Performance Analysis of Number Theoretic Transform for Lattice-Based Cryptography

Ayman W. Mohsen
Computer & Systems Engineering Dept.
Ain Shams University
Cairo, Egypt
ayman.wagih@eng.asu.edu.eg

Mohamed A. Sobh
Computer & Systems Engineering Dept.
Ain Shams University
Cairo, Egypt
mohamed.sobh@eng.asu.edu.eg

Ayman M. Bahaa-Eldin
Misr International University
On leave from Ain Shams University
ayman.bahaa@eng.asu.edu.eg

***Abstract*—This study concentrates on the performance of number theoretic transform (NTT) which is a technique used to speed up polynomial multiplication in ring-LWE based cryptosystems. The NTT is the most time-consuming operation in such cryptosystems. Different implementations of the NTT were tested on SSE and AVX2 architectures.**

***Keywords*—Ring-LWE, number theoretic transform, polynomial multiplication, SSE, AVX2.**

I. Introduction

One of the most promising fields of Post-Quantum Cryptography is Lattice-based cryptography [1]. It has been a subject of study for many years and has produced Public Key Cryptosystems, and Key Encapsulation Mechanisms. The modern lattice-based cryptosystems started with the Learning with Errors (LWE) problem [2], which was developed in 2005. There are two types of lattice-based cryptosystems: matrix-based and polynomial ring-based [4]. The matrix-based cryptosystems are based on the original LWE problem and are older and less efficient. They have key sizes of $O(n^2)$. While the polynomial ring cryptosystems are based on the newer ring-LWE problem that uses cyclic lattices [3]. This enables storing and operating on polynomials of size O(n) instead of matrices of size $O(n^2)$. Still, the lattice-based cryptosystems have a disadvantage: ciphertext is larger in size than the plaintext by a factor of 30 [10]. This ratio is called the ciphertext blowup factor. There is a tradeoff between size efficiency and cryptographic security as seen in the NewHope KEM [6].

In this paper we concentrate on the efficiency of modern ring-LWE cryptosystems. Since such cryptosystems use polynomial rings, most operations are done on polynomials. Hence the most time-consuming operation in these cryptosystems is polynomial multiplication [7]. Polynomial multiplication can be done efficiently by taking the discrete Fourier transform (DFT) of the arguments and taking the inverse-DFT of the product. The DFT can be modified to work purely with modular arithmetic. This modified DFT is called the number theoretic transform (NTT). The time efficiency of the implementation of the NTT has the greatest effect on the time efficiency of the ring-LWE cryptosystems. The NTT can be optimized in similar fashion as the DFT to have a time complexity of O (n log n). In addition, most modern processors have SIMD instructions which can be used to further speed up the NTT.

A. Preliminaries

In modular arithmetic, the number q that is usually prime is called the modulus. The modular reduction operation calculates the remainder of division by q. The polynomial ring used here is defined as a set of all polynomials with integer coefficients modulo q. The polynomials in the set are themselves reduced modulo a certain chosen polynomial $\varphi(x)$, using long division. The polynomial ring is denoted as $Rq=Zq[x]/\varphi(x)$. Where Zq is the set of numbers modulo q, while Zq[x] is the set of polynomials in x with coefficients from Zq, and $Zq[x]/\varphi(x)$ is the set of polynomials from Zq[x] modulo the polynomial $\varphi(x)$. The polynomial rings in lattice-based cryptosystems have $\varphi(x)$ usually chosen as X^n+1 or X^n-1. The polynomial modular reduction with such polynomial $\varphi(x)$ (the long division by $\varphi(x)$) is a trivial operation. The first polynomial $\varphi(x) = X^n+1$ is more widely used [11], but it adds an additional step when performing multiplications using the number theoretic transform [12], [13], and [14].

B. Outline

Section II is about polynomial multiplication, the number theoretic transform, and cache efficiency. Section III is about various methods for fast modular reduction. Section IV is about the performance analysis of different implementations of the NTT algorithm.

