Efficient Ring-LWE Encryption on 8-bit AVR Processors

Zhe Liu[1], Hwajeong Seo[2], and Sujoy Sinha Roy[3], Johann Großschädl[1], Howon Kim[2], Ingrid Verbauwhede[3]

[1] University of Luxembourg,
Laboratory of Algorithmics, Cryptology and Security (LACS),
6, rue R. Coudenhove-Kalergi, L–1359 Luxembourg-Kirchberg, Luxembourg
{zhe.liu,johann.groszschaedl}@uni.lu

[2] Pusan National University,
School of Computer Science and Engineering,
San-30, Jangjeon-Dong, Geumjeong-Gu, Busan 609–735, Republic of Korea
{hwajeong,howonkim}@pusan.ac.kr

[3] ESAT/COSIC and iMinds, KU Leuven
Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium
{sujoy.sinharoy, ingrid.verbauwhede}@esat.kuleuven.be

Abstract. Lattice-based cryptography is considered to be a big challenge to implement on resource-constraint microcontrollers. In this paper, we focus on efficient arithmetic that can be used for the ring variant of the Learning with Errors (ring-LWE) encryption scheme on 8-bit AVR processors. Our contributions include the following optimizations: for the Number Theoretic Transform (NTT) based polynomial multiplication, (1) we propose the MOV-and-ADD and Shifting-Addition-Multiplication-Subtraction-Subtraction (SAM-S2) techniques for speeding up the modular coefficient multiplication, (2) we exploit the incomplete arithmetic for representing the coefficient to reduce the number of reduction operations, (3) and we reduce the running memory requirement of NTT multiplication with a refined memory-access scheme, finally, we propose to perform the Knuth-Yao Gaussian distribute sampler with a byte-wise scanning strategy to reduce the memory footprint of the probability matrix. For medium-term security level, our high-speed optimized ring-LWE implementation requires only $590K$, $666K$ and $299K$ clock cycles for key-generation, encryption and decryption, respectively. Similarly for long-term security level, the key-generation, encryption and decryption take $2.3M$, $2.7M$ and $700K$ clock cycles, respectively. These achieved results speed up the previous fastest LWE implementation by a factor of 4.5, while at least one order of magnitude faster than state of the art RSA and ECC implementations on the same platform.

Keywords: Ring learning with errors (Ring-LWE), software implementation, public-key encryption, 8-bit AVR, number theoretic transform, discrete Gaussian sampling

1 Introduction

Today's widely used cryptosystems are mainly based on integer factorization and discrete logarithm problems, which are believed to be intractable with classical computers. However, these hard problems can be solved by using Shor's algorithm [33] and its variant on a quantum computer. Lattice-based cryptography is considered as a premier candidate for post-quantum cryptosystems. Its security is based on worst-case computational assumptions in lattices that remain hard even for quantum computers. The Internet is currently in the midst of a transition from a network that connects commodity computers (e.g. PCs, laptops) to a network of smart objects ("things"). Even today, there are more "non-traditional" computing devices connected to the Internet than "conventional" computers [14]. Among the smart devices that are (or will soon be) populating the Internet are all kinds of sensors, actuators, meters, consumer electronics, medical monitors, household appliances, vehicles, and even items of clothing. Many of these devices are very constrained in terms of computing power and memory resources. For example, a typical wireless sensor node, such as the widely-used MICAz mote, features an 8-bit AVR ATmega processor clocked at 8 MHz and a few kB RAM. However, in order to communicate securely with such devices, they need to be able to execute public-key cryptography as otherwise end-to-end authentication and end-to-end key establishment would not be possible. Implementing public-key algorithms on 8-bit processors poses a big challenge, not only for RSA and ECC, but also post-quantum techniques like lattice-based cryptography. Therefore, it is necessary to study how well "cryptosystems of the future" are suited for the "Internet of the future." In other words, it is necessary to study how well lattice-based cryptography can be implemented on 8-bit processors such as the AVR series processors [3].

