

# Efficient Software Implementation of Ring-LWE Encryption

Ruan de Clercq, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede

KU Leuven, Department of Electrical Engineering - ESAT/COSIC and iMinds, Belgium  
firstname.lastname@esat.kuleuven.be

**Abstract.** Present-day public-key cryptosystems such as RSA and Elliptic Curve Cryptography (ECC) will become insecure when quantum computers become a reality. This paper presents the new state of the art in efficient software implementations of a post-quantum secure public-key encryption scheme based on the ring-LWE problem. We use a 32-bit ARM Cortex-M4F microcontroller as the target platform. Our contribution includes optimization techniques for fast discrete Gaussian sampling and efficient polynomial multiplication. This implementation beats all known software implementations, on any architecture, by at least one order of magnitude. We further show that our scheme beats all ECC-based public-key encryption schemes by at least one order of magnitude. At 128-bit security we require 121166 cycles per encryption and 43324 cycles per decryption, while at a 256-bit security we require 261939 cycles per encryption and 96520 cycles per decryption. Gaussian sampling is done at an average of 28.5 cycles per sample.

## 1 Introduction

Our present day public-key schemes use number theoretic problems such as the factoring or the discrete logarithm for providing security, authenticity and privacy. For sufficiently large key sizes, these public-key cyptosystems are practically unsolvable using present day computers or even using super computers or special hardware clusters. In 1994 Peter Shor proposed a quantum algorithm [1] for integer factorization. Later a modified version of Shor's algorithm appeared that can solve the elliptic curve discrete logarithm problem (ECDLP). Shor's algorithms cannot be used on classical computers, and can only execute on *powerful* quantum computers to solve the factoring or the discrete logarithm problems in polynomial time. In the present decade, significant research is being performed to develop powerful quantum computers. As a consequence, the cryptography research community has become interested in developing post-quantum-secure public-key schemes.

The *learning with errors* (LWE) problem and its *efficient* ring variant (the ring-LWE problem) are related to well known worst-case problems over lattices, and hence are considered to be secure in the post-quantum world. In this paper we will describe an efficient software implementation of an encryption scheme based on the ring-LWE problem [2]. The ring-LWE encryption scheme is computation intensive, and uses polynomial arithmetic and discrete Gaussian sampling as primitive functions. While addition and subtraction of large polynomials are easy to implement, efficient design deci-

sions are essential for implementing the polynomial multiplication algorithm. Similarly, an efficient discrete Gaussian sampler improves the performance of encryption.

In recent years several practical implementations based on the ring-LWE problem were published in literature. The first implementation [3] includes a hardware-based architecture and a high-level software implementation. It uses a Number Theoretic Transform (NTT) based polynomial multiplier and a rejection sampler. In [4–8] more efficient hardware implementations reduced the area and timing requirements. Even though there exists several high-level software implementations [3, 13] and hardware implementations of the ring-LWE based encryption scheme in literature, efficient software implementations on resource constrained microcontrollers have received less attention [12]. Our everyday lives are permeated by these devices, and they are increasingly becoming interconnected, even over the Internet. This raises security concerns, as these devices handle sensitive information and are sometimes critical for the safety of human lives.

***Our Contributions:*** This paper presents an implementation of the ring-LWE encryption scheme on the ARM Cortex-M4F microcontroller. Our design goals include high speed and low memory consumption. Our contributions are as follows:

**Fast discrete Gaussian Sampling** We use the Knuth-Yao sampling algorithm to implement a fast discrete Gaussian sampler. We investigate acceleration techniques to improve the sampler based on the architecture of the microcontroller. The platform’s built-in True Random Number Generator (TRNG) is used to generate random numbers. Lookup tables are used to accelerate the sampling algorithm in the most frequently used regions of the Gaussian distribution. This allows us to sample at an average of 28.5 cycles per sample.

**Efficient Polynomial Multiplication** We use the negative-wrapped NTT along with computational optimizations from [7] to implement the polynomial multiplication. The architecture’s large word size is used to store multiple coefficients in each processor word, and the basic negative-wrapped iterative NTT algorithm is unrolled by a factor two. This reduces the number of memory accesses and loop overhead by 50%. We demonstrate that a polynomial multiplication of 256 elements can be done in 108147 cycles.

The paper is organized as follows. First, we provide a mathematical background to help readers understand the paper. Next, we discuss the implementation details. We provide an analysis of bottlenecks in standard algorithms, and provide solutions to overcome these, with a specific focus on the target platform. Afterwards, our results are presented and compared with related works. Finally, we provide a short conclusion together with ideas for future work.

