Efficient Software Implementation of Ring-LWE Encryption on IoT Processors

Zhe Liu, *Member, IEEE*, Reza Azarderakhsh, *Member, IEEE*, Howon Kim, *Member, IEEE*, and Hwajeong Seo

Embedded processors have been widely used for building up Internet of Things (IoT) platforms, in which the security issue is becoming critical. This paper studies efficient techniques of lattice-based cryptography on these processors and presents the first implementation of ring-LWE encryption on ARM NEON and MSP430 architectures. For ARM NEON architecture, we propose a vectorized version of Iterative Number Theoretic Transform (NTT) for high-speed computation of polynomial multiplication on ARM NEON platforms and a 32-bit variant of SAMS2 technique for fast reduction. For MSP430 architecture, we propose an optimized SWAMS2 reduction technique, which consists of five different basic operations, including Shifting, Swapping, Addition, as well as two Multiplication-Subtractions. Regarding of the sampling from the discrete Gaussian distribution, we adopt Knuth-Yao sampler, accompanied with optimized methods such as Look-Up Table (LUT) and byte-scanning. Subsequently, a full-fledged implementation of Ring-LWE is presented by both taking advantage of our proposed method and previous optimization techniques re-designed for desired platforms. Ultimately, our ring-LWE implementation of encryption/decryption at a classical security level of at least 128 bits requires only $149.4k/32.8k$ clock cycles on ARM NEON, and $2126.3k/244.5k$ clock cycles on MSP430. These results are at least 7 times faster than the fastest ECC implementation on desired platforms with same security level.

***Index Terms*—Lightweight Implementation, Lattice-based Cryptography, Internet of Things, ARM–NEON, MSP430**

I. Introduction

The Internet of Things (IoTs) is the most commonly described as an ecosystem of technologies monitoring the status of physical objects, capturing meaningful data, and communicating that information through IP networks to software applications. In particular, embedded technologies can be seen as the foundation of the IoT. However, due to the resource, computing, and environmental constraints, it is a challenging task to do efficient processing and helping maintain data privacy and security for the IoT. Efficient processing not only means streamlined devices and longer battery life, it means reducing latency, network traffic and server loads by enabling processing closer to the data. The need to secure IoT is also critical, as it is becoming an increasingly attractive target for cyber criminals.

Embedded processors with low cost and high performance, such as 32-bit ARM [1] or 16-bit MSP430, have been widely used for building up IoT platforms. The 32-bit ARM architecture (i.e., ARMv6 [2] or ARMv7 [3]) is the most widely used architecture in mobile devices. The 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. On the other hand, the MSP430 is a 16-bit microcontroller family of Reduced Instruction Set Computer (RISC) architecture from Texas Instruments (TI), which is famous for its extremely low power consumption, and thus can be used for low powered (even less than 1 μA) embedded devices. MSP430 microcontrollers incorporate 16 registers and support 27 core instructions as well as 7 addressing modes. Moreover, a hardware multiplier is peripheral designed for 16×16-bit multiplication in MSP430.

Cryptography primitives have been exploited for securing IoT applications based on these embedded processors, such as AES [39], [17], RSA [36], Elliptic Curve Cryptography (ECC) [5], pairing-based cryptography [18] and lattice-based cryptography [25]. Public-key cryptography (a.k.a PKC) for embedded devices is currently predominantly relying on RSA and ECC, and in many applications with tight bounds on computational costs and storage, ECC-based cryptosystems have already replaced previous solutions based on RSA. Bernstein and Schwabe in CHES'12 presented their work on ARM NEON architecture [5], which firstly implied that NEON supports high-security ECC 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. also emphasized the advantage of NEON for high-speed binary ECC. They 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 $\mathbb{F}_{2^m}$ [10].

Unlike the RSA or ECC which are easily attacked by a quantum computer, some lattice-based cryptosystems appear to be resistant to attack by both classical and quantum computers. Despite recent research progress [16], [32], [25], [13],

Z. Liu is with Nanjing University of Aeronautics and Astronautics, China and SnT, University of Luxembourg, Luxembourg.
E-mail: sduliuzhe@gmail.com
R. Azarderakhsh is with Department of Computer and Electrical Engineering and Computer Science, at Florida Atlantic University, USA.
E-mail: azarderakhsh@gmail.com
H. Kim is with the College of Computer Engineering, Pusan National University, Republic of Korea.
E-mail: howonkim@pusan.ac.kr
H. Seo is with the IT Department from Hansung University, Republic of Korea.
E-mail: hwajeong84@gmail.com
Manuscript received 2017; revised , 2017. Corresponding author: H. Seo

[8], [9], [28], [22], efficient implementation of lattice based cryptographic algorithm on 32-bit ARM (in particular ARM NEON) or 16-bit MSP430, is still an interesting and challenge topic.

