High-speed FPGA Implementation of the NIST Round 1 Rainbow Signature Scheme

Ahmed Ferozpuri
George Mason University
Fairfax, Virginia 22030
aferozpu@gmu.edu

Kris Gaj
George Mason University
Fairfax, Virginia 22030
kgaj@gmu.edu

***Abstract*—Round 1 of the the NIST post-quantum cryptography (PQC) standardization effort began on November 30th, 2017. The competition aims to select the most promising quantum-resistant algorithms, which are currently secure against large scale quantum computers. Multivariate cryptosystems belong to a promising group of PQC schemes and are based on multivariate polynomials over finite fields. Among them are the Unbalanced Oil and Vinegar (UOV) and Rainbow signature schemes, which have been extensively studied since 1999 and 2005, respectively. The main advantage of UOV is high confidence in its security, and the disadvantages include large key and signature sizes. Rainbow is a multi-layer version of UOV that offers better performance, smaller keys, and smaller signatures. This paper presents a high-speed FPGA implementation for the NIST Round 1 PQC submission of Rainbow. We discuss a high-speed design that uses a parameterized system solver, which can solve an n-by-n system in n clock cycles. Compared to the previous state-of-the-art, we reduce the number of required multipliers by almost half, speed up execution, and implement Rainbow for higher security levels. Our design supports many parameter sets, which require operations in the fields GF(16) and GF(256). Additionally, in order to make benchmarking easier and fairer, our design follows a universal PQC hardware API, which allows for fair comparison with other post-quantum signature schemes. This design is being made open-source to increase transparency and speed up further optimization.**

***Index Terms*—Post-Quantum, Multivariate, UOV, Rainbow, FPGA, high-speed, Gaussian, System Solver**

I. Introduction

In the mid-1990s Peter Shor published his ground-breaking paper detailing a quantum algorithm capable of solving factoring and discrete logarithm problems in polynomial time. The idea of a quantum computer may have seemed more like science fiction then, but recent developments indicate that we may be close to a reliable and scalable quantum computer. In 2015, the United States National Security Agency (NSA) announced that those who have not yet switched over to Suite B recommendations, should hold off and instead wait for future post-quantum algorithms [1]. In December 2016 the National Institute of Standards and Technology (NIST) made an announcement regarding their Post-Quantum Cryptography Project, where they were accepting proposals for quantum-resistant public-key cryptographic algorithms [2] [3].

This paper is partially based upon work supported by the U.S. Department of Commerce / National Institute of Standards and Technology under Grant no. 60NANB15D058.

The cryptography community has made significant efforts recently to establish quantum-resistant algorithms to be ready for the threat posed by a scalable quantum computer. Round 1 of the NIST PQC competition had 69 complete submissions, and there are many likely candidates based on hash functions, codes, lattices, elliptic curve isogenies, and multivariate quadratic systems. The competition will include various cryptographic primitives such as key encapsulation mechanism (KEM), public key encryption, and digital signature schemes. Those that have withstood attacks for the longest are the most promising. There is a NIST submission available for Rainbow with new parameter sets, for which no hardware implementation results are available. Our work is the first that will provide feedback for the competition regarding the hardware performance of Rainbow at the category I parameter sets. Additionally, we provide the first fully constant-time architecture that does not require swapping and consequently does not leak side-channel information during Gaussian Elimination (see Section 3.3 in [4]).

II. Background

Multivariate cryptography uses a system of non-linear multivariate polynomials to perform signature generation, encryption, and key exchange. In multivariate cryptography, a set of polynomials, $\mathcal{P}$, represents the public key and is made up of m polynomials in n variables as shown in (1). Note, the variable n will be used to define the number of variables in the multivariate or linear system of equations described in this paper. Additionally, the degree of the system, d, is 2, in order to constrain the key size and the amount of computations.

$$\begin{aligned} p^{(1)}(x_1,...,x_n) &= \sum_{i=1}^{n}\sum_{j=i}^{n} p_{ij}^{(1)} x_i x_j + \sum_{i=1}^{n} p_i^{(1)} x_i + p_0^{(1)} \\ p^{(2)}(x_1,...,x_n) &= \sum_{i=1}^{n}\sum_{j=i}^{n} p_{ij}^{(2)} x_i x_j + \sum_{i=1}^{n} p_i^{(2)} x_i + p_0^{(2)} \\ &\dots \\ p^{(m)}(x_1,...,x_n) &= \sum_{i=1}^{n}\sum_{j=i}^{n} p_{ij}^{(m)} x_i x_j + \sum_{i=1}^{n} p_i^{(m)} x_i + p_0^{(m)} \end{aligned} \tag{1}$$

For all multivariate public key cryptosystems, the size of the public key is based on the number of terms in the polynomial,

978-1-7281-1968-7/18/$31.00 ©2018 IEEE

which depends directly on m, n, and d. Using (1), there are $\binom{n}{2} + n = \binom{n+1}{2}$ $p_{ij}^{(k)}$ terms, n $p_i^{(k)}$ terms and 1 constant for any given kth polynomial in $\mathcal{P}$. Therefore, the size of the public key, s, for m polynomials, is shown in (2).

$$s = m\left(\binom{n+1}{2} + n + 1\right) = m\binom{n+2}{2} \tag{2}$$

As shown in (1) the basic operations are addition and multiplication of elements $p_{ij}^{(k)}, p_i^{(k)}, x_i, x_j$, which are elements of finite field $\mathbb{F}_q$. In Rainbow, q = 16, 31, or 256. Our implementation supports q = 16 and 256. For these values, addition is equivalent to the XOR operation. Multiplication between elements from $GF(2^4)$ and $GF(2^8)$ is performed using the irreducible polynomials x^4+x+1 and $x^8+x^6+x^3+x^2+1$, respectively. Additionally, the system of multivariate equations must be solved during signature generation, and system solvers are covered in section IV.

In order to generate the public and private key pairs, invertible affine transformations must be selected, however, this process is assumed to occur in software as specified in [5], and the hardware receives the inverted affine transformations as part of the private key input. Additionally, to meet the EUF-CMA requirements, the NIST Rainbow submission uses $msg\_in = \mathcal{H}(\mathcal{H}(doc)\|r)$ as an input to Rainbow, where doc is a document to be signed, r is a random salt, and $\mathcal{H}$ is a hash function [4].

A. Rainbow Overview

The original Rainbow scheme [6] is similar to UOV [7], but introduces a layering technique to reduce the public and private key sizes.

