NEON PQCryto: Fast and Parallel Ring-LWE Encryption on ARM NEON Architecture

Reza Azarderakhsh, Zhe Liu, Hwajeong Seo, and Howon Kim

Recently, ARM NEON architecture has occupied a significant share of 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. In particular, we propose a vectorized version of Iterative Number Theoretic Transform (NTT) for high-speed computation. We present a 32-bit variant of SAMS2 technique, original proposed in CHES'15, for fast reduction. A combination of proposed and previous optimizations results in a very efficient implementation. For 128-bit security level, our ring-LWE implementation requires only $145,200$ clock cycles for encryption and $32,800$ cycles for decryption. These result are more than 17.6 times faster than the fastest ECC implementation on ARM NEON with same security level.

***Index Terms*—Lightweight Implementation, Lattice-based Cryptography, ARM NEON Architecture**

I. Introduction

The 32-bit ARM processor [1] is the most widely used embedded processor in almost all application markets, especially mobile devices, e.g., 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 permits certain operations to execute twice or four times as quickly, without implementing 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.

A typical ARM processor, such as the Cortex v7 [3], features 13 general-purpose 32-bit registers (`R0-R12`), and an additional three 32-bit registers which have special names and usage models, `R13(SP)` for stack pointer, `R14(LR)` for link register as well as `R15(PC)` for program counter. ARMv7-M supports a large number of 32-bit instructions that were introduced as Thumb-2 technology into the Thumb instruction set. For example, both of the `ADD` and `UMULL` instructions cost one clock cycle. The NEON register bank consists of 32 64-bit registers, it can be seen as sixteen 128-bit quadword registers, `Q0-Q15` or thirty-two 64-bit doubleword registers, `D0-D31` [4]. The NEON instructions provide data processing and load/store operations only, and are integrated into the ARM and Thumb instruction sets. The number of elements operated on is indicated by the specified register size. For example, `VADD.I16 q0, q1, q2` indicates an addition operation on 16-bit integer elements stored in 128-bit `Q` registers. This means that the addition operation is on eight 16-bit lanes in parallel. Some instructions can have different size input and output registers. For example, `VMULL.S16 Q0, D2, D3` multiplies four 16-bit lanes in parallel, producing four 32-bit products in a 128-bit destination vector.

The ARM platform occupies a significant share of the worldwide smartphone and tablets market and other security-critical segments of the embedded systems industry. This has made ARM the most common target platform for research projects in the area of efficient implementation of cryptographic primitives for embedded devices. The literature contains papers dealing with RSA [5], Elliptic Curve Cryptography (ECC) [6], pairing-based cryptography [21], AES [7] as well as lattice-based cryptography [8]. Despite recent research progress, efficient implementation of cryptographic algorithm on 32-bit ARM, in particular ARM NEON, is still an interesting and challenge topic.

A. 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 two 64-bit binary polynomials 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

R. Azarderakhsh is with Rochester Institute of Technology, United States. E-mail: {rxaeec}@rit.edu

Z. Liu is with University of Luxembourg and University of Waterloo. E-mail: {zhe.liu}@uni.lu

H. Seo and H. Kim are with Pusan National University, South Korea. E-mail: {hwajeong,howon}@pusan.ac.kr

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 ARMv8 [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 LWE 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. 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 [16] implemented ring-LWE encryption scheme on the identical ARM processors, 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 [17], [18]. The authors evaluated four lattice-based authentication protocols on both 8-bit AVR and 32-bit ARM processors. Very recently, Pöppelmann et al. [19] and Liu et al. [20] 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.

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 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.

B. 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 and signature. 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.
- 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.

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 results outperform the previous ARM implementation (without NEON) by a factor of 2.07. 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 III, 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 IV, we report the implementation results and compare with the state-of-the-art NTT implementations.

II. Background

In this section, we briefly recap the ring-LWE encryption scheme, polynomial multiplication and discrete Gaussian distribution used in our implementation.

