Efficient Implementation of Ring-LWE Encryption on High-end IoT Platform*

Zhe Liu[1,2], Reza Azarderakhsh[3], Howon Kim[4], and Hwajeong Seo[5]**

[1] College of Computer Science and Technology,
Nanjing University of Aeronautics and Astronautics, China
[2] Department of Combinatorics and Optimization, University of Waterloo, Canada
sduliuzhe@gmail.com
[3] Computer and Electrical Engineering and Computer Science (CEECS) and
I-SENSE, USA
azarderakhsh@gmail.com
[4] Pusan National University, Republic of Korea
howonkim@pusan.ac.kr
[5] Institute for Infocomm Research (I2R), Singapore
hwajeong84@gmail.com

Abstract. ARM NEON architecture has occupied a significant share of high-end Internet of Things platforms such as mini computer, tablet and smartphone markets due to its low cost and high performance. This paper studies efficient techniques of lattice-based cryptography on ARM processor and presents the first implementation of ring-LWE encryption on ARM NEON architecture. We propose a vectorized version of Iterative Number Theoretic Transform (NTT) for high-speed computation and present a 32-bit variant of SAMS2 technique, original from Liu et al. in CHES2015, for fast reduction. Subsequently, we present a full-fledged implementation of Ring-LWE by taking advantage of proposed and previous optimization techniques. Ultimately, our ring-LWE implementation requires only $145k$ clock cycles for encryption and $32.8k$ cycles for decryption for $n = 256$. These results are more than 17.6 times faster than the fastest ECC implementation on ARM NEON with same security level.

Keywords: Lightweight Implementation, Lattice-based Cryptography, ARM NEON Architecture

1 Introduction

The 32-bit ARM processor [1] is the most widely used embedded processor in almost all high-end Internet of Things (IoT) platforms, e.g., mini computer,

* This work was supported by the NSERC CREATE Training Program in Building a Workforce for the Cryptographic Infrastructure of the 21st Century (CryptoWorks21), and Public Works and Government Services Canada.
** Corresponding Author

tablets and smartphones, thanks to its low cost and high performance. ARMv6 [2] architecture introduces a small set of *SIMD* instructions, operating on multiple 16-bit or 8-bit values packed into standard 32-bit general purpose registers. This nice feature permits some certain operations can be executed in at least double speed, without using any additional computation units. From ARMv7 architecture [3], ARM introduces the Advanced SIMD extension, called "*NEON*". It extends the SIMD concept by defining groups of instructions operating on vectors stored in 64-bit `D`, doubleword, registers and 128-bit `Q`, quadword, vector registers. In the literature, many papers presented cryptography primitives on the embedded processor such as RSA [5], Elliptic Curve Cryptography (ECC) [6], pairing-based cryptography [26], AES [7] as well as lattice-based cryptography [8]. Despite recent research progress, efficient implementation of lattice based cryptographic algorithm on 32-bit ARM, in particular ARM NEON, is still an interesting and challenge topic.

1.1 Related Work

The first evaluation of cryptographic algorithm on ARM NEON architecture belonged to Bernstein and Schwabe in CHES'12 [6]. The authors showed that NEON supports high-security elliptic curve cryptography at surprised high speeds. They also summarized the useful instructions set for high-speed cryptography and presented the experimental results of NaCl library on Cortex A8 core. In 2013, Câmaraand et al. employed the `VMULL.P8` instruction to describe a novel software multiplier for performing a polynomial multiplication of 64-bit binary polynomial and obtained a fast software multiplication in the binary field $\mathbb{F}_{2^m}$ [9]. Their results emphasized the advantage of NEON for high-speed binary ECC. In SAC'13, Bos et al. in [10] presented a parallel approach to compute interleaved Montgomery multiplication, which is suitable to be computed on 2-way single instruction, multiple data platforms, e.g., ARM NEON. Seo et al. revisited the work in [10], and introduced the Cascade Operand Scanning (COS) method for multi-precision multiplication with the goal of reducing Read-After-Write (RAW) dependencies in the propagation of carries and the number of pipeline stalls [11]. As a follow up work, Seo et al. proposed a novel Double Operand Scanning (DOS) method to speed-up multi-precision squaring with non-redundant representations on SIMD architecture and investigated RSA-1024 and RSA-2048 on ARM Cortex A9 and A15 cores [5]. Besides public-key algorithm, cryptographic engineers also evaluated the impact of performance for symmetric ciphers on ARM NEON architecture. In [12], Seo et al. evaluated and proposed a parallel implementation of block cipher LEA on ARM NEON and achieved a speed up of roughly 50% compared to previous fastest implementation on ARM without NEON. In 2014, Saarinenand et al. presented the results of authenticated encryption algorithms, e.g., WHIRLBOB and STRIBOB on NEON platform [13]. In CT-RSA'15, Gouvêa and López used NEON instructions `vmull` to multiply two 64-bit binary polynomials and presented an optimized yet timing-resistant implementation of GCM over AES-128 on AR-

Mv8 [14]. Similarly, Wang et al. chose the ARM NEON platform and presented a high order masked AES implementation in [7].

