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. Our implementation beats all known software implementations of ring-LWE encryption by a factor of at least 7. We further show that our scheme beats ECC-based public-key encryption schemes by at least one order of magnitude. At medium-term security we require 121 166 cycles per encryption and 43 324 cycles per decryption, while at a long-term security we require 261 939 cycles per encryption and 96 520 cycles per decryption. Gaussian sampling is done at an average of 28.5 cycles per sample.

1 Introduction

Our present day public-key cryptography schemes use number theoretic problems such as factoring or discrete logarithms for providing digital signatures, key-exchange, and confidentiality. For sufficiently large key sizes, these public-key cyptosystems are practically unbreakable with present day computers, super computers, and special hardware clusters. In 1994 Peter Shor proposed a quantum algorithm [1] for integer factorization. A modified version of Shor's algorithm 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. Therefore, it is imperative that efficient quantum secure public-key algorithms are used when quantum computers become a reality.

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 describe an efficient software implementation of an encryption scheme based on the ring-LWE problem [2]. The ring-LWE encryption scheme is computationally 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 decisions are

key for implementing polynomial multiplication. 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. 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. While several high-level software implementations of the ring-LWE based encryption scheme exist [3, 9], there are few efficient software implementations on resource constrained microcontrollers [10–12]. 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[1] 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 108 147 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 analyse the bottlenecks in standard algorithms, and provide solutions to overcome these, with a specific focus on the target platform. Afterwards, we present our results and finally we draw a conclusion.

2 Mathematical Background

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

[1] Source code available at http://homes.esat.kuleuven.be/~ssinharo

2.1 The ring-LWE Encryption Scheme

In the ring-LWE problem [13] two polynomials a and s are chosen uniformly from a polynomial ring $R_q = Z_q[\mathbf{x}]/\langle f \rangle$ where f is an irreducible polynomial of degree $n-1$. An error polynomial 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 σ. The ring-LWE distribution $A_{s,\mathcal{X}}$ over $R_q \times R_q$ consists of tuples (a,t) where $t = a \cdot s + e \in R_q$. Given a polynomial number of sample pairs (a,t) from $A_{s,\mathcal{X}}$, it is very difficult to find s. This problem is known as the *search ring-LWE* problem. An encryption scheme based on the ring-LWE problem was proposed in [2]. In this paper, we use an efficient version of the encryption scheme [7] which minimizes the number of costly NTT operations. We denote NTT of a polynomial z by $\tilde{z}$.

1. $KeyGeneration(\tilde{a})$: Two polynomials r_1 and r_2 are sampled from $\mathcal{X}_\sigma$ using a discrete Gaussian sampler. The following computations are performed.

$$\tilde{r_1} \leftarrow NTT(r_1);\ \tilde{r_2} \leftarrow NTT(r_2);\ \tilde{p_1} \leftarrow \tilde{r_1} - \tilde{a} * \tilde{r_2}$$

   The private key is $\tilde{r_2}$ and the public key is $(\tilde{a}, \tilde{p})$.
2. $Encryption(\tilde{a}, \tilde{p}, m)$: The input message m is encoded to a polynomial $\bar{m} \in R_q$. Error polynomials $e_1, e_2, e_3 \in R_q$ are generated from $\mathcal{X}_\sigma$ using a discrete Gaussian sampler. The following computations are performed to compute the ciphertext $(\tilde{c_1}, \tilde{c_2})$.

$$\tilde{e_1} \leftarrow NTT(e_1); \quad \tilde{e_2} \leftarrow NTT(e_2)$$
$$(\tilde{c_1}, \tilde{c_2}) \leftarrow \big(\tilde{a} * \tilde{e_1} + \tilde{e_2};\ \tilde{p} * \tilde{e_1} + NTT(e_3 + \bar{m})\big)$$

3. $Decryption(\tilde{c_1}, \tilde{c_2}, \tilde{r_2})$: The inverse NTT is performed to compute $m' = INTT(\tilde{c_1} * \tilde{r_2} + \tilde{c_2}) \in R_q$. The original message m is recovered from m' by using a decoder.

We use the parameter sets (n, q, σ) from [3], namely $P_1 = (256, 7\ 681, 11.31/\sqrt{2\pi})$ and $P_2 = (512, 12\ 289, 12.18/\sqrt{2\pi})$ that have medium-term and long-term security 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 σ of the distribution. A detailed comparative analysis of the sampling algorithms can be found in [14]. We use the Knuth-Yao algorithm [15] 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 [14, 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 $R_q = Z_q[\mathbf{x}]/\langle f \rangle$ with $f = x^n + 1$, and $n = 2^k$ and q a prime such that $q \equiv 1 \mod 2n$. Such a choice of the parameters allows us to use an n-point NTT instead of a $2n$-point NTT during a polynomial multiplication in R_q. However there is an additional scaling overhead associated with the negative wrapped convolution. This technique is known as 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 operation samples three error polynomials, computes three forward NTTs, 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 component has a big performance impact on the ring-LWE cryptosystem. In our implementation, each encryption operation requires $3n$ samples.