Let $\mathbb{F}_q$ be a finite field and $\mathcal{S}$ be the set $\{1,\ 2,\ \ldots, n\}$. Let $v_1,\ v_2 \ldots, v_n$ be integers such that $0 < v_1 < v_2 \ \ldots \ < v_u < v_{u+1} = n$. Define sets $\mathcal{S}_i = \{1,\ 2,\ \ldots, v_i\}$ for $i = \{1,\ 2,\ \ldots, u\}$. Let $o_i = v_{i+1} - v_i$ and $\mathcal{O}_i = \{v_i + 1,\ v_i + 2,\ \ldots, v_{i+1}\}$ for $i \in \mathcal{U}$ such that $\mathcal{U} = \{1,\ 2,\ \ldots, u\}$. Therefore, $|\mathcal{S}_i| = v_i$ and $|\mathcal{O}_i| = o_i$. Let $\mathbf{x} = x_1,\ x_2,\ \ldots\ x_n$, where each $x_i \in \mathbb{F}_q$.

The central map, $\mathcal{F} : \mathbb{F}_q^n \to \mathbb{F}_q^m$, contains $m = n - v_1$ polynomials, $p^{(1)},\ p^{(2)},\ \ldots, p^{(m)} \in \mathbb{F}[\mathbf{x}]$ of the form;

$$p^{(k)}(\mathbf{x}) = \sum_{i,j \in S_\ell, i \le j} \alpha_{ij}^{(k)} x_i x_j + \sum_{i \in O_\ell, j \in S_\ell} \beta_{ij}^{(k)} x_i x_j + \sum_{i \in S_\ell \cup O_\ell} \gamma_i^{(k)} x_i + \delta^{(k)}, \tag{3}$$

where k is a specific polynomial in the map and ℓ is the only integer such that $k \in \mathcal{O}_\ell$.

Let $\mathbf{S}_\ell$ be the set of variables $(x_i \mid i \in \mathcal{S}_\ell)$ called the Vinegar variables for layer ℓ. Let $\mathbf{O}_\ell$ be the set of variables $(x_i \mid i \in O_\ell)$ called the Oil variables for layer ℓ. Additionally, let $\mathbf{S}$ and $\mathbf{O}$ be the set of all vinegar and oil variables, respectively.

B. Public and Private Key

Public Key: In order to hide the composition of the central map, two affine transformations are composed with $\mathcal{F}$, $\mathcal{L}_1 : \mathbb{F}_q^m \to \mathbb{F}_q^m$ and $\mathcal{L}_2 : \mathbb{F}_q^n \to \mathbb{F}_q^n$. Therefore, the public key is defined as $\mathcal{P} = \mathcal{L}_1 \circ \mathcal{F} \circ \mathcal{L}_2$ where $\mathcal{P} : \mathbb{F}_q^n \to \mathbb{F}_q^m$.

Private Key: The private key is made up of $\mathcal{F}$, $\mathcal{L}_1^{-1}$ and $\mathcal{L}_2^{-1}$, and therefore its size is $|\mathcal{F}| + |\mathcal{L}_1| + |\mathcal{L}_2|$. $\mathcal{L}_1$ and $\mathcal{L}_2$ are ensured to be invertible. Let $\mathcal{F}_\ell$ be the set of polynomials for a given layer, and $f_\ell = |\mathcal{F}_\ell|$. Let $l1 = |\mathcal{L}_1|$ and $l2 = |\mathcal{L}_2|$.

Since $\mathcal{L}_1$ is an affine transformation, it is made up of a matrix $\mathbf{M_{L_1}} \in \mathbb{F}_q^{m \times m}$ and a vector $\mathbf{c_{L_1}} \in \mathbb{F}_q^m$, $|\mathcal{L}_1| = m \cdot (m+1)$. Since $\mathcal{L}_2$ is an affine transformation, it is made up of a matrix $\mathbf{M_{L_2}} \in \mathbb{F}_q^{n \times n}$ and a vector $\mathbf{c_{L_2}} \in \mathbb{F}_q^n$, $|\mathcal{L}_2| = n \cdot (n+1)$.

Using (3), the types of the coefficients in a given polynomial will be identified to derive the size of $\mathcal{F}$. Let terms corresponding to coefficients

- α_{ij} be denoted as the vinegar-vinegar (VV) type,
- β_{ij} be denoted as the vinegar-oil (VO) type,
- γ_i be denoted as the vinegar-oil-only (VOO) type,
- δ be denoted as the constant (C) type.

Therefore, the number of coefficients of each type is given by; $|VV| = \binom{v+1}{2}$, $|VO| = v \cdot o$, $|VOO| = v + o = n$, $|C| = 1$. For each polynomial, $p^{(k)}$,

$$|p^{(k)}| = |VV| + |VO| + |VOO| + |C| = \binom{v+1}{2} + v \cdot o + n + 1 \tag{4}$$

Since $\mathcal{F}$ contains o polynomials, $|\mathcal{F}| = o \cdot |p^{(k)}|$.

$$o \cdot \left(\binom{v+1}{2} + v \cdot o + n + 1\right) + n \cdot (n+1) \tag{5}$$

Using (5) and extending it to Rainbow's layering scheme, the size of the private key is:

$$\sum_{l=1}^{u} o_l \cdot \left(\binom{v_l+1}{2} + v_l \cdot o_l + v_{l+1} + 1\right) + m \cdot (m+1) + n \cdot (n+1) \tag{6}$$

C. Signature Generation and Verification

Signature Generation: Prior to generating the signature, a hash, $\mathbf{msg\_in}$, of the message, $\mathbf{msg}$, is generated, such that, $\mathcal{H} : \{0,1\}^\star \to \mathbb{F}_q^m$ and $\mathbf{msg\_in} = \mathcal{H}(\mathbf{msg}) \in \mathbb{F}_q^m$. In order to calculate the signature, $\mathbf{sgn\_out}$, the following calculations using the private key are performed: $\mathbf{sgn\_out} = (\mathcal{L}_2^{-1} \circ \mathcal{F}^{-1} \circ \mathcal{L}_1^{-1})(\mathbf{msg\_in})$. We assume $u = 2$, and therefore $\ell \in (1, 2)$. The first step is to apply a matrix multiplication and vector addition using *Affine Transformation* to find $\mathbf{Y} = \mathcal{L}_1^{-1}(\mathbf{msg\_in})$, where $\mathbf{Y} = \mathbf{Y_1}|\mathbf{Y_2}$, such that $\mathbf{Y}_l \in \mathcal{F}^{o_l}$. Next, the central map must be inverted through a recursive process that produces each layer's Oil set $\mathbf{O}_\ell = \mathcal{F}_\ell^{-1}(\mathbf{Y}_\ell)$. *Polynomial Evaluation* is used to reduce a layer's quadratic polynomials into a linear system of equations, called $\mathcal{F}'_\ell$, which can be solved by the *System Solver* to obtain

$\mathbf{O}_\ell$. This process continues until we obtain the last set of Oil variables $\mathbf{O}_2$. Next, matrix multiplication and vector addition is performed in *Affine Transformation* to find the signature $\mathbf{sgn\_out} = \mathcal{L}_2^{-1}(\mathbf{S}|\mathbf{O})$. If an inconsistent system is detected, all steps must be performed again with a new set of randomly chosen vinegar variables (i.e., $\mathbf{S_1}$).