Another interesting research line is to evaluate lattice-based cryptography (e.g., Ring-LWE) on different platforms. The first practical evaluations of L-WE and ring-LWE based encryption schemes were presented by Göttert et al. in CHES'12 [15]. The authors concluded that the ring-LWE based encryption scheme is faster by at least a factor of four and requires less memory in comparison to the encryption scheme based on the standard LWE problem. Sujoy et al. [30] proposed a complete ring-LWE based encryption processor that uses the Number Theoretic Transform (NTT) algorithm for polynomial multiplication. The architecture is designed to have small area and memory requirement, but is also optimized to keep the number of cycles small. Oder et al. in [8] presented the first efficient implementation of Bimodal Lattice Signature Schemes (BLISS) on a 32-bit ARM processor. The most optimal variant of their implementation cost $6M$ cycles for signing, $1M$ cycles for verification and $368M$ cycles for key generation, respectively, at a medium-term security level. In DATE'15, De Clercq et al. in [18] implemented ring-LWE encryption scheme on the identical ARM processors, they investigated acceleration techniques to improve the sampler based on the architecture of the microcontroller. Namely, the platform built-in True Random Number Generator (TRNG) is used to generate random numbers. As a result, their implementation required $121K$ cycles per encryption and $43.3K$ cycles per decryption at medium-term security level while $261K$ cycles per encryption and roughly $96.5K$ cycles per decryption for long-term security level. The first time when a lattice-based cryptographic scheme was implemented on an 8-bit processor belonged to Boorghany et al. in [19, 20]. The authors evaluated four lattice-based authentication protocols on both 8-bit AVR and 32-bit ARM processors. Very recently, Pöppelmann et al. [21] and Liu et al. [22] studied and compared implementations of Ring-LWE encryption and the Bimodal Lattice Signature Scheme (BLISS) on an 8-bit platform and presents efficient ring-LWE results, respectively.

1.2 Motivation

Lattice-based cryptography is often considered a premier candidate for realizing post-quantum cryptosystems [32]. Its security relies on worst-case computational assumptions in lattices that will remain hard even for quantum computers. Although some work has been done, the design and implementation of post-quantum cryptosystems and protocols is still a big challenge. For example, it has been recognized in a recent Microsoft Research project [23] and the Canada "CryptoWorks21" project [24] as well as the European project "PQCrypto" [25]. However, we were surprised to find there exists no previous work about evaluating Ring-LWE encryption or signature scheme on ARM NEON architecture, which was reported, in 2014, to be present in 95 % of mini computers, tablets and smartphones [14]. This raises one interesting question that how well this "cryptosystems of the future" are suited for today's most widely used mobile devoices and one aspect of this question is the performance and memory consumption of lattice-based cryptosystems on 32-bit ARM NEON platform. In this paper, we are going to fill the implementation gap and give our answer for this open problem.

1.3 NEON PQCryto

This paper studies efficient techniques of lattice-based cryptography and presents an efficient ring-LWE implementation on ARM NEON architecture, called "NEON PQCrypto". NEON PQCrypto includes support for core ring-LWE functions necessary to implement most popular ring-LWE based schemes, i.e. encryption scheme. In particular, NEON PQCrypto supports the computation of two most important operations:

- We propose parallel Number Theoretic Transform (NTT) to reduce the execution time for coefficient multiplication. This method introduces 4-way NTT computations over SIMD architecture.
- We introduce the 32-bit wise Shifting-Addition-Multiplication-Subtraction-Subtraction (SAMS2) approach for reduction operation. The approach replaced the expensive division operation into shifting, addition and multiplication operations.
- We exploit the incomplete arithmetic for representing the coefficients and perform the reduction operation in a lazy fashion. This technique avoids one time of subtraction in each reduction stage.
- Efficient implementation of Gaussian distribution sampler. We employ Knuth-Yao sampler, LUT and byte-scanning methods. Our implementation exploits the PRNG based on block cipher, which achieved the high performance with parallel and pipelined techniques.

NEON PQCrypto achieves high performance without compromising security. By a combination of proposed and previous optimizations (e.g., Incomplete arithmetic), we present high speed implementations of ring-LWE encryption for 128-bit security level on ARM NEON. For 128-bit security level, it only requires $145,200$ and $32,800$ clock cycles for encryption and decryption. The decryption result outperforms the previous ARM implementation (without NEON) by a factor of 1.32. When compared with ECC implementation with same security level, our ring-LWE is 17.6 faster on identical platform.

The rest of this paper is organized as follows. In the next section, we review the background of Ring-LWE. In Section 2, we introduce the optimization techniques for Ring-LWE on ARM-NEON processors. In particular, we propose several optimization techniques to reduce the execution time in SIMD architecture. In Section 4, we report the implementation results and compare with the state-of-the-art implementations.