A. The Ring-LWE Encryption Scheme

The encryption schemes we use in this paper are based on the ring version of the learning with errors (ring-LWE) problem. The more general form of the problem, i.e. the LWE problem is parameterized by a dimension $n \geq 1$, a modulus q, and an error distribution. The error distribution is generally taken as a discrete Gaussian distribution $\mathcal{X}_\sigma$ with standard deviation σ and mean 0 to achieve best entropy/standard deviation ratio [26]. In the literature the LWE problem is defined as following:

Two polynomials $\mathbf{a}$ and $\mathbf{s}$ are chosen uniformly from $\mathbb{Z}_q^n$. The first polynomial is a global polynomial, whereas the second polynomial is kept as a secret. The LWE distribution $A_{s,\mathcal{X}}$ is defined over $\mathbb{Z}_q^n \times \mathbb{Z}_q$ and comprises of the elements $(\mathbf{a}, t)$ where $t = \langle \mathbf{a}, \mathbf{s} \rangle + e \bmod q \in \mathbb{Z}_q$ for some error polynomial e sampled from the error distribution $\mathcal{X}_\sigma$. In the

search version of the LWE problem, an attacker is provided a polynomial number of $(\mathbf{a}, t)$ pairs sampled from $A_{s,\mathcal{X}}$ and he (she) tries to find the secret polynomial $\mathbf{s}$. Similarly in the *decision* version of the LWE problem, an attacker tries to distinguish between a polynomial number of samples from $A_{s,\mathcal{X}}$ and the same number of samples from $\mathbb{Z}_q^n \times \mathbb{Z}_q$.

In 2010, Lyubashevshy et al. proposed an encryption scheme based on a more practical algebraic variant of the LWE problem defined over polynomial rings $R_q = \mathbb{Z}_q[\mathbf{x}]/\langle f \rangle$ with an irreducible polynomial $f(x)$ and a modulus q. In the ring-LWE problem, the elements a, s and t are polynomials in the ring R_q. The ring-LWE encryption scheme proposed by Lyubashevshy et al. was later optimized in [25] by Roy et al.. Their variant aims at reducing the cost of polynomial arithmetic. In particular, the polynomial arithmetic during a decryption operation requires only one Number Theoretic Transform (NTT) operation. Beside this computational optimization, the scheme performs sampling from the discrete Gaussian distribution using a Knuth-Yao sampler. In the next subsection we will first present the mathematical concepts of the NTT operation and then we will describe the steps used in the Roy et al's version of the encryption scheme. For efficiency, our implementation adopts the Knuth-Yao (KY) for sampling from a discrete Gaussian distribution and the Number Theoretic Transform (NTT) for polynomial multiplication. We denote the NTT of a polynomial a by $\tilde{a}$. Roy et al's ring-LWE encryption scheme is computed as follows:

B. The Encryption Scheme

In this section we describe the steps used in the encryption scheme proposed by Roy et al. [25]. We denote the NTT of a polynomial a by $\tilde{a}$.

- The key generation stage **Gen($\tilde{a}$)**: Two error polynomials $r_1, r_2 \in R_q$ are sampled from the discrete Gaussian distribution $\mathcal{X}_\sigma$ by applying the Knuth-Yao sampler twice.

  $$\tilde{r_1} = NTT(r_1), \tilde{r_2} = NTT(r_2)$$

  and then an operation $\tilde{p} = \tilde{r_1} - \tilde{a} \cdot \tilde{r_2} \in R_q$ is performed. The public key is polynomial pair $(\tilde{a}, \tilde{p})$ and the private key is polynomial $\tilde{r_2}$.
- The encryption stage **Enc($\tilde{a}$, $\tilde{p}$, M)**: The input message $M \in \{0,1\}^n$ is a binary vector of n bits. This message is first encoded into a polynomial in the ring R_q by multiplying the bits of message by $q/2$. Three error polynomials $e_1, e_2, e_3 \in R_q$ are sampled from $\mathcal{X}_\sigma$. The ciphertext is computed as a set of two polynomials $(\tilde{C_1}, \tilde{C_2})$:

  $$(\tilde{C_1}, \tilde{C_2}) = (\tilde{a} \cdot \tilde{e_1} + \tilde{e_2}, \tilde{p} \cdot \tilde{e_1} + NTT(e_3 + M'))$$

- The decryption stage **Dec($\tilde{C_1}$, $\tilde{C_2}$, $\tilde{r_2}$)**: One inverse NTT is performed to recover M':

  $$M' = INTT(\tilde{r_2} \cdot \tilde{C_1} + \tilde{C_2})$$

  and then a decoder is used to recover the original message M from M'.