A. Motivation

Lattice-based cryptography is often considered a premier candidate for realizing post-quantum cryptosystems [30]. Its security relies on the worst-case computational assumptions in lattices which are still considered to be 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 [24] and the Canada "CryptoWorks21" project [11] as well as the European project "PQCrypto" [15]. 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 [17]. 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 or other embedded processor based platform such as 16-bit MSP430.

B. Related Work

The implementations of lattice based cryptography (e.g., Ring-LWE) algorithms flourish in very recent years, and many researchers have reported their experimental results on embedded processors. The first practical evaluations of LWE and ring-LWE based encryption schemes were presented by Göttert et al. in CHES'12 [16]. 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. [32] 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 [25] 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 $5,927k$ cycles for signing, $1,002k$ cycles for verification and $367,859k$ cycles for key generation, respectively, at a medium-term security level. In DATE'15, De Clercq et al. in [13] 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 [8], [9]. The authors evaluated four lattice-based authentication protocols on both 8-bit AVR and 32-bit ARM processors. Very recently, Pöppelmann et al. [28] and Liu et al. [22] studied the implementations of Ring-LWE encryption and the Bimodal Lattice Signature Scheme (BLISS) on an 8-bit platform, as well as they presented efficient ring-LWE results.

C. Target Platforms

The ARM Cortex A9 is a 32-bit processor with full implementations of the ARMv7 architecture including NEON engine. The Register sizes of such processor are 64-bit/128-bit for double(D)/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. The ARM Cortex A9 provides highly configurable L1 caches, and optional NEON and Floating-point extensions. Specially, the NEON SIMD instruction set extension performs up to 16 8-bit wise operations per instruction.

The MSP430 is a 16-bit microcontroller family of Reduced Instruction Set Computer (RISC) architecture from Texas Instruments (TI), which is famous for its extremely low power consumption. MSP430 microcontrollers incorporate 16 registers (namely, R0 through R15). Four of them are reserved for special purpose. Specifically, register R0 (PC), R1 (SP), R2 (SR/CG1) and R3 (CG2) are served as *Program Counter Register, Stack Pointer Register, Status Register* and *Constant Generator Register*, respectively. The other 12 registers, namely from R4 to R15, are all general-purpose registers that can be used by programmers. There are three different multiply modes in MSP430 hardware multiplier: MPY (unsigned 8-bit and 16-bit operands multiplication), MPYS (signed 8-bit and 16-bit operands multiplication) and MAC (unsigned 8-bit and 16-bit operands multiply-and-accumulate multiplication).

D. Contributions

This paper studies efficient techniques of lattice-based cryptography and presents efficient ring-LWE implementations on ARM NEON and MSP430 architectures. Our implementations include core ring-LWE functions which are necessary to implement most popular ring-LWE based encryption scheme. In particular, they support the computation of two most important operations:

- Polynomial multiplication in R_q:
  1) 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.
  2) We introduce the 32-bit wise Shifting $\rightarrow$ Addition $\rightarrow$ Multiplication $\rightarrow$ Subtraction $\rightarrow$ Subtraction (SAMS2) approach for reduction operation on 32-bit ARM NEON processors. The approach replaced

the expensive division operation into shifting, addition and multiplication operations.
3) We propose an optimized SWAMS2 reduction technique on 16-bit MSP processors. The reduction process consists of five different basic operations, namely, Shifting → Swapping → Addition → Multiplication → Subtraction → Subtraction (SWAMS2).
4) 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.

- Gaussian distribution $\mathcal{X}_\sigma$ sampler:
  1) In order for efficient implementation of Gaussian distribution sampler, we employ Knuth-Yao sampler, LUT and byte-scanning methods.
  2) Our experiment also exploits the Pseudo Random Number Generator (PRNG) based on block cipher, which achieved the high performance with parallel and pipelined techniques.

