A High-Speed Constant-Time Hardware Implementation of NTRUEncrypt SVES

Farnoud Farahmand, Malik Umar Sharif, Kevin Briggs, Kris Gaj

Department of Electrical and Computer Engineering, George Mason University, Fairfax, VA, U.S.A.

{ffarahma, msharif2, kbriggs2, kgaj}@gmu.edu

Abstract—In this paper, we present a high-speed constant-time hardware implementation of NTRUEncrypt Short Vector Encryption Scheme (SVES), fully compliant with the IEEE 1363.1 Standard Specification for Public Key Cryptographic Techniques Based on Hard Problems over Lattices. Our implementation follows an earlier proposed Post-Quantum Cryptography (PQC) Hardware Application Programming Interface (API), which facilitates its fair comparison with implementations of other PQC schemes. The paper contains the detailed flow and block diagrams, timing analysis, as well as results in terms of latency (in clock cycles), maximum clock frequency, and resource utilization in modern high-performance Field Programmable Gate Arrays (FPGAs). Our design takes full advantage of the ability to parallelize the major operation of NTRU, polynomial multiplication, in hardware. As a result, the execution time bottleneck shifts to the hash function, SHA-256, which is sequential in nature and as a result cannot be easily sped up in hardware. The obtained FPGA results for NTRU Encrypt SVES are compared with the equivalent results for Classic McEliece, a competing, well-established Post-Quantum Cryptography encryption scheme, with a long history of unsuccessful attempts at breaking. Our code for NTRUEncrypt SVES is being made open-source to speed-up further design-space exploration and benchmarking on multiple hardware platforms.

Index Terms—NTRU, lattice-based, hardware, API, P1363.1

I. Introduction

NTRUEncrypt is a polynomial ring-based public-key encryption scheme that was first introduced at Crypto'96. The first formal paper describing this scheme was published at ANTS III [1]. In 2008, an extended version of this algorithm was published as the IEEE 1363.1 Standard Specification for Public Key Cryptographic Techniques Based on Hard Problems over Lattices [2]. Within the standard, the described algorithm is called Short Vector Encryption Scheme – SVES. Since the core of this algorithm is known in the academic literature as NTRUEncrypt, we will refer to the full cryptosystem as NTRUEncrypt SVES. The recent renewed interest in NTRU is at least partially driven by its presumed resistance to any efficient attacks using quantum computers. In December 2016, National Institute of Standards and Technology (NIST) published an official Call for Proposals and Request for Nominations for Public-Key Post-Quantum Cryptographic Algorithms, followed by the submission and acceptance of 69 candidates to the first Round of the NIST standardization

process a year later. Among the candidates, there are new, substantially modified versions of NTRUEncrypt. However, in an attempt to characterize an already standardized algorithm, in this paper, we focus on the still unbroken version of the algorithm published in 2008. We are not aware of any previous high-speed hardware implementation of the entire NTRUEncrypt SVES scheme reported in the scientific literature or available commercially. Our implementation is also unique in that it is the first implementation of any PQC scheme following our newly proposed PQC Hardware API [3]. As such, it provides a valuable reference for any future implementers of PQC schemes, which is very important in the context of the ongoing NIST standard candidate evaluation process.

II. Previous Work

Software implementations of NTRUEncrypt are available in eBACS [4]. The full open source implementation of the IEEE P1363.1 standard, developed by Security Innovation, Inc., is available in [5]. The majority of these implementations support multiple parameter sets with security levels of 112, 128, 192 and 256 bits. Bailey et al. [6] implemented NTRU on a wide variety of microcontrollers, as well as Xilinx Virtex-E 1000 family of FPGAs. O'Rourke et al. [7] presented a scalable architecture to perform NTRU multiplication, and proposed a unified architecture based on Montgomery multiplication. Kaps proposed a scalable low-power design for NTRU polynomial multiplication [8]. In [9], Atici et al. presented a compact and low power NTRU design that was suitable for pervasive security applications, such as RFIDs and sensor nodes. Kamal et al. [10] investigated several hardware implementation options for NTRUEncrypt, and analyzed its performance on Virtex-E FPGA devices. The design was configurable to perform modular reduction using Mersenne Prime and the look-up table based architecture. The first attempt at the hardware modeling of the entire functionality of P1363.1 was reported in [11], [12]. A hybrid behavioral and structural VHDL model was developed. Unfortunately, no implementation results (maximum clock frequency, execution time, resource utilization, etc.) were reported, which most likely indicates that the developed model was not synthesizable. The most complete and efficient high-speed hardware implementation of NTRUEncrypt itself (without the remaining components of SVES), supporting all major parameter sets, was reported in [13], [14]. Only encryption is fully supported. No operations specific to decryption are explicitly supported.