2 Implementation of NTT

In this section, we describe several optimization techniques to reduce the execution time of Ring-LWE on ARM NEON architectures. We choose the parameter sets (n, q, σ) with $(256, 7681, 11.31/\sqrt{2\pi})$ for security level of 128-bit. These parameter sets were also used in most of the previous hardware implementations, e.g., [15, 30] and software implementations, e.g., [19, 20, 18, 21, 22]. This also helps us to compare our work with previous works.

2.1 Vectorized Iterative algorithm

Previous implementations on RISC processors, e.g., [18, 21, 22], executed the NTT computation in a sequential fashion. Namely, the coefficient multiplication is performed in sequence in each iteration. In the following, we propose a vectorized variant of iterative NTT algorithm, which significantly speeds up the execution time of NTT operations on ARM NEON. The core idea is to take the advantages of SIMD instruction set and implement NTT computation in a hybrid fashion. In particular, when the number of consecutive coefficient multiplication satisfies the minimum width of SIMD, we compute the SIMD based vectored computations. Otherwise, when the number of consecutive coefficient multiplication is smaller than width of SIMD, we simply adopt the sequential fashion in ARM instruction.

The vectorized variant of NTT computation is given in Algorithm 1. As shown in steps 3 to 12, in the innermost k loop, the index value of consecutive coefficient multiplication between two coefficients $(a[k+j], a[k+j+i/2])$ are only 1 and 2 for $i = 2$ and $i = 4$ cases, respectively. Thus, we conduct these coefficient multiplication in a sequential way. On the other hand, the cases $i > 4$ have at least four consecutive coefficient multiplication operations, we perform these coefficient multiplications in a parallel fashion. Specifically, we first conduct the whole twiddle factors (ω) in consecutive array form (steps 15 $\sim$ 18). Observing that the twiddle factors are fixed variables, we simply compute these values off-line and store them into a look-up table. Thereafter, in steps 19 $\sim$ 28, the coefficient variables are loaded into registers in consecutive array form such as U_{array}, V_{array} and ω_{array}. We conduct the four different modular multiplications with $\omega_{array}[p : p + 3] \cdot a[k + j + i/2 : k + j + 3 + i/2]$. After then, the pointer address of p increases by 4 (i.e. the SIMD width)[6]. Finally, the multiple number of coefficient variables are added and subtracted each other, simultaneously.

2.2 Parallel Coefficient Multiplication

The coefficient multiplication is one of the most expensive operations of NTT computation, since each NTT computation requires $\frac{n}{2}log_2 n$ coefficient multiplications. In our implementation, the coefficient is at most 13-bit long, which can be kept in one 32-bit ARM register. As mentioned before, it is possible to store two coefficients into one register as De Clercq did in [18]. However, we decide to store only one coefficient in a register since the product of a coefficient multiplication can be (at most) 26-bit long. In this case, storing 26-bit in a register will result in some extra cost to extract the 13-bit operand out of 26-bit before

[6] For AVX256 and AVX512, we can extend to 8 and 16 respectively.

Algorithm 1: Vectorized Iterative Number Theoretic Transform

Require: A polynomial $a(x) \in \mathbb{Z}_q[x]$ of degree $n-1$ and n-th primitive $\omega \in \mathbb{Z}_q$ of unity
Ensure: Polynomial $a(x) = NTT(a) \in \mathbb{Z}_q[x]$

$a = BitReverse(a)$ {`BitReverse computation`}
for i from 2 by $i = 2i$ to n **do**
  $\omega_i = \omega_n^{n/i}$, $\omega = 1$ {`Setting twiddle factors`}
  if $i = 2$ or $i = 4$ **then**
    for j from 0 by 1 to $i/2 - 1$ **do**
      for k from 0 by i to $n-1$ **do**
        $U = a[k+j]$ {`sequential computations`}
        $V = \omega \cdot a[k+j+i/2]$ {`single multiplication`}
        $a[k+j] = U + V$ {`single addition`}
        $a[k+j+i/2] = U - V$ {`single subtraction`}
      end for
      $\omega = \omega \cdot \omega_i$ {`computation of single twiddle factors`}
    end for
  else
    $\omega_{array}[0] = \omega$
    for p from 1 by 1 to $i/2 - 1$ **do**
      $\omega = \omega \cdot \omega_i$, $\omega_{array}[p] = \omega$ {`computations of multiple twiddle factors`}
    end for
    for j from 0 by i to $n-1$ **do**
      $p = 0$
      for k from 0 by 4 to $i/2 - 1$ **do**
        $U_{array} = a[k+j : k+j+3]$ {`parallel computations`}
        $V_{array} = \omega_{array}[p : p+3] \cdot a[k+j+i/2 : k+j+3+i/2]$ {`multiple multiplications`}
        $p = p + 4$ {`index increment`}
        $a[k+j : k+j+3] = U_{array} + V_{array}$ {`multiple additions`}
        $a[k+j+i/2 : k+j+3+i/2] = U_{array} - V_{array}$ {`multiple subtractions`}
      end for
    end for
  end if
end for
return a

performing the next step. For ARM NEON, the 128-bit `Q` register is able to store four 32-bit wise variables. We load four different aligned consecutive variables and then conduct the four different multiplications with one single vectorized `vmull` instruction.