The introduction of learning with errors (LWE) problem [29] and its ring variant (ring-LWE) [23] provide an efficient way to build lattice based public key cryptosystems. The first practical software implementation of an LWE-based cryptosystem was presented in CHES'12 [15]. Göttert et al. presented both a hardware and a software implementation of ring-LWE. Oder et al. in [24] presented an efficient implementation of Bimodal Lattice Signature Schemes (BLISS) on a 32-bit ARM Cortex-M4F microcontroller; they achieved execution times of 35.3 and 6 ms for signature generation and verification, respectively, at a medium-term security level. Recently, De Clercq et al. in [11] 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. To the best of our knowledge, the first time that a lattice-based cryptographic scheme was implemented on an 8-bit processor belonged to Boorghany et al. in [7, 8]. In [7, 8], the authors evaluated four lattice-based authentication protocols on both 8-bit AVR and 32-bit ARM processors. In particular, for 8-bit AVR implementation, their implementation needed 754668 cycles and 2207787 cycles for Fast Fourier Transform (FFT) transform at medium-term and long-term security level, respectively. Based on efficient implementation of polynomial multiplication and Gaussian sampler function, their implementation of LWE based encryption scheme required 2770592 clock cycles for key generation, 3042675 clock cycles for encryption as well as 1368969 clock cycles for decryption at medium-term security level.

1.1 Research Contributions

This paper continues the line of research on the efficient implementation of the ring-LWE encryption scheme on an 8-bit AVR processor. The core contributions are several optimizations to reduce the execution time and running RAM requirements of ring-LWE encryption scheme. More specifically, our contributions are listed as follows:

1. The efficiency of coefficient modular multiplication is a pre-requisite for high-speed NTT operation. We propose the MOV-and-ADD technique for coefficient multiplication and Shifting-Addition-Multiplication-Subtraction-Subtraction (SMAS2) approach for reduction operation. The MOV-and-ADD method aims at reducing the number of addition operations by rescheduling the order of byte multiplications, while the SMAS2 approach performs the reduction by using a sequence of Shifting-Addition-Multiplication-Subtraction-Subtraction operations. For $q = 7681$, a coefficient modular multiplication using a combination of proposed methods can be performed in 53 clock cycles in average. (See Subsection 3.3 and 3.4 for details)
2. In the NTT computation, the majority of the execution time is spent on computing the modular reduction operation since it is the most frequent operation in the innermost loop. We exploit the incomplete arithmetic for representing the coefficients and perform the reduction operation in a lazy fashion. Our practical results show this approach reduces roughly 6% of the reduction operations in average.(See Subsection 3.5 for details)
3. The intermediate coefficient during the computation of an NTT requires a large amount of RAM. We propose a refined memory access scheme by making full use of the memory space. For $q = 7681$, we store 16 13-bit coefficients in 26 bytes memory space so that all the memory space is efficiently used for storing the coefficients. The proposed scheme allows to save roughly $15 \sim 19\%$ RAM requirements. (See Subsection 3.6 for details)
4. The Knuth-Yao algorithm requires a large probability matrix to store the probabilities of sampling a random number at a discrete position from the Gaussian distribution. In order to reduce the memory consumption of storing this matrix, we propose a byte-scanning technique using sliding window method (of width 8) to check the results in a byte level instead of bit-level in previous work. We are able to reduce 12.6% of the size of the probability matrix while achieving fast execution time. The method can be easily exploited to the other sampler algorithms in signature scheme. (See details in Subsection 4.1)
5. Based on the above optimization techniques, we present two implementations of ring-LWE encryption scheme for both medium-term and long-term security levels on an 8-bit AVR processor. The first one is high-speed (HS) oriented, while the second is memory-efficient (ME) oriented. For medium-term security level, the former one only requires $590K$, $666K$ and $299K$ clock cycles for key-generation, encryption and decryption, respectively. Similarly for long-term security level, the key-generation, encryption and decryption of HS implementation take $2.3M$, $2.7M$ and $700K$ clock cycles, respectively. Both of the HS and ME implementations improve the speed records for ring-LWE encryption scheme on 8-bit AVR processors. (See Table 1 in Subsection 5.2 for details).