A. Background on NTRUEncrypt SVES

The flow diagrams of the NTRUEncrypt SVES encryption and decryption operations are shown in Fig. 1. The notation used and the names of basic operations, inputs, outputs, and intermediate variables are explained in Tables 1 and 2.
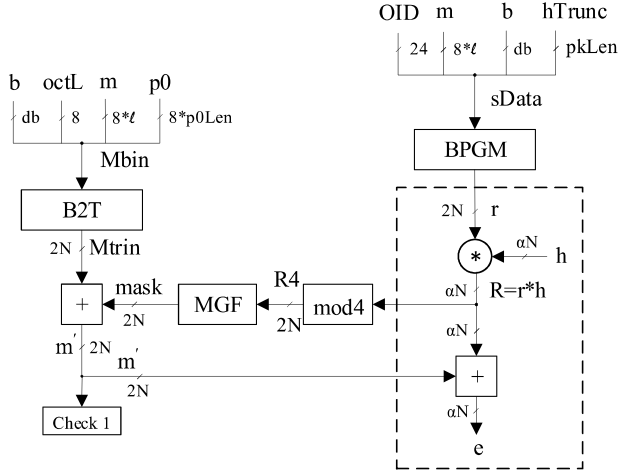
(a) Encryption
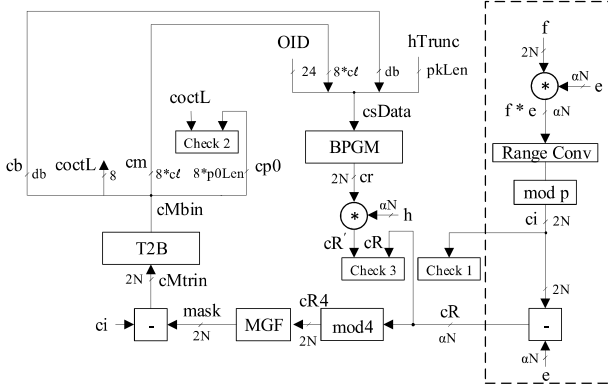
(b) Decryption

Fig. 1: Flow diagram of NTRUEncrypt SVES.

In Fig. 1, the operations of the core NTRUEncrypt scheme, known from the early literature on the topic, such as [6], are shown in dashed boxes. NTRUEncrypt has three major parameters (N, p, q) such that a) N is prime, b) p and q are

relatively prime, $gcd(p, q) = 1$, and c) q is much larger than p. For the purpose of efficiency p is typically chosen to be 3, and q as a power of two. The scheme is based on polynomial additions and multiplications in the ring $R = Z[X]/X^N - 1$. We use the $*$ to denote a polynomial multiplication in R, which is the cyclic convolution of the coefficients of two polynomials. After completion of a polynomial multiplication or addition, the coefficients of the resulting polynomial need to be reduced either modulo q or p. The key creation process also requires two polynomial inversions, which can be computed using the Extended Euclidean Algorithm. The procedures are briefly outlined below. During the key generation, the user chooses two random secret polynomials $F \in R$ and $g \in R$, with so called "small" coefficients, i.e., coefficients reduced modulo p (typically chosen to be in the integer range from -1 to $+1$, and thus limited to -1, 0, and 1. The private key f is computed as $f = 1 + pF$. The public key h is calculated as

$$h = f^{-1} * g \cdot p \ \text{ in } (Z/qZ)[X]/(X^N - 1) \tag{1}$$

The message m is assumed to be a polynomial with small coefficients. The ciphertext is computed as

$$e = r * h + m \ (mod \, q) \tag{2}$$

where $r \in R$ is a randomly chosen polynomial with small coefficients. The decryption procedure requires the following three steps:

- calculate $f * e \, (mod \, q)$
- shift coefficients of the obtained polynomial to the range $[-q/2, q/2)$,
- reduce the obtained coefficients $mod \, p$.

h and e are naturally polynomials with so called "big" coefficients, i.e., coefficients in the range from 0 to q-1. In the SVES encryption scheme shown in Fig. 1, m is replaced

TABLE I: Basic operations of Encryption and Decryption

Name	Description
Poly Mult, *	Polynomial Multiplication (ring multiplication in $Z[X]/(X^N - 1)$)
BPGM	Blinding Polynomial Generation Method
MGF	Mask Generation Function
Range Conv	Range Conversion from [0, q] to [-q/2, q/2]
B2T	Conversion of each group of three bits to two ternary coefficients
T2B	Conversion of two ternary coefficients to a group of three bits
Poly Add, +	Polynomial Addition
Poly Sub,	Polynomial Subtraction
Check 1	Checking whether an input polynomial with small coefficients contains at least dm0 1s, -1s, and 0s. If not, setting fail=1
Check 2	Checking whether all bytes of padding after decryption are 0s.
Check 3	Comparing the values of cR and cR. If they are different setting fail=1

TABLE II: Inputs, Outputs, and Intermediate Variables

Name	Role	Description
OID	in	Object identifier specifying uniquely an algorithm and parameter set used
b	in	Random data (binary string)
m	in	Message (binary string)
octL	in	Length of message m in bytes (single byte)
p0	var	Zero padding (binary string)
hTrunc	in	First pkLen bits of the public key h (binary string)
r	var	Random polynomial with small coefficients
h	in	Public key (polynomial with big coefficients)
e	out/in	Ciphertext (polynomial with big coefficients)
Mbin, cMbin sData, csData	var	Intermediate variables (binary strings)
Mtrin, mask, m, cMtrin, mask, ci	var	Intermediate variables (polynomials with small coefficients)
R, cR, cR	var	Intermediate variables (polynomials with big coefficients)
cb	var	Decrypted random data (binary string)
cm	out	Decrypted message (binary string)
cOctL	out	Length of decrypted message (single byte)
cp0	var	Decrypted padding (to be verified)
F	in/var	Polynomial with small coefficients (can be used as an input representing uniquely private key f)
f=1+pF	in/var	Private key (can be replaced as an input by F)