Algorithm 1 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 $a(x) = NTT(a) \in \mathbb{Z}_q[x]$

```
1:  a = BitReverse(a)
2:  for i from 2 by i = 2i to n do
3:      ω_i = ω_n^{n/i}, ω = 1
4:      for j from 0 by 1 to i/2 − 1 do
5:          for k from 0 by i to n − 1 do
6:              U = a[k + j]
7:              V = ω · a[k + j + i/2]
8:              a[k + j] = U + V
9:              a[k + j + i/2] = U − V
10:         end for
11:         ω = ω · ω_i
12:     end for
13: end for
14: return a
```

C. Number Theoretic Transform

Our implementation adopts the Number Theoretic Transform (NTT) for performing the polynomial multiplication. An NTT can be seen as a variant of Fast Fourier Transform (FFT) but performs in a finite ring $\mathbb{Z}_q$. Instead of using the complex roots of unity, NTT evaluates a polynomial multiplication $a(x) = \sum_{i=0}^{n-1} a_i x^i \in \mathbb{Z}_q$ in the n-th roots of unity ω_n^i for $i = 0, \ldots, n-1$, where ω_n denotes a primitive n-th root of unity. Algorithm 1 shows the iterative version of NTT algorithm, which is originally from Cormen et al. in [23]. As shown in Algorithm 1, the iterative NTT algorithm consists of three nested loops. The outermost loop (i-loop, line $2 \sim 11$) starts from $i = 2$ and increases by doubling i, and the loop stops when $i = n$, thus it has only $log_2 n$ iterations. In each iteration, the value of twiddle factor ω_i are computed by executing a power operation $\omega_i = \omega_n^{n/i}$, and the value of ω is initialized by 1. Compared to i-loop, the j-loop (line $4 \sim 10$) executes more iterations, the number of iteration can be seen as a sum of a geometric progression for 2^i where i starts from 0 and has a maximum value of $log_2(n-1)$, thus, the j-loop has $n-1$ iterations. In each iteration of j-loop, the twiddle factor ω is updated by performing a coefficient modular multiplication in line 10. Apparently, the innermost loop (k-loop, line $5 \sim 9$) occupies most part of the execution time of NTT algorithm since it is executed roughly $\frac{n}{2} log_2 n$ times. In each iteration of the innermost loop (line $6 \sim 9$), two coefficients $a[i+j]$ and $a[i+j+i/2]$ are loaded from memory into registers, and then $a[i+j+i/2]$ are multiplied by the twiddle factor ω, after that, the value of $a[k+j]$ and $a[k+j+i/2]$ are updated and stored in the memory.

In NTT computations, the majority of the processor's clock cycles are spent on modular operations. The straight-forward method of evaluating modular arithmetic is to perform an integer division. However, constrained devices often do not have any dedicated hardware to divide the variables which generates a bulk of code containing loops, multiplications

and additions. For this reason, the optimal modular implementation is the important consideration for high-speed NTT implementations. In [17], [18], Boorghany et al. introduces the technique that calculates the approximation of $\lfloor a/q \rfloor$ and then executes the $q = a - q \times \lfloor a/q \rfloor$. The results show acceptable approximation of modular operation. The following works by [19], [20] applied the approximation techniques to 8-bit AVR processor in assembly level. The technique optimizes the number of addition and shift operations by taking advantages of 8-bit word and instruction sets. Furthermore, it only utilizes the temporal registers without saving which avoids push/pop operations before/after function call. In [16], De Clercq et al. presented memory efficient polynomial multiplication. When storing single 13-bit coefficient variable into single 32-bit word in ARM processor, it is possible to utilize the remaining 19-bit. The authors store two coefficients into single word, which reduces the number of memory load and store instructions by 50%.