For a comparison with related work, our HS implementations speed up the previous fastest work on 8-bit AVR processors in [8] by a factor of 4.5, while somewhat memory efficient (see details in Table 3). It is also worth to note that all of the proposed optimizations can also be used to speed up LWE signature schemes, for example [26], on the identical platform.

The rest of this paper is organized as follows. In the next section, we review the background of ring-LWE encryption schemes including the NTT and Knuth-Yao sampler. In Section 3, we focus on the optimization techniques for NTT on 8-bit AVR processors. In particular, we propose several optimizations to reduce the execution time and memory consumption of NTT. In Section 4, we propose the optimizations for Knuth-Yao sampler. In Section 5, we report the implementation results and compare with the state-of-the-art public-key cryptography implementations on same platform, including the LWE, RSA as well as ECC. Finally, we draw our conclusions in Section 6.

2 Background

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

2.1 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 [12]. 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 [30]. Roy et al.'s 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 and the Knuth-Yao sampling operations and then we will describe the steps used in the Roy et al's version of the encryption scheme.

2.2 Number Theoretic Transform

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:     $\omega_i = \omega_n^{n/i}$, $\omega = 1$
 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:             $V = \omega \cdot a[k + j + i/2]$
 8:             $a[k + j] = U + V$
 9:             $a[k + j + i/2] = U - V$
10:         $\omega = \omega \cdot \omega_i$
11: return $a$
```

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 [10].

As shown in Algorithm 1, the iterative NTT algorithm consists of three nested loops. The outermost loop (i-loop, line 2-11) 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 4-10) 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 5-9) 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 6-9), 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.

2.3 The Gaussian Sampler

The ring-LWE cryptosystem needs samples from a discrete Gaussian distribution to provide the error polynomials during the key generation and encryption operations. There are several methods for sampling from a discrete Gaussian distribution. Among them we selected Knuth-Yao algorithm. The Knuth-Yao algorithm stores probabilities of the sample points and performs a random walk by following a binary tree, namely the discrete distribution generating (DDG) tree [19, 31, 13]. A DDG tree efficiently counts the visited non-zero nodes to find the sample based on probability.

Algorithm 2 Knuth-Yao Sampling

