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

Kevin Briggs

***Abstract*— The NTRUEncrypt SVES algorithm, published as IEEE Standard P1363.1 in 2008, codifies a standardized form for implementing the NTRUEncrypt lattice-based cryptographic scheme that was first introduced in 1996. Recently, a high speed constant time implementation was developed in [FSBG18]. This report covers details relating to the constant time nature of the implementation and to the data conversion units used in the design.**

I. Introduction

The advent of quantum computers has brought about a new fear in the data security and privacy world. These computing devices are proven to be capable of defeating a multitude of cryptographic schemes that protect the vast majority of internet traffic today, in internet protocols such as SSH, IPSec, and TLS. In these widely used protocols, traditional public key cryptography such as RSA is commonly employed. RSA and other commonly used asymmetric cryptography techniques are reliant on the inability of classical computers to solve various mathematical problems efficiently. Quantum computers however break these schemes efficiently, and therefore the digital arms race has begun to find new schemes to protect our data moving into the future.

Cryptographic schemes under investigation for the next generation of data security fall under the category coined Post-Quantum Cryptography (PQC), and several families of these schemes exist. One such family is the lattice-based family, wherein one of the oldest and perhaps most promising schemes resides - NTRUEncrypt.

II. NTRUEncrypt

NTRUEncrypt is a scheme first proposed by J. Hoffstein, J. Pipher, and J. Silverman at Crypto '96 and later appearing as [HPS98] that is based on polynomial multiplications over an integer ring, such as:

$$R = Z[X]/(X^N - 1) \tag{1}$$

The NTRUEncrypt scheme is afforded security through a well known hard problem over lattices known as the shortest vector problem. The shortest vector problem is the computational challenge of finding the shortest vector in a lattice that is defined by 2 arbitrary base vectors, which has as of yet always been a generally difficult problem for classical computers when dealing with lattices of high dimension TODO put a ref here.

NTRUEncrypt is notably different from popular classical schemes in several regards. For one, NTRUEncrypt is an example of what is known as a probabilistic encryption scheme. This term refers to a category of schemes wherein probabilities, i.e. a random element, is involved in the actual encryption and decryption process and not just key generation. In the case of the original NTRUEncrypt, this random element is known in the original literature as ϕ, or in the IEEE 1363.1 standard as *r*, and additionally in the SVES standard a plaintext mask value is generated randomly.

The probabilistic nature of NTRUEncrypt presents an interesting challenge in terms of making a constant-time implementation, which is explored in further detail below.

III. IEEE 1363.1 - NTRUEncrypt SVES

The first published standard involving the NTRUEncrypt algorithm was released as IEEE Standard 1363.1. This standard defines choices for various parameter sizes and sequences to use when implementing the NTRUEncrypt SVES algorithm. It is this standard that the implementation in [FSBG18] conforms to[IEE09]. Several components in [FSBG18] are the subject of review in this report, namely a brief overview of how the design fulfills a constant-time requirement, and how several of the input and output data conversion units are used to adhere to the GMU CERG API[FFD$^+$].

A. Parameters

The major parameters in NTRUEncrypt SVES are shown in figure 1. 2 parameter sets are shown, which are referred to as *ees1499ep1* and ees1087ep1. These refer to the value of the N parameter, and are 192 and 256 bit equivalent security levels, respectively.

B. Encryption

The encryption operation is defined as:

$$e = r * h + m(modq) \tag{2}$$

The *r* term is the randomly selected element mentioned earlier, and is known as the blinding polynomial, while *h* is the user's public key, and *m* is the user's message.

C. Decryption

The decryption operation is defined by several steps, which are:

$$f * e(modq) \tag{3}$$

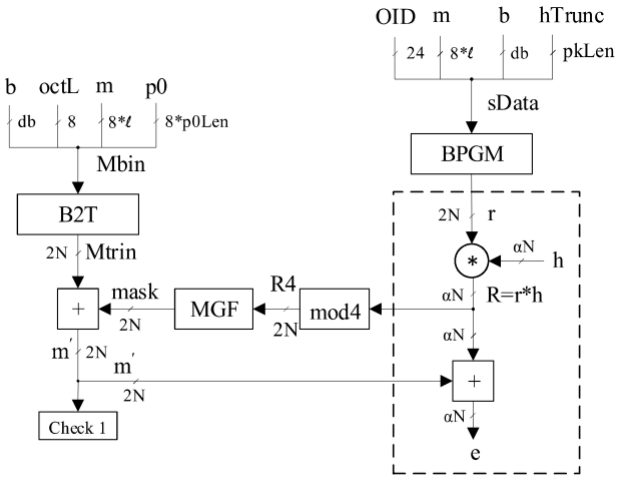
shift to range

$$[-q/2, q/2)$$

coefficient reduction mod p

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=$log_2 q$	No. of bits used to represent "big" coef.	11	11
b=$log_2 N$	No. of bits used to represent an index of a polynomial coefficient	11	11
cthr	Index generation threshold = $2^c - (2^c \bmod N)$, 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. 1. NTRUEncrypt SVES Parameters



(a) Encryption

Fig. 2. Encryption Data Flow

IV. Constant-Time Implementation

As mentioned in section 2, the NTRUEncrypt algorithm is probabilistic, meaning an element of randomness is present during the encryption process, and not just key generation as is the case with some well known classical public key schemes.