## 2 Mathematical Background

In this section we provide a brief mathematical overview and related references that might be helpful for the reader.

## 2.1 The ring-LWE Encryption Scheme

In the ring-LWE problem [9] two polynomials  $\mathbf{a}$  and  $\mathbf{s}$  are chosen uniformly from a polynomial ring  $R_q = \mathbb{Z}_q[\mathbf{x}]/\langle f \rangle$  where  $f$  is an irreducible polynomial of degree  $n$ . An error polynomial  $\mathbf{e}$  of degree  $n$  is sampled from an error distribution  $\mathcal{X}$ . The error distribution is usually a discrete Gaussian distribution  $\mathcal{X}_\sigma$  with standard deviation  $\sigma$ . The ring-LWE distribution  $A_{s,\mathcal{X}}$  over  $R_q \times R_q$  consists of tuples  $(\mathbf{a}, \mathbf{t})$  where  $\mathbf{t} = \mathbf{a} \cdot \mathbf{s} + \mathbf{e} \in R_q$ . Given a polynomial number of sample pairs  $(\mathbf{a}, \mathbf{t})$  from  $A_{s,\mathcal{X}}$ , it is very difficult to find  $\mathbf{s}$ . This problem is known as the *search ring-LWE* problem. An encryption scheme based on the ring-LWE problem was proposed in [2]. Efficient implementations of this encryption scheme are available in [?,7]. We use the parameter sets  $(n, q, \sigma)$  from [3], namely  $P_1 = (256, 7681, 11.31/\sqrt{2\pi})$  and  $P_2 = (512, 12289, 12.18/\sqrt{2\pi})$  that have security levels of 128-bit and 256-bit respectively.

## 2.2 Discrete Gaussian Sampling

The ring-LWE cryptosystem requires samples from a discrete Gaussian distribution to construct the error polynomials during the key generation and encryption operations. There are various methods for sampling from a discrete Gaussian distribution. The most well known techniques are rejection sampling, inversion sampling, and the random bit model. Efficiency (time and space) of the sampling algorithms depends on the standard deviation  $\sigma$  of the distribution. Detailed comparative analysis of the sampling algorithms can be found in [10]. We use the Knuth-Yao algorithm [11] to sample from a discrete Gaussian distribution. The Knuth-Yao algorithm is based on the random-bit model and uses, on average, a near-optimal number of random bits. This algorithm requires storage of the probabilities of the sample points. The small standard deviation in the ring-LWE encryption scheme means that the memory requirement is small and can easily be satisfied on microcontrollers.

*Tail and precision bounds:* The tail of a discrete Gaussian is infinitely long and the probability values have infinitely large precision. Thus for a practical implementation, there is a tail and precision bound for a required bit-security. We use sufficiently large precision and tail-bound [10, 6] to maintain a maximum statistical distance of  $2^{-90}$  to the true distribution.

## 2.3 Polynomial Multiplication

For large polynomial multiplications, the Fast Fourier Transform (FFT) is considered as the fastest algorithm due to its  $\mathcal{O}(n \log n)$  complexity. In this paper we use the Number Theoretic Transform (NTT) which corresponds to a FFT where the primitive roots of unity are from a finite ring (thus integers) instead of complex numbers. For efficient implementation, we perform polynomial arithmetic in  $R_q = \mathbb{Z}_q[\mathbf{x}]/\langle f \rangle$  with  $f = x^n + 1$ , and  $n = 2^k$  and  $q$  a prime such that  $q \equiv 1 \pmod{2n}$ . Such a choice of the parameters allows us to use an  $n$ -point NTT instead of an  $2n$ -point NTT during a polynomial multiplication in  $R_q$ . This technique is known as the *negative wrapped convolution*. However there is an additional scaling overhead associated with the negative wrapped convolution. We use the optimizations from [7] during the NTT computation.

In this paper we use the steps from [7] to implement a fast encryption scheme. The encryption samples three error polynomials, computes three forward NTTs and two coefficient-wise polynomial multiplications, and three polynomial additions. The decryption computes one coefficient-wise polynomial multiplication, one polynomial addition and one inverse NTT. For more details the reader is referred to [7].

### 3 Implementation

In this section we describe our ring-LWE implementation based on an ARM platform. After introducing the target device, we describe techniques for efficient sampling from a Gaussian distribution, efficient polynomial multiplication, and finally random number generation.