```
Require: Probability matrix $P_{mat}$, random number $r$, modulus $q$
Ensure: Sample value $s$
 1: for $col$ from 0 by 1 to $MAXCOL$ do
 2:     $d \leftarrow 2d + (r\&1)$
 3:     $r \leftarrow r \gg 1$
 4:     for $row$ from $MAXROW$ by $-1$ to 0 do
 5:         $d \leftarrow d - P_{mat}[row][col]$
 6:         if $d = -1$ then
 7:             if $(r\&1) = 1$ then
 8:                 return $q - row$
 9:             else
10:                 return $row$
11: return 0
```

As shown in Algorithm 2, the DDG tree is constructed on-the-fly, eliminating the need for storing the entire tree. A random walk is performed from the root of the DDG tree. Each random walk checks a random bit to explore from one level of the tree to the next level. The distance counter d represents the number of intermediate nodes to the right side of the visited node. Each non-zero node that is visited, decrements the distance counter by one. When the distance counter is finally decremented to below zero, the terminal node is found, and the current row number of the probability matrix represents the sample. As the probability matrix only contains the positive half of the Gaussian distribution, a random bit is used to decide the sign of the sample. As our scheme performs all operations modulo q, the negative number is found by $q - row$. Algorithm 2 requires consecutive accesses to elements from different rows in the same column in P_{mat}. To keep the number of memory accesses low, P_{mat} is suggested to be stored in a column-wise form [11].

2.4 The Encryption Scheme

In this section we describe the steps used in the encryption scheme proposed by Roy et al. [30]. 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'.

Another optimized variant of the encryption scheme is called YASHE scheme [9]. YASHE scheme makes efforts to reduce the size of ciphertext, which further reduces the communication cost in the practical usage of ring-LWE.

2.5 Parameter Selection

Our implementation adopts the parameter sets (n, q, σ) with $(256, 7681, 11.31/\sqrt{2\pi})$ and $(512, 12289, 12.18/\sqrt{2\pi})$ for security levels of 128-bit and 256-bit, respectively. 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 hardware implementations, e.g., [15, 30] and software implementations, e.g., [7, 8, 11]. This also helps us to compare our work with previous work.

3 Optimization Techniques for NTT Computation

In this section, we describe several optimization techniques to reduce the execution time and memory consumption of NTT and inverse NTT on 8-bit AVR processors. Throughout the paper, we represent the coefficient of the polynomial using lowercase letters, and each coefficient, for example a, is represented using two bytes a_H for the higher byte, a_L for the lower byte.

3.1 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 both the memory-access operations of ω and ω_i (including of loading and storing) and its coefficient modular multiplication. In total, $n-1$ times of coefficient modular multiplications and $2(n-1)$ times of loading and storing operations are required. On the other hand, storing all the intermediate twiddle factors ω and ω_i into RAM is extremely expensive considering that the 8-bit AVR processor only have several kilo bytes of RAM.

However, 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. Based on this observation, our solution is to store all the twiddle factors ω into ROM which is similar to the technique used in [27] for hardware implementation. More specifically, we pre-computed the twiddle factor "off-line" and store them in a look-up table in flash ROM instead of RAM. We only need to transfer the twiddle factor that is required for the current iteration of the j-loop from ROM to RAM. In this way, we do not require to compute the power of ω_n in i-loop and the coefficient modular multiplication in j-loop. On the other hand, two bytes of RAM are sufficient for keeping the twiddle factor ω during the whole NTT computation.

3.2 Algorithmic Optimization

