terasic DE5-Net FPGA development board.

Parameters and PRNG Inputs. First, we chose a set of parameters, which were usually the system parameters of the cryptosystem (m, t, and n, with $k = n - mt$). In addition, we picked two design parameters, N_R and N_H, which configure the size of the processor arrays in the GF(2^m) and GF(2) Gaussian systemizers. In order to guarantee a deterministic output, we randomly picked seeds for the PRNGs and used the same seeds for corresponding tests on different platforms. Given the parameters and random seeds as input, we used Sage code to generate appropriate input data for each design module.

Sage Reference Results. For each module, we provide a reference implementation in Sage using built-in Sage functions for field arithmetic, etc. Given the parameters, seeds, and input data, we used the Sage reference implementation to generate reference results for each module.

iVerilog Simulation Results. We simulated the Verilog HDL code of each module using a "testbench" top module and the iVerilog simulator. At the end of the simulation, we stored the simulation result in a file. Finally, we compared the simulation result with the Sage reference result. If these reference and simulation results matched repeatedly for different inputs, we assumed the Verilog HDL code to be correct.

FPGA Results. After we tested the hardware design through simulation, we synthesized the design for an Altera Stratix V FPGA using the Altera Quartus 16.1 tool chain. We used a PCIe interface for communication with the FPGA. After a test finished, we wrote the FPGA output to a file. Then we compared the output from the FPGA testrun with the output of the iVerilog simulation and the Sage reference results. If the outputs matched, we assumed the hardware design to be correct.

7 Evaluation

We synthesized the final design for an Altera Stratix V FPGA (5SGXEA7N) and for comparison for a Xilinx UltraScale+ VUP9 FPGA (e.g., used in the Amazon EC2 F1 instances). Based on the PQCRYPTO project [20] recommendations, the following system parameters were selected: $m = 13$, $t = 119$, $n = 6960$ and

Case	N_H	N_R	Cycles	Logic	Time×Area	Mem.	Fmax	Time
				Altera Stratix V				
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
				Xilinx Virtex Ultrascale+				
logic	40	1	$11,121,220$	$42,632$	$4.74 \cdot 10^{11}$	348.5	200 MHz	55.64 ms
bal.	80	2	$3,062,942$	$60,989$	$1.87 \cdot 10^{11}$	356	221 MHz	13.85 ms
time	160	4	$896,052$	$112,845$	$1.01 \cdot 10^{11}$	375	225 MHz	3.98 ms

Table 8: Performance of the key generator for parameters $m = 13$, $t = 119$, and $n = 6960$. All the numbers in the table come from compilation reports of the Altera and Xilinx tool chains respectively. For Xilinx, logic utilization is counted in LUTs.

Design	m	t	n	Cycles (avg.)	Freq.	Time (avg.)	Arch.
Shoufan et al. [24]	11	50	2048	$1.47 \cdot 10^7$	163 MHz[a]	90 ms	Virtex V
this work	11	50	2048	$2.72 \cdot 10^6$	168 MHz[b]	16 ms	Virtex V
Chou [7]	13	128	8192	$1.24 \cdot 10^9$	1–4 GHz[c]	1236–309 ms	Haswell
this work	13	128	8192	$4.30 \cdot 10^6$	215 MHz[a]	20 ms	Stratix V

Table 9: Comparison with related work. Cycles and Time are average values, taking into account failure cases. [a]Actual frequency running on FPGAs. [b]Fmax reported by the Xilinx tool chain. [c]Available for a range of frequencies.

$k = 5413$ (note $k = n - mt$). These parameters were specified in [5] for a target public key size of about 1 MB. They provide a classical security level of about 266-bit which corresponds to a post-quantum security level of at least 128-bit.

Due to the large size of the permuted parity check matrix H, generating the public key K by doing matrix systemization on the binary version of H is usually the most expensive step both in logic and cycles in the key-generation algorithm. In our key generator, independently of the security parameters, the design can be tuned by adjusting N_R and N_H, which configure the size of the processor array of the GF(2^m) and GF(2) Gaussian systemizer respectively. Table 6 and Table 7 show that by adjusting N_R and N_H in the two Gaussian systemizers, we can achieve a trade-off between area and performance for the key generator.

Table 8 shows performance data for three representative parameter choices: The *logic* case targets to minimize logic consumption at the cost of performance, the *time* case focuses on maximising performance at the cost of resources, and the *balanced* case (bal.) attempts to balance logic usage and execution time.

Comparison of our results with other Niederreiter key-generator implementations on FPGAs is not easy. Table 9 gives an attempt of comparing our result to the performance data given in [24]. The design in [24] occupies about 84% of the target FPGA for their entire Niederreiter-cryptosystem implementation including key generation, encryption, decryption, and IO. Our design requires only about 52% of the logic (for $N_H = 30$ and $N_R = 2$), but only for the key generation. The design in [24] practically achieves a frequency of 163 MHz while we can only report estimated synthesis results for Fmax of 168 MHz for our design. Computing a private-public key pair using the design in [24] requires about 90 ms on average (their approach for generating the Goppa polynomial is not constant time and the key-generation procedure needs to be repeated several times until the Gaussian systemization of the public key succeeds). Our design requires about 16 ms on average at 168 MHz.