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.

Knuth-Yao Algorithm This algorithm [15] 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 of the tree 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 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, 14].

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 5 995 bits in our design for $s = 11.31$. The following discusses several optimization

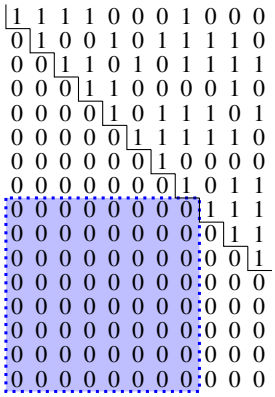

Fig. 1. The partial contents of a probability matrix. We avoid storing zero words (represented by the blue box) as they do not need to be processed.

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 desireable to only process and store the region of the matrix that contains 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.

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 zero words 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 another solution 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 within the first few levels of the tree is already very high. 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.

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 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.

Input: Probability matrix P, random number r, modulus q
Output: Sample value s

```
1  index ← r&255
2  r ← r ≫ 8
3  s ← LUT1[index]
4  if msb(s) = 0 then
5      if (r&1) = 1 then
6          return q − s
7      else
8          return s
9  d ← s&7
10 col ← 0
11 for col ← 8 to MAXCOL do
12     d ← 2d + (r&1)
13     r ← r ≫ 1
14     for row ← MAXROW downto 0 do
15         d ← d − P[row][col]
16         if d = −1 then
17             if (r&1) = 1 then
18                 return q − row
19             else
20                 return row
21 return 0
```

Algorithm 2: Knuth-Yao Sampling with an LUT

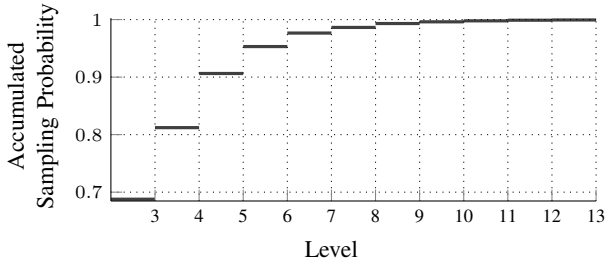

Fig. 2. The probability that the Knuth-Yao sampling algorithm finds a terminal node within x levels for a Gaussian distribution with $\sigma = 11.31/\sqrt{2\pi}$.

