CRC-Oriented Error Detection Architectures of Post-quantum Cryptography Niederreiter Key Generator on FPGA

Alvaro Cintas-Canto
School of Technology and Innovation
Marymount University
Arlington, VA, USA
acintas@marymount.edu

Mehran Mozaffari-Kermani
Department of Computer Science and Engineering
University of South Florida
Tampa, FL, USA
mehran2@usf.edu

Reza Azarderakhsh
Department of Computer and Electrical Engineering and Computer Science
Florida Atlantic University
Boca Raton, FL, USA
razarderakhsh@fau.edu

Kris Gaj
Department of Electrical and Computer Engineering
George Mason University
Fairfax, VA, USA
kgaj@gmu.edu

2022 IEEE Nordic Circuits and Systems Conference (NorCAS) | 979-8-3503-4550-6/22/$31.00 ©2022 IEEE | DOI: 10.1109/NORCAS57515.2022.9934378

***Abstract*—Providing error detection constructions for Internet of nano-Things in constrained applications is of prominent importance. The Niederreiter cryptosystem falls into the category of code-based public-key cryptography. It is a relatively well-established scheme, but its key size and performance overheads have traditionally hindered its efficiency to be utilized for traditional computers. However, with the arrival of quantum computers, the Niederreiter cryptosystem is believed to be secure against attacks enabled by such computers, even though it has been previously shown that it is still vulnerable to fault injection and natural hardware defects. In this paper, we present fault detection schemes for the different blocks in the key generation of the Niederreiter cryptosystem using binary Goppa codes. These blocks perform finite field operations such as addition, multiplication, squaring, and inversion. The schemes are derived for different parameter sizes in order to have more flexibility and be able to choose according to the overheads to be tolerated and the required level of security. Moreover, we implement our fault detection schemes on Xilinx field-programmable gate array (FPGA) family Kintex UltraScale+ (device xcku5p-ffvd900-1-i) to benchmark the overhead induced of the proposed approaches.**

***Index Terms*—Fault detection, field-programmable gate array (FPGA), Niederreiter cryptosystem, post-quantum cryptography.**

I. Introduction

Deeply-embedded hardware architectures with tight constraints are wide-spread in today's Internet of nano-Things era. Providing security and fault detection schemes for such constrained usage models, i.e., aerospace and defense systems, medical electronics, or scientific instruments, is of prominent importance. With the arrival of quantum computers, traditional public key cryptosystems will be no longer secure against attacks enabled by such computation power [1]. In the past few years, Post-Quantum Cryptography (PQC) has gained attention, especially after the National Institute of Standards and Technology (NIST) began the process to standardize quantum-resistant public-key cryptographic algorithms [2].

The McEliece cryptosystem was one of the first public-key cryptography schemes. It was published in 1978 [3]. In 1986, a related cryptosystem was proposed by Niederreiter et al. [4]. While the McEliece cryptosystem used as its primary building block the binary Goppa code, the Niederreiter cryptosystem was initially based on the Reed-Salomon code. However, after the Reed-Salomon variant was shown insecure in [5], the Niederreiter cryptosystem was revised and became based on the same Goppa code as the McEliece cryptosystem. In 2017, a submission to the NIST standardization process, called Classic McEliece, was developed. Classic McEliece is a slightly revised version of the Niederreiter cryptosystem. Compared to the Niederreiter cryptosystem, it extends the public-key encryption scheme to a key encapsulation mechanism (KEM). However, it retains all other major features of the Niederreiter cryptosystem, including security guarantees. Thus, the Niederreiter cryptosystem can be treated as a subset of the Classic McEliece and use the same parameters as this NIST PQC candidate. In Round 1, Classic McEliece had only two parameter sets, both at the security level 5, equivalent to the security of AES-256. In Round 2, 8 additional sets have been added, two at security level 1 (corresponding to the security of AES-128), two at security level 3 (corresponding to the security of AES-192), and four at security level 5.

The sensitivity of both the original McEliece cryptosystem and revised (binary Goppa-code-based) Niederreiter cryptosystem to fault analysis was demonstrated in [6]. Shortly after, timing side-channel attacks were investigated in [7] and [8]. Most recently, fixed-vs-random test vector leakage assessment (TVLA), partial key exposure attacks, and backdoors were used to recover a secret key of the McEliece cryptosystem in [9], [10], and [11], respectively.

In this paper, we evaluate the resistance of the key generation function of the Niederreiter cryptosystem against fault attacks. One of the primary parameters of this cryptosystem affecting our study is m, which defines the size of the underlying Galois field $GF(q)$ through the relation $q = 2^m$. In the NIST process, the Round 1 Classic McEliece used $m = 13$. Starting from Round 2, values $m = 12$ and 13 were employed. Additionally, there exists prominent earlier work with different security parameters based on security necessities and system constraints [12]-[14]. Therefore, our work provides flexibility and tunable security by covering such works as well through the choice of the range $11 \leq m \leq 14$.

There has been previous work on countering fault attacks and providing reliability for traditional cryptography and PQC [15]-[26]. Moreover, fault detection techniques for composite fields and finite fields are investigated in [27]-[29]. In [27], multi-bit parity prediction is used to detect faults in the composite fields used in the McEliece cryptosystem. Error detection schemes based on cyclic redun-

dancy checks (CRC-5 and CRC-10) are introduced for the finite field multipliers used by the Luov's cryptosystem in [28] and by cryptosystems using the NIST field $GF(2^{163})$ in [29]. This work adds two new CRC schemes, i.e., CRC-3 and CRC-4, overcoming the limitations of parity schemes, vulnerable to intelligent fault injections and a 50% error coverage. The CRC sizes selected in this paper are lower than those in [28] and [29] to minimize the overheads, since this work is intended to provide fault detection to deeply-embedded systems. Additionally, we implement for the first time error detection schemes for the Key Generator of the Niederreiter cryptosystem.

II. Preliminaries

McEliece proposed the first code-based public-key encryption system in 1978 [3]. The McEliece cryptosystem's private key is a randomly selected irreducible Goppa binary code with a generator matrix that corrects errors of up to t. In 1986, Niederreiter presented a dual version of the McEliece cryptosystem, using a parity check matrix *H* instead of a generator matrix for encryption. For the Niederreiter cryptosystem, the message is encoded as a 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.

There are three main operations in the Niederreiter cryptosystem: (i) the generation of a private key and a public key (key generation), (ii) the creation of a ciphertext to prevent unauthorized access to the message (encryption), and (iii) the decoding of the message previously encrypted (decryption). Our work focuses on the key generation, which is the most expensive operation in the Niederreiter cryptosystem. In this process, a public key needed for the encryption of the message and a private key needed to decrypt the message are created. There are four important parameters which can highly affect how secure the cryptosystem is: m, which is the extension field dimension; t, which is the number of correctable errors; the code length n; and finally, the dimension k.