2.3 Fast Reduction

In NTT computation, the majority of the execution time is spent on computing reduction operation since it is performed in the innermost k-loop (three times nested). Thus, fast reduction operation is an essential for high-speed implementation of NTT algorithm. Our implementation chooses the prime modulus $q = 7681$ (i.e. `0x1e01` in hexadecimal representation).

One of the efficient method for reduction belongs to SAMS2 method, which was originally proposed in an 8-bit AVR implementation [22]. This method has optimized the register usages and computation complexity. Since it replaces expensive operation (e.g., `division`) with relatively cheaper instructions (e.g., `addition`, `shifting`, `multiplication`), the execution time is significantly improved. However, compared to RISC architecture, ARM NEON has more distinguished features. First, the length of a word is bigger, i.e. 32-bit per word. This feature allows us to readily compute the 13-bit wise multiplication in single instruction and up-to 31-bit shifting can be performed in single cycle. Second, ARM NEON supplies SIMD instructions, which perform multiple operations (up-to four 32-bit multiplications) in parallel using single instruction. Therefore, we have craftily design an enhanced variant of SAMS2 method on ARM NEON architecture.

We propose an optimized 32-bit wise SAMS2 reduction technique for performing the mod 7681 operation. The SAMS2 method is introduced in [22] and the method is highly optimized in 8-bit AVR processors in terms of register utilization and the number of operations. However, ARM NEON processor has two distinguished features over 8-bit AVR. First the processor provide 32-bit word size. We can readily compute the 13-bit wise multiplication in single instruction and up-to 31-bit shift is available within single cycle. Second multiple number of operations are conducted at once by exploiting SIMD instructions. With these features in mind, we redesign the original SAMS2 for ARM NEON architecture.

This main idea of SAMS2 is to first estimate the quotient of $t = \frac{a}{q}$, and then perform the subtraction $a - t \cdot q$ where the value of t is $(a \gg 13) + (a \gg 17) + (a \gg 21)$. The reduction process consists of four different basic operations, namely, 32-bit wise Shifting $\rightarrow$ Addition $\rightarrow$ Multiplication $\rightarrow$ Subtraction $\rightarrow$ Subtraction (SAMS2). As shown in Figure 1, one `Q` register consists of four 32-bit registers. Among them, multiplication over one 32-bit long register ($r0$, a quarter of NEON register) is described in detail. Since remaining three 32-bit registers and $r0$ register is packed in the `Q` register, four identical SAMS2 method is conducted simultaneously. The colorful parts mean that the storage has been occupied while the white part is not. The reduction with 7681 using SAMS2 approach can be performed as follows:

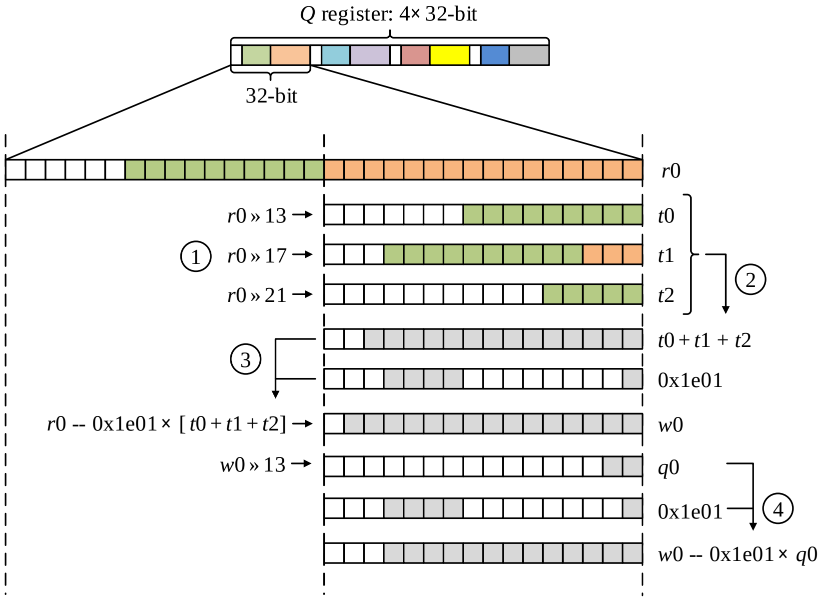

Fig. 1. Fast reduction operation with 32-bit wise SAMS2 method for $q = 7681$. ①: shifting; ②: addition; ③: multiplication & subtraction; ④: multiplication and subtraction.