We follow the parameter sets from [30] for the ring-LWE encryption scheme where the modulus q is a prime and is $q \equiv 1 \bmod 2n$. In this setting, a polynomial multiplication can be performed using only n-point NTTs/INTT. This special technique is known as the negative wrapped convolution theorem and has been applied in the hardware implementations [26, 30]. We remark that for a more generic implementation, such restrictions on the parameter set may not be applicable. Our choice to use such restricted parameter set is mainly due to the fact that our target platform (which is an AVR microcontroller) is computationally weak, and hence computation of $2n$-point NTT (or INTT) during a polynomial multiplication costs both time and dynamic memory requirement. Beside the application of the negative wrapped convolution, we apply other small computational optimizations that were used to accelerate hardware architectures. We find these techniques very suitable for the software implementation too. These optimizations include the interchanging of the j and k-loops in the NTT algorithm [5], merging the scaling operation by n^{-1} with the chain of multiplications during the post processing operation in the inverse NTT [30].

3.3 Efficient Coefficient Multiplication

The coefficient multiplication is one of the most performance-critical operations of NTT computation in terms of "computational complexity" since each NTT computation requires $\frac{n}{2} log_2 n$ co-efficient multiplications.

In our implementation, the coefficient is at most 13-bit and 14-bit long, which can each be kept in two 8-bit registers. Inspired by the hybrid technique for multi-precision multiplication [17], we propose the "MOV-and-ADD" technique to perform the coefficient multiplication. The MOV-and-ADD multiplication aims at minimizing the number of `adc` instructions. As shown in Figure 1, the two coefficients a and b are each loaded from RAM into two registers. We first multiply the lower byte of a (i.e. a_L) by the lower byte of b (i.e. b_L) and move the product to two result

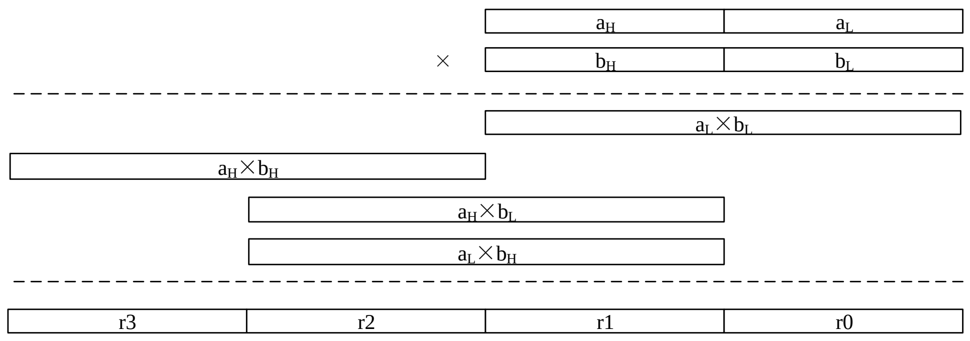

Fig. 1: The MOV-and-ADD coefficient multiplication.

registers $r1$ and $r0$ with help of the `movw` instruction. Next, we form the product $a_H \cdot b_H$ and move the result to the registers $r2$ and $r3$. Thereafter, we multiply a_H by b_L, add the resulting 16-bit product $a_L \cdot b_H$ to $r1$, $r2$, and propagate the carry from the last addition to the register $r3$. Finally, we perform the byte multiplication of a_L with b_H, and then add the product into the register $r1$ and $r2$, the carry bit will be added into $r3$. In summary, the processing of the MOV-and-ADD coefficient multiplication in Figure 1 requires four `mul`, two `movw`, and a total of 6 `add` or `adc` instructions, respectively.

3.4 Fast Reduction

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 the perquisite for high-speed implementation of NTT algorithm.

We propose an optimized SMAS2 reduction technique for performing the mod 7681 and mod 12289 operations. This main idea is to first estimate the quotient of $t = \frac{a}{q}$, and then perform the subtraction $a - t \cdot q$. Finally, the correct result can be obtained by a correction process with a final subtraction. Observing that $2^{13} \equiv 2^9 - 1 \bmod 7681$, it is not difficult to conjecture the approximating value of t is $(a \gg 13) + (a \gg 17) + (a \gg 21)$. More precisely, the reduction process consists of four different basic operations, namely, Shifting $\rightarrow$ Addition $\rightarrow$ Multiplication $\rightarrow$ Subtraction $\rightarrow$ Subtraction (SAMS2). As shown in Figure 2, we keep the product in four registers $(r3, r2, r1, r0)$, which has been marked by different colors. Each of $r3, r2, r1, r0$ is 8-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 SAMS2 approach can be performed as follows:

1. Shifting. We first right shift $r3, r2, r1$ by one bit, and store the intermediate result of $r3, r2$ in $t0$. After that, we right shift by 4-bit to get the results $t1$ and $t2$. As shown in the figure, both $t0$ and $t1$ consist of two temporary registers while $t2$ is 8-bit long which is stored in one temporary register.
2. Addition. We then perform the addition of $t0 + t1 + t2$. Apparently, the sum result is less than 16-bit, which can be kept in two registers.

3. Multiplication. The third step is to multiply the constant `0x1e` (i.e. the higher byte of 7681) by $(t0 + t1 + t2)$, which is a 16×8-bit multiplication.
4. Subtraction. Thereafter, we subtract both the sum of $t0+t1+t2$ and the product obtained from Step 3 from r.
5. Subtraction. However, the result we get in step 4 may still be larger than $p = 7681$, thus, we do the correction by subtracting the modulus p at most twice.

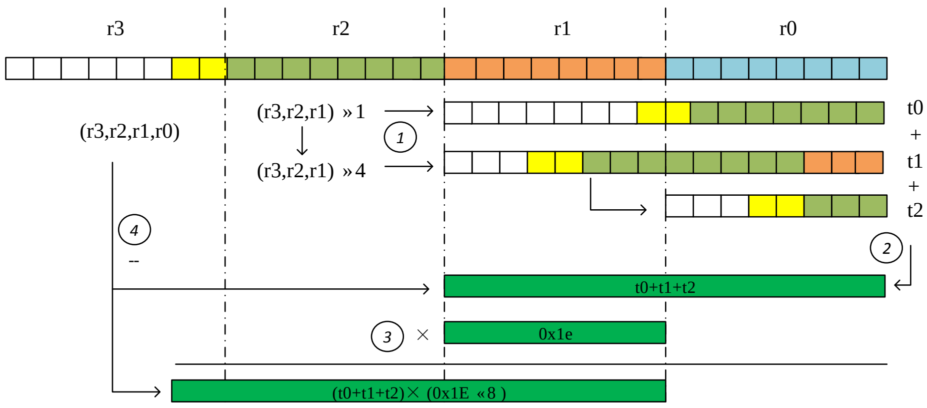

Fig. 2: Fast reduction operation with SMAS2 method for $q = 7681$. ①: shifting; ②: addition; ③: multiplication; ④: subtraction.