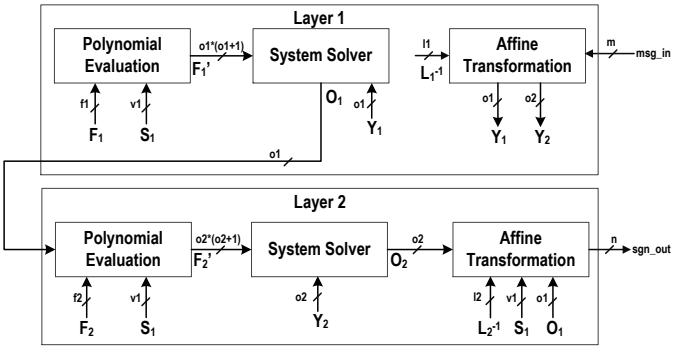

Fig. 1. Rainbow Signature Generation Flow Diagram

Signature Verification: Prior to verifying a signature is valid, a hash is computed such that $\mathbf{msg\_in} = \mathcal{H}(\mathbf{msg})$. Next the signature, $\mathbf{sgn\_in}$, is used to produce $\mathbf{msg\_out} = \mathcal{P}(\mathbf{sgn\_in})$ through *Polynomial Evaluation*. If $\mathbf{msg\_in} = \mathbf{msg\_out}$ the signature is valid and the *is_valid* signal is set to 1, otherwise 0. This process is illustrated in Fig. 2.

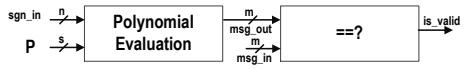

Fig. 2. Rainbow Signature Verification Flow Diagram

D. Parameter Sets

The category I parameter sets for field sizes GF(16) and GF(256) were selected for implementation in this paper and are summarized below in Table I. These parameter sets were taken from the reference documentation available in the zip file of the Rainbow submission [4].

TABLE I
RAINBOW PARAMETER SELECTION

Sec level (bits)	Parameters (Cat[1]: $\mathbb{F}$, v_1,o_1,o_2)	*Pub Key size (kB)*	*Priv Key size (kB)*	*Hash size (bits)*	*Sign size (bits)*
80	N/A: (GF(256), 18, 12, 12)	22.2	17.4	192	336
143[2]	Ia: (GF(16), 32, 32, 32)	148.5	97.9	256	512
143[2]	Ic: (GF(256), 40, 24, 24)	187.7	140.0	384	832

[1]NIST Security Category with letter representing finite field used [4].
[2]Classical gates security level stated as specified in [4].

III. DESIGN OVERVIEW

As shown in Figs. 1 and 2, the main functional units required to execute signature generation and verification in Rainbow are: Polynomial Evaluation (PEval), System Solver (SS), and Affine Transformation (AT). Our primary optimization target is minimum latency, expressed as Execution Time (μs).

A. Rainbow Operations

TABLE II
DESCRIPTION OF RAINBOW OPERATIONS

Operation	*Description*
AT	Performs vector addition (VA) followed by matrix-vector multiplication (MVM), which requires multiplication followed by addition to combine like terms.
PEval	Performs the substitution of known values in the polynomial, which amounts to multiplication followed by addition to combine like terms. For example, the VV and VOO terms that add to either the corresponding VO terms or the C term described in (4).
SS	Performs the steps required for Gauss-Jordan elimination described in Algorithm 1.

The detailed steps required for signature generation are given in Table III. Signature verification is similar to steps 3 and 5, and require only a PEval operation followed by an equality check, as shown in Fig. 2.

TABLE III
RAINBOW SIGNATURE STEPS

Step	*Operation*	*Expression*	*Result*
1	AT1-VA	$msg\_in + c_1$	a_1
2	AT1-MVM	$L_1^{-1} \times a_1$	$Y = Y_1 \vert Y_2$
3	PEval1	$F_1(S_1)$	F_1'
4	SS1	Solve $F_1' = Y_1$	O_1
5	PEval2	$F_2(S_1 \vert O_1)$	F_2'
6	SS2	Solve $F_2' = Y_2$	O_2
7	AT2-VA	$(S_1 \vert O_1 \vert O_2) + c_2$	a_2
8	AT2-MVM	$L_2^{-1} \times a_2$	sgn_out

AT1-VA is the VA required in the first AT operation.
AT1-MVM is the MVM required in the first AT operation.
AT2-VA and AT2-MVM correspond to the second AT operation.

B. Top-level Architecture Overview

The operations used in Rainbow are described in Table II. The top-level data path is shown in Fig 3. The data path contains four central components: the PreProcessor, Rainbow Core, MEMORY, and PostProcessor. The PreProcessor is used to load external data into SipoRamLBS, the Rainbow core executes signature generation and verification, MEMORY (internal) is used to store temporary matrix data, and the PostProcessor prepares external output. The Pre/Post processor provide I/O compliance with the Universal PQC Hardware API in [5].

Performing SS: The Linear System Solver (LSS) component is used to perform Gauss-Jordan elimination. Its use and behavior is described in detail in Section IV.