1. Shifting. We right shift $r0$ by 13-bit, 17-bit and 21-bit. This outputs results $t0$, $t1$ and $t2$.
2. Addition. We then perform the addition of $t0 + t1 + t2$.
3. Multiplication and Subtraction. The third step is to multiply the constant `0x1e01` by $(t0 + t1 + t2)$, which is a 16×13-bit multiplication and then subtract the product from $r0$.
4. Multiplication and Subtraction. However, the result we get in step 3 may still be larger than $p = 7681$, thus, we do the correction by subtracting the modulus p multiplied by intermediate result larger than 13-bit.

In Algorithm 2, pseudo codes for vectorized NTT computation with constant time reduction is described. Firstly four coefficients (`q3`) and four twiddle factors (`q1`) are multiplied in Step 1. From Steps 2 $\sim$ 6, the intermediate results are shifted to right by 13, 17 and 21-bit and accumulated. In Step 7, we conduct multiplication with modulo (`d0[0]`) and intermediate result (`q4`). This process is readily available by using `vmls` instruction, which conducts four different multiplication and then subtract operations from the destination (`q3`). From Steps 8 $\sim$ 9, results over 13-bit are shifted and then reduced once again. In case of coefficient addition, two operands (`q2` and `q3`) are added and then one time of reduction is follows in Steps 10 $\sim$ 12. For subtraction, we firstly calculate the value (4×modulus) in Step 13. After then the value is added to operand (`q2`). Since the operand (`q3`) is placed within $[0, 2^{\lceil log_2 p \rceil}]$, the subtraction in Step 15 does not introduce negative values. Conveniently we can conduct one time of reduction that is same with addition case.

2.4 Coefficient Addition and Subtraction

We employ the incomplete arithmetic to represent the intermediate result of coefficient. Our implementation of coefficient addition works as follows. We first perform a normal coefficient addition, after that, we conduct the 13-bit shift to the right and perform the modular reduction by multiplying the modulus with the shifted results. Similarly, for incomplete coefficient subtraction, we first perform a normal coefficient subtraction, after that, we add $4 \times p$ and then conduct the 13-bit shift to the right and perform the modular reduction by multiplying the modulus with the shifted results. This approach replaces the subtraction into addition which avoids the negative cases.

Algorithm 2: Assembly codes of vectorized NTT for innermost loop

Require: Eight 32-bit coefficients $A[0:3]$(q2), $B[0:3]$(q3), ω(q1), modulo(q0).

Ensure: Eight 32-bit results C(q5,q10).

```
vmul.i32 q3, q3, q1 {Four 32-bit wise parallel multiplications}
vshr.u32 q4, q3, #13                        {SAMS2 ①:shifting}
vshr.u32 q5, q3, #17                        {SAMS2 ①:shifting}
vshr.u32 q6, q3, #21                        {SAMS2 ①:shifting}
vadd.i32 q4, q4, q5                         {SAMS2 ②:addition}
vadd.i32 q4, q4, q6                         {SAMS2 ②:addition}
vmls.i32 q3, q4, d0[0]  {SAMS2 ③:multiplication & subtraction}
vshr.u32 q4, q3, #13                        {SAMS2 ④:shifting}
vmls.i32 q3, q4, d0[0]  {SAMS2 ④:multiplication & subtraction}
vadd.i32 q5, q2, q3        {coefficient addition ①: addition}
vshr.u32 q4, q5, #13       {coefficient addition ②: shifting}
vmls.i32 q5, q4, d0[0] {coefficient addition ③: multiplication
   & subtraction}
vshl.i32 q1, q0, #2     {coefficient subtraction ①: 4×modulo}
vadd.i32 q2, q2, q1      {coefficient subtraction ②: 4×modulo
   addition}
vsub.i32 q10, q2, q3  {coefficient subtraction ③: subtraction}
vshr.u32 q14, q10, #13   {coefficient subtraction ④: shifting}
vmls.i32 q10, q14, d0[0]           {coefficient subtraction ⑤:
   multiplication & subtraction}
```

2.5 Look-Up Table for the Twiddle Factors

A straightforward computation of $\omega = \omega \cdot \omega_i$ on-the-fly needs to perform $n-1$ times of coefficient modular multiplications. Both of the computations of the power of ω_n in i-loop and twiddle factor $\omega = \omega \cdot \omega_i$ in j-loop can be considered as fixed costs. We can pre-compute the all twiddle factors ω into RAM which is similar to the technique used in [22]. Fortunately, ARM-NEON process provides huge RAM size ($1 \sim 4$GB) and the storing all the intermediate twiddle factors ω into RAM is very cheap approach. We only need to transfer the twiddle factor that is required for the current iteration. For vectorized operation, whole twiddle factors are stored in aligned vector form which ensures efficient memory access pattern and vector operations as well.

3 Implementation of Gaussian Sampler

Both key-generation and encryption require the operation of Gaussian samplers, thus efficient implementation of the Knuth-Yao sampler is another important factor for a high-speed ring-LWE encryption scheme. In this section, we describe optimization techniques that can be used to reduce the execution time of the Knuth-Yao sampler on ARM NEON processors.