In NTRUEncrypt SVES, this probabilistic element is found in 2 components - the BPGM, or Blinding Polynomial Generation Method, and the MGF, or Mask Generation Function. In the specification, both of these components effectively fill the role of what is known as an IGF, or Index Generation Function, which is a function that generates integer indices of random polynomials. In the case of BPGM, the randomly generated polynomial is r. In the case of MGF, the randomly generated polynomial is a plaintext masking value. The specification states 2 possibilities for implementing an IGF, which is through either a random bit generator, or through a deterministic hash function, either SHA-1 or SHA-256 [IEE09].

In [FSBG18], the selected underlying IGF implementation is SHA-256, which is realized for both the BPGM and MGF. Because of the common use of SHA-256, both of these components are combined into the same functional block in the design, shown in figure 3.

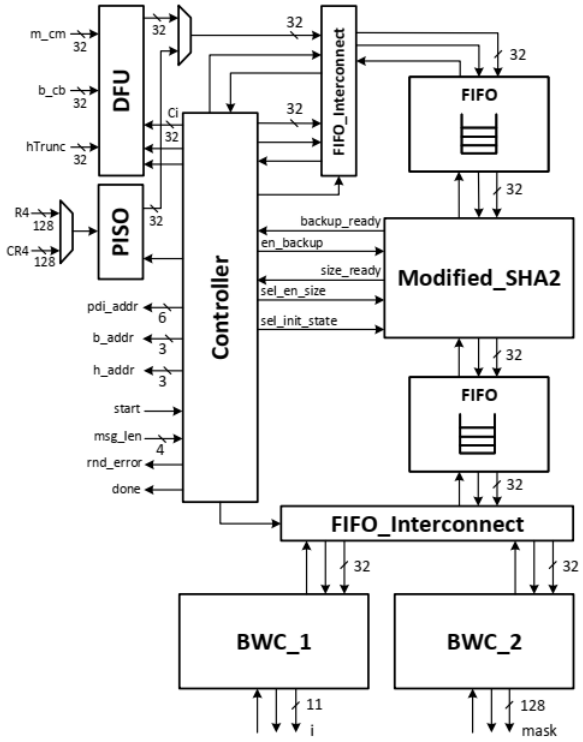


Fig. 3. BPGM/MGF internal diagram

A. BPGM

As was shown in equation 2, the encryption process involves multiplying the *h* polynomial, or public key, with the *r* polynomial, which is the *blinding polynomial*, which must be generated randomly. The *r* polynomial is a sparse polynomial, having specifically only *dr* indices with coefficients equal to 1, and another *dr* indices with coefficients set to -1, where *dr* is defined by the parameter set selection. The BPGM portion of the combined BPGM/MGF unit is responsible for creating this sparse polynomial.

The BPGM functionality does this by first starting with a buffer representing the polynomial where all coefficients are initially set to 0. The internal SHA-256 block is executed, producing 256-bit output blocks pseudo-randomly from a seed value. This 256-bit output block is segmented into 8 32-bit blocks and stored in an intermediate FIFO for

consumption by a BWC, or bus width conversion unit. The BWC1 unit is utilized by BPGM, while the BWC2 unit is utilized by the MGF functionality.

1) BWC1: The BWC1 (Bus Width Converter 1), shown in figure 4, is a special component, because it must be able to operate in real-time as the SHA-256 output blocks become available, and it must not skip over any data. There are 2 specific facets of this component which cause the total amount of time required to generate 2 * *dr* coefficients to be non-deterministic, which are 1) A threshold check value and 2) the fact that indices can only be selected once.