#### 3.1 Target Device

Our design was implemented on the ARM Cortex-M4F, which is a popular and powerful embedded platform. It has a 32-bit word size, 13 general-purpose registers, its instruction set supports performing single-cycle 32-bit multiplications, 16-bit SIMD arithmetic, and a division instruction that requires between 2-12 cycles, depending on the input parameters. We make use of STMicroelectronics' STM32F407 chip, which includes an ARM Cortex-M4F with a maximum clock speed of 168 MHz, 192 KB of SRAM, and a hardware-based TRNG.

#### 3.2 Gaussian Sampler

The discrete Gaussian sampler deserves special attention, as the efficient implementation of this building block has a big performance impact on the ring-LWE cryptosystem. In our implementation, each encryption operation requires  $3n$  Gaussian samples.

**Knuth-Yao Algorithm** This algorithm [11] performs a random walk along a binary tree known as the discrete distribution generating (DDG) tree. A DDG tree is related to the probabilities of the sample points. The binary expansions of the probabilities are written in matrix form with binary elements, referred to as the probability matrix  $P_{mat}$ . Each row of  $P_{mat}$  corresponds to the probability of sampling a random number at a discrete position from the Gaussian distribution. Each element of  $P_{mat}$  represents a node in the binary tree, with each non-zero element corresponding to a terminal node. Each column of  $P_{mat}$  corresponds to a level in the DDG tree.

Alg. 1 shows a listing of the algorithm. The DDG tree is constructed on-the-fly, eliminating the need for storing the entire tree. During a sampling operation a random walk is performed starting from the root of the DDG tree. Each random walk consumes a single random bit to travel from one level to the next. The distance counter  $d$  represents the number of intermediate nodes to the right side of the visited node. Each non-zero node that is visited, decrements the distance counter by one. When the distance counter is finally decremented to below zero, the terminal node is found, and the current row

**Input:** Probability matrix  $P$ , random number  $r$ , modulus  $q$

**Output:** Sample value  $s$

```
1 for col ← 0 to MAXCOL do
2   d ← 2d + (r&1)
3   r ← r ≫ 1
4   for row ← MAXROW downto 0 do
5     d ← d - P[row][col]
6     if d = -1 then
7       if (r&1) = 1 then
8         return q - row
9       else
10        return row
11 return 0
```

Algorithm 1: Knuth-Yao Sampling.

number of the probability matrix represents the sample. As the probability matrix only contains the positive half of the Gaussian distribution, a random bit is used to decide the sign of the sample. As our scheme performs all operations modulo  $q$ , the negative number is found by  $q - row$ . For more details the reader is referred to [6, 10].

The bit-scanning operation in Alg. 1 is expensive: in the inner loop each bit of a column is extracted from  $P_{mat}$ , subtracted from  $d$ , after which the sign of  $d$  is checked. Loop overhead exists in each inner loop iteration: the  $row$  index needs to be updated, and checked against its lower bound. We further need to read multiple words from each column. Therefore, each iteration of the inner loop requires at least 8 cycles for: updating and bounds checking of  $d$  and the loop overhead from the  $row$  index. While this seems like a small amount, it becomes significant when you consider that  $P_{mat}$  consists of 5995 bits in our design for  $s = 11.31$ . The following discusses several optimizations techniques for software implementations of the Knuth-Yao algorithm. These techniques mostly focus on reducing the number of scanned bits.

**Storing the probability matrix in column form** Alg. 1 requires consecutive accesses to elements from different rows in the same column in  $P_{mat}$ . To keep the number of memory accesses low,  $P_{mat}$  is stored in a column-wise form. Storage of  $P_{mat}$  with  $s = 11.31$  requires 55 rows and 109 columns to provide a precision of  $2^{-90}$ . As each column contains only 55 bits, 9 bits are wasted when a column is stored in two 32-bit words. The following optimizations assume that the matrix is stored in column-wise form.

**Reducing the number of stored elements** From Alg. 1 we can see that processing a zero has no effect, as the distance  $d$  is not altered. It is therefore desirable to process and store the region of the matrix that is known to contain non-zero elements. The Hamming weight between two consecutive columns increases at most by one; thereby causing a large number of zeros to exist in the bottom left-hand corner of the probability matrix, as shown in Fig. 1.

```

1 1 1 1 0 0 0 1 0 0 0
0 1 0 0 1 0 1 1 1 1 0
0 0 1 1 0 1 0 1 1 1 1
0 0 0 1 1 0 0 0 0 1 0
0 0 0 0 1 0 1 1 1 0 1
0 0 0 0 0 1 1 1 1 1 0
0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 1 0 1 1
0 0 0 0 0 0 0 0 0 1 1 1
0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0

```

**Fig. 1.** The partial contents of a probability matrix. We avoid storing zero words (represented by the blue box) as they do not need to be processed.