Performing AT and PEval: LSS is reused to perform SS and the multiplications required in AT and PEval, and it is described in detail in Section IV. In this mode of LSS, operands are scheduled as vector inputs and produce the result of element-wise vector multiplication. There are $o_\ell \times (o_\ell + 1)$

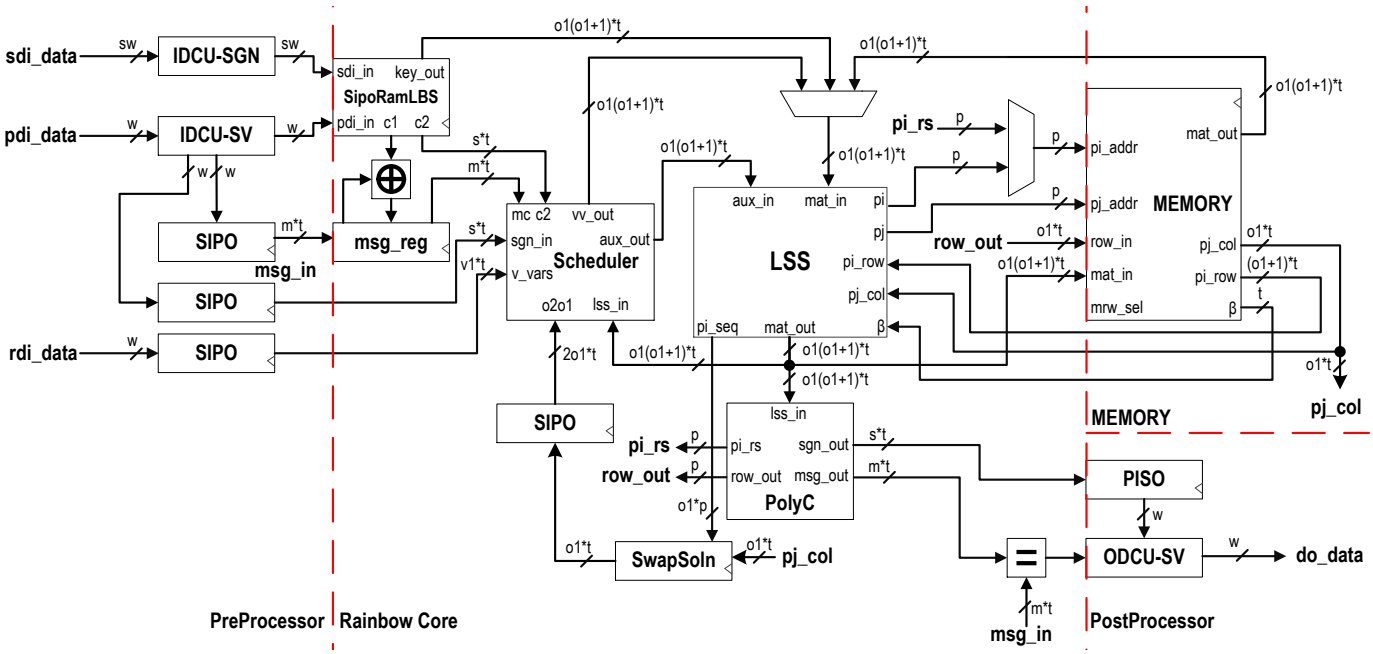

Fig. 3. Rainbow Top-Level Diagram - Note: Bus widths are in bits

multipliers in LSS, that multiply elements in the field $GF(2^t)$, where t is the power of 2 in the scheme, either 8 or 4.

The inputs correspond to a portion of the matrix-vector multiplication from AT or substitutions from PEval. Since only multipliers are re-used in LSS, the remaining additions for combining like terms occur in the Polynomial Combiner (PolyC) component (see Section III-D). The result of this combination is $Y1$ and $Y2$, or either sgn_out, row_out or msg_out, where row_out corresponds to a given row in F_1' or F_2'. These signals are in the Result column of Table III for steps 2, 3, 5 and 8, and msg_out is used in verification.

The complete execution of the Rainbow Signature scheme on this architecture is shown in Table IV.

C. Scheduler

The input sequence generator circuit ($Scheduler$) is able to provide the appropriate inputs to LSS that are paired with coefficients contained in the current line of memory being processed from $SIPORamLBS$ for the private or public key based operations. It provides precise multiplicands corresponding to each byte in the current line of memory. For example, during AT operations, the Scheduler will provide the correct sequence of inputs required to perform the matrix-vector multiplication. In Table III, the matrices are L_1^{-1} and L_2^{-1}, and the vectors are a_1 and a_2, respectively.

The sequences required for aux_in are generated from variables and are stored internally. Then, using a circular shift register that rotates by the multiplier width, $o_\ell \times (o_\ell + 1)$, each clock cycle the precise inputs are generated. Similar to AT, matching inputs for each line of memory is used in polynomial evaluation. However, there is a SIPO available in Scheduler that stores the results of VV multiplications, which are reused

TABLE IV
RAINBOW SIGNATURE STEPS ON THIS ARCHITECTURE

Step	*Operation*	*Description*	*Processing Compo-nent(s)*	*Result Stored to*
1	LOAD	Use preprocessor to load msg_in and the key.	PreProcessor	SipoRamLBS, msg_reg
2	AT1-VA	See Table III.	XOR	Scheduler
3	AT1-MVM	Schedule block inputs, perform multiplications and additions.	Scheduler, LSS, PolyC	PolyC
4	PEval1	Schedule block inputs, perform multiplications and additions.	Scheduler, LSS, PolyC	MEMORY
5	SS1	Obtain F_1' from MEMORY and Solve for O_1	MEMORY, LSS, SwapSoln	Scheduler
6	PEval2	Schedule block inputs, perform multiplications and additions.	Scheduler, LSS, PolyC	MEMORY
7	SS2	Obtain F_2' from MEMORY and Solve for O_2	MEMORY, LSS, SwapSoln	Scheduler
8	AT2-VA	See Table III.	XOR	Scheduler
9	AT2-MVM	Schedule block inputs, perform multiplications and additions.	Scheduler, LSS, PolyC	PostProcesor
10	OUTPUT	Using PostProcessor, transmit data externally	PostProcesor	External

for subsequent polynomial evaluations - effectively eliminating the need to perform these multiplications with each polynomial as done in [8].

D. PolyC

All outputs from LSS are sent to PolyC for bulk elimination. For AT, all lines of memory are stored in a SIPO. The AT operations produce $Y = Y_1|Y_2$, which is stored in PolyC, the signature, sgn_out, and the verification message, msg_out. For PE, only one polynomial is reduced at time, and the matrix is updated accordingly (per row). The input lss_in is used to read the output of LSS of size $o_\ell \times (o_\ell + 1)$. These results are stored in a $SIPO$, which is used to sum L_1 or L_2 or a row corresponding to a polynomial in F_1, F_2 or PK (the public map). In the case of PK, each summation results in an element that is stored in the SIPO used to generate msg_out. The summation of L_1 and L_2 is stored internally while row updates are sent directly to MEMORY. The incremental constants within row updates for F_1 and F_2 are combined with the corresponding element from Y_1 or Y_2, respectively.

IV. LSS Overview

In order to solve an $n \times n$ linear system of equations, the Gauss-Jordan elimination method is used, such that the number of variables is $n = o1 = o2$ and the number of polynomials is also n. To speed up the multiplication process for AT and PEval, nearly all multipliers are re-used every clock cycle, which requires a dual-mode of operation at the cost of multiplexers and an additional input port denoted aux_in. Note, adders in LSS are not reused. Additionally, multiplication is performed using the architecture described in [9].