by m', which is an intermediate variable, dependent on the binary message m, length of m (denoted by $octL$), random data b, public key h, and the Object identifier, OID, representing uniquely a given encryption scheme and parameter set. Additionally, r is not selected completely at random, but rather generated by a deterministic function, called the Blinding Polynomial Generation Method (BPGM), based on a standardized hash algorithm, with inputs in the form of OID, message m, random data b, and the first $pkLen$ bits of the public key h ($hTrunc$). B2T is a conversion of each group of three bits to two ternary coefficients, using the look-up table defined in the IEEE standard. In the SVES decryption scheme shown in Fig. 1, the decrypted value is denoted by ci, and must be still unmasked in order to recover the actual decrypted binary message cm. Three checks are performed on the decryption side. If any of these checks fails, the result of decryption is considered invalid. Check 1 is to verify whether ci, which should be identical with m' on the encryption side, has a sufficient number (at least $dm0$) of 1s, -1s, and 0s (where $dm0$ is a part of a given parameter set, and is given in Table III). Check 2 is to determine whether $cMbin$ on the decryption side, which should be the same as $Mbin$ on encryption side, has a proper format, i.e., its padding bytes (the last $maxMsgLenBytes - cOctL$ bytes) are all equal to zeros. Finally, Check 3 is the most comprehensive check, used to verify whether the value of cR' is equal to cR, where cR' is calculated using the same formulas as R during decryption, with the message m replaced by decrypted message cm, and the random input b replaced by the decrypted random data cb. The other parts of the input to BPGM, namely OID and $hTrunc$, remain the same as during encryption. T2B is an inverse of the B2T conversion function.

III. HARDWARE DESIGN

A. Assumptions

To ensure compatibility among implementations of the same algorithm by different designers, our NTRU implementation is designed based on the hardware API proposed in [3]. Encryption and decryption share the same circuit. Key generation is assumed to be performed externally, e.g., in software. The public key and private key are loaded in advance, before the first encryption/decryption. They are stored internally and can be used for processing of multiple messages/ciphertexts.

The primary optimization target is the minimum latency (in absolute time units) for encryption and decryption. However, in case any design choices can lead to the same or only marginally greater latency, with the circuit area decreased substantially, these design choices are pursued as well in order to keep the cost and energy consumption of the circuit as low as possible. Since our implementation is intended primarily for high-end servers supporting a very large number of TLS, IPSec, and other secure protocol transactions per second, no attempt was made to introduce any countermeasures against side channel attacks other than timing attacks.

The implementation supports two parameter sets, specified in [2], denoted as ees1087ep1 and ees1499ep1, optimized for

speed, with security levels of 192 and 256 bits, respectively. SHA-256 is used as a basis for the implementation of the Blinding Polynomial Generation Method (BPGM) and the Mask Generation Function (MGF) of SVES. The remaining major parameters of both sets are summarized in Table III. Both polynomial r (for encryption) and polynomial F (for decryption) are represented using indices of all their coefficients equal to 1 and -1.

B. Top-Level Block Diagram

The top-level hardware block diagram is shown in Fig. 2. The two major functional units, which determine the speed and area of the circuit are Poly_Mult and BPGM_MGF. The latter of these units is used to implement both BPGM and MGF, because of the similarity between both operations, their sequential non-overlapping functionality, and because of the reliance on a single hash function core, implementing SHA-256. The public key h is stored inside of Poly_Mult (for both encryption and decryption). Indices of non-zero coefficients of F, uniquely determining the private key f, are stored in the RAM following the Input Data Conversion Unit for Decryption (IDCU-D). Range Conversion and modulo p reduction are naturally combined together. The mod p (mod 3) operation is optimized in such a way to use just 10 LUTs per 11-bit coefficient. Poly Add (+), Poly Sub (-), Range_Conv & mod_p, T2B, Check 1 and Check 3 are all performed on

TABLE III: Parameters of the algorithm, architecture, and input affecting the execution time, for two parameter sets ees1499ep1 and ees1087ep1

Parameter Set		ees1499 ep1	ees1087 ep1
Name	Description		
PARAMETERS OF ALGORITHM BASIC			
N	Dimension (rank) of the polynomial ring	1499	1087
dr	No. of 1s and no. of -1s in r	79	63
df	No. of 1s and no. of -1s in F	79	63
db	No. of random bits of b	256	192
$dm0$	The minimum number of 0s, 1s and -1s in m and ci, used in Check 1	79	63
$maxMsg$ $LenBytes$	Maximum message length in bytes	247	178
$pkLen$	No. of bits of h to include in sData	256	192
q	"Big" modulus	2048	2048
p	"Small" modulus	3	3
c	Polynomial index generation constant	13	13
$hiLen$	Hash function input block size in bits	512	512
$hoLen$	Hash function output block size in bits	256	256
PARAMETERS OF ALGORITHM DERIVED			
$a=log_2q$	No. of bits used to represent "big" coef.	11	11
$b=log_2N$	No. of bits used to represent an index of a polynomial coefficient	11	11
$cthr$	Index generation threshold $= 2^c - (2^c \, mod \, N)$, used by BPGM	7495	7609
$cval$	Probability that a randomly generated c-bit unsigned integer is smaller than cthr	0.9149	0.9288
$bthr$	Threshold $= 3^5$, used by MGF	243	243
$bval$	Probability that a randomly generated 8-bit unsigned integer is smaller than bthr	0.9492	0.9492
PARAMETERS OF ARCHITECTURE			
$cphi$	Clock cycles per hash input block	65	65
w	Width of the PDI and DO data buses	64	64
sw	Width of the SDI data bus	16	16
rw	Width of the RDI data bus	32	32
PARAMETERS OF INPUT			
l	Message length in bytes	variable	variable

Fig. 2: Top-level block diagram of the developed hardware architecture of NTRUEncrypt SVES.

only $2w = 64$ (rather than N) coefficients at a time. 64 small coefficients amount to 128 bits, and 64 big coefficients to 704 bits. Since the aforementioned operations do not limit the latency of either Encryption or Decryption (as long as performed at least with the speed of unloading final results), the narrower datapaths of these units help to minimize the area and energy consumption of the circuit without affecting performance.