3.1 Pseudo-Random Number Generation

Gaussian sampler needs random sequences. As ARM NEON does not support the build-in TRNG, our implementation adopts the PRNG algorithm, which runs the block cipher in counter mode, i.e. it encrypts successive values of an incrementing counter. There are a number of lightweight block ciphers that can be used for generating random numbers. Recently, ATmel company introduced AES peripheral based PRNG [34]. This module is available in modern XMega products which can be used for high performance of PRNG and Seo et al. in WF-IoT'14 implemented the AES accelerator based PRNG implementation on XMega processor [35].

Our implementation exploits the LEA block cipher [33] for random generations. LEA is a new lightweight and low-power encryption algorithm. This algorithm has a certain useful features which are especially suitable for parallel hardware and software implementations, i.e., simple ARX operations, non-S-BOX architecture, and 32-bit word size. We follow the parallel implementation of LEA introduced by [12]. ARM NEON processor supports 128-bit register which consists of four different 32-bit registers. By assigning four different 128-bit wise data into four 128-bit registers, we can conduct four different encryption computations in parallel fashion. Finally, the implementation results achieved 10.06 cycle/byte for encryption by computing four different encryptions at once.

3.2 Look-up Table for Probability matrix

In order to ensure a precision of 2^{-90} for dimension $n = 256$, the Knuth-Yao algorithm is suggested to have a probability matrix P_{mat} of 55 rows and 109 columns [18]. On 32-bit ARM processor, we stored each 55-bit column in two words, where each word size is 32-bit long. In this case, 9-bit is wasted per each column and the probability matrix only occupies 872 bytes in total.

3.3 Byte-wise scanning

The bit-scanning operation requires to check each bit and decreases the distance (d) whenever the bit is set. Instead of executing the scanning operation in a bit-level, we perform the scanning operation in a byte-wise fashion [22] . The byte-wise scanning method counts the number of bits in the byte and decreases the distance by the number of bits. Since the byte-wise method does not conduct the subtraction by each bit, it only requires eight additions, one subtraction and one conditional branch statements, saving seven conditional branch statements at the cost of one subtraction rather than bit-wise scanning.

3.4 Efficiently skip the consecutive leading zeros

The probability matrix includes an occurrence of consecutive leading zeros. In order to skip the consecutive leading zeros, we conduct the simple comparison between zero and bit counter. One time of byte comparison can decide that the probability matrix has leading zeros or not by byte wise. This approach can skip one byte-scanning at the cost of one conditional branch statement, if the counter is zero.

3.5 Look-up table in DDG tree

We exploit the Look-Up Table (LUT) approaches proposed in [18] into byte-wise scanning implementations. First, we perform sampling with an 8-bit random number as an index to the LUT in the first 8 levels for a Gaussian distribution with $\sigma = 11.31/\sqrt{2\pi}$. If the most significant bit of the lookup result is reset, then the algorithm returns the LUT result successfully. Otherwise, the most significant bit of the LUT result is one, then a LUT failure occurs, and the next level of sampling will execute. Similarly, a second LUT will be used for level $9 \sim 13$ in the same Gaussian distribution. Since two levels of LUT method shows about 99% hit ratio, this is the computation efficient approach.

4 Performance Evaluation and Comparison

4.1 Experimental Platform

The ARM Cortex A9 is full implementations of the ARMv7 architecture including NEON engine. Register sizes are 64-bit and 128-bit for double(`D`) and quadruple(`Q`) word registers, respectively. Each register provides short bit size computations such as 8-bit, 16-bit, 32-bit and 64-bit. This feature provides more precise operation and benefits to various word size computations. We complied our implementation with speed optimization option -O3. In order to obtain accurate timings, we ran each operation at least 1000 times and calculated the average cycle count for one operation.

Table 1. Performance comparison of software implementation of Number Theoretic Transform on different processors.

Implementations	`NTT/FFT`
8-bit AVR processors, e.g., ATxmega64, ATxmega128:	
Boorghany et al. [20]	1,216,000
Boorghany et al. [19]	754,668
Pöppelmann et al. [21]	334,646
Liu et al. [22]	193,731
32-bit ARM processors, e.g., Cortex-M4F, ARM7TDMI:	
Boorghany et al. [19]	109,306
DeClercq et al. [18]	31,583
32-bit ARM-NEON processors, e.g., Cortex-A9:	
This work	**25,574**

Table 2. Performance comparison of software implementation of lattice-based cryptosystems on different processors (clock cycle 10^3).

