LEARNING WITH ERRORS CRYPTOGRAPHY IN THE ARDUINO UNO

Keshav Sharma: 2017A7PS0140P

Aditya Ramaswamy: 2017A7PS0130P

TABLE OF CONTENTS

OBJECTIVE

Learning with errors has come up recently as a hard problem on which ciphers can be designed. Since this problem is directly dependent on certain well studied Lattice problems assumed to be worst case hard, systems built on this problem are assumed to work well even in the era of post quantum cryptography. Systems designed for low processing power environments will become even more useful, as such environments proliferate. With that in mind, our aim has been to understand fundamental concepts in the field of LWE cryptography and try to implement a cipher based on this field. This cipher has also been designed to work in 8 bit AVR processors, so we also intended to test our work on an Arduino.

MOTIVATION

Both the students involved in this project have had a brush with the Arduino Uno and with assembly programming to some extent. Also not having done anything too significant in cryptography earlier, learning about this scheme and implementing it on an Arduino seemed to be a very good opportunity for us to improve our skills with the Arduino and learn about Lattice cryptography and LWE cryptography. That being said, there is also inherent motivation in learning about the described scheme, analysing it and implementing it. This process serves as a benchmark to test our ability in both understanding and implementing ciphers.

INTRODUCTION

Cryptography for AVR processors adds additional exigent restrictions to the already imminent issues with the advent of quantum computing; any lasting solutions need to be developed keeping in mind both post quantum cryptography and the relatively scant processing power on these devices. It was presumably with this in mind that the proposed solution (described in some detail in the section **METHODOLOGY** given below) was designed.

This solution depends on the hardness of the Learning with Errors problem in Lattice cryptography (how the problem is related to this field is explored in **LEARNING WITH ERRORS**). The solution also requires the use of the analogue to the Discrete Fourier Transform in a finite field, the Number Theoretic Transform. Finally, to generate random samples from a Gaussian distribution, the Knuth-Yao sampler has been used. All of these are described further below.

Our project has implemented the cipher in Arduino C. Initially we planned to implement it in Arduino assembly, but we encountered massive space constraints for some of the requirements for the code. Also, due to inconsistencies in parts of the paper and our inexperience in assembly programming, we were spending too much time on relatively trivial issues, like debugging and implementing arithmetic operations. It was due to this that we decided to shift to C++. We have enclosed our assembly code in this report as well.

THEORY

LEARNING WITH ERRORS

The learning with errors problem can be described as the problem of solving a series of linear equations modulo some prime number, each of which have some small error associated with them. This error immediately invalidates the use of Gaussian Elimination, as the errors only accumulate with either addition or subtraction. This problem is the Learning with Errors Problem.

Slightly more formally, given a series of samples of ordered pairs (a, $< a, s > +e$) where a and s are polynomials belonging to the finite field $\mathbb{Z}q$ [x] where q is a prime number and e is an error chosen according to an error probability distribution, and **a** is public while **s** is kept unknown, the problem is to find the polynomial s. The RLWE problem assumes the elements belong to a polynomial ring instead of a polynomial field.

The most optimal solutions to this problem are of exponential complexity in either time or number of samples required. The most naive solution for instance looks for samples (**a**, b) until an **a** of the form (1, 0, 0, 0…..) is obtained. It can be shown that it takes poly(n) samples until one of this form is obtained. With this sample, s_1 can be obtained. Similarly, all of s_2, s_3,… s_n can be obtained. This solution requires $2^{O(n \log n)}$ samples and running time.

Another algorithm relies on the fact that after roughly O(n) samples, the only possible solution is the one that roughly satisfies all the equations. This thus also takes $2^{O(n \log n)}$ running time. The most optimal solution is $2^{O(n)}$ in both number of samples and time complexity.

The hardness of this problem relies on the hardness of standard lattice problems GAPSVP (the decision version of the shortest vector problem) and SIVP (shortest independent vectors problem). Hardness based on the SIVP is crucial for cryptographic applications.

NUMBER THEORETIC TRANSFORM

Polynomial multiplication is well known to take $O(n^2)$ time complexity by brute force, and O(n logn) complexity by using the DFT. The idea behind this is explained as follows.