Multiple processor words are required to store each column (section 3.2), and as we would like to avoid the ineffectual bit-scanning operation on zero words, we store only the non-zero words of the probability matrix, as they require no processing. For  $s = 11.31$  this technique allowed us to reduce the number of stored words from 218 to 180.

**Skipping Unnecessary Bit Scanning** A terminal node is located in a particular level when the value of the distance  $d$  is less than the Hamming weight of the associated column of the probability matrix. The authors in [6] proposed to store each column's Hamming weight in memory. The Hamming weights are used to avoid performing bit scanning on any column that would not reduce the distance  $d$  to a negative number, thereby avoiding columns in which a terminal node will not be found. This method requires storage for the Hamming weights, as well as logic in the outer loop to check if a terminal node will be discovered in the current level.

We propose to avoid the bit-scanning operation completely on all non-zero elements of the probability matrix. A useful feature to do this on our target microcontroller is an instruction that counts the number of leading zeros, called `clz`. The current column word is stored in a register, and in each iteration of the inner loop `clz` is used to skip the consecutive leading zeros of the elements of a column that still requires processing. Next, the processed bits are left-shifted, and the inner loop repeats until the register is equal to zero.

**Lookup Tables** Fig. 2 shows that the probability of sampling from the distribution is highest at the mean. For  $s = 11.31$  a terminal node has a probability of 97.27% to be located within the first 8 levels and 99.87% to be located within the first 13 levels of the DDG tree. This can be exploited by using a lookup table (LUT) to represent the levels of the DDG tree that are closest to the mean, as they have a much higher likelihood of containing a terminal node.

**Input:** Probability matrix  $P$ , random number  $r$ , modulus  $q$

**Output:** Sample value  $s$

```

1   $index \leftarrow r \& 255$ 
2   $r \leftarrow r \gg 8$ 
3   $s \leftarrow LUT1[index]$ 
4  if  $msb(s) = 0$  then
5      if  $(r \& 1) = 1$  then
6           $\quad \left| \quad \text{return } q - s$ 
7      else
8           $\quad \left| \quad \text{return } s$ 
9   $d \leftarrow s \& 7$ 
10  $col \leftarrow 0$ 
11 for  $col \leftarrow 8$  to  $MAXCOL$  do
12      $d \leftarrow 2d + (r \& 1)$ 
13      $r \leftarrow r \gg 1$ 
14     for  $row \leftarrow MAXROW$  downto  $0$  do
15          $d \leftarrow d - P[row][col]$ 
16         if  $d = -1$  then
17             if  $(r \& 1) = 1$  then
18                  $\quad \left| \quad \text{return } q - row$ 
19             else
20                  $\quad \left| \quad \text{return } row$ 
21 return  $0$ 

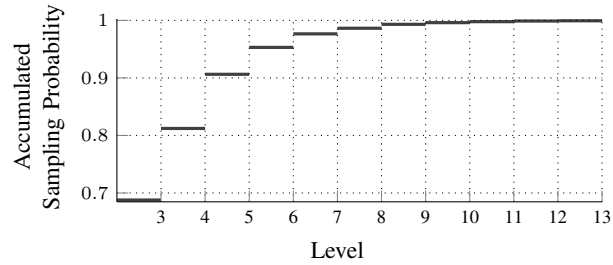
```

Algorithm 2: Knuth-Yao Sampling with an LUT

An LUT that represents the first 8 columns is generated by using an 8-bit index (instead of a random number) as an input to Alg. 1, and the results are saved in a 256-element lookup table. There is a small probability that a terminal node will not be discovered in the first 8 levels. Each index that does not lead to a terminal node within the first 8 levels, will cause a lookup failure, which is indicated in the most significant bit of the corresponding lookup table element.

Alg. 2 shows how the costly bit-scanning operation for the first 8 levels is replaced by an inexpensive table lookup operation. Sampling is performed with an 8-bit random number as an index to the LUT. The lookup is successful if the most significant bit of the lookup result is zero, after which the algorithm returns the lookup result. A lookup failure occurs if the most significant bit of the lookup result is one. The distance  $d$  is then assigned to the lookup result with the most significant bit cleared, followed by the bit-scanning operation (Alg. 1).