III. Optimization Techniques for Ring-LWE

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], [25] and software implementations, e.g., [17], [18], [16], [19], [20]. This also helps us to compare our work with previous works.

A. Vectorized Iterative Number Theoretic Transform

Previous implementations on RISC processors, e.g., [16], [19], [20], executed the NTT computation (Algorithm 1) 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 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.

The vectorized variant of NTT computation is given in Algorithm 2. 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). Finally, the multiple number of coefficient variables are added and subtracted each other.

B. 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 register. As mentioned before, it is possible to store two coefficients into one register as De Clercq did in [16]. 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 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.

C. 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 [20]. 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 significatnly 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 in parallel using one 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 $\mathrm{mod}\, 7681$ operation. The SAMS2 method is introduced in [20] 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

Algorithm 2 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 $a(x) = NTT(a) \in \mathbb{Z}_q[x]$

1: $a = BitReverse(a)$ {BitReverse computation}
2: **for** i from 2 by $i = 2i$ to n **do**
3: $\omega_i = \omega_n^{n/i}$, $\omega = 1$ {Setting twiddle factors}
4: **if** $i = 2$ or $i = 4$ **then**
5: **for** j from 0 by 1 to $i/2 - 1$ **do**
6: **for** k from 0 by i to $n - 1$ **do**
7: $U = a[k + j]$ {sequential computations}
8: $V = \omega \cdot a[k + j + i/2]$ {single multiplication}
9: $a[k + j] = U + V$ {single addition}
10: $a[k + j + i/2] = U - V$ {single subtraction}
11: **end for**
12: $\omega = \omega \cdot \omega_i$ {computation of single twiddle factors}
13: **end for**
14: **else**
15: $\omega_{array}[0] = \omega$
16: **for** p from 1 by 1 to $i/2 - 1$ **do**
17: $\omega = \omega \cdot \omega_i$, $\omega_{array}[p] = \omega$ {computations of multiple twiddle factors}
18: **end for**
19: **for** j from 0 by i to $n - 1$ **do**
20: $p = 0$
21: **for** k from 0 by 4 to $i/2 - 1$ **do**
22: $U_{array} = a[k + j : k + j + 3]$ {parallel computations}
23: $V_{array} = \omega_{array}[p : p + 3] \cdot a[k + j + i/2 : k + j + 3 + i/2]$ {multiple multiplications}
24: $p = p + 4$ {index increment}
25: $a[k + j : k + j + 3] = U_{array} + V_{array}$ {multiple additions}
26: $a[k + j + i/2 : k + j + 3 + i/2] = U_{array} - V_{array}$ {multiple subtractions}
27: **end for**
28: **end for**
29: **end if**
30: **end for**
31: **return** a

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 SMAS2 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, we keep the product in 32-bit long register ($r0$, a quarter of NEON register). 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:

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 multiplicatio 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 intermeidate result larger than 13-bit.

In Algorithm 3, 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 hifted 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 ($4\times$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.