Our implementations achieve 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 at around 128-bit classical security level on desired platforms[1]. For 128-bit security level, it only requires $149,400/32,800$ clock cycles for encryption/decryption on ARM NEON, and $2,126,300/244,500$ clock cycles for encryption/decryption on MSP430. 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 at least 7 times faster on identical platforms.

The rest of this paper is organized as follows. In the next section, we review the basic knowledge of Ring-LWE required in this literature. In Section III and Section IV, we introduce the optimization techniques for Ring-LWE on ARM-NEON processors and MSP430 processors. In particular, we propose several optimization techniques to reduce the execution time of polynomial multiplication and discrete Gaussian sampling. In Section V, we report the implementation results and compare with the state-of-the-art PKC implementations. We conclude our work in Section VI at last.

II. Preliminaries

A. The Ring-LWE Encryption Scheme

The general form of the learn with errors (a.k.a. LWE) problem is parameterized by a dimension $n \geq 1$, a modulus q, and an error distribution. In this work, we focus on the ring version of the learning with errors (ring-LWE) problem. 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 [14].

The ring-LWE based cryptosystems are built on a polynomial ring $R_q = \mathbb{Z}_q[x]/\langle f(x)\rangle$, where $f(x)$ is irreducible. Usually we choose $f(x) = x^n + 1$ with $n = 2^k$ ($k \in \mathbb{N}^+$) and q is a prime with $q \equiv 1 \bmod 2n$ for faster arithmetic in R_q. The ring-LWE distribution on $R_q \times R_q$ consists pairs (a, t) with $a \in R_q$ chosen uniformly random and $t = as + e \in R_q$, where $s \in R_q$ is a fixed secret element and e has small coefficients sampled from the discrete Gaussian above. The resulting distribution on R_q is also denoted as $\mathcal{X}_\sigma$.

We implemented the ring-LWE encryption scheme (`Gen`, `Enc`, `Dec`) proposed by Lyubashevshy et al. [23] and later optimized by Roy et al. [32]. Suppose the message m is well encoded into R_q as $\bar{m}$ (i.e., $\bar{m} \in R_q$ with its i-th coefficient is $(q-1)/2$ iff the i-th bit of m is 1 and 0 otherwise), then the desired encryption scheme is given as follows:

1) $\texttt{Gen}(a)$: Sample $r_1, r_2 \in R_q$ from the distribution $\mathcal{X}_\sigma$, and compute $p = r_1 - a \cdot r_2 \in R_q$. The public key is (a, p) and the private key is r_2.
2) $\texttt{Enc}(a, p, \bar{m})$: Sample $e_1, e_2, e_3 \in R_q$ from $\mathcal{X}_\sigma$. The cipher text consists $(c_1, c_2) \in R_q \times R_q$, where $c_1 = a \cdot e_1 + e_2$ and $c_2 = p \cdot e_1 + e_3 + \bar{m}$.
3) $\texttt{Dec}(c_1, c_2, r_2)$: Compute $m' = c_1 \cdot r_2 + c_2 \in R_q$, and recover m from m'.

B. Related Algorithms

The costly operations in above scheme are sampling from $\mathcal{X}_\sigma$ and polynomial multiplication in R_q. In the following, we perform sampling from the discrete Gaussian distribution using a Knuth-Yao sampler and implement polynomial multiplication via Number Theoretic Transform (NTT) algorithm, as have been adopted by several literatures.

Discrete Gaussian sampling is an integral part of Ring-LWE algorithm. An efficient and secure implementation of discrete Gaussian sampling is a key towards achieving practical implementations of Ring-LWE. To achieve efficiency, the sampler architecture should be small and fast. Previous implementations of Gaussian sampler are usually based on the Knuth-Yao random walk algorithm, which requires a near-optimal number of random bits to generate a sample point in the average case. However, such Knuth-Yao sampler is not secure against timing and power analysis based attacks, since it uses a non constant bit/byte scanning operation, in which the sample generated is related to the number of probability-bits scanned during a sampling operation and its timing provides secret information to an adversary about the value of the sample. In [31], Roy et al. suggested a random shuffle method to protect the Gaussian distributed polynomial against such attacks. In our implementation, we perform the Knuth-Yao sampler by both taking efficiency and security into consideration.