The efficiency of the algorithm can further be increased by using a second lookup table to represent level 9 up to level 13 of the DDG tree. The second LUT is generated in a similar way to the first LUT. The 8-bit index now consists of a 5-bit random number concatenated with the 3-bit distance  $d$ . After a failed lookup in the first LUT, a lookup in the second LUT checks if the terminal node lies inside level 9 through level 13. The most significant bit of the lookup result is checked for a lookup failure. If the most



**Fig. 2.** The probability that the Knuth-Yao sampling algorithm finds a terminal node within  $x$  levels for a Gaussian distribution with  $\sigma = 11.31/\sqrt{2\pi}$ .

significant bit of the lookup result is zero, the lookup was successful, and the algorithm quits by returning the lookup result. However, if the most significant bit of the lookup result is one, a lookup failure has occurred. Next, the distance  $d$  is set to the low 4 bits of the lookup result, followed by the bit-scanning operation on the remaining levels.

For a Gaussian distribution with  $\sigma = 11.31/\sqrt{2\pi}$  we found that all failed lookups from the first LUT has a distance  $d$  between 0 and 6. As the second LUT index, which consists partly of the parameter  $d$ , now has a limited range of values that will ever be set, the LUT can be stored in only 224 elements.

### 3.3 Polynomial Multiplication

In this section we discuss the building blocks for fast multiplication based on the Number Theoretic Transform (NTT).

The basic negative-wrapped iterative algorithm for NTT computation is shown in Alg. 3. This algorithm has inefficiencies coming from memory accesses in non-consecutive locations. First, two coefficients are read from non-consecutive locations. Next, some processing is performed on these two coefficients, and finally the two resulting coefficients are written to non-consecutive locations. Each inner loop iteration requires two pointer calculations to access the elements stored in the non-consecutive memory locations. The inner loop further has loop overhead from updating the index  $k$ , and checking it against its upper bound.

Coefficients require only 13-bits or 14-bits of storage respectively for  $q = 7681$  and  $q = 12289$ , and therefore fit into halfwords. On the target platform, a memory access requires 2 cycles, regardless of whether it is to a 16-bit halfword, or to a full 32-bit word. It is therefore wasteful to access halfwords, as two coefficients can be stored in a 32-bit word, which can then be accessed in the minimum number of cycles. The expensive calculation of twiddle factors can be avoided by storing precomputed twiddle factors, and inverse twiddle factors in a lookup table.

### 3.4 Efficient Polynomial Multiplication

We can reduce the number of memory accesses, pointer operations, and loop overhead by 50% by performing a two-fold unrolling of the inner loop, as shown in Alg. 4.



**Input:** Polynomial  $a(x) \in \mathbb{Z}_q[x]$  of degree  $n - 1$  and the  $n$ -th primitive root  $w_n \in \mathbb{Z}_q$  of unity

**Output:** Polynomial  $A(x) \in \mathbb{Z}_q[x] = NTT(a)$

```

1  $A \leftarrow BitReverse(a)$ 
2 for  $m = 2$  to  $n/2$  step  $2m$  do
3    $w_m \leftarrow primitiveroot(m)$  //  $w_m = w_n^{n/m}$ 
4    $w = \sqrt{w_m}$ 
5   for  $j = 0$  to  $m/2 - 1$  do
6     for  $k = 0$  to  $n - 1$  step  $m$  do
7        $t \leftarrow w \cdot A[m/2 + j + k] \bmod q$ 
8        $u \leftarrow A[j + k]$ 
9        $A[j + k] \leftarrow u + t \bmod q$ 
10       $A[m/2 + j + k] \leftarrow u - t \bmod q$ 
11     $w \leftarrow w \cdot w_m$ 

```

Algorithm 3: Negative-Wrapped Iterative Fwd NTT

Memory operations are reduced by storing two coefficients in a single 32-bit word. In each iteration of the inner loop the following occurs: two 32-bit words, each containing two coefficients are read from memory, followed by arithmetic with these coefficients, and finally the two resulting 32-bit words, each containing two coefficients, are written to two different memory locations. The inner loop still requires two pointer calculations, but now at least each memory operation will access two coefficients in a single memory location. The reduction in the loop overhead comes from the two-fold unrolling of the inner loop, as index  $k$  requires 50% fewer updates.

During encryption, three NTT operations are performed, one after the other, on three different sets of coefficients. The loop overhead and calculation of the parameter  $w$  has a non-negligible cost. This cost can be reduced by performing the three NTT operations in parallel inside the same inner loop. By doing this, the cost is effectively reduced by 33%. Each of the three sets of coefficients requires a pointer to its data set. However, as the target only has 13 general-purpose registers, we found it infeasible to keep all three pointers simultaneously inside registers. To solve this, we propose to store the three sets of coefficients in three consecutive memory locations separated by  $n/2$  addresses. This allows us to store only the first coefficient set's address in a register, as the remaining two sets' addresses can easily be calculated.