Consider two polynomials A(x) and B(x) of the same degree n for now. Given 2n points $x_1,... x_{2n}$ evaluate A(x) and B(x) at all these points. $C(x_i) = A(x_i)B(x_i)$. Interpolating C(x) from these 2n polynomials in O(n log n) time gives us the final answer. This time complexity is possible only by taking the 2n points to be the $2n^{th}$ roots of unity. This process of evaluating the polynomial at all these roots is called the DFT. Interpolating C(x) from the 2n points is called the inverse DFT. The inverse DFT is the same algorithm for the DFT with the multiplicative inverse of the roots of unity used instead. It also includes a constant time division by 2n also.

LWE operates in the field GF(q), and the DFT in this field takes the name NTT (number theoretic transform). The idea is to take roots of unity in this field instead of the complex roots of unity. This can be calculated in constant time and hardcoded, as long as q is fixed. For instance, for any value of q, we have $q(q-2) = (q-1)^2 - 1$. Thus, we have $(q-1)^2 \equiv 1 \pmod q$ for every value of q. This way, we have q-1 is a square root of unity in GF(q) for every value of q. Other roots of unity can be found using simple iteration for any value of q. If x is an n^{th} root of unity in GF(q) then $1 \equiv x^n \pmod q$. This means that $x^{-1} \equiv x^{n-1} \pmod q$. This way is an easy way to calculate the inverse of all our roots as well.

KNUTH YAO SAMPLER

It is required to sample random error polynomials in the key generation and encryption phases. This sampling should also be dependent on the error probability distribution. The Knuth Yao sampler samples according to a probability distribution. This sampler does so by pursuing a random walk across a DDG tree. This is explained below.

Consider the problem of sampling from a set {1, 2, 3, 4}. The probabilities of choosing an element i, p_i, are {.010010, .001000, .000101, .100001} (in binary) respectively. Consider the tree below:

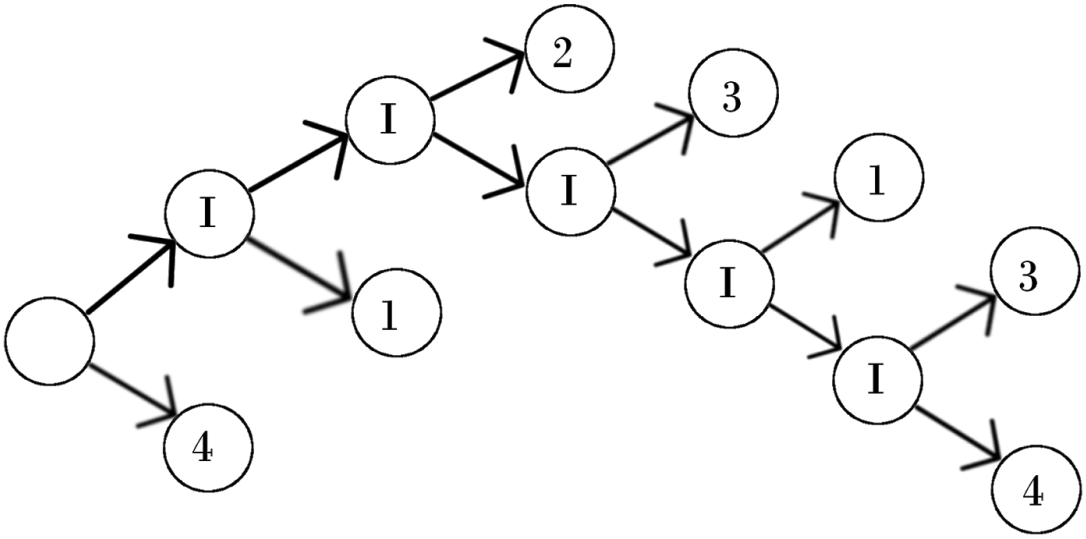

Fig 1: The DDG tree for the given probabilities.

Starting at the root node, at each node, choose a child to travel to based on a different random bit. If a leaf node has been reached, return the value at that node. What is the probability of returning a 4? It is returned either at the first or the last level. Thus, the probability is $2^{-1} + 2^{-6}$ which is 0.100001 in binary, as required. The advantage of this method is that it uses the least number of random bits to generate according to a distribution.

METHODOLOGY

The cipher proceeds as follows:

1. Key Generation(NTT(a)):

Two error polynomials r1 and r2 $\in Rq$ are sampled from the Gaussian Probability Distribution by applying the Knuth Yao sampler twice. The operation NTT(p) = NTT(r1) – NTT(a)•NTT(r2) is performed. The public key is now (NTT(a), NTT(p)) and the private key is (NTT(r2)).