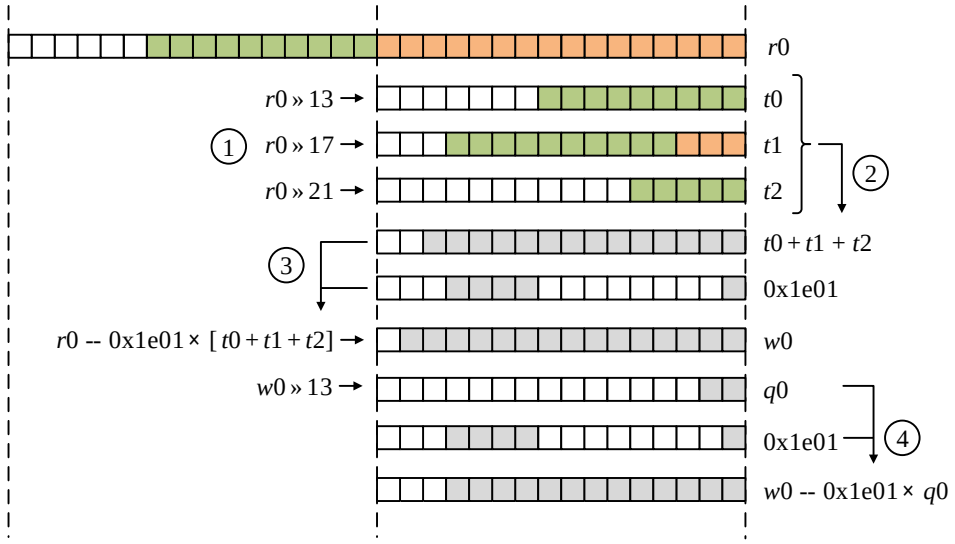

Fig. 1. Fast reduction operation with 32-bit wise SAMS2 method for $q = 7681$. ①: shifting; ②: addition; ③: multiplication & subtraction; ④: multiplication and subtraction.

D. 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.

E. Look-Up Table for the Twiddle Factors

In each iteration of the i-loop, a new twiddle factor ω (line 3 of Algorithm 1) is computed by performing a modular multiplication. The total number of times a new ω is computed in an NTT operation is n. In each iteration of the j-loop, the twiddle factor ω is computed as shown in line 10 of Algorithm 1. 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 [20]. 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.

F. Pseudo-Random Number Generation

Gaussian sampler needs random sequences. Our implementation adopts the PRNG algorithm, which runs the block cipher in counter mode, i.e. it encrypts successive values of an incrementing counter. For block cipher, we exploit parallel implementation of LEA block cipher introduced by [12]. The implementation results achieved 10.06 cycle/byte for encryption by computing four different encryptions at once.

IV. Performance Evaluation and Comparison

A. 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. In particular, the main structure of NTT and interface are written in C while the modular operations are implemented in Assembly Language. 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 I
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. [18]	1,216,000
Boorghany et al. [17]	754,668
Pöppelmann et al. [19]	334,646
Liu et al. [20]	193,731
32-bit ARM processors, e.g., Cortex-M4F, ARM7TDMI:	
Boorghany et al. [17]	109,306
DeClercq et al. [16]	31,583
32-bit ARM-NEON processors, e.g., Cortex-A9:	
This work	**25,574**

Algorithm 3 Pseudo codes of vectorized NTT computation 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).

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

TABLE II
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. [18]	1,216.0	N/A	N/A	5,024.0	2,464.0
Boorghany et al. [17]	754.7	N/A	2,770.6	3,042.7	1,369.0
Pöppelmann et al. [19]	334.6	N/A	N/A	1,315.0	381.3
Liu et al. [20]	193.7	26.8	589.9	671.6	275.6
Implementations on 32-bit ARM processors:					
DeClercq et al. [16]	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**

B. Experimental Results

Table II 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 the 8-bit AVR and 32-bit platforms, the previous works [17], [18], [19], [16], [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 [16]. This can be explained that the authors in [16] 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 [16], the key generation and encryption are slightly slower while the decryption is faster.

TABLE III
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**

Table III 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.

V. FUTURE WORK

The paper presents the first round result of our ring-LWE implementation on ARM NEON. For future, we plan to optimize the current implementation and propose a constant-time version.

REFERENCES

[1] ARM achitectures. 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. Availabe 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] 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.

[17] 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.

[18] 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.

[19] T. Pöppelmann, Tobias Oder, and T. Güneysu. Speed Records for Ideal Lattice-Based Cryptography on AVR. In http://eprint.iacr.org/2015/382.pdf.

[20] 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.

[21] 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.

[22] 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.

[23] T. Cormen, C. Leiserson, and R. Rivest. *Introduction To Algorithms*. http://staff.ustc.edu.cn/$\sim$csli/graduate/algorithms/book6/toc.htm.

[24] 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.

[25] 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.

[26] L. Ducas. Lattice based signatures: Attacks, analysis and optimization. Ph.D Thesis, 2013. http://cseweb.ucsd.edu~lducas/Thesis/index.html.