To create the private and public keys needed for the encryption and decryption processes, n elements in $GF(2^m)$ are chosen randomly, i.e., $\alpha_0, \alpha_1, ..., \alpha_{n-1}$, and a random permuted list of indices *P* is produced. Next, a monic irreducible polynomial in the form $g(x) = x^t + g_{t-1}x^{t-1} + ... + g_1x + g_0$, with degree *t* is created. The irreducible polynomial (denoted as Goppa polynomial) and the permuted list of indices *P*, form the private key $(g(x), (\alpha_0, \alpha_1, ..., \alpha_{n-1}))$. To derive the public key *K*, a parity check matrix *H* is then computed with the following form:

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

Computing *H* is one of the most time-consuming processes in the Niederreiter cryptosystem since it needs addition, multiplication, and inversion in $GF(2^m)$. Each element of the matrix *H* is then replaced with a column of m bits using the permuted list of indices *P*, obtaining a binary parity check matrix *H'*. Lastly, the key generator transforms the modified $mt \times n$ *H'* matrix to its systematic form $[I_{mt}|K]$, where I_{mt} is the identity matrix of size mt, returning the public key *K*.

In the process of encryption, the sender encodes the message as an error vector e of length n and weight at most t and computes the ciphertext by multiplying the extended public key $[I_{mt}|K]$ with the plaintext. Such operation produces a specific syndrome which is sent to the receiver as the ciphertext c. When the receiver obtains such ciphertext, the original plaintext is recovered by using the private key $(g(x), (\alpha_0, \alpha_1, ..., \alpha_{n-1}))$.

III. Proposed Fault Detection Architectures

In this work, we propose fault detection schemes based on CRC-3 and CRC-4. These schemes aim to detect transient and permanent internal faults on the Key Generator due to natural or malicious faults, e.g., differential fault analysis. Due to technological limitations, an adversary may not be able to flip precisely one bit to capture sensitive information. As a result, techniques that can identify multiple stuck-at faults (stuck-at 0 and stuck-at 1) in addition to single faults need to be explored [30].

CRC was first proposed in 1961 and it is based on the theory of cyclic error-correcting codes. To implement CRC, a generator polynomial $g_p(\beta)$ is selected and a fixed number of check bits are appended to the data, which are inspected when the output is received to detect any errors. In our proposed error schemes, we obtain formulations for predicted CRC checksums and they are compared via XOR gates with the actual CRC. The security parameters used are shown in Table I, where it can be seen how m varies from 11 to 14 bits [31]. Moreover, the set for $m = 13$ corresponds to the Classic McEliece NIST submission; although in the last submission, $m = 12$ with $t = 64$ and $n = 3488$ is also proposed [32].

A. Functional Units of the Key Generator

As mentioned earlier, the main operations within the Key Generator are $GF(2^m)$ addition, multiplication, and inversion. Fig. 1 shows the overall architecture of the Key Generator (adopted from [33]). In Fig. 1, the circles are memory blocks, while the squares are functional units. As it is shown, the Key Generator uses PRNGs to allow deterministic testing, which would have to be replaced with a cryptographically secure random number generator for real scenarios. The private key is formed by *G_out* and *P_out*, which are provided by the *R Generator* and the *P Generator*, respectively. Such generators provide a generator matrix G and a permutation matrix P needed to obtain the private key. Next, the $g(x)$ *Evaluation* block is used to perform high degree polynomial multiplications efficiently and lastly, the *H Generator* computes a parity check matrix *H* needed to obtain the public key. *Gaussian Systemizers* are also used in this design for matrix systematization over $GF(2^m)$ and $GF(2)$, needed to generate both public and private keys.

The scope of this paper is to propose error detection schemes capable of detecting natural and malicious faults in different finite field arithmetic of the Key Generator within the Niederreiter cryptosystem. Since every unit besides the *P Generator* performs finite field arithmetic, our schemes

Table I
SECURITY PARAMETERS USED TO DERIVE THE DIFFERENT FAULT DETECTION SCHEMES

m	t	n	Security Level
11	27	2,048	81
12	64	3,488	146
13	119	6,960	263
14	15	16,384	90

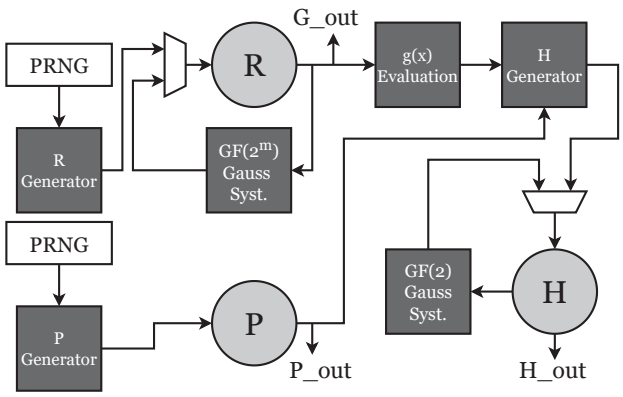

Figure 1. Overall architecture of the Key Generator.

provide error detection to the entire Key Generator besides the *P Generator*, which mainly swaps elements and can be done be rewiring. Next, we present an overview of all the different blocks used in the Key Generator. Readers interested in knowing more about the other blocks of the Niederreiter cryptosystem can refer to [33].

1) R Generator: As shown in Fig. 1, a PRNG is employed initially, generating *t* random *m*-bit vectors for the coefficients of $r(x) = \sum_{i=0}^{t-1} r_i x^i$. The coefficient matrix *R* is then computed by calculating the powers of $1, r, ..., r^t$, using a polynomial multiplier that performs the classical schoolbook multiplication utilizing $GF(2^m)$ multipliers. After each multiplication, the results are stored in the $GF(2^m)$ *Gaussian Systemizer* until the entire coefficient matrix *R* has been initialized.

2) Gaussian Systemizer: Two different *Gaussian Systemizer* units are used in the Key Generator for matrix systemization, i.e., $GF(2)$ *Gaussian Systemizer* and $GF(2^m)$ *Gaussian Systemizer*. The first one is needed to calculate the public key *K* while the latter one is utilized to generate the Goppa polynomial $g(x)$. To perform matrix systemization over $GF(2)$, once the matrix *H'* is completely stored in the $GF(2)$ *Gaussian Systemizer* unit, *R* is divided into two sub-matrices. Through Gaussian elimination and backward substitution, its left part is then reduced into an $mt \times mt$ identity matrix and its right part becomes the public key matrix *K* of size $mt \times k$. In the matrix systemization over $GF(2^m)$, a binary field inverter and binary field multipliers are added to support arbitrary binary fields. Once the coefficient matrix *R* is completely stored in the memory of the $GF(2^m)$ *Gaussian Systemizer*, the Gaussian elimination process starts. This process transforms *R* into its reduced echelon form, containing all of the unknown coefficients of the minimum polynomial $g(x)$ in the right portion of such matrix.

3) P Generator: The *P Generator* produces a permuted list of indices *P* needed for the calculation of both public key (precisely to generate the permuted binary check matrix *H')* and private key (formed by an irreducible Goppa polynomial $g(x)$ and *P*). The Fisher-Yates shuffle is used to perform the permutation. To do so, a dual-port memory block of depth 2^m formed by m vectors is initialized. The port *A* address decrements from $2^m - 1$ to 0 and as long as the output is greater than address *A*, a PRNG produces new random numbers for each address *A*. This output is utilized as the address for port *B* whenever the PRNG output is smaller than address *A*. The contents of the *A* and *B* addresses are then swapped to finally obtain a shuffled array *A*.

4) $G(x)$ Evaluation: The $G(x)$ *Evaluation* unit uses the Gao-Mateer Additive FFT scheme. Gao-Mateer Additive FFT is a recursive algorithm that reduces the amount of finite field multiplications ever further by transforming the polynomial $g(x)$ into the form $g(x) = g^{(0)}(x^2 + x) + xg^{(1)}(x^2 + x)$, where $g^{(0)}(x)$ and $g^{(1)}(x)$ are half-degree polynomials, using radix conversion [34].