II. Polynomial Multiplication and The Number Theoretic Transform

A. Polynomial multiplication

The most time-consuming operation in ring-LWE cryptosystems is polynomial multiplication [7]. The naive implementation has n^2 multiplications with polynomials of size n. While if we look at the polynomials as vectors of coefficients or as finite discrete signals, then the multiplication operation is equivalent to the convolution of these vectors of coefficients. This can be exploited by taking the Fourier transform of these vectors, so that the $O(n^2)$ convolution operation becomes a linear O(n) element-wise multiplication in frequency domain. And since polynomial rings use modular arithmetic, a modification of the Fourier transform is used that

works purely with modular integers. That is, it doesn't have complex numbers or floating points. This modified transform is called the Number Theoretic Transform (NTT). It is achieved by replacing the kernel of the Fourier transform with one from modular arithmetic. More specifically the N complex roots of unity are replaced with N numbers coming from the powers of a generator w modulo q. For example, if the polynomials are of size 4, then a suitable modulus for the coefficients is q=5, to produce an NTT with a generator w=2. The generator w produces a sequence of 4 powers of w mod q, which are {1, 2, 4, 3}. This sequence is produced by raising the generator w to power of the index and reducing the result modulo q. This sequence can be used to take the NTT of a size 4 vector of coefficients mod 5.

After taking the number theoretic transform of the two vectors of coefficients they are multiplied element-wise producing a third vector. By taking the inverse NTT of that, we get the product of the two original polynomials. The NTT can be implemented with n log n multiplications using the Cooley-Tukey FFT algorithm. Thus the new multiplication method consists of two NTT operations of O (n log n), an element-wise multiplication of O(n), and an inverse-NTT operation of O (n log n),. With a total number of multiplications of 3nlogn+n, the polynomial multiplication is now faster than the naive method with n^2 multiplications. The speedup is more significant with a large size n. For example, in the NewHope key encapsulation mechanism [6], the size is 1024, and the NTT multiplication is about 33 times faster than the naive method.

The previously described method for polynomial multiplication is valid for multiplications in the polynomial ring $Zq[X]/(X^n-1)$. A more widely used polynomial ring is $Zq[X]/(X^n+1)$. It is used in the majority of ring-LWE key encapsulation mechanisms such as the BCNS [5], NewHope [6], and Kyber [8] KEMs. This method for polynomial multiplication needs a slight modification to be used in the ring $Zq[X]/(X^n+1)$. In the polynomial ring with $\varphi(X)=X^n-1$, modular reduction is done by wrapping around the extra coefficients and adding them up, which is equivalent to the convolution operation. While in the ring with $\varphi(X)=X^n+1$, the coefficients are negated when wrapped around. This is equivalent to negative wrapped convolution. To compensate this, the coefficients are multiplied by the powers of the square root of the generator w before taking the NTT, and after the inverse-NTT.

In order to have an integer square root of the generator w, it has to be a square of an integer. Therefore, the modulus q and sqrt(w) are chosen to get an order of double the size of the polynomials. For example, if the polynomials are of size 4 and from the ring $Zq[X]/(X^n+1)$, then a suitable modulus is q=17, and a generator w=4. The NTT roots are {1, 4, 16, 13}. And when working with polynomials from the ring $Zq[x]/(X^n+1)$, the coefficients should be multiplied by $\{1, w^{1/2}, w, w^{3/2}\} = \{1, 2, 4, 8\}$ before taking the NTT, and by $\{1, -w^{3/2}, -w, -w^{1/2}\} = \{1, -8, -4, -2\}$ after taking the inverse-NTT. This is the modification that is needed when working with this polynomial ring.

B. The NTT Algorithm

Then Number Theoretic Transform (NTT) is a modification of the Discrete Fourier Transform (DFT) where the only difference is that the kernel $w=\exp(-2\pi/N)$ with the N complex roots was replaced with a generator w from modular arithmetic, and all operations are now in modular arithmetic. All other details are the same as in the DFT algorithm. In both transforms $w^N = 1$ and $w^{N/2} = -1$. The DFT operation is multiplication of the DFT matrix is by the input vector. The DFT matrix contains roots raised to power the product of time and frequency indexes. The naive method has n^2 multiplications.

The Cooley-Tukey Fast Fourier Transform (FFT) is a famous optimized implementation of the DFT algorithm with n log n multiplications [9]. It can be understood as the bit-reverse permuted n by n DFT matrix split into log n stage matrices. The product of these stage matrices gives the DFT matrix of weights in a bit-reverse permuted form. The first step of the FFT algorithm is applying a bit-reverse permutation on the input vector. Then, each stage matrix performs operations on pairs of coefficients of the bit-reverse permuted vector. The operations are grouped in identical blocks. The blocks of operations are doubling in size and halving in count with each new stage: The first stage has N/2 blocks, where each block has only one operation: it takes the sum and difference of each consequent pair of elements. The second stage has N/4 blocks and each block has 2 operations, and so on. The last stage has only one block with N/2 operations. The FFT algorithm can be applied in-place (on the same memory buffer that contains the input data).