2. Encryption(NTT(a), NTT(p), M):

The message M is assumed to be an n bit binary vector. It is first encoded into a polynomial M' by multiplying each bit by q/2. Three error polynomials e1, e2, e3 $\in Rq$ are now sampled by applying the sampler again. The cipher text is two polynomials (NTT(C1), NTT(C2)) where

$NTT(C1) = NTT(a) \cdot NTT(e1) + NTT(e2)$

$NTT(C2) = NTT(p) \cdot NTT(e1) + NTT(e3 + M')$

3. Decryption(NTT(c1), NTT(c2), NTT(r2)):

M' is obtained as

$M' = INTT(NTT(r2) \cdot NTT(C1) + NTT(C2))$

M' is then decoded to obtain M.

The code is included below.

WORK REPORT

Initially we began our project with an intention to implement our code on an Arduino Uno and then measure our execution statistics against those given by the author elsewhere, but unfortunately, we encountered multiple issues in doing so.

Firstly, we required to code multiple arrays and look up tables for various purposes (for instance, the polynomials themselves) and we ran into severe memory constraints. The Arduino only has 2KB of RAM, and 32 KB of flash memory, so encoding the probability distribution (7681 x 16 bytes) alone would have required some very heuristic approaches that would deviate from the cipher itself.

Further, coding these relatively complex algorithms at an assembly level resulted in us having to spend a huge amount of time just debugging our code. It also required that we spend a lot of time trying to implement basic arithmetic operations, which again seemed to pull us off track.

With these issues in mind we have written our code in standard C++ and enclosed it in this report. We have also enclosed the assembly code in our report below.

OUTCOME

We have a C++ program that implements this cipher. We have also learnt quite a bit about LWE cryptography, and about processes that are used frequently in Lattice cryptography and LWE cryptography, like Gaussian sampling and the NTT. Apart from these, our knowledge about assembly programming and AVR programming has improved through our unsuccessful attempt at implementing the algorithm in the Arduino.

APPENDIX 1: C++ CODE ENCLOSED

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <vector>
#include <time.h>


using namespace std;

int KnuthYao(int Pmat[][16], int r, int q, int MAXROW, int
MAXCOL)
{
     int d = 0;
     for(int col = 0; col<MAXCOL; col++)
     {
          d = 2*d + (r&1);
          r = r >> 1;
          for(int row = MAXROW-1; row>=0; row--)
          {
               d = d-Pmat[row][col];
               if(d==-1)
               {
                    if((r&1)==1)
                         return q-row;
                    else
                         return row;
               }
          }
     }
     return 0;
}


vector<int> generatePoly()
{
     int P[7681][16];
/********************** This array P stores a probability
distribution to be used in Knuth Yao sampler. It is NOT
initialized here because the initialization takes 7600 lines of
code. ****************************************/
```

```
     int q = 7681;
     int arr[256];
     for(int i=0; i<256; i++)
     {
         int r = rand()%7681;
          int res = KnuthYao(P, r, q, 7681, 16);
          arr[i] = res;
     }
     vector <int> ans(begin(arr), end(arr));
     return ans;
}


int inverse(int n)
{
     for(int i=0; i<7681; i++)
     {
          if((i*n)%7681==1)
               return i;
     }
     return 1;
}


vector<int> recfft(vector<int> a)
{
    int N = a.size();
    int omegN, omega = 1;
    if(N==1)
        return a;
    else if(N==256)
        omegN = 198;
    else if(N==128)
        omegN = 799;
    else if(N==64)
        omegN = 878;
    else if(N==32)
        omegN = 2784;
    else if(N==16)
        omegN = 527;
    else if(N==8)
        omegN = 1213;
    else if(N==4)
        omegN = 4298;
    else if(N==2)
```

```
        omegN = 7680;

    vector<int> a0;
    for(int i=0; i<N/2; i++)
        a0.push_back(a[2*i]);
    vector<int> a1;
    for(int i=0; i<N/2; i++)
        a1.push_back(a[2*i+1]);

    vector<int> y0= recfft(a0);
    vector<int> y1= recfft(a1);
    vector<int> y(N, 0);
    for(int k=0; k<N/2; k++)
    {
        y[k] = y0[k] + (omega*y1[k])%7681;
        y[k+N/2] = y0[k] - (omega*y1[k])%7681;
        omega = (omega*omegN)%7681;
    }
    if(y.size()==256)
    {
        for(int i=0; i<y.size(); i++)
        {
            y[i] %= 7681;
              y[i] += 7681;
              y[i] %= 7681;
         }
    }
    return y;
}


vector<int> recinfft(vector<int> a)
{
    int N = a.size();
    int omegN, omega = 1;
    if(N==1)
        return a;
    else if(N==256)
        omegN = 1125;
    else if(N==128)
        omegN = 5941;
    else if(N==64)
        omegN = 1286;
    else if(N==32)
        omegN = 2381;
```