5) H Generator: After the $G(x)$ *Evaluation* unit is done and the permuted list of indices *P* is set, the indices *P* become the new indices of the α elements from the *H* matrix to calculated the permuted binary parity check matrix *H'*, e.g., α_0 becomes α_{p_0}(where p_i's when $0 \leq i \leq n-1$ are the permuted indices *P*). Each entry is calculated by using a finite field inverter and they get stored in the $GF(2)$ *Gaussian Systemizer* which will reduced *H'* to obtain the public key matrix *K*.

B. Error Detection Schemes

1) $GF(2^m)$ Addition and $GF(2^m)$ Multiplication: To perform $GF(2^m)$ addition, we use one of the modules used to perform $GF(2^m)$ multiplication as described in [35], the *sum* module. $GF(2^m)$ multiplication is done using three different modules: *sum*, α, and *pass-thru* modules. The *sum* module adds two elements in $GF(2^m)$ using m two-input XOR gates; the *pass-thru* module multiplies a $GF(2^m)$ element by a $GF(2)$ element; and the α module multiplies an element of $GF(2^m)$ by α and it reduces the result modulo $f(\alpha)$. The multiplication of any element in $GF(2^m)$ by α gives

$$A(\alpha) \cdot \alpha = a_{m-1}\alpha^m + a_{m-2}\alpha^{m-1} + ... + a_0\alpha, \qquad (1)$$

where $\alpha^m = f_{m-1}\alpha^{m-1} + f_{m-2}\alpha^{m-2} + ... + f_0 \bmod f(\alpha)$.

In multiplication using polynomial basis, the inputs A and B are elements of $GF(2^m)$ in the form of $A = \sum_{i=0}^{m-1} a_i\alpha^i,\ a_i \epsilon \{0, 1\}$ and $B = \sum_{i=0}^{m-1} b_i\alpha^i,\ b_i \epsilon \{0, 1\}$, where a^i and b^i are the coordinates of each input. The multiplication of these two elements can be represented as:

$$A \cdot B = A \cdot \sum_{i=0}^{m-1} b_i\alpha^i = \sum_{i=0}^{m-1} b_i(A\alpha^i).$$

This, in turn, obtains the output C as follows:

$$C = A \cdot B \bmod f(\alpha) = \sum_{i=0}^{m-1} b_i X^{(i)},$$

where $X^{(i)} = \alpha \cdot X^{(i-1)} \bmod f(\alpha)$ for $1 \leq i \leq m-1$, and $X^{(0)} = A$.

For the α module, $g_{p_1}(\beta) = \beta^3 + \beta + 1$ is used as the generator polynomial with CRC-3, and $g_{p_2}(\beta) = \beta^4 + \beta + 1$ is used as the generator polynomial with CRC-4. To find the different checksums for each *m,* these fixed polynomials are used as follows, where we denote them as $g_{p_1}(\beta)$ and $g_{p_2}(\beta)$, respectively:

Table II
OUR DERIVED CRC CHECKSUMS FOR $m = 11$, $m = 12$, $m = 13$, AND $m = 14$ IN THE α MODULE OF THE PRESENTED SCHEME

m	$f(\alpha)$	CRC	Predicted CRC Checksum	Actual CRC Checksum
11	$\alpha^{11} + \alpha^2 + 1$	3	$(a_{10} + a_8 + a_5 + a_4 + a_3 + a_1)\alpha^2 + (a_9 + a_7 + a_4 + a_3 + a_2 + a_0)\alpha + (a_{10} + a_9 + a_6 + a_5 + a_4 + a_2)$	$(\gamma_9 + \gamma_6 + \gamma_5 + \gamma_4 + \gamma_2)\alpha^2 + (\gamma_{10} + \gamma_8 + \gamma_5 + \gamma_4 + \gamma_3 + \gamma_1)\alpha + (\gamma_{10} + \gamma_7 + \gamma_6 + \gamma_5 + \gamma_3 + \gamma_0)$
		4	$(a_8 + a_6 + a_5 + a_2)\alpha^3 + (a_{10} + a_9 + a_7 + a_5 + a_4 + a_1)\alpha^2 + (a_9 + a_8 + a_6 + a_4 + a_3 + a_0)\alpha + (a_{10} + a_9 + a_7 + a_6 + a_3)$	$(\gamma_9 + \gamma_7 + \gamma_6 + \gamma_3)\alpha^3 + (\gamma_{10} + \gamma_8 + \gamma_6 + \gamma_5 + \gamma_2)\alpha^2 + (\gamma_{10} + \gamma_9 + \gamma_7 + \gamma_5 + \gamma_4 + \gamma_1)\alpha + (\gamma_{10} + \gamma_8 + \gamma_7 + \gamma_4 + \gamma_0)$
12	$\alpha^{12} + \alpha^6 + \alpha^4 + \alpha + 1$	3	$(a_{10} + a_8 + a_5 + a_4 + a_3 + a_1)\alpha^2 + (a_{10} + a_9 + a_7 + a_4 + a_3 + a_2 + a_0)\alpha + (a_9 + a_6 + a_5 + a_4 + a_2)$	$(\gamma_{11} + \gamma_9 + \gamma_6 + \gamma_5 + \gamma_4 + \gamma_2)\alpha^2 + (\gamma_{11} + \gamma_{10} + \gamma_8 + \gamma_5 + \gamma_4 + \gamma_3 + \gamma_1)\alpha + (\gamma_{10} + \gamma_7 + \gamma_6 + \gamma_5 + \gamma_3 + \gamma_0)$
		4	$(a_{11} + a_{10} + a_8 + a_6 + a_5 + a_2)\alpha^3 + (a_{11} + a_{10} + a_9 + a_7 + a_5 + a_4 + a_1)\alpha^2 + (a_{10} + a_9 + a_8 + a_6 + a_4 + a_3 + a_0)\alpha + (a_9 + a_7 + a_6 + a_3)$	$(\gamma_{11} + \gamma_9 + \gamma_7 + \gamma_6 + \gamma_3)\alpha^3 + (\gamma_{11} + \gamma_8 + \gamma_6 + \gamma_5 + \gamma_2)\alpha^2 + (\gamma_{11} + \gamma_{10} + \gamma_9 + \gamma_7 + \gamma_5 + \gamma_4 + \gamma_1)\alpha + (\gamma_{10} + \gamma_8 + \gamma_7 + \gamma_4 + \gamma_0)$
13	$\alpha^{13} + \alpha^4 + \alpha^3 + \alpha + 1$	3	$(a_{12} + a_{11} + a_{10} + a_8 + a_5 + a_4 + a_3 + a_1)\alpha^2 + (a_{12} + a_{11} + a_{10} + a_9 + a_7 + a_4 + a_3 + a_2 + a_0)\alpha + (a_{11} + a_9 + a_6 + a_5 + a_4 + a_2)$	$(\gamma_{12} + \gamma_{11} + \gamma_9 + \gamma_6 + \gamma_5 + \gamma_4 + \gamma_2)\alpha^2 + (\gamma_{12} + \gamma_{11} + \gamma_{10} + \gamma_8 + \gamma_5 + \gamma_4 + \gamma_3 + \gamma_1) \cdot \alpha + (\gamma_{12} + \gamma_{10} + \gamma_7 + \gamma_6 + \gamma_5 + \gamma_3 + \gamma_0)$
		4	$(a_{12} + a_{11} + a_{10} + a_8 + a_6 + a_5 + a_2)\alpha^3 + (a_{11} + a_{10} + a_9 + a_7 + a_5 + a_4 + a_1)\alpha^2 + (a_{11} + a_{10} + a_9 + a_8 + a_6 + a_4 + a_3 + a_0)\alpha + (a_{11} + a_9 + a_7 + a_6 + a_3)$	$(\gamma_{12} + \gamma_{11} + \gamma_9 + \gamma_7 + \gamma_6 + \gamma_3)\alpha^3 + (\gamma_{12} + \gamma_{11} + \gamma_{10} + \gamma_8 + \gamma_6 + \gamma_5 + \gamma_2)\alpha^2 + (\gamma_{12} + \gamma_{11} + \gamma_{10} + \gamma_9 + \gamma_7 + \gamma_5 + \gamma_4 + \gamma_1)\alpha + (\gamma_{12} + \gamma_{10} + \gamma_8 + \gamma_7 + \gamma_4 + \gamma_0)$
14	$\alpha^{14} + \alpha^8 + \alpha^6 + \alpha + 1$	3	$(a_{13} + a_{12} + a_{11} + a_{10} + a_8 + a_5 + a_4 + a_3 + a_1)\alpha^2 + (a_{11} + a_{10} + a_9 + a_7 + a_4 + a_3 + a_2 + a_0)\alpha + (a_{12} + a_{11} + a_9 + a_6 + a_5 + a_4 + a_2)$	$(\gamma_{13} + \gamma_{12} + \gamma_{11} + \gamma_9 + \gamma_6 + \gamma_5 + \gamma_4 + \gamma_2) \cdot \alpha^2 + (\gamma_{12} + \gamma_{11} + \gamma_{10} + \gamma_8 + \gamma_5 + \gamma_4 + \gamma_3 + \gamma_1)\alpha + (\gamma_{13} + \gamma_{12} + \gamma_{10} + \gamma_7 + \gamma_6 + \gamma_5 + \gamma_3 + \gamma_0)$
		4	$(a_{13} + a_{12} + a_{11} + a_{10} + a_8 + a_6 + a_5 + a_2)\alpha^3 + (a_{12} + a_{11} + a_{10} + a_9 + a_7 + a_5 + a_4 + a_1)\alpha^2 + (a_{13} + a_{11} + a_{10} + a_9 + a_8 + a_6 + a_4 + a_3 + a_0)\alpha + (a_{12} + a_{11} + a_9 + a_7 + a_6 + a_3)$	$(\gamma_{13} + \gamma_{12} + \gamma_{11} + \gamma_9 + \gamma_7 + \gamma_6 + \gamma_3)\alpha^3 + (\gamma_{13} + \gamma_{12} + \gamma_{11} + \gamma_{10} + \gamma_8 + \gamma_6 + \gamma_5 + \gamma_2)\alpha^2 + (\gamma_{12} + \gamma_{11} + \gamma_{10} + \gamma_9 + \gamma_7 + \gamma_5 + \gamma_4 + \gamma_1)\alpha + (\gamma_{13} + \gamma_{12} + \gamma_{10} + \gamma_8 + \gamma_7 + \gamma_4 + \gamma_0)$