The Number Theoretic Transform (NTT) can be viewed as a variation of Fast Fourier Transform (FTT), where the roots of unity are taken from a finite ring instead of the complex numbers. Given $u, v \in R_q$, we can easily compute their product $w = u \cdot v$ as $w = NTT^{-1}(NTT(u) * NTT(v))$ (here $*$ denotes point-wise multiplication). Since we have $f(x) | x^{2n} - 1$, the $2n$-point NTT immediately leads to a fast (reduced) multiplication operation in R_q. In the later, we will give optimizations in detail for this operation according to the processor features.

[1]Our implementation provides 58 bits of post-quantum security based on https://eprint.iacr.org/2015/1092.pdf.

C. Parameters Setting

For security level at around 128-bit, we choose the parameter sets (n, q, σ) with $(256, 7681, 11.31/\sqrt{2\pi})$.

The discrete Gaussian sampler is limited to 12σ to achieve a high precision statistical difference from the theoretical distribution, which is less than 2^{-90}. These parameter sets were also used in most of the previous software implementations [8], [9], [13], [22].

In the NTT computation, the majority of the execution time is spent on computing reduction operation since it is performed in the innermost k-loop. Thus, fast reduction operation is important for high-speed implementation of NTT algorithm. Our implementation chooses the prime modulus $q = 7681$ (i.e. `0x1e01` in hexadecimal representation) for 128-bit security ring-LWE encryption schemes.

III. Optimization Techniques for ARM–NEON

In this section, we describe several optimization techniques to reduce the execution time of Ring-LWE on ARM NEON architecture.

A. NTT Algorithm

Previous implementations on RISC processors, e.g., [13], [28], [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)[2]. Finally, the multiple number of coefficient variables are added and subtracted each other, simultaneously.

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 [13]. 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. 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 $\bmod 7681$ operation, as shown in Figure 1. 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 Figure 1 illustrated,

[2]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

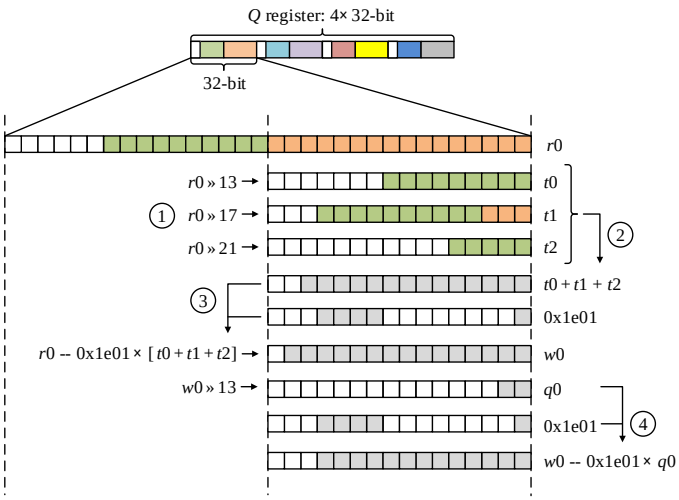


Fig. 1. Fast reduction operation with 32-bit wise SAMS2 method for $q = 7681$. ①: shifting; ②: addition; ③: multiplication & subtraction; ④: multiplication and subtraction.

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:

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.

Pseudo codes for vectorized NTT computation with constant time reduction is described in Algorithm 2. Firstly four

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).

```
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}
```

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 ($4\times$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.

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.

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.

B. 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.

Gaussian sampler needs random sequences. As the ARM NEON does not support the build-in True Random Number Generator (TRNG), our implementation adopts the Pseudo Random Number Generator (PRNG) algorithm, which runs the block cipher in counter mode, i.e. it encrypts successive values of an incrementing counter. We firstly set a secret initial vector (counter) and perform encryption to generate the randomized output. There are a number of lightweight block ciphers that can be used for generating random numbers. Recently, ATmel company introduced AES peripheral based PRNG [29]. 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 [34].

Our implementation exploits the LEA block cipher [20] for random generations. The 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 [33]. The 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.

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 [13]. On a 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.

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.

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.

We exploit the Look-Up Table (LUT) approaches proposed in [13] 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, then it follows the computationally efficient approach.

When it comes to security, we used secure countermeasure against timing and power analysis based attacks, since the Knuth-Yao sampler uses a non-constant bit/byte scanning operation. We used random shuffle suggested by [31] to protect the Gaussian distributed polynomial against such attacks. The random shuffle operation swaps all random samples, which removes any timing information. We firstly perform the previous Knuth-Yao sampler with byte-scanning. Then all generated samples are randomly shuffled.