A. Interface and Operation

The LSS component shown in Fig. 3 has inputs mat_in and aux_in, which represent either a system to be solved or an element-wise vector multiplication (corresponding to parts of the AT and PEval). Memory uses pi and pj to select pj_col, pi_row and β synchronously after storing the input mat_out. The output of LSS, called mat_out, represents either a solved linear system or the element-wise vector product of mat_in and aux_in.

The data signals are assumed to be a single row vector representing either a matrix or a vector. The elements inside each signal will be referred to in the form $data(i, j)$, $data(i)$ *or* $data(j)$ $\forall i \in (0, \ldots, n-1)$ *and* $\forall j \in (o, \ldots, n)$. Additionally, signals pi, pj, pi_seq and $valid$ signals will be discussed in Section IV-C, and $p = \lceil log_2(n) \rceil$ bits. Figures in this section may have a subscript notation that is equivalent to the above index notation, for example, $data(i, j)$ is equivalent to $data_{i,j}$.

B. Multiplication Operation

When $mode = 1$, LSS is in the batch multiplication mode, where all data inputs are used. By utilizing all multipliers in the PEs, there are $n \times (n + 1)$ multiplications possible per clock cycle. For each clock cycle the result is available on the data output bus mat_out corresponding to the element-wise multiplication of mat_in and aux_in.

C. System Solving Operation

When $mode = 0$, LSS is in the system solving mode, where only mat_in contains the system that needs to be solved. Algorithm 1 summarizes the Gauss-Jordan elimination algorithm, which requires calculating the pivot, multiplicative inversion, normalization and elimination. There is one main loop that iterates over each column in the input system of linear equations, A. Columns are represented by a counter pj, which represents the index of the current pivot column, pj_col. Algorithm 2 shows the steps required to calculate the location of the current pivot, pi, which amounts to searching for the first non-zero element in the current pivot column, pj_col. Additionally, pi is used to select the current pivot row pi_row.

Lines 4-9 of Algorithm 1 contain a loop that iterates n times and computes the current pivot, performs normalization and elimination. This loop will be described here in detail. First, the current pivot is computed and then β is selected using pj and pi. Line 7 executes Normalize, which is defined in Algorithm 3 - where multiplicative inversion is used to compute the inverse of the current pivot, β. Normalize amounts to multiplication of the inverse of the current pivot, β^{-1}, with every element in pi_row, and produces the normalized pivot row, N. Line 8 executes Eliminate, which is defined in Algorithm 4. Eliminate uses N from the previous step to update each element of A based on whether the current element's row is the pivot row. As shown in Algorithm 4, if it is the pivot row, it is assigned the corresponding value from N (Line 7), and otherwise it is eliminated as shown in Line 9.

Algorithm 1 Solving an $n \times n$ system of linear equations using Gauss-Jordan Elimination in n iterations without swapping

```
1: INPUT: A∈ GF(2^t)^{n×(n+1)}
2: OUTPUT: Reduced Row Echelon Form (RREF) of A
3:
4: for pj from 0 to n-1 do
5:     pi ← PivotCalc(pj_col)
6:     β ← A(pi, pj)
7:     N ← Normalize(β, pi_row)
8:     A ← Eliminate(pi, pj_col, N, A)
9: end for
```

LSS executes each iteration of the Algorithm 1 in parallel by using $n \times (n + 1)$ processing elements (PE). Additionally, by utilizing appropriate hardware components, all loop-based operations shown in Algorithm 2, Algorithm 3 and Algorithm 4 also occur in parallel. The top-level diagram of LSS is shown in Fig. 4. A PivotCalc circuit was created to execute Algorithm 2 and is described in Section IV-D1. The multiplicative inverse component, I, is described in Section IV-D2. Algorithm 3 is executed by a row of $n+1$ multipliers shown in

Algorithm 2 PivotCalc

1: **INPUT:** $pj\_col \in GF(2^t)^n$
2: **OUTPUT:** $pi \in \mathbb{N}$
3:
4: **for** i from 0 to n-1 **do**
5: **if** $pj\_col(i) \neq 0$ and i not assigned to pi before **then**
6: $pi \leftarrow i$
7: break
8: **end if**
9: **end for**

Algorithm 3 Normalize

1: **INPUTS:** $\beta \in GF(2^t), pi\_row \in GF(2^t)^{n+1}$
2: **OUTPUT:** $N \in GF(2^t)^{n+1}$
3:
4: $\beta^{-1} \leftarrow I(\beta)$ ▷ Compute inverse of β
5: **for** j from 0 to n **do**
6: $N(j) \leftarrow \beta^{-1} \times pi\_row(j)$
7: **end for**

the top of Fig. 4. Finally, Algorithm 4 is executed by uniform PEs, that are described in Section IV-D3.

D. LSS Components

This section will describe the components used in LSS that are shown in Fig. 4.

1) PivotCalc: The diagram for $PivotCalc$ is shown in Fig. 5. It takes as input, each element in the current pivot column, pj_col. Each element is input into n comparators, denoted by a "$! = 0$" block with a corresponding enable, en_i. If the en_i signal is 0, then the corresponding output i of the comparator is 0. Otherwise, the not-equal-zero comparator operates normally. Based on the output of the comparators, an n bit signal, $neq0i$ is generated, and sent to a priority encoder that outputs the pivot row index, pi.

In order to generate eni, the output pi is sent to a decoder to produce a local signal dec_out, which produces an n-bit signal *op* that sets the *operation* of each PE. Also, the decoder is necessary because $neq0i$ may have more than one non-zero

Algorithm 4 Eliminate

1: **INPUTS:** $pi \in \mathbb{N}, pj\_col \in GF(2^t)^n, N \in GF(2^t)^{n+1}$, A$\in GF(2^t)^{n\times(n+1)}$
2: **OUTPUT:** A after Elimination
3:
4: **for** i from 0 to n-1 **do**
5: **for** j from 0 to n **do**
6: **if** i = pi **then**
7: $A(i,j) \leftarrow N(j)$
8: **else**
9: $A(i,j) \leftarrow A(i,j) - pj\_col(i) \times N(j)$
10: **end if**
11: **end for**
12: **end for**