C. The Radix-4 NTT Algorithm

The previously described FFT algorithm is called the radix-2 FFT. This is because each operation takes two elements and has two outputs. A radix-4 FFT algorithm can be obtained when each two consequent stages are combined as one stage. That is when looking at the FFT stages as a series of multiplications by the stage matrices, each two consequent matrices are multiplied together forming a radix-4 stage matrix. In the radix-4 FFT, each operation has 4 inputs and 4 outputs, and is similar to a matrix multiplication by a size 4 DFT matrix. The bit reverse permutation is different in the radix-4 FFT. Instead of reversing the bits of the index, the digits of the index are reversed in base-4 representation.

D. Decimation in Time and Decimation in Frequency

The FFT algorithm can be performed in reversed order. In the decimation in time (DIT) FFT the values are first permuted, then they pass through stages with blocks doubling in size in case of radix-2 and quadrupling in size in radix-4 version. While in decimation in frequency version of the FFT algorithm, the input values first pass through the transposed

stages of the FFT in reversed order and lastly, they get permuted in frequency domain. Here the first stage has 1 block of operations, then the blocks are halving in size with each operation in radix-2 version and a quarter in size in radix-4 version. The last stage in the DIF FFT takes the sum and difference of consequent pairs, same as the first stage in the DIT FFT algorithm.

By performing a decimation in time in the forward FFT and decimation in frequency in the inverse FFT, the same code can be used for both the forward FFT and the inverse FFT, but with different roots.

E. Cache Efficiency

Cache-oblivious algorithms such as matrix multiplication and the Fast Fourier Transform have issues with performance on most computing platforms used nowadays. This is because they access the memory in a non-continuous manner. Modern computers use hierarchical memory. The random-access memory is much slower than the CPU. The frequently used memory blocks are loaded into the cache which is in the same die as the CPU and is much faster than RAM. The problem with cache oblivious algorithms such as the FFT is that they access the memory in far apart locations. This presents a problem for the cache mechanism. Each time a continuous memory block is loaded only a few values are accessed then the algorithm tries to access a memory location that is far away, and it is not in the cache. The algorithm operations have to wait till the proper memory block is loaded from the slow RAM. This is called a cache miss, and it is a very expensive performance-wise because it takes several thousand CPU cycles. This performance impact is not visible algebraically. Instead the code has to be reviewed by its memory reads. To decrease the cache misses it is preferable that the memory should be accessed in continuous or at least spatially close locations, and the times that access the same location should be consequent as well.

In order to make the best use of SIMD processing, the loaded values have to be spatially close. The performance problem with the FFT algorithm arises in the later stages where the operations take values far apart. With each stage, the operations have arguments spread by a factor of 2 in the radix-2 algorithm and by 4 in the radix-4 version. The code can be sped up by loading 8 consequent values in the SSE registers. This increases the number of operations to 8 per one memory load, therefore decreasing the number of memory loads by 8.

III. Modular Reduction

The modular reduction operation is finding the remainder of division by the modulus q. In the C language it is the operator %. This modulus operator should not be used for cryptographic applications because it contains a division operation that uses a variable time instruction, which presents a security risk [7]. Instead, the reduction should be done as a fixed time operation.

A. Barrett Modular Reduction

There are several ways to do modular reduction. The definition of modular reduction is given by: $r = x - \text{floor}(x/q) * q$. In modular arithmetic, the modulus q is constant. So instead of dividing by q, we can multiply by 1/q. The problem here is that 1/q is a small fraction less than one, and it is a rational number. It can be stored in fixed point representation. This means that the bits of the value $m = 2^k/q$ are stored as an integer. And now the operation x/q is replaced by $x*m >> k$. That is, the value x is multiplied by the bits of $2^k/q$ in integer form, then the result is shifted right k bits to cancel the 2^k. A simple implementation of Barrett reduction is shown in Algorithm 1.