According to $g_{p_1}(\beta)$:

$$\begin{gathered}\beta^3 \equiv \beta + 1 \bmod g_{p_1}(\beta)\\ \beta^4 \equiv \beta^2 + \beta \bmod g_{p_1}(\beta)\\ \vdots\\ \beta^{13} \equiv \beta^3 + \beta^2 + \beta \equiv \beta^2 + 1 \bmod g_{p_1}(\beta).\end{gathered}$$

According to $g_{p_2}(\beta)$:

$$\begin{gathered}\beta^4 \equiv \beta + 1 \bmod g_{p_2}(\beta)\\ \beta^5 \equiv \beta^2 + \beta \bmod g_{p_2}(\beta)\\ \vdots\\ \beta^{13} \equiv \beta^4 + \beta^3 + \beta^2 + \beta \equiv \beta^3 + \beta^2 + 1 \bmod g_{p_2}(\beta).\end{gathered}$$

Applying these generator polynomials into (1), we obtain the CRC-3 and CRC-4 checksums for the different *m*'s as shown in Table II. The case for $m = 11$ is explained below, and the derivations for other cases can be derived following the same approach.

If $m = 11$, from (1) we have

$$A(\alpha) \cdot \alpha = a_{10}\alpha^{11} + a_9\alpha^{10} + ... + a_1\alpha^2 + a_0\alpha.$$

Then, applying the irreducible polynomial $f(\alpha) = \alpha^{11} + \alpha^2 + 1$, one obtains

$$\begin{gathered}A(\alpha) \cdot \alpha \equiv a_{10}\alpha^2 + a_{10} + a_9\alpha^{10} + a_8\alpha^9\\ + a_7\alpha^8 + a_6\alpha^7 + a_5\alpha^6 + a_4\alpha^5 + a_3\alpha^4\\ + a_2\alpha^3 + a_1\alpha^2 + a_0\alpha \bmod f(\alpha).\end{gathered} \tag{2}$$

To calculate the predicted CRC-3 checksum for $m = 11$ in the α module ($PCRC3_{11}$), the generator polynomial is applied to obtain

$$\begin{gathered}PCRC3_{11} = (a_{10} + a_8 + a_5 + a_4 + a_3\\ + a_1)\alpha^2 + (a_9 + a_7 + a_4 + a_3 + a_2 + a_0)\\ \cdot\alpha + (a_{10} + a_9 + a_6 + a_5 + a_4 + a_2)\end{gathered} \tag{3}$$

Lastly, to calculate the actual CRC-3 checksum for $m = 11$ in the α module ($ACRC3_{11}$), we rename the coefficients of (2): a_9 as γ_{10}, ... , a_0 as γ_1, and a_{10} as γ_0, to obtain

$$\begin{gathered}A(\alpha) \cdot \alpha = \gamma_{10}\alpha^{10} + \gamma_9\alpha^9 + \gamma_8\alpha^8 + \gamma_7\alpha^7 + \gamma_6\alpha^6\\ + \gamma_5\alpha^5 + \gamma_4\alpha^4 + \gamma_3\alpha^3 + \gamma_2\alpha^2 + \gamma_1\alpha^1 + \gamma_0,\end{gathered} \tag{4}$$

and the generator polynomial is applied to obtain

$$\begin{gathered}ACRC3_{11} = (\gamma_9 + \gamma_6 + \gamma_5 + \gamma_4 + \gamma_2)\alpha^2\\ + (\gamma_{10} + \gamma_8 + \gamma_5 + \gamma_4 + \gamma_3 + \gamma_1)\alpha\\ + (\gamma_{10} + \gamma_7 + \gamma_6 + \gamma_5 + \gamma_3 + \gamma_0).\end{gathered} \tag{5}$$

For the *sum* module, which adds elements $A(x)$ and $B(x)$ over $GF(2^m)$, both CRC checksums are similar as the actual CRC checksums from Table II, but since $B(x)$ is added, its coefficients would be added as well. Lastly, for the *pass-thru* module, which adds element $A(x)$ over $GF(2^m)$ with a $GF(2)$ element *b*, both CRC checksums are also similar as those from Table II, but since *b* is multiplied, the CRC checksums from Table II are multiplied as well with *b*.