```
    else if(N==16)
        omegN = 583;
    else if(N==8)
        omegN = 1925;
    else if(N==4)
        omegN = 3383;
    else if(N==2)
        omegN = 7680;

    vector<int> a0;
    for(int i=0; i<N/2; i++)
        a0.push_back(a[2*i]);
    vector<int> a1;
    for(int i=0; i<N/2; i++)
        a1.push_back(a[2*i+1]);

    vector<int> y0= recinfft(a0);
    vector<int> y1= recinfft(a1);
    vector<int> y(N, 0);
    for(int k=0; k<N/2; k++)
    {
        y[k] = y0[k] + (omega*y1[k])%7681;
        y[k+N/2] = y0[k] - (omega*y1[k])%7681;
        omega = (omega*omegN)%7681;
    }
    if(y.size()==256)
    {
        for(int i=0; i<y.size(); i++)
        {
            y[i] %= 7681;
            y[i] += 7681;
            y[i] %= 7681;
          y[i] *= inverse(y.size());
          y[i] %= 7681;
         }
    }
    return y;
}

vector<vector<int> > KeyGen(vector<int> acap)
{
    vector<int> r1 = generatePoly();
    vector<int> r2 = generatePoly();
    cout << endl;
    vector<int> r1cap = recfft(r1);
```

```
    vector<int> r2cap = recfft(r2);
    cout << endl;
    vector<int> pcap;
    for(int i=0; i<256; i++)
    {
        int ans = r1cap[i]-acap[i]*r2cap[i];
        ans %= 7681;
        ans += 7681;
        ans %= 7681;
        pcap.push_back(ans);
    }
    vector<vector<int> > result;
    result.push_back(acap);
    result.push_back(pcap);
    result.push_back(r2cap);
    return result;
}

vector<vector<int> > Encryption(vector<int> acap, vector<int>
pcap, vector<int> M)
{
    for(int i=0; i<M.size(); i++)
        M[i] *= 3840;
    vector<int> e1 = generatePoly();
    vector<int> e2 = generatePoly();
    vector<int> e3 = generatePoly();
    vector<int> C1cap;
    vector<int> C2cap;
    for(int i=0; i<M.size(); i++)
        M[i] = (M[i] + e3[i])%7681;
    vector<int> Mt = recfft(M);
    vector<int> e1cap = recfft(e1);
    vector<int> e2cap = recfft(e2);
    for(int i=0; i<M.size(); i++)
    {
        C1cap[i] = (acap[i]*e1cap[i] + e2cap[i])%7681;
        C2cap[i] = (pcap[i]*e1cap[i] + Mt[i])%7681;
    }
    vector<vector<int> > result;
    result.push_back(C1cap);
    result.push_back(C2cap);
    return result;
}
```

```
vector<int> Decryption(vector<int> C1cap, vector<int> C2cap,
vector<int> r2cap)
{
    vector<int> dec;
    for(int i=0; i<C1cap.size(); i++)
        dec[i] = (r2cap[i]*C1cap[i] + C2cap[i])%7681;
    vector<int> result = recinfft(dec);
    return result;
}
```

APPENDIX 2: ASSEMBLY CODE ENCLOSED

```
.dseg
polynomial: .byte 512

.cseg
rcall START
/*R19 R18 * R17 R16*/
MA:
    MUL   R18, R16
    MOVW   R22, R0

    MUL   R19, R17
    MOVW   R24, R0

    MUL   R18, R17
    ADD   R23, R0
    ADC   R24, R1

    MUL   R19, R16
    ADD   R23, R0
    ADC   R24, R1

    RET

    //ASSUMING INITIAL RO IN REGISTERS R5:R2
SAMS2:
    MOV   R9, R5
    MOV   R8, R4
    MOV   R7, R3
    MOV   R6, R2
//  T0->R11 R10   T1->R13 R12   T2->R15 R14
//  0X1E->R17   0X01->R16