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 ← BitReverse(a)
2  for m = 2 to n/2 step 2m do
3      w_m ← primitiveroot(m)                          //  w_m = w_n^{n/m}
4      w = √w_m
5      for j = 0 to m/2 − 1 do
6          for k = 0 to n − 1 step m do
7              t ← w · A[m/2 + j + k] mod q
8              u ← A[j + k]
9              A[j + k] ← u + t mod q
10             A[m/2 + j + k] ← u − t mod q
11         w ← w · w_m
```

Algorithm 3: Negative-Wrapped Iterative Fwd NTT

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 $q = 7\ 681$ and $q = 122\ 89$, and therefore fits into halfwords. On the target platform a memory access requires 2 cycles, regardless of whether it is to a halfword or a full 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. 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,

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

$A \leftarrow BitReverse(a)$
$l \leftarrow 1$
for $m = 2$ *to* $n/2$ *step* $2m$ **do**
    $w_m \leftarrow primitive\_root[l]$ // $w_m = w_n^{n/m}$
    $w \leftarrow primitive\_root[l+1]$ // $w = \sqrt{w_m}$
    for $j = 0$ *to* $m-1$ *step* 2 **do**
        for $k = 0$ *to* $n/2-1$ *step* $2m$ **do**
            $(u_1, t_1) \leftarrow (A[j+k], A[j+k+1])$
            $(u_2, t_2) \leftarrow (A[m+j+k], A[m+j+k+1])$
            $t_1 \leftarrow w \cdot t_1 \bmod q$
            $t_2 \leftarrow w \cdot t_2 \bmod q$
            $A[j+k] \leftarrow u_1 + t_1 \bmod q$
            $A[j+k+1] \leftarrow u_2 + t_2 \bmod q$
            $A[m+j+k] \leftarrow u_1 - t_1 \bmod q$
            $A[m+j+k+1] \leftarrow u_2 - t_2 \bmod q$
            $w \leftarrow w \cdot w_m$
    $l \leftarrow l + 1$
$w_m \leftarrow primitive\_root[l]$ // $w_m = w_n$
$w \leftarrow primitive\_root[l+1]$ // $w = \sqrt{w_m}$
for $k = 0$ *to* $n/2-1$ **do**
    $(u_1, t_1) \leftarrow (A[2k], A[2k+1])$
    $t_1 \leftarrow w \cdot t_1 \bmod q$
    $A[2k] \leftarrow u_1 + t_1 \bmod q$
    $A[2k+1] \leftarrow u_1 - t_1 \bmod q$
    $w \leftarrow w \cdot w_m$

Algorithm 4: Memory Efficient Negative-Wrapped Fwd NTT

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 is clocked at 168 MHz, and can perform other computations while waiting 12 cycles between each random number request. According to [16] 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 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. After all the fresh bits are used, the register becomes one, and a new random number is fetched.

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 10 000 runs.

4.1 Performance Results

Table 1 shows that the NTT-based operations are at least 123% more for expensive for P_2 when compared with P_1, while Knuth-Yao sampling requires an average of 28.5 cycles per sample for both parameter sets. 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 seperate 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 require 27.5% less cycles than [10], and one order of magnitude less cycles than [11]. The scheme proposed by [9] has parameters $n = 1024, \sigma = 8/\sqrt{2\pi}$ while providing medium-term security. The

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

$P_1 = (256, 7\,681, 11.31/\sqrt{2\pi})$, $P_2 = (512, 12\,289, 12.18/\sqrt{2\pi})$

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

$P_1 = (256, 7\,681, 11.31/\sqrt{2\pi})$, $P_2 = (512, 12\,289, 12.18/\sqrt{2\pi})$

same security level can be provided with P_1 which uses three times fewer coefficients. Our result for NTT transformation is 72% faster than the Cortex-M4F based [10], while being a factor 15.8 slower than [17], which uses a much more powerful desktop processor. Our Gaussian sampler is faster than any other software-based Gaussian sampler by a factor of 7.6.

Table 4 shows that our implementation of the encryption and decryption operations are faster than those of [12] by a factor of 7.25 and 5.22 respectively.

In order to draw a comparison of our cryptosystem with more estabelished public-key cryptosystems, we compare our implementation to an existing ECC implementation. We consider Elliptic Curve Integrated Encryption Scheme (ECIES) [18], which has few known software implementations. The cycle cost of the encryption operation is estimated based on the most performance hungry operations, namely two point multiplications. As our scheme provides medium-term security, we therefore compare our results to an ECC implementation of 233-bits. In [19] the authors found that a 233-bit point multiplication requires 2 761 640 cycles on the ARM Cortex-M0+. This means that an ECIES implementation would require roughly 5 523 280 cycles for encryption. Therefore, our implementation is faster than ECIES by more than one order of magnitude.

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

Operation	Platform	Cycles	*n*, *q*, σ
NTT transform [17]	Core i5-3210M	4 480	P_5
NTT transform [17]	Core i3-2310	4 484	P_5
NTT multiplication [17]	Core i5-3210M	16 052	P_5
NTT multiplication [17]	Core i3-2310	16 096	P_5