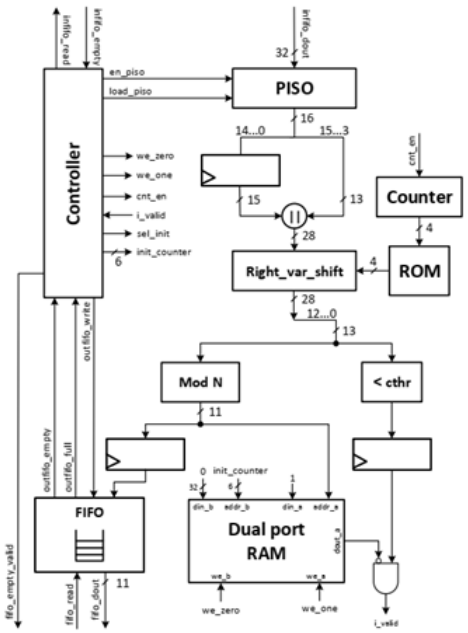


Fig. 4. BWC1 Block Diagram

The unit is a variable shifting design which takes as input 32-bit chunks of data from the SHA-256 output FIFO, and converts these into valid coefficient indices of the *r* polynomial. A 32-bit chunk is divisioned into 2 16-bit chunks. Each 16-bit chunk is treated as an unsigned integer, allowing possible value from 0 to 8191. As per the specification, each integer which is $\geq$ *cthr* must be discarded. For the N=1499 parameter set, *cthr* = 7495. Once a sample value has passed the threshold check, it is reduced mod N to be within range of the desired polynomial, and if not already set to 1 or -1, the value is accepted and stored in RAM.

2) Constant-Time Solution: Because of the aforementioned threshold check and the fact that coefficient indices cannot be generated more than once, and given that indices are generated pseudo-randomly, then there is no way to guarantee that a specified number of indices, in this case 2**df*, will be generated within a finite number of SHA-256 output blocks. Instead, a reasonable approach which can still be used to achieve a constant time implementation is to assume that a number of outputs will be invalid, and to plan accordingly by preparing to generate a fixed number of SHA-256 output blocks which is greater that strictly necessary. In essence, time and thus performance must be traded to achieve a constant-time implementation given the inherent probabilistic element.

3) Probability Verification: The remaining question after resorting to coefficient over-generation is a matter of how many additional blocks of SHA-256 need to be generated, and the answer affects the performance of the design. To resolve this, the effective probability of generating the required number of coefficients from a set number of SHA-256 output blocks was investigated. The primary investigative method used was simple Monte Carlo simulation through the use of a C program.

In addition to needing to determine the raw number of output blocks necessary, another concern with the implementation is associated with the amount of time that could potentially be wasted if a given run of the algorithm is turning out to generate a low number of coefficients, making the outlook for generating the final required amount very unlikely. For this reason, a number of checkpoints have been created where a proportionate amount of coefficients are needed in order to continue the process. This allows for a "look-ahead", to ascertain ahead of time whether the entire polynomial generation process needs to be restarted.

The probability results for each checkpoint and the final probability of success, generated using simulation with a C program, are included in figure 5. These results show that the likelihood of success is relatively high. It is important to note that in the event that a single checkpoint fails, or the final number of coefficients is not realized, the entire polynomial generation process must be restarted with an entirely new seed value. Thus the implementation achieves a constant operation time in the sense that upon a successful run where the required number of coefficients has been generated, the operation will have taken a known constant amount of time.

#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

Fig. 5. BWC1 Probability Table

B. MGF

The MGF component is similar to the BPGM. It uses the internal SHA-256 implementation in a similar manner to produce coefficients of a masking polynomial. The details are not explored here, however the concerns are the same as with the BPGM in that the implementation must accordingly generate a pre-defined number of output blocks in order to expect a correct number of coefficients to be generated. Just as with the BPGM component, the MGF checkpoint

probabilities and total success probability was determined through simulation. These values are shown in figur x.

#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. 6. BWC2 Probability Table

V. PQC Hardware API

As the race to produce the most secure and most efficient PQC algorithms takes off, the importance of having well defined APIs is paramount, as without them, inter-operability and therefore rate of adoption will be affected in a time when it may not be afforded. To this end, the implementation in [FSBG18] strives to be compliant with the recently proposed GMU CERG PQC Hardware API[FFD$^+$]. This API defines a number of minimum compliance criteria, including:

- Key generation details
- Padding and acceptable message sizes
- Timing characteristics
- External memory constraints
- Interface ports

The relevant encryption and decryption elements are shown in figures 7 and 8.