    MOV   R12, R9
    MOV   R11, R8
    MOV   R10, R7
    LDI   R27, 3      // NUMBER OF TIMES WE NEED TO SHIFT
LABEL1:
    CLC
    ROL   R10
    ROL   R11
```

```
    ROL   R12
    SUBI  R27, 1
    BRNE  LABEL1
    MOV   R10, R11
    MOV   R11, R12      // T0

    MOV   R13, R11
    MOV   R12, R10
    LDI   R27, 4
LABEL2:
    CLC
    ROR   R13
    ROR   R12
    SUBI  R27, 1
    BRNE  LABEL2      // T1


    MOV   R15, R13
    MOV   R14, R12
    LDI   R27, 4
LABEL3:
    CLC
    ROR   R15
    ROR   R14
    SUBI  R27, 1
    BRNE  LABEL3      // T2

    ADD   R14, R10
    ADC   R15, R11
    ADD   R14, R12
    ADC   R15, R13
//  SUM OF TO T1 T2 IN R15 R14

    LDI   R16, 0x01
    LDI   R17, 0x1e
    MOV   R18, R14
    MOV   R19, R15
    CALL  MA
    //PRODUCT IN R25 R24 R23 R22

    SUB   R6, R22
    SBC   R7, R23
    SBC   R8, R24
    SBC   R9, R25
//  R - 0X1E01*[TO+T1+T2] -> R7 R6    (W0)
```

```
    MOV   R21, R7
    MOV   R20, R6             // TEMPORARILY
    LDI   R27, 3
LABEL4:
    CLC
    ROL   R6
    ROL   R7
    ROL   R8
    SUBI  R27, 1
    BRNE  LABEL4
//R8  ->  Q0  (W0>>13)

    LDI   R16, 0x01
    LDI   R17, 0x1e
    MOV   R18, R8
    LDI   R19, 0
    CALL  MA
    //PRODUCT IN R25 R24 R23 R22

    SUB   R20, R22
    SBC   R21, R23
//  W0 - 0X1E01*Q0  ->  R21 R20
    MOVW  R24, R20

    RET

//"div8u" - 8/8 Bit Unsigned Division
//This function is taken from AVR Application Notes AVR200.[5]

BitReverse:
//Parameters
//1st The address of the array to be bit reversed
//2nd The number of elements in the array
//Y: Index
//Z: Reverse of index
//X: Stores the value in Y
//r4:r5 stores the value in Z
push r26
push r27
push r28
push r29
push r30
push r31
ldi r29, 0
```

```
ldi r31, 0

ldi r28, 0
steBeg:
cp r28, r22
cpc r29, r23
brlo ste1
jmp stePostEv

ste1:
ror r28
brcs CarSet1
rol r30
clc
jmp ste2
Carset1:
rol r30
sec

ste2:
ror r28
brcs CarSet2
rol r30
clc
jmp ste3
Carset2:
rol r30
sec

ste3:
ror r28
brcs CarSet3
rol r30
clc
jmp ste4
Carset3:
rol r30
sec

ste4:
ror r28
brcs CarSet4
rol r30
clc
jmp ste5
```

```
Carset4:
rol r30
sec

ste5:
ror r28
brcs CarSet5
rol r30
clc
jmp ste6
Carset5:
rol r30
sec

ste6:
ror r28
brcs CarSet6
rol r30
clc
jmp ste7
Carset6:
rol r30
sec

ste7:
ror r28
brcs CarSet7
rol r30
clc
jmp ste8
Carset7:
rol r30
sec

ste8:
ror r28
brcs CarSet8
rol r30
clc
jmp ste9
Carset8:
rol r30
sec

ste9:
```

```
ror r28

steEnd:
cp r30, r28
brlo steInc
breq steInc
//Generate addresses for information
clc
rol r28
rol r29
rol r30
rol r31
add r28, r24
add r29, r25
add r30, r24
add r31, r25
//Load values
ld r4, Y+
ld r5, Y
ld r26, Z+
ld r27, Z
//Store values
st Y, r27
st -Y, r26
st Z, r5
st -Z, r4
//Regenerate indices
sub r28, r24
sub r29, r25
sub r30, r24
sub r31, r25
clc
ror r29
ror r28
ror r31
ror r30
steInc:
adiw r28,1
jmp steBeg

stePostEv:
pop r31
pop r30
pop r29
pop r28
```

```
pop r27
pop r26
ret

Exp:
//Parameters
//r27 has the power to which to raise the value
//r26 has the result
//r18 has the modulus q
mul r26,r26
mov r26,r0
call div8u
mov r26,r16       //Can be removed later
lsr r27
cpi r27, 1
brne Exp
ret