Algorithm 1 : Barrett reduction, with q=12289.

```
short barrett_reduction(int v)
{
    const int q=12289, m=10921, k=27;

    long long temp = v*m;
    temp >>= k;
    temp *= q;
    return v - temp;
}
```

The values m and k are chosen according to the desired range for the input v. If v exceeds this range the result will be wrong. This is because the value m does not have the exact value of 1/q, but the first k bits of it.

It should be noted that the value v to be reduced is usually a result of multiplication of two values mod q, so the input v has the range of $[-q^2, q^2[$. And when q is large enough as in this case, the result v*m can be greater than 2^{32}, and has to be stored in 64 bits.

The range for v is determined as follows: the error of the reciprocal m is given by: $\text{error} = 1/q - m/2^k$. The maximum value v that gives the correct result is 1/error. So, in case of q=12289, we need to reduce values up to $q^2 = 151019521$. The first values of m and k that give a limit that exceeds q^2 are: k=27, and $m=2^{27}/q=10921$.

The problem with this reduction method is that it needs a large 64bit integer to store the temporary product of the input value and the approximate reciprocal bits m. This can be avoided by using the SSE instruction mulhi, since the product will be shifted right by k which is at least 16 bits. Another problem is that the result is in the range [0, 2q[instead of the desired [0, q[. This can be corrected by subtracting q under the condition that the result is greater than q. This is shown in

Algorithm 2.

Algorithm 2 : Range correction.

```
int range_correction(int v)
{
    int mask = v > q;
    mask = -mask;
    mask &= q;
    return v -= mask;
}
```

It is not necessary to do range correction after each modular reduction, but it is just to keep the values from overflowing the 16bit after several additions and subtractions.

B. Short Barrett Reduction

A fast implementation of Barrett reduction exists for modular reduction of small single-word integers. In case of small integers, the approximate reciprocal m and the shift amount k are small, and the shift amount can be chosen as 16, the size of a single machine word. This way the multiplication and the shift can be replaced with a single mulhi instruction. This instruction calculates the product of the values in two registers and stores the high part of the product. This is equivalent to multiplying the values and shifting the product right by 16 bits. This way the short Barrett reduction can be implemented as just 3 SSE instructions as shown in Algorithm 3. The short Barrett reduction was used in the NewHope KEM [ADPS'16].

Algorithm 3 SSE short Barrett reduction for q=12289. Note that the code is in a function only for presentation.

```
__m128i short_barrett_reduction(__m128i v)
{
    const __m128i bar_m=_mm_set_epi16(5);
    const __m128i m_q=_mm_set_epi16(12289);

    __m128i temp=_mm_mulhi_epi16(v, bar_m);
    __m128i temp=_mm_mullo_epi16(temp, m_q);
    return v=_mm_sub_epi16(v, temp);
}
```

C. Montgomery Modular Reduction

The Montgomery reduction is a fast-modular reduction method that can be implemented in just 3 instructions. Usually a machine has a word size of $L = \log(\beta) = 16$, and β is equal to 2^{16}. The single word integer is denoted as a short integer. The large value v to be reduced is stored in an integer of size $2*\log(\beta)$. It is split into high and low parts v_hi and v_lo, each of size $\log(\beta)$. Then the low part v_lo is multiplied by the inverse of q mod β. In this example $q^{-1} \bmod \beta = 12289^{-1} \bmod 65536 = 53249$, which is equal to -12287 in signed short representation. Then the result m is multiplied by q, and divided by β. This is equivalent to taking the high 16 bits from the multiplication m*q. Then the result is subtracted from the high part v_hi. Montgomery reduction is explained in Algorithm 4.

Algorithm 4 Montgomery reduction.

```
short mon_reduction(int v)
{
    const short q=12289, q_1=-12287;

    short *pv = (short*)&v;
    short temp = pv[0]*q_1;
    temp = temp*q >> 16;
    return pv[1] - temp;
}
```

It should be noted that the result needs to be multiplied by $\beta \bmod q$ to get the correct modular reduction. Each Montgomery reduction should be compensated by another multiplication by $\beta \bmod q = 4091$. This compensation can be done as follows: when multiplying the value by a constant, the constant is pre-multiplied by $\beta \bmod q$. Or if no multiplication is found before Montgomery reduction, all reductions can be compensated by multiplying the values by $\beta^n \bmod q$, if the values passed through n Montgomery reductions.