IV. Optimization Techniques for MSP430

For faster implementation of NTT algorithm and Gauss Sampler on MSP430 processors, we also illustrate our methods in this section.

A. NTT algorithm

In order to optimize the performance of NTT algorithm, we propose an optimized SWAMS2 reduction technique for performing the $\bmod 7681$ operation on 16-bit MSP processors. The main idea is from previous SAMS2 reduction technique on 8-bit AVR processors [22], which estimates the quotient of $t = \frac{a}{q}$, and then perform the subtraction $a - t \cdot q$. More precisely, the reduction process consists of five different basic operations, namely, Shifting $\rightarrow$ Swapping $\rightarrow$ Addition $\rightarrow$ Multiplication $\rightarrow$ Subtraction $\rightarrow$ Subtraction (SWAMS2).

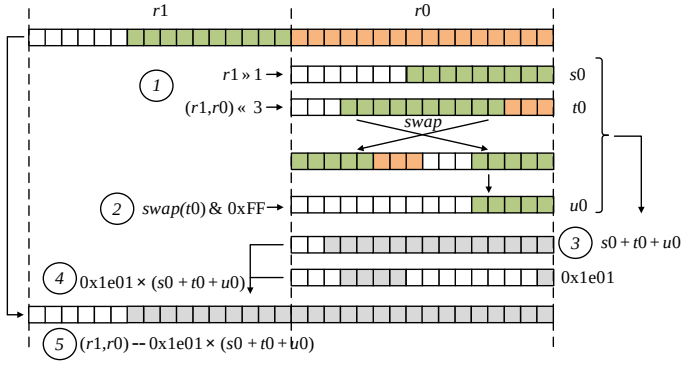


Fig. 2. Fast reduction operation with SWAMS2 method for $q = 7681$. ①: shifting; ②: swapping; ③: addition; ④: multiplication; ⑤: subtraction.

As shown in Figure 2, we keep the product in two registers $(r1, r0)$, which has been marked by different colors. Each of $r1, r0$ is 16-bit long. The colorful parts mean that this bit has been occupied while the white part means the current bit is empty. The reduction with 7681 using SWAMS2 approach can be performed as follows:

1) Shifting. We first right shift $r1$ by one bit, and store the result in $s0$. After that we left shift by 3-bit to get the results $t0$.
2) Swapping. We perform the swapping of $t0$. After that the lower byte of intermediate result is extracted.
3) Addition. We then perform the addition of $s0 + t0 + u0$. Apparently, the sum result is less than 16-bit, which can be kept in one register.
4) Multiplication. The fourth step is to multiply the constant `0x1e01` by $(s0 + t0 + u0)$, which is a 16×16-bit multiplication.
5) Subtraction. Thereafter, we subtract the product obtained from Step 4.
6) Subtraction. However, the result we get in step 5 may still be larger than $p = 7681$, thus, we do the correction by subtracting the modulus p at most twice.

For implementing NTT on 16-bit MSP processors, we adopted the original iterative number theoretic transform, as given in Algorithm 3.

The inner loop of NTT in 16-bit MSP assembly code is described in Algorithm 4. As shown in Step 1 and 2, coefficient (`OP_A`) and twiddle factor (`OP_B`) is multiplied and the result is stored in multiplier's memory (`&RESLO`, `&RESHI`). In Step 3 and 4, the result is moved from memory to register. In Step 5, 6, and 7, the results are copied twice.