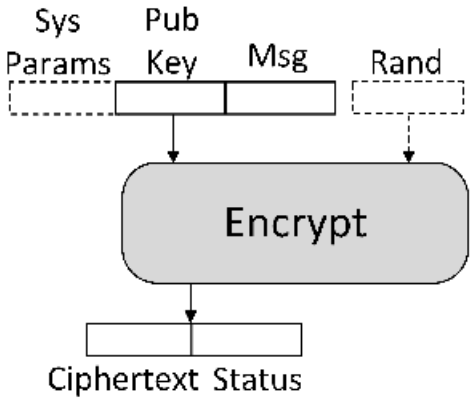


Fig. 7. PQC API - Encryption Data Elements

VI. Data Conversion Units

The data conversion units are responsible for conforming to the PQC API input and output specifications, and moving and reformatting different types of data as necessary to respective internal components in a format that is most easily used during the processes in which they are needed. There are 5 different data conversion units, of which 3 are input units and 2 are output units. Both a single input and output unit are dedicated to both encryption and decryption, while a 5th unit is used during both encryption and decryption.

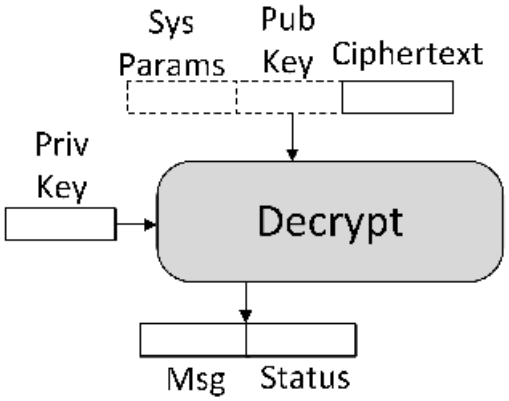


Fig. 8. PQC API - Decryption Data Elements

A. IDCU-E

The input data conversion unit - encryption. This component is responsible for leading the message, *m*, into an internal RAM block over the pdi_data bus. It also loads a random seed value, *b*, into RAM over the rdi_data bus.

This unit is a simple pass-through of the pdi_data bus to the message output signal to RAM, as well as a pass-through of rdi_data to the b output to RAM. It additionally includes an internal register for saving the *octL* value to present to the global controller.

B. IDCU-ED

The input data conversion unit - encryption/decryption. This component is responsible for loading the public key, *h*, into an internal SIPO component inside the polynomial multiplier, and another chunk of data, *hTrunc*, into RAM.

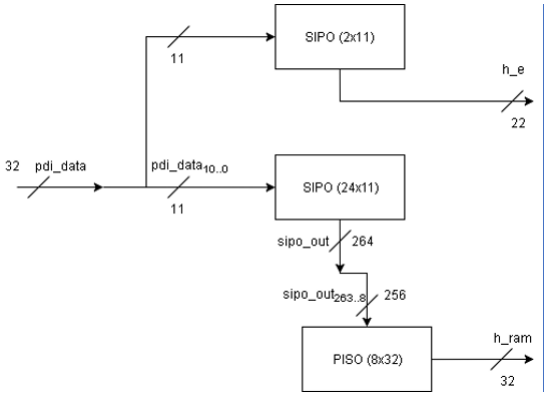


Fig. 9. IDCU-ED

This unit uses a simple SIPO to PISO interconnect to convert between incoming 11 bit coefficient values and the output 32-bit bus width. It also uses another SIPO for storing coefficient values to feed into the polynomial multiplier, 2 coefficients at a time.

C. IDCU-D

The input data conversion - decryption. This component is the simplest, and reads in indices of non-zero coefficients in the private key polynomial, *f*, over the sdi_data bus. Each coefficient comes in 11 bits at a time, thus its functionality is merely a truncation of the incoming bits.

D. ODCU-E

The output data conversion unit - encryption. This component is responsible for formatting and outputting the encrypted ciphertext over the pdo_data bus. Internally, it uses a PISO to convert from a 704 bit ciphertext chunk into 11 bit output pieces.

E. ODCU-D

The output data conversion unit - decryption. This component is responsible for formatting and outputting the decrypted message over the pdo_data bus. It uses a PISO internally to accept 96-bit chunks of the output message, *cm*, and conver them to 32-bit output chunks.

REFERENCES

[FFD+] Ahmed Ferozpuri, Farnoud Farahmand, Viet Dang, Malik Umar Sharif, Jens-Peter Kaps, and Kris Gaj. Hardware api for post-quantum public key cryptosystems. https://cryptography.gmu.edu/athena/PQC/PQC_HW_API.pdf.

[FSBG18] Farnoud Farahmand, Malik Umar Sharif, Kevin Briggs, and Kris Gaj. A high-speed constant-time hardware implementation of ntruencrypt sves. In *2018 International Conference on Field Programmable Technology (ICFPT)*, December 2018.

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. Ntru: A ring-based public key cryptosystem. In Joe P. Buhler, editor, *Algorithmic Number Theory*, pages 267–288, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[IEE09] IEEE. Ieee standard specification for public key cryptographic techniques based on hard problems over lattices. *IEEE Std 1363.1-2008*, pages C1–69, March 2009.