2) $GF(2^m)$ Inversion : The multiplicative inverse of an element $A \neq 0$ in the field $GF(2^m)$ is defined as the process of finding the unique element $A^{-1}\,\epsilon\, GF(2^m)$ such that $A \cdot A^{-1} = 1$. The inversion can be derived with squarings and multiplications by employing the Itoh-Tsujii multiplicative inverse algorithm (ITA) [36], which reduces the complexity of the polynomial variant of Fermat's Little Theorem (FLT) to obtain a better performance. To perform $GF(2^m)$ inversion, we based our schemes on FLT and ITA. FLT specifies that the inverse of an element A can be derived as $A^{2^m-2} \equiv A^{-1} \bmod f(\alpha)$, requiring $m-2$ multiplications and $m-1$ squarings.

Following FLT basics, Itoh and Tsujii introduced the method ITA to reduce the complexity of such theorem. ITA yields dramatic reductions in the number of multiplications

Table III
STEPS NEEDED TO PERFORM THE INVERSE OF $A \epsilon GF(2^{11})$ USING ADDITION CHAIN

Step	$\beta_{V_i}(\alpha)$	$\beta_{V_j+U_k}(\alpha)$	Exponentiation
1	$\beta_1(\alpha)$		A
2	$\beta_2(\alpha)$	$\beta_{1+1}(\alpha)$	$(\beta_1)^{2^1}\beta_1 = A^{2^2-1}$
3	$\beta_4(\alpha)$	$\beta_{2+2}(\alpha)$	$(\beta_2)^{2^2}\beta_2 = A^{2^4-1}$
4	$\beta_5(\alpha)$	$\beta_{4+1}(\alpha)$	$(\beta_4)^{2^1}\beta_1 = A^{2^5-1}$
5	$\beta_{10}(\alpha)$	$\beta_{5+5}(\alpha)$	$(\beta_5)^{2^5}\beta_5 = A^{2^{10}-1}$

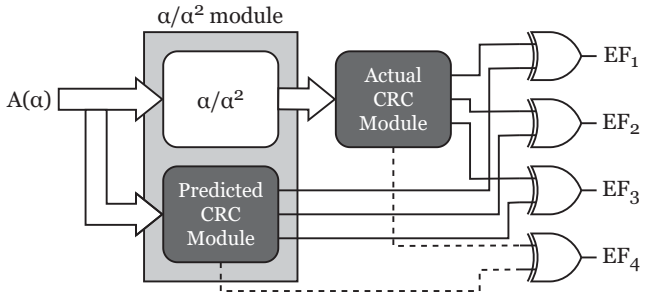

Figure 2. The proposed error detection of the α and α^2 modules using CRC.

needed in the exponentiation by an efficient use of addition chains [37]-[39]. The inverse can be rewritten as $A^{-1} = [\beta_{m-1}(A)]^2$, where $\beta_k(A) = A^{2^k-1} \epsilon GF(2^m)$ and $k \epsilon \mathbb{N}$. In [39], a recursive sequence is used with an addition chain for $m-1$ to compute $\beta_{m-1}(A)$. To calculate an addition chain $C = \{c_1, c_2, \ldots, c_t\}$ with a field polynomial $f(\alpha)$ of m degree, we have $c_1 = 1$ and $c_t = m-1$. If c_i is even, $c_{i-1} = c_i/2$ and if c_i is odd, $c_{i-1} = c_i - 1$.

The addition chain C obtained for $GF(2^{11})$, $GF(2^{12})$, $GF(2^{13})$, and $GF(2^{14})$ are $C_{11} = \{1,\ 2,\ 4,\ 5,\ 10\}$, $C_{12} = \{1,\ 2,\ 4,\ 5,\ 10,\ 11\}$, $C_{13} = \{1,\ 2,\ 3,\ 6,\ 12\}$, and $C_{14} = \{1,\ 2,\ 3,\ 6,\ 12,\ 13\}$, respectively. The computational steps to calculate the inverse of $A \epsilon GF(2^{11})$ using such addition chains is illustrated in Table III, where V_i are the integers in the addition chain, $V_j = V_{i-1}$, and $U_k = V_i - V_j$.

Fault detection schemes for $GF(2^m)$ squaring are presented below. $GF(2^m)$ squaring uses the *sum* module and an α^2 module instead of the α module presented previously. In the α^2 module, an element *A* is multiplied by α^2 to achieve:

$$A(\alpha) \cdot \alpha^2 = a_{m-1}\alpha^{m+1} + a_{m-2}\alpha^m + ... + a_0\alpha^2, \quad (6)$$

where $\alpha^{m+1} = f_{m-1}\alpha^m + f_{m-2}\alpha^{m-1} + ... + f_0\alpha \bmod f(\alpha)$ and $\alpha^m = f_{m-1}\alpha^{m-1} + f_{m-2}\alpha^{m-2} + ... + f_0 \bmod f(\alpha)$.

a) Cyclic Redundancy Check: The element A is multiplied by α^2; therefore, the actual and predicted CRC checksums for the different values of *m* differ. In Table IV, the predicted CRC checksums for the α^2 module are presented (the actual CRC checksums of the α^2 module are the same as the ones from the α module, presented in Table II). To clarify this process, $m = 14$ is used as an example to show how the predicted CRC-4 and actual CRC-4 are calculated. We have

$$A(\alpha) \cdot \alpha^2 = a_{13}\alpha^{15} + a_{12}\alpha^{14} + ... + a_1\alpha^3 + a_0\alpha^2$$

Then, applying the irreducible polynomial $f(\alpha) = \alpha^{14} + \alpha^8 + \alpha^6 + \alpha + 1$, one obtains

$$\begin{aligned} A(\alpha) \cdot \alpha^2 = {} & a_{13}\alpha^9 + a_{13}\alpha^7 + a_{13}\alpha^2 + a_{13}\alpha \\ & + a_{12}\alpha^8 a_{12}\alpha^6 + a_{12}\alpha + a_{12} + a_{11}\alpha^{13} + a_{10}\alpha^{12} \\ & + a_9\alpha^{11} + a_8\alpha^{10} + a_7\alpha^9 + a_6\alpha^8 + a_5\alpha^7 + a_4\alpha^6 \\ & + a_3\alpha^5 + a_2\alpha^4 + a_1\alpha^3 + a_0\alpha^2 \bmod f(\alpha) \end{aligned} \quad (7)$$

To calculate the predicted CRC-4 checksum for $m = 14$ in the α^2 module ($PCRC4_{14_2}$), the generator polynomial is applied to obtain

$$\begin{aligned} PCRC4_{14} = {} & (a_{12} + a_{11} + a_{10} + a_9 + a_7 \\ & + a_5 + a_4 + a_1)\alpha^3 + (a_{13} + a_{11} + a_{10} + a_9 + a_8 \\ & + a_6 + a_4 + a_3 + a_0)\alpha^2 + (a_{13} + a_{12} + a_{10} + a_9 \\ & + a_8 + a_7 + a_5 + a_3 + a_2)\alpha + (a_{13} + a_{11} + a_{10} \\ & + a_8 + a_6 + a_5 + a_2). \end{aligned} \quad (8)$$