Implementations	`NTT/FFT`	`Sampling`	`Gen`	`Enc`	`Dec`
Implementations on 8-bit AVR processors, e.g., ATxmega64, ATxmega128:					
Boorghany et al. [20]	1,216.0	N/A	N/A	5,024.0	2,464.0
Boorghany et al. [19]	754.7	N/A	2,770.6	3,042.7	1,369.0
Pöppelmann et al. [21]	334.6	N/A	N/A	1,315.0	381.3
Liu et al. [22]	193.7	26.8	589.9	671.6	275.6
Implementations on 32-bit ARM processors:					
DeClercq et al. [18]	31.6	7.3	117.0	121.2	43.3
Implementations on 32-bit ARM-NEON processors, e.g., Cortex-A9:					
This work	**25.5**	**18.8**	**123.2**	**145.2**	**32.8**

4.2 Experimental Results

Table 2 summarizes the execution times of Number Theoretic Transform, Gaussian sampling, key generation, encryption and decryption of the proposed implementation for medium-term security level. Our parallel NTT operations only require $25,574$ clock cycles for 128-bit security level. We also compare software implementations of Number Theoretic Transform on different processors. For

Table 3. Comparison of Ring-LWE encryption schemes with RSA and ECC on ARM NEON processors (Enc and Dec in clock cycles)

Implementation	Scheme	Enc	Dec
Seo et al. [5]	RSA-2048	535,020	20,977,660
Bernstein et al. [6]	ECC-255	1,157,952	578,976
This work	LWE-256	**145,200**	**32,800**

the 8-bit AVR and 32-bit platforms, the previous works [19–21, 18, 8] and our implementations adopt the same parameter sets. The most suitable comparison is 32-bit ARM implementations, since the target processor shares similar ARM instructions of ARMv7. A comparison of our implementation (parallel) with De Clercq's implementation (sequential) clearly show the advantage of NEON engine, roughly 19 % enhancements can be achieved for NTT computation. For Gaussian sampling, our current implementation is slower than the work in [18]. This can be explained that the authors in [18] adopted build-in true random number generator (in hardware) and our implementation simply adopts the pseudo random number generator using software implementation. For 128-bit security level, our ring-LWE implementation requires only 145, 200 clock cycles for encryption and 32, 800 cycles for decryption. Comparing with the implementation on ARM Cortex M4 in [18], the key generation and encryption are slightly slower while the decryption is faster.

Table 3 compares the results of our ring-LWE encryption scheme with some classical public-key encryption algorithms, in particular recent RSA and ECC implementations for ARM NEON platform. The to-date fastest RSA software for an ARM NEON processor was reported in [5]; it achieves an execution time of approximately 20.9 M clock cycles for RSA-2048 decryption at the 96-bit security level. For comparison, our LWE-256 implementation requires only 32.8 k cycles for decryption, which is more than 639 times faster despite a much higher (i.e. 128-bit) security level. The fastest implementation ECC software implementations on NEON belongs to Bernstein et al.[6]. For comparison, our implementation of ring-LWE is roughly 8 times faster for encryption and 17.6 for decryption.

5 Conclusion

This paper presented several optimizations for efficiently implementing ring-LWE encryption scheme on high-end IoT platform, 32-bit ARM NEON architecture. In particular, we proposed three optimizations to accelerate the execution time of the NTT-based polynomial multiplication. A combination of these optimizations results in a very efficient NTT computation, which is 19% faster than the previous best implementation. All of these achieved results set new speed records for ring-LWE encryption implementation on 32-bit ARM NEON platforms. Finally, a comparison of our implementation with traditional public-key cryptography (i.e. RSA, ECC) also sheds some new light on practical application of ring-LWE on 32-bit ARM NEON processors.

References

1. ARM architectures. `http://www.arm.com/products/processors/index.php`.
2. ARM Limited, Cortex-V6 technical reference manual. Available in `http://ecee.colorado.edu/ecen3000/labs/lab3/files/DDI0419C_arm_architecture_v6m_reference_manual.pdf`
3. ARM Limited, Cortex-V7 technical reference manual. Available in `https://web.eecs.umich.edu/~prabal/teaching/eecs373-f10/readings/ARMv7-M_ARM.pdf`.
4. Introducing NEON Development Article. Available in `https://software.intel.com/sites/default/files/m/b/4/c/DHT0002A_introducing_neon.pdf`
5. H. Seo, Z. Liu, J. Großschädl and H. Kim Efficient Arithmetic on ARM-NEON and Its Application for High-Speed RSA Implementation. Available in IACR ePrint `http://eprint.iacr.org/2015/465.pdf`, 2015.
6. D.J. Bernstein and P. Schwabe. NEON crypto. *Cryptographic Hardware and Embedded Systems –CHES 2012*, pages 320–339, Springer Berlin Heidelberg, 2012.
7. J. Wang, P.K. Vadnala, J. Großschädl and Q. Xu. Higher-Order Masking in Practice: A Vector Implementation of Masked AES for ARM NEON. *Topics in Cryptology – CT-RSA 2015*, pages 181–198, Springer, 2015.
8. 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*, 2014.
9. D. Câmaraand, C.PL Gouvêa, J. López and R. Dahab. Fast Software Polynomial Multiplication on ARM Processors using the NEON Engine, *Security Engineering and Intelligence Informatics*, pages 137–154, Springer, 2013.
10. J.W. Bos, P.L. Montgomery, D. Shumow, and G. M. Zaverucha. Montgomery multiplication using vector instructions. *Selected Areas in Cryptography – SAC 2013*, pages 471-489, Springer Berlin Heidelberg, 2013.
11. H. Seo, Z. Liu, J. Großschädl, J. Choi and H. Kim Montgomery Modular Multiplication on ARM-NEON revisited. *Information Security and Cryptology – ICISC 2014*, pages 328–342, Springer, 2014.
12. H. Seo, Z. Liu, T. Park, H. Kim, Y. Lee, J. Choi and H. Kim. Parallel Implementations of LEA. *Information Security and Cryptology – ICISC 2013*, pages 256-274, Springer International Publishing, 2014.
13. M.J.O Saarinenand and B.B. Brumley Lighter, Faster, and Constant-Time: WhirlBob, the Whirlpool variant of StriBob. IACR ePrint `https://eprint.iacr.org/2014/501.pdf`, 2014.
14. C.PL Gouvêa and J. López Implementing GCM on ARMv8. *Topics in Cryptology – CT-RSA 2015*, pages 167–180, Springer, 2015.
15. 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*, 7428:512–529, 2012.
16. T. Güneysu, V. Lyubashevsky and T. Pöppelmann. Practical Lattice-Based Cryptography: A Signature Scheme for Embedded Systems. *Cryptographic Hardware and Embedded Systems – CHES 2012*, 7428:530–547, 2012.