Algorithm 4 Inner loop of NTT in 16-bit MSP assembly codes

Input: 32-bit coefficient A(OP_A), 32-bit twiddle factor B(OP_B)
Output: 32-bit result C(TMP0).

```
1: MOV OP_A, &MPY                          {16-bit wise multiplication}
2: MOV OP_B, &OP2                          {16-bit wise multiplication}
3: MOV &RESLO, TMP0                        {loading 32-bit result}
4: MOV &RESHI, TMP1                        {loading 32-bit result}
5: MOV TMP0, TMP2                          {copying 32-bit result}
6: MOV TMP1, TMP3                          {copying 32-bit result}
7: MOV TMP3, OP_A                          {copying 32-bit result}
8: RLA TMP2                                {SWAMS2 ①:shifting}
9: RLC TMP3                                {SWAMS2 ①:shifting}
10: RLA TMP2                               {SWAMS2 ①:shifting}
11: RLC TMP3                               {SWAMS2 ①:shifting}
12: RLA TMP2                               {SWAMS2 ①:shifting}
13: RLC TMP3                               {SWAMS2 ①:shifting}
14: RRA OP_A                               {SWAMS2 ①:shifting}
15: ADD TMP3, OP_A                         {SWAMS2 ③:addition}
16: SWPB TMP3                              {SWAMS2 ②:swapping}
17: AND #0X00FF, TMP3                      {SWAMS2 ②:swapping}
18: ADD TMP3, OP_A                         {SWAMS2 ③:addition}
19: MOV #0X1E01, OP_B                      {SWAMS2 ④:multiplication}
20: MOV OP_A, &MPY                         {SWAMS2 ④:multiplication}
21: MOV OP_B, &OP2                         {SWAMS2 ④:multiplication}
22: SUB &RESLO, TMP0                       {SWAMS2 ⑤:subtraction}
```

Algorithm 3 Iterative Number Theoretic Transform

Input: A polynomial $a(x) \in \mathbb{Z}_q[x]$ of degree $n-1$ and n-th primitive $\omega \in \mathbb{Z}_q$ of unity
Output: 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: **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: $V = \omega \cdot a[k + j + i/2]$ {multiplication}
8: $a[k + j] = U + V$ {addition}
9: $a[k + j + i/2] = U - V$ {subtraction}
10: **end for**
11: $\omega = \omega \cdot \omega_i$ {computing twiddle factors}
12: **end for**
13: **end for**
14: **return** a

From Step 8 to 14, intermediate results are shifted to right by 13, 17 and 21-bit and accumulated. From Step 15 to 18, the intermediate result is swapped and added. In Step 19 $\sim$ 21, modulus is multiplied. Lastly the results are subtracted from the intermediate result.

B. Gaussian Sampler

Here we describe optimization techniques that can be used to reduce the execution time of the Knuth-Yao sampler on MSP processors. Similar to the case of ARM NEON processor, MSP processor does not support the build-in TRNG, too. Our implementation adopts the PRNG algorithm, which runs the block cipher in counter mode.

The parameter setting for Knuth-Yao algorithm on MSP430 architecture is the same as that on ARM architecture (i.e., a precision of 2^{-90} for dimension $n = 256$, and a probability matrix P_{mat} of 55 rows and 109 columns [13]). On 16-bit MSP processor, we stored each 55-bit column in four words, where each word size is 16-bit long. In this case, 9-bit is wasted per column and the probability matrix only occupies 872 bytes in total. Similar to the implementation on ARM NEON, We also adopted random shuffle operation suggested by [31] to protect the Gaussian distributed polynomial against timing and power analysis based attacks on MSP430.

V. Performance Evaluation and Comparison

This section presents the performance results of our implementation. 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 summarizes the execution times of Number Theoretic Transform, Gaussian sampling, key generation, encryption and decryption of the proposed implementation for medium-term (around 128-bit) security level. Specially, on 32-bit ARM NEON processors, our parallel NTT operation only requires $25,5k$ clock cycles for 128-bit security level, while on 16-bit MSP430 processor it requires $178.1k$ clock cycles. We also compare software implementations of Number Theoretic Transform on different processors. For the 8-bit AVR and 32-bit platforms, the previous works [8], [9], [28], [13], [25]

TABLE I
PERFORMANCE COMPARISON OF SOFTWARE IMPLEMENTATION OF LATTICE-BASED CRYPTOSYSTEMS ON DIFFERENT PROCESSORS (CLOCK CYCLE 10^3).

Implementations	NTT/FFT	Sampling	Gen	Enc	Dec
8-bit AVR processors, e.g., ATxmega64, ATxmega128:					
Boorghany et al. [9]	1,216.0	N/A	N/A	5,024.0	2,464.0
Boorghany et al. [8]	754.7	N/A	2,770.6	3,042.7	1,369.0
Pöppelmann et al. [28]	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
16-bit MSP processors, e.g., MSP430F1611:					
This work	**178.1**	**397.0**	**1,296.3**	**2,126.3**	**244.5**
32-bit ARM processors, e.g., Cortex-M4F, ARM7TDMI:					
DeClercq et al. [13]	31.6	7.3	117.0	121.2	43.3
32-bit ARM-NEON processors, e.g., Cortex-A9:					
This work	**25.5**	**19.9**	**126.6**	**149.4**	**32.8**

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 [13]. This can be explained that the authors in [13] 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 on 32-bit ARM processors requires only $149.4k$ clock cycles for encryption and $32.8k$ cycles for decryption. Comparing with the implementation on ARM Cortex M4 in [13], the key generation and encryption are slightly slower while the decryption is faster.