Lastly, to calculate the actual CRC-4 checksum for $m = 14$ in the α^2 module ($ACRC4_{14_2}$), we rename the coefficients of (7): a_{12} as γ_{13}, ..., $a_{13} + a_0$ as γ_1, and a_{13} as γ_0,and again, the generator polynomial is applied to obtain

$$\begin{aligned} ACRC4_{14_2} = {} & (\gamma_{12} + \gamma_{11} + \gamma_{10} + \gamma_9 + \gamma_7 + \gamma_5 \\ & + \gamma_4 + \gamma_1)\alpha^3 + (\gamma_{13} + \gamma_{11} + \gamma_{10} + \gamma_9 + \gamma_8 + \gamma_6 \\ & + \gamma_4 + \gamma_3 + \gamma_0)\alpha^2 + (\gamma_{13} + \gamma_{12} + \gamma_{10} + \gamma_9 + \gamma_8 \\ & + \gamma_7 + \gamma_5 + \gamma_3 + \gamma_2)\alpha + (\gamma_{13} + \gamma_{11} + \gamma_{10} + \gamma_8 \\ & + \gamma_6 + \gamma_5 + \gamma_2). \end{aligned} \quad (9)$$

In Fig. 2, the proposed architecture with CRC-3 and CRC-4 is presented. As shown in Fig. 2, three or four error indication flags denoted as EF are obtained to indicate if an error has been found using CRC-3 or CRC-4, respectively. This figure represents the α module as well as the α^2 module. The XOR gates are used to compare the outputs of the CRC modules.

IV. ERROR COVERAGE AND FPGA IMPLEMENTATIONS

To calculate the error coverage provided by the different proposed schemes, the total number of operations need to be taken into account. For instance, to calculate the error coverage computing *H*, a total of n finite field inversions and $(n \times t) - n$ finite field multipliers are utilized. Depending on the value of *m*, a different number of multiplications and squarings will be needed to perform each inversion. Each finite field multiplication requires a total of $m-1$ α modules, $m-1$ sum modules, and m $pass-thru$ modules and each finite field squaring uses $m-1$ α^2 modules and $m-1$ sum modules.

Each of the modules requires three or four checksums depending on the choice of CRC-3 or CRC-4, respectively. For $m = 11$, the error coverage percentage computing *H* is calculated as follows: A total of $2,048$ finite field inversions and $79,872$ finite field multiplications are needed to compute *H*; each inverse calculation can be derived by 10 squaring and 4 multiplication operations using ITA, which is a total of $20,480$ squarings and $8,192$ multiplications; therefore, a total of $20,480 \cdot (10+10) + 8,192 \cdot (10+10+11) + 79,872 \cdot (10+10+11)$ or close to 1.2×10^7 or 1.6×10^7 checksums are needed for CRC-3 or CRC-4, respectively. For $m = 12$, close to 3.5×10^7 or 4.7×10^7 checksums are needed for CRC-3 or CRC-4, respectively. For $m = 13$, close to 1.3×10^8 or 1.8×10^8 checksums are needed for CRC-3 or CRC-4, respectively. Lastly, for $m = 14$, close to 6.7×10^7, or 9.0×10^7 checksums are needed for CRC-3 or CRC-4, respectively. The lowest percentage (representing the worst case scenario) of error coverage are obtained applying CRC-3 with $m = 11$, where the error coverage percentage is $100 \cdot (1 - (\frac{1}{2})^{1.2 \times 10^7})\%$. Moreover, we add our proposed error detection schemes to the finite field multipliers of the Niederreiter's Key Generator. The design from [33] is used as basis. Other most recent hardware

Table IV
OUR DERIVED CRC CHECKSUMS FOR $m = 11$, $m = 12$, $m = 13$, AND $m = 14$ IN α^2 MODULE OF THE PRESENTED SCHEMES

m	$f(\alpha)$	CRC	Predicted CRC Checksum
11	$\alpha^{11} + \alpha^2 + 1$	3	$(a_9 + a_7 + a_4 + a_3 + a_2 + a_0)\alpha^2 + (a_8 + a_6 + a_3 + a_2 + a_1)\alpha + (a_{10} + a_9 + a_8 + a_5 + a_4 + a_3 + a_1)$
		4	$(a_{10} + a_7 + a_5 + a_4 + a_1)\alpha^3 + (a_9 + a_8 + a_6 + a_4 + a_3 + a_0)\alpha^2 + (a_{10} + a_8 + a_7 + a_5 + a_3 + a_2)\alpha + (a_9 + a_8 + a_6 + a_5 + a_2)$
12	$\alpha^{12} + \alpha^6 + \alpha^4 + \alpha + 1$	3	$(a_9 + a_7 + a_4 + a_3 + a_2 + a_0)\alpha^2 + (a_9 + a_8 + a_6 + a_3 + a_2 + a_1)\alpha + (a_8 + a_5 + a_4 + a_3 + a_1)$
		4	$(a_{11} + a_{10} + a_9 + a_7 + a_5 + a_4 + a_1)\alpha^3 + (a_{10} + a_9 + a_8 + a_6 + a_4 + a_3 + a_0)\alpha^2 + (a_{11} + a_9 + a_8 + a_7 + a_5 + a_3 + a_2)\alpha + (a_{11} + a_8 + a_6 + a_5 + a_2)$
13	$\alpha^{13} + \alpha^4 + \alpha^3 + \alpha + 1$	3	$(a_{12} + a_{11} + a_{10} + a_9 + a_7 + a_4 + a_3 + a_2 + a_0)\alpha^2 + (a_{12} + a_{11} + a_{10} + a_9 + a_8 + a_6 + a_3 + a_2 + a_1)\alpha + (a_{12} + a_{10} + a_8 + a_5 + a_4 + a_3 + a_1)$
		4	$(a_{11} + a_{10} + a_9 + a_7 + a_5 + a_4 + a_1)\alpha^3 + (a_{10} + a_9 + a_8 + a_6 + a_4 + a_3 + a_0)\alpha^2 + (a_{12} + a_{10} + +a_9 + a_8 + a_7 + a_5 + a_3 + a_2)\alpha + (a_{12} + a_{10} + a_8 + a_6 + a_5 + a_2)$
14	$\alpha^{14} + \alpha^8 + \alpha^6 + \alpha + 1$	3	$(a_{12} + a_{11} + a_{10} + a_9 + a_7 + a_4 + a_3 + a_2 + a_0)\alpha^2 + (a_{13} + a_{10} + a_9 + a_8 + a_6 + a_3 + a_2 + a_1)\alpha + (a_{13} + a_{11} + a_{10} + a_8 + a_5 + a_4 + a_3 + a_1)$
		4	$(a_{12} + a_{11} + a_{10} + a_9 + a_7 + a_5 + a_4 + a_1)\alpha^3 + (a_{13} + a_{11} + a_{10} + a_9 + a_8 + a_6 + a_4 + a_3 + a_0)\alpha^2 + (a_{13} + a_{12} + a_{10} + a_9 + a_8 + a_7 + a_5 + a_3 + a_2)\alpha + (a_{13} + a_{11} + a_{10} + a_8 + a_6 + a_5 + a_2)$

Table V
OVERHEADS OF THE PROPOSED ERROR DETECTION SCHEMES FOR THE ENTIRE KEY GENERATOR USING THE PARAMETERS $m = 13$, $t = 119$, AND $n = 6,960$ ON XILINX FPGA FAMILY KINTEX ULTRASCALE+ (DEVICE XCKU5P-FFVD900-1-I)

Architecture	Area (CLBs)	Delay (ns)	Power (mW) @50 MHz	Throughput (Gbps)	Efficiency ($Gbps/CLBs$)
Key Generator	45,358	9.176	0.916	7.520	1.658×10^{-4}
Key Generator with CRC-3	46,874 (3.34%)	9.202 (0.28%)	0.919 (0.33%)	7.498 (-0.29%)	1.600×10^{-4} (-3.50%)
Key Generator with CRC-4	46,634 (2.81%)	9.668 (5.36%)	0.921 (0.55%)	7.137 (-5.09%)	1.530×10^{-4} (-7.72%)

implementations of Classic McEliece are described in [40]-[41].