17. T. Pöppelmann, L. Ducas and T. Güneysu. Enhanced Lattice-Based Signatures on Reconfigurable Hardware. *Cryptographic Hardware and Embedded Systems – CHES 2014*, 8731:353–370, 2014.
18. R. De Clercq, S. S. Roy, F. Vercauteren, and I. Verbauwhede. Efficient Software Implementation of Ring-LWE Encryption. *18th Design, Automation & Test in Europe Conference & Exhibition – DATE 2015*, 2015.
19. S. B. S. Ahmad Boorghany and R. Jalili. On constrained implementation of lattice-based cryptographic primitives and schemes on smart cards. Cryptology ePrint Archive, Report 2014/514, 2014. https://eprint.iacr.org/2014/514.pdf.
20. 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. https://eprint.iacr.org/2014/078.pdf.
21. T. Pöppelmann, Tobias Oder, and T. Güneysu. Speed Records for Ideal Lattice-Based Cryptography on AVR. In http://eprint.iacr.org/2015/382.pdf.
22. Z. Liu , H. Seo, S. S. Roy, J. Großschädl, H. Kim and I.Verbauwhede Efficient Ring-LWE Encryption on 8-Bit AVR Processors. *Cryptographic Hardware and Embedded Systems – CHES 2015*, 9293:663–682, 2015.
23. Microsoft Research, "Latticed-Based Cryptography", http://research.microsoft.com/en-us/projects/lattice/
24. University of Waterloo, Canada. "CryptoWork21", avaiable in http://cryptoworks21.albertoconnor.ca/about/
25. "Post-quantum cryptography for long-term security PQCRYPTO ICT-645622" http://pqcrypto.eu.org/index.html
26. G. Grewal, R. Azarderakhsh, H. Lee, D. Jao, P. Longa Efficient Pairings on ARM Processors. *Selected Areas in Cryptography – SAC 2012*, 7707:149–165, 2012.
27. J. W. Bos, K. E. Lauter, J. Loftus, and M. Naehrig. Improved security for a ring-based fully homomorphic encryption scheme. In *Cryptography and Coding - 14th IMA International Conference, IMACC 2013, Oxford, UK, December 17-19, 2013. Proceedings*, pages 45–64, 2013.
28. T. Cormen, C. Leiserson, and R. Rivest. *Introduction To Algorithms*. http://staff.ustc.edu.cn/$\sim$csli/graduate/algorithms/book6/toc.htm.
29. T. Yanık, E. Savaş, and Ç. K. Koç. Incomplete reduction in modular arithmetic. *IEE Proceedings – Computers and Digital Techniques*, 149(2):46–52, Mar. 2002.
30. 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*, volume 8731, pages 371–391. 2014.
31. L. Ducas. Lattice based signatures: Attacks, analysis and optimization. Ph.D Thesis, 2013. http://cseweb.ucsd.edu~lducas/Thesis/index.html.
32. 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*, STOC '05, pages 84–93, New York, NY, USA, 2005. ACM.
33. D. Hong. LEA: A 128-bit block cipher for fast encryption on common processors. In*Information Security Applications*, WISA'14, pages 3-27, 2014.
34. T. Prescott. Random Number Generation using AES. http://www.atmel.com/zh/cn/Images/article_random_number.pdf
35. H. Seo, J. Choi, H. Kim, T. Park, H. Kim. Pseudo Random Number Generator and Hash Function for Embedded Microprocessors. In *IEEE World Forum on Internet of Things*, WF-IoT'14, pages 37–40, Seoul, Korea, 2014. IEEE.