### 3.5 Random Number Generation

The target platform's TRNG uses a 48 MHz clock and can generate a 32-bit random number every 40 clock cycles. The microprocessor runs at a clock speed of 168 MHz, and can perform other computations while waiting 12 cycles between each random number request. According to [?] the TRNG passes all the NIST statistical tests for secure random number generation.

Each call to the Knuth-Yao sampling algorithm requires a varying amount of random bits. To reduce the number of accesses to the TRNG, and thereby increase the efficiency of our implementation, we propose to fetch a fresh random number from the

**Input:** Polynomial  $a(x) \in \mathbb{Z}_q[x]$  of degree  $n - 1$ , and lookup table *primitive\_root* with the  $m$ -th primitive roots  $w_m \in \mathbb{Z}_q$  of unity

**Output:** Polynomial  $A(x) \in \mathbb{Z}_q[x] = NTT(a)$

```

1  $A \leftarrow BitReverse(a)$ 
2  $l \leftarrow 1$ 
3 for  $m = 2$  to  $n/2$  step  $2m$  do
4    $w_m \leftarrow primitive\_root[l]$  //  $w_m = w_n^{n/m}$ 
5    $w \leftarrow primitive\_root[l + 1]$  //  $w = \sqrt{w_m}$ 
6   for  $j = 0$  to  $m - 1$  step  $2$  do
7     for  $k = 0$  to  $n/2 - 1$  step  $2m$  do
8        $(u_1, t_1) \leftarrow (A[j + k], A[j + k + 1])$ 
9        $(u_2, t_2) \leftarrow (A[m + j + k], A[m + j + k + 1])$ 
10       $t_1 \leftarrow w \cdot t_1 \bmod q$ 
11       $t_2 \leftarrow w \cdot t_2 \bmod q$ 
12       $A[j + k] \leftarrow u_1 + t_1 \bmod q$ 
13       $A[j + k + 1] \leftarrow u_2 + t_2 \bmod q$ 
14       $A[m + j + k] \leftarrow u_1 - t_1 \bmod q$ 
15       $A[m + j + k + 1] \leftarrow u_2 - t_2 \bmod q$ 
16       $w \leftarrow w \cdot w_m$ 
17    $l \leftarrow l + 1$ 
18  $w_m \leftarrow primitive\_root[l]$  //  $w_m = w_n$ 
19  $w \leftarrow primitive\_root[l + 1]$  //  $w = \sqrt{w_m}$ 
20 for  $k = 0$  to  $n/2 - 1$  do
21    $(u_1, t_1) \leftarrow (A[2k], A[2k + 1])$ 
22    $t_1 \leftarrow w \cdot t_1 \bmod q$ 
23    $A[2k] \leftarrow u_1 + t_1 \bmod q$ 
24    $A[2k + 1] \leftarrow u_1 - t_1 \bmod q$ 
25    $w \leftarrow w \cdot w_m$ 

```

Algorithm 4: Memory Efficient Negative-Wrapped Fwd NTT

**Table 1.** Measured results of major operations.

Operation	$P_1$ (Cycles)	$P_2$ (Cycles)
NTT transform	31 583	73 406
Parallel NTT transform	84 031	188 150
Inverse NTT transform	39 126	90 583
Knuth-Yao sampling	7 294	14 604
NTT multiplication	108 147	248 310

$P_1 = (256, 7681, 11.31/\sqrt{2\pi})$ ,  $P_2 = (512, 12289, 12.18/\sqrt{2\pi})$

TRNG only when we run out of fresh random bits. The random bits are stored in a register, and all used bits are right-shifted out of the register. When the register contains an insufficient number of fresh bits for the next operation, a new random number is fetched. A register could be used to keep a count of the number of fresh random bits in the register. However, a limited number of registers are available, and we can do better than this by counting the number of fresh bits with the `clz` instruction. To ensure that `clz` reports the correct number of used bits, we set each fresh random number’s most significant bit to one.

## 4 Results

In this section, we present the performance results of our ring-LWE implementation on the ARM Cortex-M4F. The elapsed clock cycles are measured with the platform’s built-in debug and trace unit (DWT), which contains a cycle count register (`DWT_CYCCNT`). All routines were benchmarked by obtaining the average cycle counts for 10000 runs.