TABLE II
COMPARISON OF RING-LWE ENCRYPTION SCHEMES WITH RSA AND ECC ON ARM NEON PROCESSORS (ENC AND DEC IN CLOCK CYCLES)

Implementation	Scheme	Enc	Dec
16-bit MSP processors, e.g., MSP430F1611:			
Wang et al. [38]	RSA-1024	6,320,000	172,000,000
Zhe et al. [21]	ECC-255	15,619,715	10,945,048
This work	LWE-256	**2,126,288**	**244,491**
32-bit ARM-NEON processors, e.g., Cortex-A9:			
Seo et al. [36]	RSA-2048	535,020	20,977,660
Bernstein et al. [5]	ECC-255	1,157,952	578,976
This work	LWE-256	**149,400**	**32,800**

Table II 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 and MSP430 platform. The to-date fastest RSA software for an ARM NEON processor was reported in [36]; it achieves an execution time of approximately $20,978k$ clock cycles for RSA-2048 decryption at the 96-bit security level. For comparison, our LWE-256 implementation requires only $32.8k$ 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.[5]. For comparison, our implementation of ring-LWE is roughly 8 times faster for encryption and 17.6 for decryption. When it comes to the MSP430 platform, our implementation of ring-LWE is at least 7 times faster than the latest ECC (and even faster than RSA) implementation on desired platforms with same (or even higher) security level.

A. Compatibility for MSP430X and ARMv8

Proposed methods are evaluated over MSP430 and ARMv7 processors. The techniques are also working on advanced MSP430X and ARMv8 processors. MSP430X processor share similar architecture of MSP430. Particularly, MSP430X processor supports a 32-bit multiplier and the requirement of proposed technique over MSP430 architecture is a multiplication with 16-bit multiplier. For ARMv8, all NEON instruction sets of ARMv7 are available in ARMv8 architecture. For this reason, we can directly apply the proposed technique to the ARMv8 program.

VI. CONCLUSION

This paper presented several optimizations for efficiently implementing ring-LWE encryption scheme on high-end IoT platform (32-bit ARM NEON) and low-end IoT processor (16-bit MSP). In particular, we proposed optimization techniques 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. We also showed that the implementation of ring-LWE is practically fast enough on low-end 16-bit MSP processors. All of these achieved results set new speed records for ring-LWE encryption implementation on 16-bit MSP and 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 16-bit MSP and 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] D.J. Bernstein and P. Schwabe. NEON crypto. *Cryptographic Hardware and Embedded Systems –CHES 2012*, pages 320–339, Springer Berlin Heidelberg, 2012.
[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] University of Waterloo, Canada. "CryptoWork21", avaiable in http://cryptoworks21.albertoconnor.ca/about/

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

[13] 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.

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

[15] "Post-quantum cryptography for long-term security PQCRYPTO ICT-645622" http://pqcrypto.eu.org/index.html

[16] 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.

[17] C.PL Gouvêa and J. López Implementing GCM on ARMv8. *Topics in Cryptology – CT-RSA 2015*, pages 167–180, Springer, 2015.

[18] 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.

[19] 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.

[20] D. Hong. LEA: A 128-bit block cipher for fast encryption on common processors. In*Information Security Applications*, WISA'14, pages 3-27, 2014.

[21] Z. Liu, H. Seo, Z. Hu, X. Hunag, and J. Großschädl. Efficient implementation of ECDH key exchange for MSP430-based wireless sensor networks. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 145–153. ACM, 2015.

[22] Z. Liu, H. Seo, S. S. Roy, J. Großschädl, H. Kim, and I. Verbauwhede. Efficient Ring-LWE encryption on 8-bit AVR processors. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 663–682. Springer, 2015.

[23] V. Lyubashevsky, C. Peikert, and O. Regev. On Ideal Lattices and Learning with Errors over Rings. In Advances in Cryptology EUROCRYPT 2010, volume 6110 of Lecture Notes in Computer Science, pages 1-23. Springer Berlin Heidelberg, 2010.