NTT transform [11] [1]	ATxmega64A3	2 720 000	P_3
NTT transform [10]	Cortex-M4F	122 619	P_3
NTT multiplication [10]	Cortex-M4F	508 624	P_3
NTT transform [12]	ARM7TDMI	260 521	P_3
NTT transform [12]	ATMega64	2 207 787	P_3
NTT transform *	Cortex-M4F	71 090	P_2
NTT multiplication *	Cortex-M4F	237 803	P_2
NTT transform [12]	ARM7TDMI	109 306	P_1
NTT transform [12]	ATMega64	754 668	P_1
NTT transform [11] [1]	ATxmega64A3	1 216 000	P_1
NTT multiplication [9]	Core i5 4570R	342 800	P_4
NTT transform *	Cortex-M4F	31 583	P_1
NTT multiplication*	Cortex-M4F	108 147.0	P_1
Gaussian sampling [12]	ARM7TDMI	218.6	P_3
Gaussian sampling [12]	ATmega64	1 206.3	P_3
Gaussian sampling [9]	Core i5 4570R	652.3	P_4
Gaussian sampling [10]	Cortex-M4F	1828.0	P_3
Gaussian sampling *	Cortex-M4F	28.5	P_1 and P_2

* This work, $P_1 = (256, 7\,681, 11.31/\sqrt{2\pi})$

$P_2 = (512, 12\,289, 12.18/\sqrt{2\pi})$, P_3 = (512, 12 289, 215)

P_4 = (1024, $2^{32} - 1, 8/\sqrt{2\pi}$), P_5 = (512, 8383489, -)

[1] Cycles estimated from reported execution time with a clock speed of 32 MHz.

Note: For NTT transform and NTT multiplication P_2 and P_3 are equivalent. The Gaussian sampling cycles are averaged for a single gaussian sampling operation.

Table 4. Comparison of ring-LWE encryption schemes.

Platform	Key Gen.	Encrypt	Decrypt	n, q, σ
ARM7TDMI [12]	575 047	878 454	226 235	P_1
ATMega64 [12]	2 770 592	3 042 675	1 368 969	P_1
ATxmega64A3 [11] [2]	-	5 024 000	2 464 000	P_1
Core 2 Duo [3] [1]	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] [1]	13 590 000	9 180 000	3 540 000	P_2
Cortex-M4F*	252 002	261 939	96 520	P_2

* This work

$P_1 = (256, 7\,681, 11.31/\sqrt{2\pi})$, $P_2 = (512, 12\,289, 12.18/\sqrt{2\pi})$

[1] Cycles estimated from reported execution time.

[2] Cycles estimated from reported execution time with a clock speed of 32 MHz.

5 Conclusion

In this work, we demonstrate that it is possible to implement a lattice-based post-quantum public-key cryptosystem on a device 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 optimizations allow us to beat all known implementations of ring-LWE encryption operations, and Gaussian sampling by a factor of at least 7. We further demonstrate 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, 121 166 and 43 324 cycles respectively for encryption and decryption at medium-term security, and 261 939 and 96 520 cycles respectively for long-term 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.

Acknowledgment

This work was supported in part by the Research Council KU Leuven: GOA TENSE (GOA/11/007), by iMinds, by the European Union Seventh Framework Programme (F P7/2007-2013) under grant agreement n. 609611 (PRACTICE). In addition, this work is supported in part by the Flemish Government, FWO G.00130.13N, FWO G.0876.14N, the Hercules Foundation AKUL/11/19, and by Intel. The second author is supported by an Erasmus Mundus PhD Scholarship. We are thankful to Vladimir Rozic for his insightful suggestions.

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.
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," in *Cryptographic Hardware and Embedded Systems CHES 2014*, 2014, vol. 8731, pp. 371–391.
8. T. Pöppelmann, L. Ducas, and T. Güneysu, "Enhanced Lattice-Based Signatures on Reconfigurable Hardware," in *Cryptographic Hardware and Embedded Systems CHES 2014*, 2014, vol. 8731, pp. 353–370.
9. 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.
10. T. Oder, T. Pöppelmann, and T. Güneysu, "Beyond ECDSA and RSA: Lattice-based Digital Signatures on Constrained Devices," *51st Annual Design Automation Conference–DAC*, 2014.
11. A. Boorghany and R. Jalili, "Implementation and Comparison of Lattice-based Identification Protocols on Smart Cards and Microcontrollers," Cryptology ePrint Archive, Report 2014/078, 2014.
12. A. Boorghany, S. B. Sarmadi, and R. Jalili, "On Constrained Implementation of Lattice-based Cryptographic Primitives and Schemes on Smart Cards," Cryptology ePrint Archive, Report 2014/514, 2014.
13. O. Regev, "On Lattices, Learning with Errors, Random Linear Codes, and Cryptography," in *37th Annual ACM Symposium on Theory of Computing*, 2005, pp. 84–93.
14. N. C. Dwarakanath and S. D. Galbraith, "Sampling from Discrete Gaussians for Lattice-based Cryptography on a Constrained Device," *Applicable Algebra in Engineering, Comm. and Computing*, pp. 159–180, 2014.
15. 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.
16. STMicroelectronics, "AN4230 Application note STM32F2xx, STM32F4xx Random Number Generation Validation using NIST Statistical Test Suite," 2013.
17. T. Güneysu, T. Oder, T. Pöppelmann, and P. Schwabe, "Software Speed Records for Lattice-based Signatures," in *Post-Quantum Cryptography*. Springer, 2013, pp. 67–82.
18. D. Hankerson, S. Vanstone, and A. J. Menezes, "Guide to Elliptic Curve Cryptography," pp. 189–190, 2004.
19. R. De Clercq, L. Uhsadel, A. Van Herrewege, and I. Verbauwhede, "Ultra Low-Power implementation of ECC on the ARM Cortex-M0+." *51st Annual Design Automation Conference–DAC*, 2014.