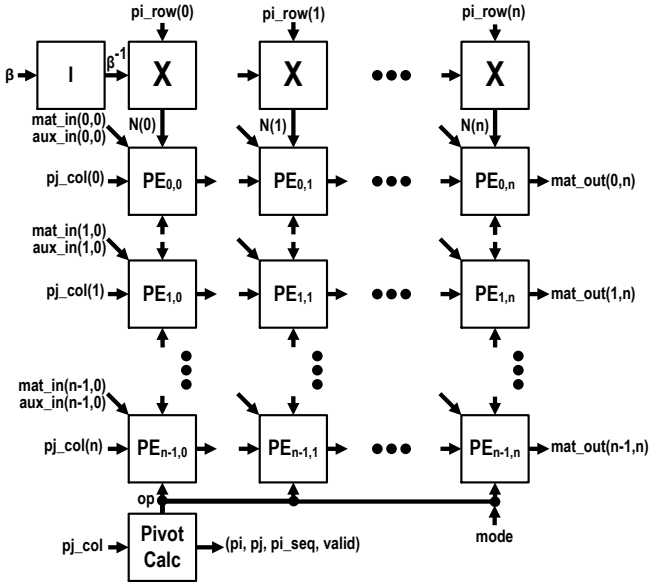

Fig. 4. LSS Top-level Diagram - Note: The left arrow entering the multipliers in the top row are β^{-1}. Each $PE_{i,j}$ has a top diagonal, top and bottom input corresponding to $mat\_in(i,j)$ and $aux\_in(i,j)$, $N(j)$, op and $mode$, respectively. The output of a given $PE_{i,j}$ is $mat\_out(i,j)$. More detail is available in Section IV-D3.

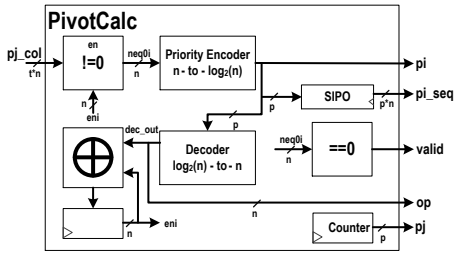

Fig. 5. PivotCalc Component

element, and therefore, op will have a 1 set at the first non-zero element for which a pivot row has not previously been used (see line 5 of Algorithm 2). Next, dec_out is XORed with the previously stored values of eni, which keeps track of previously used pivot rows, and the result is stored in the eni register. Each clock cycle, the calculated pi is stored in a SIPO and produces pi_seq once the system is solved. Additionally, the $valid$ signal indicates if the solution of the system is valid. As shown in Fig. 5, this is accomplished by checking if at any iteration no valid pivot row exists, for example, $\forall i \in (0,\ ...,\ n-1),\ neq0(i) == 0$.

2) Multiplicative Inversion: Either a 256×8 or 16×4 look-up table is used to perform multiplicative inversion for an element in the field $GF(2^8)$ or $GF(2^4)$, respectively. The component that performs inversion, denoted I in Fig. 4, takes the current pivot, β, as input and produces its inverse, β^{-1}.

3) PEs: The PE block diagram is shown in Fig. 6, which performs both elimination and normalization of any element in the matrix. It consists of one two-input multiplier, one two-input XOR gate, and 4 multiplexers that are required to reuse

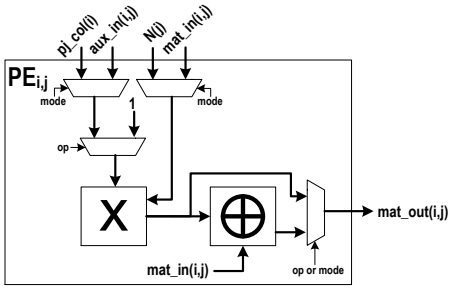

Fig. 6. Processing Element Block Diagram. Note: the $mode$ signal indicates either multiplication or system solving, and in system solving mode, the op signal indicates the operation of either normalization or elimination.

the PE for multiplication. The inputs and output elements correspond to subscript i and/or j of the current PE.

For example, $\text{PE}_{0,0}$ has the inputs $mat\_in(0,0)$, $pj\_col(0)$, $aux\_in(0,0)$, and $N(0)$. When $mode = 1$ and $op = 1$, the component performs $mat\_out(0,0) = mat\_in(0,0) \times aux\_in(0,0)$. When $mode = 0$, the PE is in system solving mode and performs either normalization or elimination on $mat\_in(0,0)$ depending on whether $op = 0$ or $op = 1$, respectively. In either case, the result is sent to $mat\_out(0,0)$. When $op = 0$, normalization is performed by $N(0) \times 1$. When $op = 1$, the inputs are the corresponding element in the pivot column element, $pj\_col(0)$ and the corresponding element in the normalized pivot row N, $N(0)$. Then, elimination is performed by $pj\_col(0) \times N(0) + mat\_in(0,0)$.

E. Operational Example

This section gives an example of solving a 5×5 system with elements in $GF(2^8)$ using LSS. The detailed steps for Gauss-Jordan elimination on a matrix is shown in Table V. For each iteration, pivot rows and columns are made bold based on the pj and pi indexes generated by PivotCalc. Additionally, each step also shows the internal values of PivotCalc that are used to control each PE. The final result is shown on in the last column of mat_out on iteration 5.