Finite field multiplications and inversions are the most complex and vulnerable to natural and malicious fault attacks. However, since [33] uses a pre-computed lookup table for the implementation of the inversion module, the implementation does not use error detection for $GF(2^m)$ inversions. Furthermore, the original Key Generator from [33] uses $GF(2^m)$ multipliers to perform finite field squarings, which means that $GF(2^m)$ squarings are also protected in our implementations as well as additions since they are followed by multiplication or squaring in the same instruction, which use our error detection schemes. The presented implementations are performed using Xilinx Vivado with the parameters $m = 13$, $t = 119$, and $n = 6,960$ on Xilinx FPGA Kintex Ultrascale+ device xcku5p-ffvd900-1-i. The implementation results for the original architectures and our presented error detection schemes are shown in Table V in terms of area (occupied slices), delay, power (at the frequency of 50 MHz), throughput, and efficiency. As seen in Table V, acceptable overheads are obtained with efficiency degradations of at most 7.72%.

There has not been any prior work done on error detection based on CRC for the Niederreiter cryptosystem to the best of our knowledge. For qualitative comparison to verify that the overheads incurred are acceptable, let us go over some case studies on error detection in $GF(2^m)$ arithmetic hardware. In [42], error detection based on parity prediction for normal basis multiplication is performed, obtaining a combined worst-case area and delay overhead of 58.1%. Additionally, normal parity provides an error detection of up to 50%, i.e., if the number of faults is even, the approach would not be able to detect the faults. This highly predictable countermeasure can be circumvented by intelligent fault injection. In our work, we propose CRC-3 and CRC-4 to overcome this problem and is intended for deeply-embedded systems where high performance, low overhead, and low energy are preferred. Reliable concurrent error detection architectures for Extended Euclidean-based division over $GF(2^m)$ are provided in [43]. The schemes utilized are based on parity prediction and they have a combined worst-case area and delay overhead of 25.18%. This and similar prior works on classical cryptography are instances to show that the proposed error detection architectures obtain similar overheads compared to other works on fault detection, achieving an acceptable overhead. These degradations are generally acceptable for providing error detection to the original architectures which lack such capability to thwart natural or malicious faults.

V. CONCLUSION

Providing constructions and schemes to thwart error injection and occurrence for Internet of nano-Things in constrained applications is of prominent importance. In this paper, we propose different fault detection schemes based on CRC-3 and CRC-4. These schemes are used in the different blocks of the key generator in the Niederreiter cryptosystem, e.g., $GF(2^m)$ multiplication, squaring, inversion, and addition. Our proposed error detection architectures provide security flexibility providing sets of closed formulations for different security parameters (m ranging from 11 to 14). Moreover, we implemented our proposed schemes on FPGA to benchmark the overhead induced in the Key Generator. Results show an acceptable overhead with efficiency degradations of at most 7.72% for the Key Generator of the Niederreiter cryptosystem providing very high error coverage.

ACKNOWLEDGMENTS

This work was supported by Marymount University through the START grant (2450100) and by the US National Science Foundation (NSF) through the award SaTC-1801488.

REFERENCES

[1] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proc. Annual Symp. Foundations of Computer Science*, Nov. 1994, pp. 124-134.

[2] D. Moody, "Post-quantum cryptography: NIST's plan for the future," Feb. 2016. [Online]. Available: https://pqcrypto2016.jp/data/pqc2016_nist_announcement.pdf.

[3] R. J. McEliece, "A public-key cryptosystem based on algebraic," *Coding Thv,* vol. 4244, pp. 114-116, 1978.

[4] H. Niederreiter, "Knapsack-type cryptosystems and algebraic coding theory," *Problems of Control and Information Theory*, vol. 15, no. 2, pp. 159-166, 1986.

[5] V. M. Sidelnikov and S. O. Shestakov, "On insecurity of cryptosystems based on generalized Reed-Solomon codes," *Discrete Mathematics and Applications,* vol. 2, no. 4, pp. 439-444, 1992.

[6] P. L. Cayrel and P. Dusart, "McEliece/Niederreiter PKC: Sensitivity to fault injection," in *Proc. Int. Conf. Future Information Technology*, May 2010, pp. 1-6.

[7] R. Avanzi, S. Hoerder, D. Page, and M. Tunstall, "Side-channel attacks on the McEliece and Niederreiter public-key cryptosystems," *Journal of Cryptographic Engineering,* vol. 1, no. 4, pp. 271-281, 2011.

[8] A. Shoufan, F. Strenzke, H. G. Molter, and M. Stöttinger, "A timing attack against Patterson algorithm in the McEliece PKC," in *Int. Conf. on Information Security and Cryptology*, 2009, pp. 161-175.

[9] Q. Guo, A. Johansson, and T. Johansson, "A key-recovery side-channel attack on classic McEliece cryptosystems," *IACR Trans. on Cryptographic Hardware and Embedded Systems*, pp. 800-827, 2022.

[10] E. Kirshanova and A. May, "Decoding McEliece with a hint-secret Goppa key parts reveal everything," *Security and Cryptography for Networks (SCN)*, pp. 3-20, 2022.

[11] A. May and C. R. T. Schneider, "How to backdoor (classical) McEliece and how to guard against backdoors," *Cryptology ePrint Archive*, 2022.

[12] D. J. Bernstein, T. Lange, and C. Peters, "Attacking and defending the McEliece cryptosystem," in *Proc. Int. Workshop Post-Quantum Cryptography*, 2008, pp. 31-46.

[13] B. Biswas and N. Sendrier, "McEliece cryptosystem implementation: theory and practice," in *Proc. Int. Workshop Post-Quantum Cryptography*, 2008, pp. 47-62.

[14] S. Heyse, and T. Güneysu, "Towards one cycle per bit asymmetric encryption: code-based cryptography on reconfigurable hardware," in *Proc. Workshop Cryptographic Hardware and Embedded Systems* (*CHES*), 2012, pp. 340-355.

[15] M. Mozaffari Kermani and A. Reyhani-Masoleh, "Reliable hardware architectures for the third-round SHA-3 finalist Grøstl benchmarked on FPGA platform," in *Proc. IEEE Int. Symp. Defect and Fault Tolerance in VLSI Systems (DFT)*, pp. 325-331, 2011.

[16] M. Mozaffari Kermani and A. Reyhani-Masoleh, "A high-performance fault diagnosis approach for the AES SubBytes utilizing mixed bases," in *Proc. IEEE Workshop Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pp. 80-87, 2011.