Another property of SMAS2 technique is its economic register usage; it occupies only 14 out of the 32 available registers [3] such that no `push`/`pop` instructions are required at the beginning/end of the function. Similarly, we give the computation process of modulus $q = 12289$ using of SMAS2 technique in the Appendix.

3.5 Reduce the Number of Modular Reduction Operations

Besides the coefficient multiplications, addition and subtraction operations are also performed in the innermost loop of NTT. In general, a coefficient addition $r = a + b \bmod p$ (resp. subtraction) can be performed by an addition (resp. subtraction) operation followed by a conditional subtraction (resp. addition) with the prime p. The intermediate result is kept in the range of $[0, p]$.

Inspired by the incomplete modular arithmetic [34], our implementation does not perform an exact comparison between r and p, but rather tolerate an incompletely reduced coefficient $r \in [0, 2^{\lceil log_2 p \rceil}]$. Taking $p = 7681$ as an example, the incomplete coefficient addition works as follows. We first perform a normal coefficient addition, after that, we compare the higher byte of r with 2^5, and perform the conditional subtraction whenever $r \geq 2^{13}$, at most two subtractions are required for keeping the intermediate result within $[0, 2^{\lceil log_2 p \rceil}]$ [20]. In the very last outermost iteration of NTT (i.e. in our case, $i = 256$), a correction process is performed to bring the final result back into the range $[0, p]$. The incomplete coefficient technique can be used for coefficient addition, subtraction as well as coefficient multiplication. Our practical results show this approach reduces roughly 6% of the modular reduction operations, thus resulting in a speed up of the execution time for the NTT computation.

3.6 Reduce the Amount of RAM Consumption

The NTT computation requires the storage of intermediate results of coefficients into RAM, which is extremely precious for an 8-bit AVR processor. In particular, the number of coefficients is very large and each coefficient occupies two bytes. Taking $p = 7681$ for dimension $n = 256$ as an example, each coefficient has a length of 13-bit and is stored in two bytes of memory, in this way, the intermediate result needs 512 bytes. Observing that the higher three bits are empty when storing a 13-bit

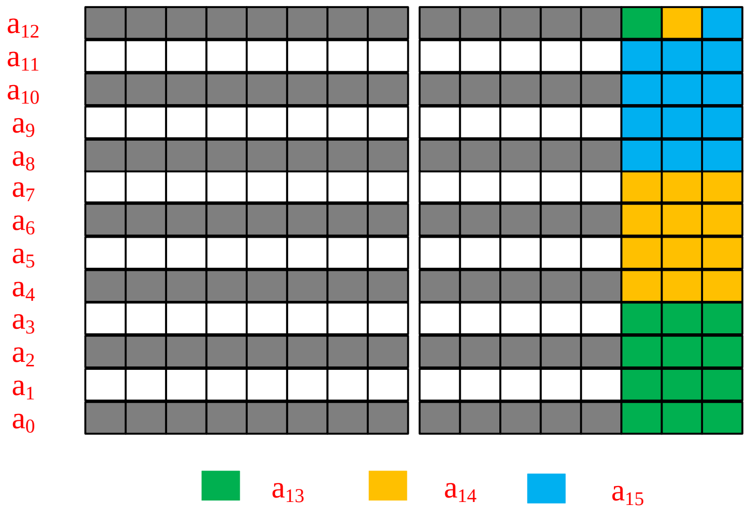

Fig. 3: Reducing the RAM consumption with refined memory-access scheme for $q = 7681$. Each row represents a memory of two bytes, while each block represents one bit.

long coefficient in two bytes memory, we propose a refined memory-access scheme for reducing the RAM requirements. The main idea is to store 16 13-bit coefficients (i.e. a_i for $0 \leq i \leq 15$) in 26 bytes memory space. As shown in Figure 3, each row represents two bytes and each block represents 1-bit. The first 13 coefficients from a_0 to a_{12} are stored into the rows in a normal way. Instead of allocating more memory space for the 14th, 15th, and 16th coefficients, we make full use of the remaining empty space for storing them. More specifically, we divide the 13-bit coefficient into two parts, the first part contains the lower 12-bit and is stored into the empty space of the rows of a_0, a_1 and a_2, while the second part, i.e. the most significant bit of a_{13}, is stored into the 14th bit of the 13th row. We mark the memory space for storing coefficients a_{13}, a_{14} and a_{15} with green, orange and blue colors, respectively. Loading the coefficients from memory into registers is not a trivial task. $a_0 \sim a_{12}$

can be obtained by an `AND` operation with `0x1fff`, while loading a_{13}, a_{14} and a_{15} requires to perform more bit rotation operations and memory accesses.

As a result, the refined memory-access scheme requires 18.75% less RAM. Similarly, the memory-access scheme of modulus $q = 12289$ is given in the Appendix.

4 Optimizations of Knuth-Yao Sampler

Both key-generation and encryption require many Gaussian samplers, thus the 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 to reduce the memory consumption and execution time of the Knuth-Yao sampler on 8-bit AVR processors.

4.1 Sliding Window Method

The Knuth-Yao algorithm requires a probability matrix P_{mat}, which contains the probabilities of sampling a random number at a discrete position from the Gaussian distribution. Our Knuth-Yao implementation mainly adopts the optimizations in [11], however, we propose a sliding window method to further reduce the memory consumption and execution time.