NTT:
//Parameters:
//1st An array base address        R25 R24
//2nd n (some power of 2)       R23 R22 (256)
//3nd An nth root of unity wn in Zq   R21 R20     (198)
//4rd The modulus q           R19 R18   (7681)
sub r2, r2
call BitReverse
movw r26, r24

movw r16, r22
add r16, r16
adc r17, r17
add r26, r16
adc r27, r17
st X+, r18
st X+, r19
st X+, r20
st X+, r21
st X+, r22
st X, r23
push r26
push r27

ldi r30,2
ldi r31,0
movw r4, r30          //r5 r4 stores the value of i
```

```
LOOP1:
cp r22,r4
brne LOOP12
cp r23, r5
breq LOOP1END
LOOP12:
  mov r26, r20
  push r27
  call Exp                    //r27 r26 has wi
  pop r27
  ldi r16,1
  ldi r17,0                  //r17 r16 has w
  mov r14,r17
  mov r15,r17                //r15 r14 has j
  movw r2, r4
  clc
  ror r3
  ror r2            //r3 r2 stores i/2

  LOOP2:
  cp r14, r2
  brne LOOP22
  cp r15, r3
  breq LOOP2END
  LOOP22:
    sub r6, r6
     sub r7, r7           //r7 r6 stores the value of k. It is
only used to check the functioning of the loop
    LOOP3:
    cp  r6, r22
     brne LOOP33
     cp r7, r23
    breq LOOP3END
    LOOP33:
      movw r30, r24     // Bass address copied
      movw r28, r6       // k+j in r29 r28
      add r28, r14
       adc r29, r15
      add r30, r28
       adc r31, r29    //Z (r31:r30) stores the address of the
element to be retrieved

      ld  r9, Z+
      ld  r10, Z-      //U-r10 r9
      add r30, r2
```

```
      adc r31, r3
      ld  r11, Z+
       ld  r12, Z-    //V-r12 r11
      push r18
       push r19
       push r25
       push r24
       push r23
       push r22
       movw r18, r11
       call MA     // V is stored in 12-11
                    // Need register r25-r22 again
       mov r13, r9
      mov r14, r10
      sub r13, r11
      sbc r14, r12    //U-V in r14:r13
      st  Z+, r13
      st  Z-, r14
      add r13, r11
      adc r14, r12
      add r13, r11
      adc r14, r12    // U+V in r14:r13
      sub r30, r2
      sbc r31, r3     //a[k+j] back in r31:r30
      st  Z+, r13
      st  Z-, r14
      add r6, r4
       adc r7, r5
      jmp LOOP3
    LOOP3END:
    mul r17, r26
    mov r17, r0
    jmp LOOP2
  LOOP2END:
  clc
  rol r5
  rol r4
  jmp LOOP1
LOOP1END:
RET



START:
    LDI r22,00
```

```
     ldi r23,01
     ldi r21,0x0D
     ldi r20,0x37
     ldi r19,0x1e
     ldi r18,01
     ldi r25, high(polynomial)
     ldi r24, low(polynomial)
     //Generating the polynomial array
     movw r26, r24
     ldi r16, 1
     ldi r17, 1
     beg1212:
     st x+, r16
     st x+, r2
     inc r16
     cpse r16, r17
     jmp beg1212
     st -x, r17                //Polynomial array has been
generated
     call NTT
```

REFERENCES

1. Z. Liu et al., “Efficient Implementation of Ring-LWE Encryption on 8-bit AVR Processors,” Int’l. Wksp. Cryptographic Hardware and Embedded Systems, Springer, 2015, pp.663–82.
2. O. Regev, "The Learning with Errors Problem (Invited Survey)," 2010 IEEE 25th Annual Conference on Computational Complexity, Cambridge, MA, 2010, pp. 191- 204.
3. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. Introduction To Algorithms. MIT Press and McGraw-Hill, 1990.
4. DWARAKANATH, N. C.—GALBRAITH, S. D.: Sampling from discrete gaussians for lattice-based cryptography on a constrained device, Appl. Algebra Engrg. Comm. Comput. 25 (2014), 159–180.
5. AVR200 AVR Application Notes (2017, February 01) Retrieved from https://www.microchip.com/wwwAppNotes/AppNotes.aspx?appnote=en591172