TABLE V
STEPS FOR SOLVING A 5×5 SYSTEM USING LSS

it#	mat_in	mat_out	*Signals*
1	$\begin{bmatrix}\mathbf{1b}\ \mathbf{43}\ \mathbf{d6}\ \mathbf{42}\ \mathbf{41}\ \mathbf{e6}\\ \mathbf{13}\ 4d\ 5e\ a9\ 17\ 81\\ \mathbf{12}\ 83\ 79\ 02\ 17\ df\\ \mathbf{f3}\ 67\ f2\ 5d\ 67\ f5\\ \mathbf{1e}\ 98\ f6\ 39\ 97\ ef\end{bmatrix}$	$\begin{bmatrix}01\ f4\ 4b\ 2b\ 07\ 11\\ 00\ 00\ 4a\ fe\ 6e\ ef\\ 00\ 3a\ 26\ 7e\ 69\ a0\\ 00\ 82\ 61\ eb\ 24\ 6a\\ 00\ 28\ 0a\ fc\ cd\ 5c\end{bmatrix}$	pj=0, pi=0 eni= $\begin{bmatrix}1&1&1&1&1\end{bmatrix}$ neq0= $\begin{bmatrix}1&1&1&1&1\end{bmatrix}$ dec_out= $\begin{bmatrix}1&0&0&0&0\end{bmatrix}$
2	$\begin{bmatrix}01\ \mathbf{f4}\ 4b\ 2b\ 07\ 11\\ 00\ \mathbf{00}\ 4a\ fe\ 6e\ ef\\ \mathbf{00\ 3a\ 26\ 7e\ 69\ a0}\\ 00\ \mathbf{82}\ 61\ eb\ 24\ 6a\\ 00\ \mathbf{28}\ 0a\ fc\ cd\ 5c\end{bmatrix}$	$\begin{bmatrix}01\ 00\ 69\ cf\ d8\ 78\\ 00\ 00\ 4a\ fe\ 6e\ ef\\ 00\ 01\ a9\ 6e\ a4\ 23\\ 00\ 00\ 10\ 2f\ 2c\ 48\\ 00\ 00\ cb\ 5d\ 89\ 10\end{bmatrix}$	pj=1, pi=2 eni= $\begin{bmatrix}0&1&1&1&1\end{bmatrix}$ neq0= $\begin{bmatrix}0&0&1&1&1\end{bmatrix}$ dec_out= $\begin{bmatrix}0&0&1&0&0\end{bmatrix}$
4	$\begin{bmatrix}01\ 00\ \mathbf{69}\ cf\ d8\ 78\\ \mathbf{00\ 00\ 4a\ fe\ 6e\ ef}\\ 00\ 01\ \mathbf{a9}\ 6e\ a4\ 23\\ 00\ 00\ \mathbf{10}\ 2f\ 2c\ 48\\ 00\ 00\ \mathbf{cb}\ 5d\ 89\ 10\end{bmatrix}$	$\begin{bmatrix}01\ 00\ 00\ ef\ e5\ 75\\ 00\ 00\ 01\ 52\ d5\ ae\\ 00\ 01\ 00\ be\ 92\ ad\\ 00\ 00\ 00\ 3b\ ba\ c0\\ 00\ 00\ 00\ 51\ fb\ 68\end{bmatrix}$	pj=2 , pi=1 eni= $\begin{bmatrix}0&1&0&1&1\end{bmatrix}$ neq0= $\begin{bmatrix}0&1&0&1&1\end{bmatrix}$ dec_out= $\begin{bmatrix}0&1&0&0&0\end{bmatrix}$
4	$\begin{bmatrix}01\ 00\ 00\ \mathbf{ef}\ e5\ 75\\ 00\ 00\ 01\ \mathbf{52}\ d5\ ae\\ 00\ 01\ 00\ \mathbf{be}\ 92\ ad\\ \mathbf{00\ 00\ 00\ 3b\ ba\ c0}\\ 00\ 00\ 00\ \mathbf{51}\ fb\ 68\end{bmatrix}$	$\begin{bmatrix}01\ 00\ 00\ 00\ b4\ 54\\ 00\ 00\ 01\ 00\ 9f\ 4e\\ 00\ 01\ 00\ 00\ aa\ 9a\\ 00\ 00\ 00\ 01\ da\ 52\\ 00\ 00\ 00\ 00\ 92\ 7e\end{bmatrix}$	pj=3 , pi=3 eni= $\begin{bmatrix}0&0&0&1&1\end{bmatrix}$ neq0= $\begin{bmatrix}0&0&0&1&1\end{bmatrix}$ dec_out= $\begin{bmatrix}0&0&0&1&0\end{bmatrix}$
5	$\begin{bmatrix}01\ 00\ 00\ 00\ \mathbf{b4}\ 54\\ 00\ 00\ 01\ 00\ \mathbf{9f}\ 4e\\ 00\ 01\ 00\ 00\ \mathbf{aa}\ 9a\\ 00\ 00\ 00\ 01\ \mathbf{da}\ 52\\ \mathbf{00\ 00\ 00\ 00\ 92\ 7e}\end{bmatrix}$	$\begin{bmatrix}01\ 00\ 00\ 00\ 00\ \mathbf{4f}\\ 00\ 00\ 01\ 00\ 00\ \mathbf{00}\\ 00\ 01\ 00\ 00\ 00\ \mathbf{25}\\ 00\ 00\ 00\ 01\ 00\ \mathbf{22}\\ 00\ 00\ 00\ 00\ 01\ \mathbf{31}\end{bmatrix}$	pj=4 , pi=4 eni= $\begin{bmatrix}0&0&0&0&1\end{bmatrix}$ neq0= $\begin{bmatrix}0&0&0&0&1\end{bmatrix}$ dec_out= $\begin{bmatrix}0&0&0&0&1\end{bmatrix}$ pi_seq= $\begin{bmatrix}0,&2,&1,&3,&4\end{bmatrix}$

V. PREVIOUS WORK

To fit different device constraints, some targeted hardware implementations for multivariate schemes have been implemented: [10], [8] and [11]. The time-area optimized implementation from [11] requires more clock cycles, but is optimal for smaller devices, such as those required in the Internet of Things (IoT). Parallel architectures for system solvers in $GF(2)$ include SMITH [12], which solves a system using Gaussian Elimination with simultaneous backward substitution. For an $n \times n$ system it takes on average $2n$ clock cycles and worst-case time complexity of $O(n^2)$.

The authors of [10] extend SMITH and call their new architecture G-SMITH. They describe an efficient method of solving linear system of equations (LSE) using shifting (instead of swapping) in a "mesh structure of smart memory cells". They also describe key components required to multiply coefficients with variables and add like-terms for the different polynomials in the AT operations (i.e. L_1 and L_2) and for evaluating the bi-linear form of F^{-1}. The core component is a mesh array made up of four types of cells and multiplexers that can be re-used as Vector Multiply and Add (VMA) and bilinear-form (BLF) units that iteratively perform AT and PEval. This also requires that inputs for the mesh array are at most of the signature size (i.e. 42 bytes), and allows each AT and PEval to have the appropriate inputs available that produce the combined result.

Compared to [10] our design can be easily adapted to a large number of parameter sets without changing the interconnections of PEs. Also, our parameterized system solver fully re-uses multipliers during all operations (i.e. AT, PEval and SS) in a uniform PE array each clock cycle. Compared to [8] we provide details of our top-level architecture, how inputs are scheduled, how processing elements can be re-used to perform AT and PEval and how data is provided to each component of the system solver. Since our system solver always executes in n clock cycles and doesn't perform swapping, it is the first hardware design that does not leak side-channel information during system solving. Additionally, our design is the first hardware implementation of the NIST Round 1 Rainbow submission at the Category I security level, and we are making it open-source [13].

VI. RESULTS

Results were generated for a Xilinx Virtex 7 (XC7VX1140) and a Kintex-7 (XC7K480) FPGA.