We also compare our design to a highly efficient CPU implementation from [7] in Table 9. The results show that our optimized hardware implementation competes very well with the CPU implementation. In this case, we ran our implementation on an Altera Stratix V FPGA. The actual frequency that we achieved fits well to the estimated frequencies for Stratix V in Table 8.

8 Conclusion

This work presents a new FPGA-based implementation of the key-generation algorithm for the Niederreiter cryptosystem using binary Goppa codes. It is the first hardware implementation of a key generator that supports currently recommended security parameters (and many others due to tunable parameters). Our design is based on novel hardware implementations of Gaussian systemizer, Gao-Mateer additive FFT, and Fisher-Yates shuffle.

Acknowledgments. We want to thank Tung Chou for his invaluable help, in particular for discussions about the additive FFT implementation.

References

1. 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, accessed June 22, 2017
2. Bernstein, D.J.: High-speed cryptography in characteristic 2, http://binary.cr.yp.to/m.html, accessed March 17, 2017
3. Bernstein, D.J., Buchmann, J., Dahmen, E. (eds.): Post-Quantum Cryptography. Springer (2009)
4. Bernstein, D.J., Chou, T., Schwabe, P.: McBits: fast constant-time code-based cryptography. In: Bertoni, G., Coron, J.S. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2013. LNCS, vol. 8086, pp. 250–272. Springer (2013)
5. Bernstein, D.J., Lange, T., Peters, C.: Attacking and defending the mceliece cryptosystem. In: Buchmann, J., Ding, J. (eds.) Post-Quantum Cryptography — PQCrypto 2008. LNCS, vol. 5299, pp. 31–46. Springer (2008)
6. 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.) Cryptographic Hardware and Embedded Systems – CHES 2013. LNCS, vol. 8086, pp. 179–196. Springer (2013)

7. Chou, T.: McBits revisited. In: Fischer, W., Homma, N. (eds.) Cryptographic Hardware and Embedded Systems. LNCS, Springer (2017), to appear with this paper.
8. Fisher, R.A., Yates, F.: Statistical tables for biological, agricultural and medical research. Oliver and Boyd (1948)
9. Gao, S., Mateer, T.: Additive fast fourier transforms over finite fields. IEEE Transactions on Information Theory 56(12), 6265–6272 (2010)
10. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: Symposium on the Theory of Computing – STOC '96. pp. 212–219. ACM (1996)
11. 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.) Advances in Cryptology – ASIACRYPT 2016. LNCS, vol. 10031, pp. 789–815. Springer (2016)
12. Heyse, S., von Maurich, I., Güneysu, T.: Smaller keys for code-based cryptography: QC-MDPC McEliece implementations on embedded devices. In: Bertoni, G., Coron, J.S. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2013. LNCS, vol. 8086, pp. 273–292. Springer (2013)
13. Hu, J., Cheung, R.C.C.: An application specific instruction set processor (ASIP) for the Niederreiter cryptosystem. Cryptology ePrint Archive, Report 2015/1172 (2015)
14. Karatsuba, A., Ofman, Y.: Multiplication of multidigit numbers on automata. Soviet Physics Doklady 7, 595–596 (1963)
15. Massolino, P.M.C., Barreto, P.S.L.M., Ruggiero, W.V.: Optimized and scalable co-processor for McEliece with binary Goppa codes. ACM Transactions on Embedded Computing Systems 14(3), 45 (2015)
16. McEliece, R.J.: A public-key cryptosystem based on algebraic coding theory. DSN Progress Report 42–44, 114—116 (1978)
17. Montgomery, P.L.: Five, six, and seven-term Karatsuba-like formulae. IEEE Transactions on Computers 54(3), 362–369 (2005)
18. Niederreiter, H.: Knapsack-type cryptosystems and algebraic coding theory. Problems of Control and Information Theory 15, 19–34 (1986)
19. PKCS #11 base functionality v2.30, ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-11/v2-30/pkcs-11v2-30b-d6.pdf (page 172), accessed June 20, 2017
20. Post-quantum cryptography for long-term security PQCRYPTO ICT-645622, https://pqcrypto.eu.org/, accessed March 17, 2017
21. Rostovtsev, A., Stolbunov, A.: Public-key cryptosystem based on isogenies. Cryptology ePrint Archive, Report 2006/145 (2006)
22. Shor, P.W.: Algorithms for quantum computation: Discrete logarithms and factoring. In: Foundations of Computer Science – FOCS '94. pp. 124–134. IEEE (1994)
23. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM review 41(2), 303–332 (1999)
24. 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)
25. Sidelnikov, V.M., Shestakov, S.O.: On insecurity of cryptosystems based on generalized Reed-Solomon codes. Discrete Mathematics and Applications 2(4), 439–444 (1992)
26. Wang, W., Szefer, J., Niederhagen, R.: Solving large systems of linear equations over GF(2) on FPGAs. In: Reconfigurable Computing and FPGAs – ReConFig 2016. pp. 1–7. IEEE (2016)