Probability matrix with small memory consumption. 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 [11]. On an 8-bit AVR processor, we stored each 55-bit column in seven words, where each word is 8-bit long. In this case, only 1-bit is wasted per column and the probability matrix only occupies 6,104 bytes in total [4].

Window method with byte-scanning. The bit-scanning operation as shown in Algorithm 2 (line 4-10) requires to check each bit and decrease the distance (d) whenever the bit is set. Instead of executing the scanning operation in a bit-level, we propose a sliding window method to perform the scanning operation in a byte-level. As shown in Algorithm 3 (line 15 $\sim$ 16), we choose the window width ($w = 8$). In order to scan 8 bits in the probability matrix P_{mat}, the proposed window method only requires eight additions, one subtraction and one conditional branch statements, which saves seven conditional branch statements at the cost of one subtraction.

Look-up table. We replace the bit counting into single 8-bit look-up table (LUT) access by using the operands as an index of look-up table. The proposed window

[4] The ROM consumption of probability matrix can be further reduced to 4352 bytes with a sophisticated window method. In order to make a balance between the execution time and ROM consumption, we decide to use the current version in our implementation.

method only needs one look-up table access, one subtraction and one branch statements to process eight bits. However, since the byte-scanning computes sampling by 8 bits, we should conduct post-processing to get correct sampling (line 17 $\sim$ 32 of Algorithm 3). If the distance d is -1, we need to determine correct results among eight cases. We trace back from $row - 7$ to row to find the set bit. On the other hand, if the distance d is lower than -1, then we trace back the index of row from $row - 7$ to row to find the case that the distance d is -1 by adding the probability matrix P_{mat} to the distance d.

Efficiently skip the consecutive leading zeros. Another issue on probability matrix is an occurrence of consecutive leading zeros. In order to skip consecutive leading zeros, we use simple sliding method by conducting the comparison between zero and bit counter (line 14 of Algorithm 3). Since the 8-bit is stored in a byte, single byte comparison can notify that the byte has leading zeros or not. This approach can skip one byte-scanning at the cost of one conditional branch statement, if the counter is zero.

Look-up table in DDG tree. We exploit the look-up table (LUT) approaches proposed in [11] into our window method implementations (shown in lines 1 $\sim$ 9 of Algorithm 3). 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 lookup result successfully. Otherwise, the most significant bit of the lookup result is one, then a lookup 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.

4.2 Pseudo Random Number Generator with AES Accelerator

We choose the PRNG algorithm suggested by [28] that runs the AES block cipher in counter mode, encrypting successive values of an incrementing counter. In order to get a true random number from AVR platform, we used the techniques presented by TrueRandom library that sets up a noisy voltage on analog pin and measures the least significant bit with von Neumann whitening algorithm [2]. Atmel's ATxmega128A1 begins to support the AES crypto-accelerator that computes the encryption with reasonable computation overheads (i.e. 375 clock cycles) and small memory footprint for AES trigger program. This is significant progress compared to the software implementation of AES on the ATmega128 processor, which requires 1993 clock cycles and 2K program [25] with pre-computations. Another attractive feature of the ATxmega128A1 is the build-in AES accelerator can be operated independently of the microprocessor, which therefore hides almost all latencies for AES encryption [32]. We exploit this idea into our Knuth-Yao sampler implementation. More specifically, we trigger the AES operation immediately after obtaining the output of AES encryption and then conduct other operations. However, AVR only supports 128-bit AES accelerator. For the case of long-term security level, we choose to use the software version of 256-bit AES developed by AVR-Crypto-Lib, where the 256-bit AES encryption takes 3521 clock cycles [4].

Algorithm 3 Knuth-Yao Sampling with Sliding Window Method ($width = 8$)