The implementation results for the designs described in this paper and for selected earlier work are summarized in Table VI. All designs are capable of performing both signature generation and signature verification. Resource utilization corresponds to the implementation of both operations with

TABLE VI
RAINBOW IMPLEMENTATION RESULTS

Algorithm	FPGA Family	Clock Cycles	*Maximum Freq (MHz)*	*Execution Time (μs)*	*100k Op/s*	LUTs/ ALUTs	Slices	FFs	DSP	BRAM
Rainbow-80 [10]	ASIC	804	67	12.00	0.83	-	-	-	-	-
Rainbow-80 [8]	Stratix II	198	50	3.96	2.53	21,740	-	1644	0	0
Rainbow-80	Virtex-7	148	200	0.61	8.10	17,048	5,878	9,033	0	18
ECC Curve-25519 [14]	Zynq-7000	34,052	100	397.00	0.32	34,009	11,277	43,875	220	22
ECC Curve-25519 [15]	Zynq-7000	61,235	175	170.00	0.65	13,595	5,697	20,924	187	110
Rainbow-Ia	Kintex-7	1,980	111	17.84	0.56	27,712	8,939	27,679	0	59
Rainbow-Ic	Kintex-7	979	90	10.88	0.92	52,895	15,112	32,476	0	67
Rainbow-Ia	Virtex-7	1,980	181	10.93	0.91	27,556	7,065	27,675	0	59
Rainbow-Ic	Virtex-7	979	167	5.86	1.71	52,721	15,976	32,475	0	67

resource sharing. The number of clock cycles, execution time, and the number of operations per second are given for signature generation only. The BRAMs used in our design store the entire public or private key. Flip-flops (FFs) are required by the various registers, SIPOs, PISOs, SipoRamLBS, Scheduler, MEMORY, PolyC, etc.

Compared to [10] and [8], our design for Rainbow-80 takes 81% and 25% fewer clock cycles, respectively, which is due to architectural improvements in our design. Unfortunately, we cannot compare either execution times or operations per second because our design does not fit on a Stratix II FPGA. Instead, we provide results for Virtex-7, which is several generations ahead of Stratix II, and accounts for the improvement in maximum frequency.

Between Rainbow-Ia and Rainbow-Ic, Rainbow-Ic has consistently lower maximum clock frequency, which is most likely due to the use of larger multipliers. However, at the same time, the number of clock cycles in Rainbow-Ic is smaller since there are fewer variables in the system, and this produces a higher number of operations per second. Therefore, we compare our Rainbow-Ic results with results for ECC Curve-25519, which has a 128-bit security level, and is implemented on the Xilinx-7 series Zynq-7020 FPGA by [14] [15]. For a similar technology, Kintex-7, we achieve about 36.5 and 15.6 times faster execution time and 2.9 and 1.4 times more op/s than [14] and [15], respectively. On the Virtex-7 FPGA we have been able to achieve even better results.

VII. CONCLUSION

In this paper, we have demonstrated a high-speed architecture capable of executing the Rainbow signature scheme. Compared to previous work, we are able to reduce the number of clock cycles and achieve a higher maximum frequency. We introduced a novel pivot calculation circuit, $PivotCalc$, which prevents side-channel leakage by eliminating the need to swap rows. Furthermore, the uniform PE array of LSS significantly reduces internal multiplexing, and therefore can be used to implement the higher security parameter sets required for the other NIST categories in the Round 1 Rainbow specification [4]. In the future, we plan to extend our design to the higher security levels and perform comparisons with other post-quantum schemes.

REFERENCES

[1] National Security Agency. (2015, Aug) Cryptography today. [Online]. Available: http://www.tinyurl.com/SuiteB

[2] National Institute of Standards and Technology. (2016, Apr) Report on post-quantum cryptography. [Online]. Available: https://csrc.nist.gov/publications/detail/nistir/8105/final

[3] National Institute of Standards and Technology. (2018, Aug) Post-Quantum Cryptography Project. [Online]. Available: https://csrc.nist.gov/Projects/Post-Quantum-Cryptography

[4] J. Ding, M.-S. Chen, A. Petzoldt, D. Schmidt, and B.-Y. Yang. (2017, November) Rainbow - Algorithm Specification and Documentation.

[5] A. Ferozpuri, F. Farahmand, V. Dang, M. U. Sharif, J.-P. Kaps, and K. Gaj. (2018, Apr) Hardware API for Post-Quantum Public Key Cryptosystems. [Online]. Available: https://cryptography.gmu.edu/athena/PQC/PQC_HW_API.pdf

[6] J. Ding and D. Schmidt, "Rainbow, a New Multivariable Polynomial Signature Scheme," in *International Conference on Applied Cryptography and Network Security*. Springer, 2005, pp. 164–175.

[7] A. Kipnis, J. Patarin, and L. Goubin, "Unbalanced Oil and Vinegar Signature Schemes," in *EUROCRYPT'99*, 1999.

[8] S. Tang, H. Yi, J. Ding, H. Chen, and G. Chen, "High-speed Hardware Implementation of Rainbow Signature on FPGAs," in *International Workshop on Post-Quantum Cryptography*. Springer, 2011, pp. 228–243.

[9] T. Zhang and K. K. Parhi, "Systematic Design of Original and Modified Mastrovito Multipliers for General Irreducible polynomials," *IEEE Transactions on Computers*, vol. 50, no. 7, pp. 734–749, 2001.

[10] S. Balasubramanian, H. W. Carter, A. Bogdanov, A. Rupp, and J. Ding, "Fast Multivariate Signature Generation in Hardware: The Case of Rainbow," in *Application-Specific Systems, Architectures and Processors, 2008. ASAP 2008. International Conference on*. IEEE, 2008, pp. 25–30.

[11] A. Bogdanov, T. Eisenbarth, A. Rupp, and C. Wolf, "Time-Area Optimized Public-Key Engines: MQ-Cryptosystems as Replacement for Elliptic Curves?" in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2008, pp. 45–61.

[12] A. Bogdanov, M. C. Mertens, C. Paar, J. Pelzl, and A. Rupp, "SMITH - A Parallel Hardware Architecture for Fast Gaussian Elimination over GF(2)," in *Workshop on Special-purpose Hardware for Attacking Cryptographic Systems (SHARCS 2006)*, 2006.

[13] A. Ferozpuri. (2018, Dec) GMU Rainbow Source Code. [Online]. Available: https://cryptography.gmu.edu/athena/index.php?id=PQC

[14] P. Sasdrich and T. Güneysu, "Implementing Curve25519 for side-channel–protected elliptic curve cryptography," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 9, no. 1, p. 3, 2015.

[15] K. Järvinen, A. Miele, R. Azarderakhsh, and P. Longa, "FourQ on FPGA: New Hardware Speed Records for Elliptic Curve Cryptography over Large Prime Characteristic Fields," in *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 2016, pp. 517–537.