This algorithm is much faster than Barrett reduction because it performs the modular reduction mod β which is taking the low part of the result since β represents the machine word size. It also does not require a large integer storage. This modular reduction can be done with just 3 SSE instructions as shown in Algorithm 5. The variables v_lo and v_hi contain the low and high parts of 8 values. This form can be easily obtained with the SSE instructions mullo and mulhi, when performing multiplications just before the modular reduction.

Algorithm 5 SSE Montgomery reduction:

```
__m128i sse_mon_reduction(__m128i v_lo, __m128i v_hi)
{
    const __m128i m_q=_mm_set1_epi16(12289);
    const __m128i m_q_1=_mm_set1_epi16(-12287);

    v_lo=_mm_mulllo_epi16(v_lo, m_q_1);
    v_lo=_mm_mulhi_epi16(v_lo, m_q);
    return v_lo=_mm_sub_epi16(v_hi, v_lo);
}
```

It should be noted that when the value to be reduced is small enough to fit entirely in the low part v_lo, then the high part v_hi should contain the sign mask of v_lo as shown in Algorithm 6. That is, the small value should be sign-extended into the high and low parts.

Algorithm 6 Montgomery reduction for a small value.

```
__m128i sse_mon_reduction_small(__m128i v_lo)
{
    const __m128i m_q=_mm_set1_epi16(12289);
```

```
    const __m128i m_q_1=_mm_set1_epi16(-12287);
    const __m128i m_zero=_mm_setzero_si128();

    __m128i v_hi=_mm_cmplt_epi16(v_lo, m_zero);
    v_lo=_mm_mulllo_epi16(v_lo, m_q_1);
    v_lo=_mm_mulhi_epi16(v_lo, m_q);
    return v_lo=_mm_sub_epi16(v_hi, v_lo);
}
```

Hence, we can perform 8 multiplications in modular arithmetic with just 5 SSE instructions as shown in Algorithm 7.

Algorithm 7 Multiplication with modular reduction:

```
__m128i sse_mul(__m128i va, __m128i vb)
{
    __m128i v_lo=_mm_mullo_epi16(va, vb);
    __m128i v_hi=_mm_mulhi_epi16(va, vb);
    v_lo=_mm_mullo_epi16(v_lo, m_q_1);
    v_lo=_mm_mulhi_epi16(v_lo, m_q);
    return v_lo=_mm_sub_epi16(v_hi, v_lo);
}
```

IV. Performance Analysis

A. Bit-Reverse Permutation

The bit-reverse permutation is considered a part of the DFT algorithm. It is known to have a significant performance impact, because it does not access memory in a cache efficient way. But when performing the forward NTT of random generated vectors, then the bit-reverse permutation can be omitted.

The bit-reverse permutation can be done as a series of swaps. But the fastest way to perform the bit-reverse permutation is by substitution from a look-up table.

B. Radix-4 FFT vs Radix-2 FFT

Usually the radix-4 FFT is faster than radix-2 FFT. But in case of a SIMD implementation, where several operations are done in parallel at once, the radix-2 turns out faster as shown in the results tables 1 and 2.

Table 1. Performance results of various NTT algorithms. The test consists of a forward transform and an inverse transform, the time shown is the average of 100000 repetitions, in microseconds. The clock cycles were measured with the RDTSC instruction. All tests were run on an Intel Core i7-6800K Broadwell-E processor.

Our attempts		
	Time (µs)	***Cycle count***
Radix-2 AVX2	2.75	9124
Radix-2 SSE	5.30	17607
Radix-4 AVX2	7.11	23613
Radix-4 SSE	10.98	36455
Radix-2	37.30	123854
Radix-4	49.87	165573
NewHope KEM by E. Alkim et al [6]:		
Radix-2 AVX2	4.59	15610
Radix-2	30.78	104638
Our attempts– separate forward and inverse transforms:		
Forward Radix-2 AVX2	1.11	3688
Inverse Radix-2 AVX2	1.42	4709
Forward Radix-2	16.28	54038
Inverse Radix-2	17.67	58654
NewHope KEM – separate forward and inverse transforms:		
Forward Radix-2 AVX2	2.11	7186
Inverse Radix-2 AVX2	2.34	7963
Forward Radix-2	14.82	50398
Inverse Radix-2	15.70	53367

Table 2. Comparison of NTT multiplication. The test consists of two forward transforms, an element-wise multiplication and an inverse transform. All tests were run on an Intel Core i7-6800K Broadwell-E processor.