[17] M. Mozaffari Kermani and A. Reyhani-Masoleh, "A low-cost S-box for the Advanced Encryption Standard using normal basis," in *Proc. IEEE Int. Conf. Electro/Information Technology (EIT)*, 2009, pp. 52-55.

[18] S. Subramanian, M. Mozaffari Kermani, R. Azarderakhsh, and M. Nojoumian, "Reliable hardware architectures for cryptographic block ciphers LED and HIGHT," *IEEE Trans. on Computer-Aided Design Integr. Circuits Syst.*, vol. 36, no. 10, pp. 1750-1758, 2017.

[19] A. Cintas-Canto, M. Mozaffari Kermani, and R. Azarderakhsh, "Reliable constructions for the key generator of code-based post-quantum cryptosystems on FPGA," *ACM Emerging Technologies in Computing Systems*, accepted, 2022.

[20] A. A. Kamal and A. M. Youssef, "Strengthening hardware implementations of NTRUEncrypt against fault analysis attacks," *Journal of Cryptographic Engineering*, vol. 3, no. 4, pp. 227-240, 2013.

[21] M. Mozaffari Kermani and A. Reyhani-Masoleh, "Fault detection structures of the S-boxes and the inverse S-boxes for the Advanced Encryption Standard," *J. Electronic Testing: Theory and Applications (JETTA)*, vol. 25, no. 4, pp. 225-245, 2009.

[22] M. Mozaffari Kermani, A. Jalali, R. Azarderakhsh, J. Xie, and R. Choo, "Reliable inversion in GF(2^8) with redundant arithmetic for secure error detection of cryptographic architectures," *IEEE Trans. on Computer-Aided Design Integr. Circuits Syst.*, vol. 37, no. 3, pp. 696-704, Mar. 2018.

[23] M. Yasin, B. Mazumdar, S. Subidh Ali, and O. Sinanoglu, "Security analysis of logic encryption against the most effective side-channel attack: DPA," in *Proc. Defect and Fault Tolerance in VLSI Systems*, 2015, pp. 97-102.

[24] M. Mozaffari Kermani, R. Azarderakhsh, A. Sarker, and A. Jalali, "Efficient and reliable error detection architectures of Hash-Counter-Hash tweakable enciphering schemes," *ACM Trans. Embedded Computing Systems*, vol. 17, no. 2, pp. 54:1-54:19, May 2018.

[25] S. Saha, D. Jap, D. Basu Roy, A. Chakraborty, S. Bhasin and D. Mukhopadhyay, "A framework to counter statistical ineffective fault analysis of block ciphers using domain transformation and error correction," *IEEE Trans. on Information Forensics and Security* vol. 15, pp. 1905-1919, 2020.

[26] M. Mozaffari-Kermani and R. Azarderakhsh, "Reliable hash trees for post-quantum stateless cryptographic hash-based signatures," in *Proc. IEEE Int. Symp. Defect and Fault Tolerance in VLSI Systems (DFT)*, 2015, pp. 103-108.

[27] A. Cintas-Canto, M. Mozaffari Kermani, and R. Azarderakhsh, "Reliable architectures for composite-field-oriented constructions of McEliece post-quantum cryptography on FPGA," *IEEE Trans. on Computer-Aided Design Integr. Circuits Syst.*, vol. 40, no. 5, pp. 999-1003, 2021.

[28] A. Cintas-Canto, M. Mozaffari Kermani, and R. Azarderakhsh, "CRC-based error detection constructions for FLT and ITA finite field inversions over $GF(2^m)$," *IEEE Trans. on Very Large Scale Integrated (VLSI) Systems,* vol. 29, no. 5, pp. 1033-1037, 2021.

[29] A. Cintas-Canto, M. Mozaffari Kermani, and R. Azarderakhsh, "Reliable CRC-based error detection constructions for finite field multipliers with applications in cryptography," *IEEE Trans. on Very Large Scale Integrated (VLSI) Systems,* vol. 29, no. 1, pp. 232-236, 2021.

[30] L. Breveglieri, I. Koren, and P. Maistri, "An operation-centered approach to fault detection in symmetric cryptography ciphers," *IEEE Trans. Computers,* vol. 56, no. 5, pp. 534-540, 2007.

[31] D. J. Bernstein, T. Chou, P. Schwabe, "McBits: fast constant-time code-based cryptography," in *Cryptographic Hardware and Embedded Systems (CHES)*, 2013, pp. 250–272, 2013.

[32] D. J. Bernstein, T. Chuo, T. Lange, I. Von Maurich, R. Misoczki, R. Niederhagen, E. Persichetti, C. Peters, P. Schwabe, N. Sendrier, J. Szefer, and W. Wang, "Classic McEliece: Conservative code-based cryptography," October 2020 [Online]. Available: https://classic.mceliece.org/nist/mceliece-20201010.pdf.

[33] W. Wang, J. Szefer, and R. Niederhagen, "FPGA-based key generator for the Niederreiter cryptosystem using binary Goppa codes," in *Int. Conf. on Cryptographic Hardware and Embedded Systems*, 2017, pp. 253-274.

[34] S. Gao and T. Mateer, "Additive fast Fourier transforms over finite fields," *IEEE Trans. on Information Theory,* vol. 56, no. 12, pp. 6265-6272, 2010.

[35] A. Reyhani-Masoleh and M. A. Hasan, "Error detection in polynomial basis multipliers over binary extension fields," in *Proc. Workshop Cryptographic Hardware and Embedded Systems* (*CHES*), 2002, pp. 515-528.

[36] T. Itoh and S. Tsujii, "A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases," *Information and Computation*, vol. 78, no. 3, pp. 171-177, 1988.

[37] J. Guajardo and C. Paar, "Itoh-Tsujii inversion in standard basis and its application in cryptography and codes," *Designs, Codes and Cryptography,* vol. 25, pp. 207–216, 2002.

[38] F. Rodriguez-Henriquez, N. A. Saqib, and N. Cruz-Cortes, "A fast implementation of multiplicative inversion over $GF(2^m)$," in *Proc. Int. Symposium on Information Technology*, 2005, pp. 574–579.

[39] C. Rebeiro, S. S. Roy, D. S. Reddy, and D. Mukhopadhyay, "Revisiting the Itoh-Tsujii inversion algorithm for FPGA platforms," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems,* vol. 19, no. 8, pp. 1508-1512, 2010.

[40] D. Fallnich, S. Zhang, and T. Gemmeke, "Efficient ASIC Architectures for Low Latency Niederreiter Decryption," *Cryptology ePrint Archive*, 2022.

[41] P. J. Chen, T. Chou, S. Deshpande, N. Lahr, R. Niederhagen, J. Szefer, and W. Wang, "Complete and improved FPGA implementation of classic McEliece," *IACR Trans. on Cryptographic Hardware and Embedded Systems*, pp. 71-113, 2022.

[42] C.Y. Lee, P. K. Meher, and J .C. Patra, "Concurrent error detection in bit-serial normal basis multiplication over $GF(2^m)$ using multiple parity prediction schemes," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems,* vol. 18, no. 8, pp. 1234-1238, 2009.

[43] M. Mozaffari-Kermani, R. Azarderakhsh, C.Y. Lee, and S. Bayat-Sarmadi, "Reliable concurrent error detection architectures for Extended Euclidean-based division over $GF(2^m)$," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 5, pp. 995-1003, 2013.