Before encrypting a message directed to a given user, this user's public key must be loaded to the Poly_Mult unit, using the pdi_data bus and the Input Data Conversion Unit for Encryption & Decryption (IDCU-ED). This unit is required to handle the control signals of the PDI bus and to perform the bus width conversion (from w to $\alpha \cdot \lfloor w/\alpha \rfloor$). Similarly, before decrypting the first message from a given user, the receiver's public key must be loaded to the circuit using the same approach. Additionally, the receivers private key value, F, must be loaded to the circuit using the sdi_data bus, and stored in the internal RAM. Since F is a polynomial with small coefficients 1, -1, and 0, only the locations of 1s and -1s must be loaded. Each of these locations is a number in the range, 0..N-1, and thus is represented using $\beta = ceil(log_2 N)$ bits. Each location is loaded in a separate clock cycle.

During encryption, the db bits of random data b are first loaded to the RAM with the input rdi_data. After being fed with b, BPGM works as a pseudorandom number generator, producing all locations i of non-zero small coefficients of the random polynomial r. Each of these locations is consumed by Poly_Mult in one clock cycle. Only after Poly_Mult processes all elements of r, the output $R = r * h$ becomes available. This output is then reduced mod 4, and the obtained values provided to the input of MGF. The MGF unit produces the mask, in the form of a polynomial with small random coefficients. This polynomial is then added to the polynomial $mTrin$ obtained by converting the extended message input

$Mbin = (b, octL, m, p0)$, using the binary to ternary conversion unit, B2T (see Fig. 1a). Finally, the obtained new message representation, m', is added to the previously generated output from Poly_Mul, R, producing the ciphertext e. The ciphertext is then released to the output do_data, after conversion to words of the width w, using the Output Data Conversion Unit for Encryption (ODCU-E).

The decryption starts from the polynomial multiplication of the private key $f = 1 + pF$ by the ciphertext e. The obtained value fe then undergoes range conversion and reduction mod p. The obtained value ci should be the same as the message representation during encryption, m'. ci undergoes Check 1 for the minimum number of 1s, -1s, and 0s. Additionally, ci is used in the calculation of $cR = ce - ci$, which should be identical to R, calculated on the encryption side. cR is then reduced mod 4 and used as an input to MGF to produce the mask. The mask is subtracted from ci to generate $cMtrin$. After $cMtrin$ is converted to $cMbin$, using the ternary to binary conversion T2B, the Output Data Conversion Unit for Decryption (ODCU-D & Check_2) checks whether the decrypted data has a correct format, including $p0Len = maxMsgLenBytes - cOctL$ bytes of zero padding. If Check 2 passes, the extended decrypted data is decoded to identify values of cb and cm, which should be the same as b and m during encryption. These values are then used as inputs to BPGM_MGF. The BPGM unit then produces the locations i of all non-zero coefficients of the random polynomial $cr = r$, which should be the same as those on the encryption side. These values are then used, together with the public key h, stored inside of Poly Mult, to calculate $cR' = cr * h$. Since for the correctly decrypted message, $cr = r$, then cR' should be equal to cR obtained earlier during the decryption process. Comparing these two values constitutes the final check (Check 3) for the correctness of decryption. Only after this test passes, the decrypted message cm is released through the output

do_data, followed by the status block with the Status field equal to Success. If any of the three decryption checks fails, all remaining calculations are preempted and only the status block with the Status field equal to Failure is released to the output do_data.

C. Major Lower-Level Components

Internal block diagrams of the Poly_Mult, BPGM_MGF, BWC_1, and BWC_2 units are shown in Figs. 3, 4, 6 and 7, respectively. The algorithm used by Poly Mult is given as Algorithm 1. A similar Fast Convolution Algorithm was reported earlier in [6]. The for loop in lines 9-11 corresponds to adding to the temporary polynomial $c = c_{N-1}..c_0$ the input polynomial $a = a_{N-1}..a_0$ rotated by b_i locations to the left, namely, $a <<< b_i = a_{N-b_i-1}..a_0 a_{N-1}..a_{N-b_i}$. Similarly, the for loop in lines 13-15 corresponds to the subtraction of the same value from c. In the block diagram shown in Fig. 3, the subtraction is accomplished by adding a complemented value of the rotated input and carry in equal to one. The choice between addition and subtraction is provided by the controller, using the signal $c0$.