Our attempts		
	Time (µs)	***Cycle count***
Radix-2 AVX2	3.86	12833
Radix-2 SSE	7.97	26476
Radix-4 AVX2	9.32	30937
Radix-4 SSE	16.90	56125
Radix-2	128.4	426303
Radix-4	136.4	453036
NewHope KEM by E. Alkim et al [6]:		
Radix-2 AVX2	6.89	15610
Radix-2	48.35	164390

The radix-2 version of the NTT turned out faster than the radix-4 one. The reason is that in radix-4 algorithm, the memory is accessed in 4 separate locations in each operation in the later stages, while the radix-2 version accesses 2 separate locations per operation. Hence the radix-2 algorithm is more cache-efficient when working with SIMD instructions, despite having double the number of stages.

We couldn't achieve the performance of the implementation by G. Seiler [7], but our performance with AVX2 is still better than the version in the NewHope KEM [6].

V. Conclusion

The Number Theoretic Transform (NTT) is a variation of the Discrete Fourier Transform (DFT) that uses modular arithmetic instead of complex numbers. The NTT is used for performing fast multiplications of polynomials in ring-LWE cryptosystems that are developed to be immune against quantum-based attacks.

Different implementations of the NTT algorithm were evaluated on different processor architectures with SIMD capabilities, in particular SSE and AVX2 architectures. The results are shown in Table 1. Also, the radix-4 implementation was compared to the radix-2 implementation and it turned out that radix-2 is slightly better. This is because the radix-2 FFT accesses memory in 2 separate locations instead of 4 locations as in radix-4. Also, different methods for fast modular reduction with SIMD instructions were presented.

References

[1] M. Rose. “Lattice-based cryptography: A practical implementation,” 2011.

[2] O. Regev. “On lattices, learning with errors, random linear codes, and cryptography.” In Journal of the ACM, 56(6) p. 34, 2009 Sept 1.

[3] R. Linder, and C. Peikert. “Better key sizes (and attacks) for LWE-based encryption.” In CT-RSA. Vol. 6558, pp. 319-339. 2011.

[4] Y. Yuan, C.-M. Cheng, S. Kyimoto, Y. Miyake, and T. Tagaki. “Portable implementation of lattice-based cryptography using JavaScript.” International Journal of Networking and Computing 6.2 (pp. 309-327). 2016.

[5] J. W. Bos, C. Costello, M. Naehrig, and D. Stebila. “Post-quantum key exchange for the TLS protocol from the RLWE problem.” In Security and Privacy (SP), 2015 IEEE Symposium on IEEE (pp. 553-570), 2015.

[6] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe. “Post-quantum key exchange – a new hope.” In USENIX Security Symposium (pp. 327-343). 2016 Jan 1.

[7] Seiler, Gregor. "Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography." IACR Cryptology ePrint Archive 2018 (2018): 39.

[8] Bos, Joppe, et al. "CRYSTALS-Kyber: a CCA-secure module-lattice-based KEM." *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2018.

[9] Postpischil, Eric. "Construction of a High Construction of a High-Performance FFT Performance FFT." 2004-08.

[10] Pöppelmann, Thomas, and Tim Güneysu. "Towards practical lattice-based public-key encryption on reconfigurable hardware." *International Conference on Selected Areas in Cryptography*. Springer, Berlin, Heidelberg, 2013.

[11] Ducas, Léo, and Alain Durmus. "Ring-LWE in polynomial rings." *International Workshop on Public Key Cryptography*. Springer, Berlin, Heidelberg, 2012.

[12] Israa Hammouda, Hazem Saied, Ayman M. Bahaa-Eldin, “Quantum Databases: Trends and Challenges”, 2016 11th International Conference on Computer Engineering & Systems (ICCES), 275-280, 2016, IEEE

[13] Israa Hammouda, Hazem Saied, Ayman M. Bahaa-Eldin, “A Generalized Grover’s Algorithm with Access Control to Quantum Databases”, 2016 11th International Conference on Computer Engineering & Systems (ICCES), 281-285, 2016, IEEE

[14] Ayman Mohsen, Mohmed Sobh, Ayman M. Bahaa-Eldin, “Lattice-Based Cryptography”, 2017 12th International Conference on Computer Engineering and Systems (ICCES), 462-467, 2018, IEEE