[24] Microsoft Research, "Latticed-Based Cryptography", http://research.microsoft.com/en-us/projects/lattice/

[25] 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.

[26] 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.

[27] T. Pöppelmann, T. Oder, and T. Güneysu. High-performance ideal lattice-based cryptography on 8-bit atxmega microcontrollers. In *International Conference on Cryptology and Information Security in Latin America*, pages 346–365. Springer, 2015.

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

[29] T. Prescott. Random Number Generation using AES. http://www.atmel.com/zh/cn/Images/article_random_number.pdf

[30] 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.

[31] S. S. Roy, O. Reparaz, F. Vercauteren, and I. Verbauwhede. Compact and side channel resistant discrete gaussian sampling, 2014.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] H. Wang and Q. Li. Efficient implementation of public key cryptosystems on mote sensors (short paper) In *International Conference on Information and Communications Security*, pages 519–528. Springer, 2006.

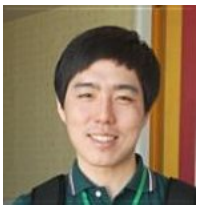
[39] J. Wang, P.K. Vadnala, J. Großschädl and Q. Xu. Higher-Order Masking in Practice: A Vector Implementation of Masked AES for ARM NEON. *Topics in Cryptology – CT-RSA 2015*, pages 181–198, Springer, 2015.

[40] 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.

Zhe Liu received his Ph.D degree Laboratory of Algorithmics, Cryptology and Security (LACS), University of Luxembourg, Luxembourg. His Ph.D thesis has received the prestigious FNR Awards 2016 – Outstanding PhD Thesis Award for his contributions in cryptographic engineering. He is a full professor in College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics (NUAA) and SnT, University of Luxembourg, Luxembourg. He has been a visiting scholar in City University of Hong Kong, COSIC, K. U. Leuven as well as Microsoft Research, Redmond. His research interests include computer arithmetic and information security. He has co-authored more than 60 research peer-reviewed journal and conference papers in the area of information security.

Reza Azarderakhsh received the Ph.D. degree in electrical and computer engineering from the University of Western Ontario, London, ON, Canada, in 2011. He was a NSERC Post-Doctoral Research Fellow with the Center for Applied Cryptographic Research and the Department of Combinatorics and Optimization, University of Waterloo, Waterloo, ON, Canada. He is currently with the Department of Computer Engineering, Rochester Institute of Technology, Rochester, NY, USA. His current research interests include finite field and its application, elliptic curve cryptography, pairing-based cryptography, and implementations on embedded devices. Prof. Azarderakhsh was the recipient of the prestigious NSERC Post-Doctoral Research Fellowship in 2012. He is a Guest Editor for the IEEE/ACM Transactions on Computational Biology and Bioinformatics for the special issue of Emerging Security Trends for Biomedical Computations, Devices, and Infrastructures (2015 and 2016). He also serves as the Guest Editor of IEEE Transactions on Dependable and Secure Computing, Special Issue on Emerging Embedded and Cyber Physical System Security Challenges and Innovations (2016).He is a supervisor member for CryptoWorks21(NSERC CREATE Training Program in Building a Workforce for the Cryptographic Infrastructure of the 21st Century).

Howon Kim received the BSEE degree from Kyungpook National University, Daegu, Republic of Korea, in 1993 and the MS and PhD degrees in electronic and electrical engineering from the Pohang University of Science and Technology (POSTECH), Pohang, Republic of Korea, in 1995 and 1999, respectively. From July 2003 to June 2004, he studied with the COSY group at the Ruhr-University of Bochum, Germany. He was a senior member of the technical staff at the Electronics and Telecommunications Research Institute (ETRI), Daejeon, Republic of Korea. He is currently working as an associate professor with the Department of Computer Engineering, School of Computer Science and Engineering, Pusan National University, Pusan, Republic of Korea. His research interests include RFID technology, sensor networks, information security, and computer architecture. Currently, his main research focus is on mobile RFID technology and sensor networks, public key cryptosystems, and their security issues. Dr. Kim is a member of the IEEE, and the International Association for Cryptologic Research (IACR).



Hwajeong Seo received the B.S.E.E. degree in 2010, and the M.S. degree in 2012 and the Ph.D degree in 2016 in Pusan National University. He is currently an assistant professor in Hansung University.