Algorithm 1 Polynomial Multiplication, Poly Mult

1: **Inputs:**
2: Polynomial a(X) with N big coefficients in the range $[0, q-1]$.
3: Polynomial b(x) with d coefficients 1 at the locations $b_0, b_1, ..., b_{d-1}$ and d coefficients -1 at the locations $b_d, b_{d+1}, ..., b_{2d-1}$, where $0 \leq b_i \leq N-1$.
4: **Output:**
5: $c(X) = a(X) * b(X) \bmod X^N - 1$
6: **Pseudocode:**
7: **for** $i := 0 \ to \ 2d-1$ **do**
8:     **if** $i < d$ **then**
9:         **for** $j := 0 \ to \ N-1$ **do**
10:             $c_j = c_j + a_{j-b_i \bmod N}$
11:         **end for**
12:     **else**
13:         **for** $j := 0 \ to \ N-1$ **do**
14:             $c_j = c_j - a_{j-b_i \bmod N}$
15:         **end for**
16:     **end if**
17: **end for**

During encryption, only one polynomial multiplication $R = r * h$ is performed, and thus, the public key h can be stored directly in the top SIPO w/PI (the Serial Input Parallel Output unit with Parallel Input). During decryption, two multiplications are performed, $f * e$ and $cR = cr * h$. As a result, during the first multiplication, h is pushed to the neighboring PISO w/PO (the Parallel Input Serial Output unit with Parallel Output), and then brought back to SIPO w/PI for the second polynomial multiplication. In the period between the two multiplications, PISO w/PO (holding the ciphertext e), feeding the serial output ce, is used for the calculation of cR (see Figs. 1 and 2).

The BPGM_MGF unit is shown in Fig. 4. It is based on the slightly modified implementation of SHA-256 [15], extended with the capability to store and retrieve the chaining value, which substantially speeds up the repeated computations of
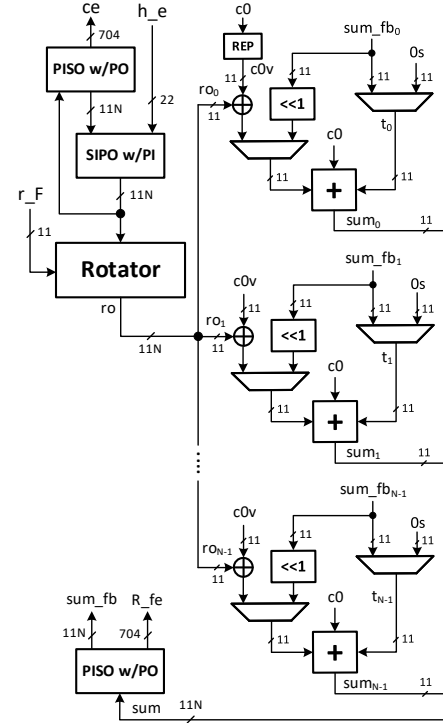

Fig. 3: Architecture of the polynomial multiplier. REP represents a component replicating input c0 11 times.

hash(sData||Ci) for multiple values of the counter Ci and sData composed of multiple input blocks of SHA-256. The speed-up method is explained in Fig. 5. The composition of sData, used during encryption, is shown in Fig. 1. The maximum size of sData, corresponding to the parameter set ees1499ep1 and the maximum message length listed in Table III, is 2512 bits. After adding 32 bits of counter Ci, we have up to 2544 bits of input to SHA-2. This input can be divided into t full 512-bit input blocks of SHA-2 containing only bits of sData (t=4 for the maximum message length), and one or two additional blocks containing the remaining bits of sData (if any) and the 32 bits of the counter Ci. Since the t first blocks of the input to SHA-2 remain unchanged in the subsequent calculations with different values of Ci, only the calculations for the blocks shown in bold in Fig. 5 must be performed. This way a substantial amount of calculations is saved, by simply storing the contents of the internal chaining variable of SHA-2 after processing of t blocks of sData during the calculation of h(sData||C1).

Our implementation of SHA-256 is a basic iterative architecture with 65 clock cycles per block. During the BPGM calculations, pdi_r and b are used in case of encryption and cm and cb will be used for decryption process. During the MGF calculations, the inputs R4 and cR4 are used, for encryption and decryption, respectively.

The BPGM data processing is performed by the Bus Width Converter 1 (BWC_1), shown in Fig. 6. For the BPGM calculations, the output of the Modified_SHA-2 is divided into c-bit blocks (with c=13 for both implemented parameter
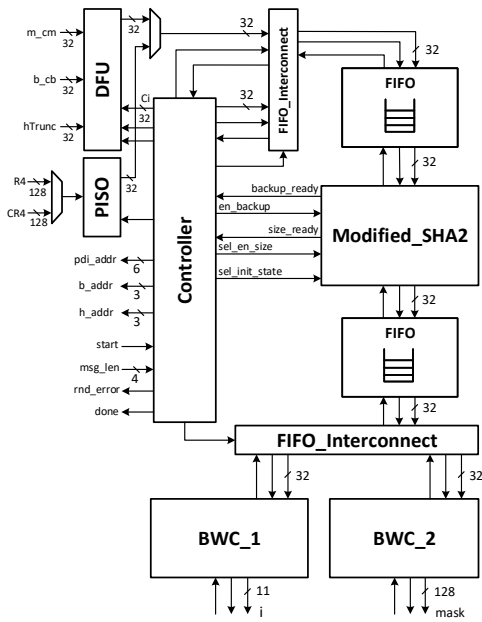
Fig. 4: Hardware architecture of the combined unit, BPGM/MGF, implementing the Blinding Polynomial Generation Method and Mask Generation Function.

Fig. 6: Hardware architecture of BWC_1.

sets). Each block is treated as an unsigned integer. If the value of this integer is greater than or equal to the index generation threshold $cthr = 2^c - (2^c \bmod N)$, then the block is discarded. Otherwise, the corresponding output i is calculated by taking the unsigned integer value of the block mod N. In our implementation, the mod N operation is performed using a 2^c x β look-up table.