### 4.1 Performance Results

Table 1 shows that the NTT-based operations are at least 123% more expensive for  $P_2$  when compared with  $P_1$ , whereas Knuth-Yao sampling requires an average of 28.5 cycles per sample for both parameters. The NTT and Inverse NTT operations have roughly the same execution times. The Parallel NTT operation consists of three NTT operations in parallel, and outperforms 3 separate NTT operations by 8.3%.

Table 2 shows that the code size has the same storage requirement for both parameter sets, while the RAM requirement increases by approx. 100% when comparing  $P_2$  with  $P_1$ . Key generation, encryption, and decryption each has a respective increase in execution time of 126%, 118%, and 117% when comparing  $P_1$  with  $P_2$ . Decryption requires 35% fewer cycles than encryption, while using 33% less RAM.

### 4.2 Performance Comparison

Table 3 shows that our results for the NTT transform requires 27.5% less cycles than [12]. The scheme proposed by [13] has parameters  $n = 1024, \sigma = 8/\sqrt{2\pi}$  while providing a security level of 128 bits. The same security level can be provided with  $P_1$  which uses 3 times less coefficients. This, together with our optimizations, allowed

**Table 2.** Measured results for our implementation of the ring-LWE encryption scheme.

Operation	Cycles	Flash (B)	RAM (B)	Parameters
Key Generation	116 772	1 552	1 596	$P_1$
Encryption	121 166	1 506	3 128	$P_1$
Decryption	43 324	516	2 100	$P_1$
Key Generation	263 622	1 552	3 132	$P_2$
Encryption	261 939	1 506	6 200	$P_2$
Decryption	96 520	516	4 148	$P_2$

$P_1 = (256, 7681, 11.31/\sqrt{2\pi})$ ,  $P_2 = (512, 12289, 12.18/\sqrt{2\pi})$

**Table 3.** Performance comparison of major building blocks in lattice-based post-quantum cryptosystems.

Operation	Platform	Cycles	$n, q, \sigma$
NTT transform[12]	ARM Cortex-M4F	122 619	$P_3$
<i>NTT transform*</i>	ARM Cortex-M4F	71 090	$P_2$
NTT multiplication[12]	ARM Cortex-M4F	508 624	$P_3$
<i>NTT multiplication*</i>	ARM Cortex-M4F	237 803	$P_2$
NTT multiplication[13]	Intel Core i5 4570R	342 800	$P_4$
<i>NTT multiplication*</i>	ARM Cortex-M4F	108 147	$P_1$
Gaussian sampling[13]	Intel Core i5 4570R	652.3	$P_4$
Gaussian sampling[12]	ARM Cortex-M4F	1828	$P_3$
<i>Gaussian sampling*</i>	ARM Cortex-M4F	28.5	$P_1$ and $P_2$

\* This work,  $P_1 = (256, 7681, 11.31/\sqrt{2\pi})$ ,  $P_2 = (512, 12289, 12.18/\sqrt{2\pi})$

$P_3 = (512, 12289, 215)$ ,  $P_4 = (1024, 2^{32} - 1, 8/\sqrt{2\pi})$

Note: For NTT transform and NTT multiplication  $P_2$  and  $P_3$  are equivalent.

us to beat their result for NTT multiplication by a factor of 3.2, and their result for Gaussian sampling by more than one order of magnitude. even though they use a much more powerful desktop processor.

Table 4 lists a comparison of our implementation with the only other known software-based ring-LWE implementation by [3]. Even though they used a desktop processor, we still beat their results by more than one order of magnitude.

In order to draw a comparison of our cryptosystem with more established public-key cryptosystems, we compare our implementation to an existing ECC implementation. We consider Elliptic Curve Integrated Encryption Scheme (ECIES) [?], which has few known software implementations. The cycle cost of both the encryption and decryption operations are estimated based on the most performance hungry operations, namely two point multiplications. As our scheme provides a security level of 128 bits, we therefore compare our results to ECC implementations of roughly 256 bits. In [15] the authors found that a 233-bit point multiplication requires 2761640 cycles on the ARM Cortex-M0+. This means that an ECIES implementation would require roughly 5523280 cycles for both encryption and decryption. Therefore, our implementation is

**Table 4.** Comparison of ring-LWE encryption schemes.

Platform	Key Gen.	Encrypt	Decrypt	$n, q, \sigma$
Core 2 Duo [3] <sup>1</sup>	9 300 000	4 560 000	1 710 000	$P_1$
Cortex-M4F*	117 009	121 166	43 324	$P_1$
Core 2 Duo [3] <sup>1</sup>	13 590 000	9 180 000	3 540 000	$P_2$
Cortex-M4F*	252 002	261 939	96520	$P_2$

\* This work,  $P_1 = (256, 7681, 11.31/\sqrt{2\pi})$ ,  $P_2 = (512, 12289, 12.18/\sqrt{2\pi})$

<sup>1</sup> Cycles estimated from reported execution time.

faster than ECIES by more than one order of magnitude for encryption and two orders of magnitude for decryption.