In order to achieve a constant-time implementation, the indices i are released in batches, after processing each subsequent output of the Modified_SHA-2. In order to make it possible, we estimate the probability of the minimum number of indices generated up to each release point (and thus available in the output FIFO), and set the size of each batch to the value for which the probability of success exceeds

Case 1: t+n-1 input blocks

h(sData||C1)=h(sData_0, ..., sData_t-1, sData_t||C1)
h(sData||C2)=h(sData_0, ..., sData_t-1, sData_t||C2)
h(sData||C3)=h(sData_0, ..., sData_t-1, sData_t||C3)
......
h(sData||Cn)=h(sData_0, ..., sData_t-1, sData_t||Cn)

Case 2: t+2(n-1) input blocks

h(sData||C1)=h(sData_0, ..., sData_t-1, sData_t||C1_0, C1_1)
h(sData||C2)=h(sData_0, ..., sData_t-1, sData_t||C2_0, C2_1)
h(sData||C3)=h(sData_0, ..., sData_t-1, sData_t||C3_0, C3_1)
......
h(sData||Cn)=h(sData_0, ..., sData_t-1, sData_t||Cn_0, Cn_1)

Fig. 5: Computations performed by the ModifiedSHA-2, assuming two different cases regarding the size of the last block of sData (sData_t).
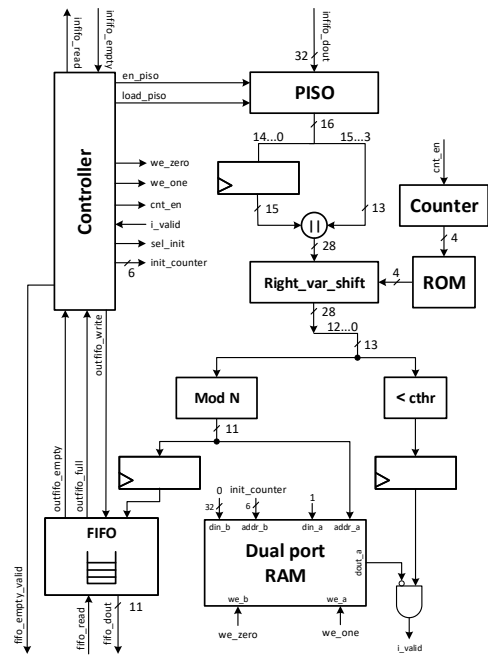
0.995. Our estimations are summarized in Table IV. We make an exception for the last batch, for which the probability of success is 0.9929, in order to avoid generating the entire additional SHA-2 output. Similarly, we estimate the probability that the number of indices in the output FIFO was sufficient after processing each subsequent SHA-2 output, from the first to tenth. The overall probability of success is estimated to be equal to 0.9794, which means that only in 2.06% of cases, the entire encryption operation will need to be repeated with the different value of seed b.

The MGF task is handled by the Bus Width Converter 2 (BWC_2), shown in Fig. 7. During the MGF calculations, the output of the Modified_SHA-2 is shifted out using 8-bit blocks. The value of each block is first compared to the threshold, $bthr = 3^5 = 243$. If the value is greater than or equal to 243, the block is discarded. Otherwise, the block is converted to the 5-digit ternary representation, with each digit encoded separately using two bits, for the total of 10 bits. The stream of 10-bit blocks is then converted into a stream of 128-bit blocks to form the output mask.

Similarly as for BWC_1, we achieve a constant time implementation, by making certain assumptions about the number of 10-bit outputs from O2T_ROM available in the following output FIFO after processing 3, 5, 7, 9, and 11 SHA-2 outputs, respectively. As shown in Table V, except for the last case of processing the 11th SHA-2 output, all previous assumptions are true with the probability of success indistinguishable from 1. The probability of meeting the last assumption is 0.9994, leading to the probability of having to repeat the encryption with a different value of seed b less than 0.06%. The overall execution time penalty for making the implementation constant-time has been estimated to be below

4% for both supported parameter sets. This penalty applies only to encryption.

TABLE IV: The assumed minimum number of indices i in the FIFO of BWC_1 after processing each subsequent SHA-256 output block for the ees1499ep1 parameter set.

#SHA-256 outputs	#13-bit output chunks	Assumed minimum # of indices i generated	Probability of success
1	19	14	0.9952
2	39	30	0.9974
3	59	47	0.9955
4	78	62	0.9976
5	98	79	0.9958
6	118	94	0.9984
7	137	110	0.9971
8	157	126	0.9969
9	177	142	0.9964
10	196	158	0.9929
BPGM successful for ees1499ep1 (N=1499, 2df=158)			0.9794

TABLE V: The assumed minimum number of 10-bit partial results in the FIFO of BWC_2 after processing each subsequent SHA-256 output block.

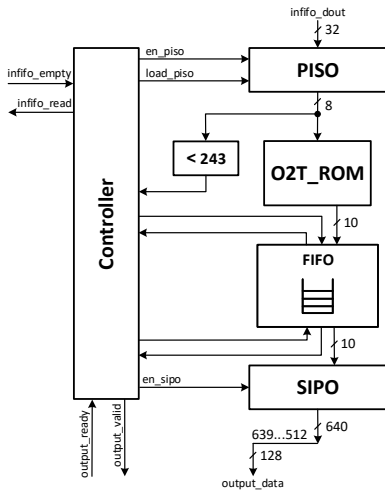| #SHA-256 outputs	#8-bit output chunks	Assumed minimum # of 10-bit chunks	Probability of success
3	96	64	1
5	160	128	1
7	224	192	1
9	288	256=1280 coefficients	1
MGF successful for ees1087ep1 with N=1087			1
11	352	320=1600 coefficients	0.9994
MGF successful for ees1499ep1 with N=1499			0.9994

Fig. 7: Hardware architecture of BWC_2.

IV. RESULTS

Our design has been described in VHDL at the Register Transfer Level (RTL). The target device has been selected as Xilinx Virtex UltraScale XCVU440-FLGA2892-3-e. All presented results are generated using Minerva hardware optimization tool [16]. In Table VI, we summarize the resource utilization (in LUTs and Slices), maximum clock frequency,

and latencies of several major building blocks. PolyMult was optimized for the minimum latency in absolute time units by introducing five pipeline stages. This unit is shown to be the most restrictive in terms of clock frequency (297 MHz) and taking a vast majority of the circuit resources (167,469 LUTs), which is over 90% of the total area. It should be stressed that for operations such as Poly Add and Poly Sub, latency represents the number of clock cycles necessary to obtain an output coefficient corresponding to the input coefficients with the same index, and not the time necessary to process all coefficients of the polynomial.

Timing analysis of our hardware implementation is shown in Table VII. Latencies in clock cycles correspond to the maximum sizes of messages allowed by a given parameter set. The hardware implementation is seriously limited by the sequential nature of the SHA-256 calculations. As a result, the operation of Poly_Mult can be almost completely overlapped with the computations of BPGM through the use of pipelining. On the other hand, in the reference software implementation, Poly Mult amounts to about 91% of the total execution time. The operations that are most critical in hardware are hash based operations of BPGM and MGF, amounting to over 99% of the encryption time and over 84% of the decryption time. For the ees1499ep1 parameter set, the total execution time of encryption is 12.60 μs and decryption 14.09 μs.

For the Intel Xeon CPU E5-2667 v3 @ 3.20GHz, 128 GB RAM, with the source code from [5] running on a single core, the encryption time was measured to be 674 μs for the reference software implementation and 64 μs for the optimized software implementations. On the same platform, the decryption time was 1264 μs for the reference and 83 μs for the optimized implementation, respectively. The optimization was based on the use of Advanced Vector Extensions 2 (AVX2) to the amd64 instruction set architecture. Both versions used the same compiler options. With 12.60 μs used for encryption by our hardware implementation, we observe the speed-up by a factor of 53.5 vs. the reference software implementation and a factor of 5.1 vs. the optimized implementation. If the number of encryptions per second, rather than latency is a primary performance metric, modern microprocessor can compensate for this speed-up with multiple cores running in parallel. Still, FPGA running at much lower frequency (297.0 MHz vs. Xeon's 3.2 GHz) will significantly outperform Xeon cores in terms of lower power consumption and lower energy usage.

In Table VIII, we compare the results of our hardware implementation of NTRUEncrypt SVES with the results of the high-speed implementation of a leading PQC candidate, Classic McEliece [17], at the same claimed security level, generated using open-source code available at [18]. Both implementations target the same FPGA device, representing Xilinx Virtex-UltraScale family. NTRUEncrypt SVES outperforms Classic McEliece in terms of the decryption speed by a factor of 2.9, while achieving approximately the same encryption speed. At the same time the high-speed implementation of NTRU requires 4.3 times more Slices, giving about 48% higher product of the Decryption Time · #Slices.

TABLE VI: Resource utilization and performance metrics of major component units generated by Minerva. Latencies correspond to the ees1499ep1 parameter set.

Operation	LUTs : Slices	Freq. [MHz]	Latency [Cycles]	Latency. LUTs
Poly Mult	167,469 : 32,310	297.0	162	27,297,447
BPGM	2,848 : 622	338.0	1,729	4,924,192
MGF			1,999	5,693,152
B2T	64 : 34	904.0	1	64
T2B	64 : 35	984.3	1	64
Poly Adds	1338 : 272	316.3	1	1338
Poly Sub	74 : 64	540.2	1	74
Poly Subc	1221 : 258	331.2	1	1221

TABLE VII: Timing analysis of our hardware implementation. Latencies in clock cycles correspond to the maximum sizes of messages allowed by a given parameter set. Notation: Calc.: Calculating, Perf.: Performing, Unload.: Unloading.

Operation	Latency [Cycles]	% of Total Time	Latency [Cycles]	% of Total Time
	ees1499ep1		ees1087ep1	
ENCRYPTION				
Perf. BPGM & calc. R	1,732	46.3%	861	44.5%
Calc. R4 and perf. MGF	1,999	53.4%	1,063	54.9%
Calc. m' & Check 1	1	0.0%	1	0.1%
Unload. ciphertext e	11	0.3%	11	0.6%
Total	**3,743**	100%	**1,936**	100%
DECRYPTION				
Loading ciphertext e	258	6.2%	187	8.2%
Calculating f*e	164	3.9%	132	5.8%
Check 1 & calc. cR4	1	0.1%	1	0.0%
Perf. MGF	1,999	47.8%	1,063	46.9%
Calc. cMbin & Check 2	1	0.1%	1	0.0%
Perf. BPGM & calc. cR'	1,732	41.4%	861	38.0%
Check 3 & Unload. cm	31	0.7%	23	1.0%
Total	**4,186**	100%	**2,268**	100%

TABLE VIII: Comparison of the NTRUEncrypt results for the ees1499ep1 parameter set with the results for Classic McEliece (m=13, t=119, N=6960, optimization for speed), supporting PQC public-key encryption at the same security level of 256-bits, targeting Xilinx Virtex-UltraScale.