## 5 Conclusion

In this work, we demonstrated that it is possible to implement a lattice-based post-quantum public-key cryptosystem on a device that is as simple as a Cortex-M4F microcontroller. We showed various implementation techniques to improve the efficiency of the Knuth-Yao Gaussian sampler and the NTT-based multiplier. These optimization allowed us to beat all known implementations of ring-lwe encryption operations, and gaussian sampling by at least one order of magnitude. We further demonstrated that our scheme beats all ECC-based public-key encryption schemes by at least one order of magnitude. Our implementation requires an average of 28.5 cycles per Gaussian sample, 121166 and 43324 cycles respectively for encryption and decryption at 128-bit security, and 261939 and 96520 cycles respectively for 256-bit security, while using a modest amount of flash and RAM.

For future work we plan to create an efficient implementation for a Single Instruction Multiple Data (SIMD) processor (e.g., ARM NEON). We further intend to extend our scheme to allow for constant-time execution. Another interesting direction is to use our optimization strategies on a signature-based protocol.

## References

1. P. Shor, “Algorithms for quantum computation: discrete logarithms and factoring,” in *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, Nov 1994, pp. 124–134.
2. V. Lyubashevsky, C. Peikert, and O. Regev, “On Ideal Lattices and Learning with Errors Over Rings,” Cryptology ePrint Archive, Report 2012/230, 2012, <http://eprint.iacr.org/>.
3. N. Göttert, T. Feller, M. Schneider, J. Buchmann, and S. Huss, “On the design of hardware building blocks for modern lattice-based encryption schemes,” *Cryptographic Hardware and Embedded Systems—CHES 2012*, vol. 7428, pp. 512–529, 2012.
4. T. Pöppelmann and T. Güneysu, “Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware,” *Progress in Cryptology—LATINCRYPT 2012*, pp. 139–158, 2012.
5. A. Aysu, C. Patterson, and P. Schaumont, “Low-cost and Area-efficient FPGA Implementations of Lattice-based Cryptography,” in *HOST*, 2013, pp. 81–86.

6. S. S. Roy, F. Vercauteren, and I. Verbauwhede, "High precision discrete gaussian sampling on FPGAs," *Selected Areas in Cryptography–SAC 2013*, pp. 383–401, 2014.
7. S. S. Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede, "Compact Ring-LWE Cryptoprocessor," *Cryptology ePrint Archive, Report 2013/866*, 2013.
8. T. Pöppelmann, L. Ducas, and T. Güneysu, "Enhanced Lattice-Based Signatures on Reconfigurable Hardware," *Cryptology ePrint Archive, Report 2014/254*, 2014, <http://eprint.iacr.org/>.
9. O. Regev, "On Lattices, Learning with Errors, Random Linear Codes, and Cryptography," in *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*, ser. STOC '05. New York, NY, USA: ACM, 2005, pp. 84–93.
10. N. C. Dwarakanath and S. D. Galbraith, "Sampling from discrete gaussians for lattice-based cryptography on a constrained device," *Applicable Algebra in Engineering, Communication and Computing*, pp. 159–180, 2014.
11. D. E. Knuth and A. C. Yao, "The complexity of nonuniform random number generation," *Algorithms and complexity: new directions and recent results*, pp. 357–428, 1976.
12. T. Oder, T. Pöppelmann, and T. Güneysu, "Beyond ECDSA and RSA: Lattice-based Digital Signatures on Constrained Devices."
13. J. W. Bos, C. Costello, M. Naehrig, and D. Stebila, "Post-quantum key exchange for the TLS protocol from the ring learning with errors problem," *Cryptology ePrint Archive, Report 2014/599*, 2014, 2014.
14. NTT Information Sharing Platform Laboratories, "PSEC-KEM Specification version 2.2," 2008.
15. R. De Clercq, L. Uhsadel, A. Van Herrewege, and I. Verbauwhede, "Ultra Low-Power implementation of ECC on the ARM Cortex-M0+." *Proceedings of the The 51st Annual Design Automation Conference–DAC 2014*, 2014.