PQC Scheme	#Slices	#BRAMs	Freq [MHz]	Enc [Cycles]	Dec [Cycles]	Enc [us]	Dec [us]
McEliece	8,236	35	426	5,413	17,055	12.70	40.04
NTRU	35,435	2	297	3,743	4,186	12.60	14.09
Ratio	4.30	0.06	0.70	0.69	0.25	0.99	0.35

V. Conclusions

We report the first high-speed constant-time hardware implementation of the full encryption scheme of the IEEE P1363.1 standard, NTRUEncrypt SVES. Our implementation is fully compliant with the proposed universal PQC Hardware API [3], and is made open-source to speed-up further design-space exploration and benchmarking on multiple hardware platforms [19]. Our results demonstrate the need to revisit the algorithmic construction of the NTRUEncrypt SVES in order to make this algorithm more parallelizable and more suitable for high-speed hardware implementations.

Our future work will involve taking advantage of any additional optimizations at the algorithmic and hardware architecture levels, as well as targeting minimum energy use. We will also develop a full constant-time hardware implementation of the NIST PQC Candidate NTRUEncrypt, which specification, published in Nov. 2017, contains substantial changes compared to the IEEE standard.

References

[1] J. Hoffstein, J. Pipher, and J. H. Silverman, "NTRU: A ring-based public key cryptosystem," in *Algorithmic Number Theory*. Springer, 1998, pp. 267–288.

[2] "IEEE Standard Specification for Public Key Cryptographic Techniques Based on Hard Problems over Lattices, P1363.1-2008," March 2009.

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

[4] D. J. Bernstein and T. Lange. eBACS: ECRYPT Benchmarking of Cryptographic Systems. Accessed August 1, 2017. [Online]. Available: http://bench.cr.yp.to

[5] Security Innovation, Inc. Open Source NTRU Public Key Cryptography Algorithm and Reference Code. [Online]. Available: https://github.com/NTRUOpenSourceProject/ntru-crypto

[6] D. V. Bailey, D. Coffin, A. Elbirt, J. H. Silverman, and A. D. Woodbury, "NTRU in Constrained Devices," in *Cryptographic Hardware and Embedded Systems — CHES*. Springer, 2001, pp. 262–272.

[7] C. Rourke, "Efficient NTRU Implementations," Master's thesis, Worcester Polytechnic Institute, 2002.

[8] J.-P. Kaps, "Cryptography for Ultra-Low Power Devices," Ph.D. dissertation, Worcester Polytechnic Institute, 2006.

[9] A. C. Atici, L. Batina, J. Fan, I. Verbauwhede, and S. B. O. Yalcin, "Low-cost implementations of NTRU for pervasive security," in *2008 International Conference on Application-Specific Systems, Architectures and Processors*, July 2008, pp. 79–84.

[10] A. A. Kamal and A. M. Youssef, "An FPGA implementation of the NTRUEncrypt cryptosystem," in *2009 International Conference on Microelectronics - ICM*, Dec 2009, pp. 209–212.

[11] F. Hu, K. Wilhelm, M. Schab, M. Lukowiak, S. Radziszowski, and X. Yang, "NTRU-based sensor network security: a low-power hardware implementation perspective," *Security and Communication Networks*, vol. 2, no. 1, pp. 71–81.

[12] F. Hu, Q. Hao, M. Lukowiak, Q. Sun, K. Wilhelm, S. Radziszowski, and Y. Wu, "Trustworthy Data Collection From Implantable Medical Devices Via High-Speed Security Implementation Based on IEEE 1363," *IEEE Transactions on Information Technology in Biomedicine*, vol. 14, no. 6, pp. 1397–1404, Nov 2010.

[13] B. Liu and H. Wu, "Efficient architecture and implementation for NTRU-Encrypt system," in *2015 IEEE 58th International Midwest Symposium on Circuits and Systems (MWSCAS)*, Aug 2015, pp. 1–4.

[14] ——, "Efficient multiplication architecture over truncated polynomial ring for NTRUencrypt system," in *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2016, pp. 1174–1177.

[15] CERG @ George Mason University. (2012, March) Source Code for the SHA-3 Round 3 Candidates & SHA-2 - The Third SHA-3 Candidate Conference Release. [Online]. Available: https://cryptography.gmu.edu/athena/index.php?id=source_codes

[16] F. Farahmand, A. Ferozpuri, W. Diehl, and K. Gaj, "Minerva: Automated Hardware Optimization Tool," in *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, Dec 2017, pp. 1–8.

[17] W. Wang, J. Szefer, and R. Niederhagen, "FPGA-based Niederreiter Cryptosystem using Binary Goppa Codes," in *International Conference on Post-Quantum Cryptography — PQCrypto*, April 2018.

[18] ——. (2018, April) FPGA-based Niederreiter Cryptosystem using Binary Goppa Codes. [Online]. Available: http://caslab.csl.yale.edu/code/niederreiter/

[19] F. Farahmad, M. U. Sharif, and K. Briggs. (2018, Dec) NTRUEncrypt SVES Source Code. [Online]. Available: https://cryptography.gmu.edu/athena/index.php?id=PQC