```
Require: Probability matrix $P_{mat}$, random number $r$, modulus $q$
Ensure: Sample value $s$
 1: $index \leftarrow r\&255$
 2: $r \leftarrow r \gg 8$
 3: $s \leftarrow LUT1[index]$
 4: if $msb(s) = 0$ then
 5:     if $(r\&1) = 1$ then
 6:         return $q - s$
 7:     else
 8:         return $s$
 9: $d \leftarrow s\&7$
10: for $col$ from 8 by 1 to $MAXCOL$ do
11:     $d \leftarrow 2d + (r\&1)$
12:     $r \leftarrow r \gg 1$
13:     for $row$ from $MAXROW$ by $-8$ to 0 do
14:         if $(P_{mat}[row][col] \| P_{mat}[row-1][col] \| ... \| P_{mat}[row-7][col]) > 0$ then
15:             $sum = \sum_{i=row}^{row-7}(P_{mat}[i][col])$
16:             $d \leftarrow d - sum$
17:             if $d < 0$ then
18:                 if $d = -1$ then
19:                     for $j$ from $row - 7$ by 1 to $row$ do
20:                         if $P_{mat}[j][col] = 1$ then
21:                             if $(r\&1) = 1$ then
22:                                 return $q - j$
23:                             else
24:                                 return $j$
25:                 else
26:                     for $j$ from $row - 7$ by 1 to $row$ do
27:                         $d \leftarrow d + P_{mat}[j][col]$
28:                         if $d = -1$ then
29:                             if $(r\&1) = 1$ then
30:                                 return $q - j$
31:                             else
32:                                 return $j$
33: return 0
```

5 Performance Evaluation and Comparison

This section presents the performance results of our implementation. We give the concrete implementation platform in Subsection 5.1. The required execution time and memory consumption of our ring-LWE implementation are given in Subsection 5.2. Finally, we show a comparison with the previous fastest implementations in Subsection 5.3.

5.1 Experimental Platform

Our implementation used ATxmega128A1 processor on an Xplain board as the target platform. This processor has a maximum frequency of 32 MHz, 128 KB flash program memory and 8 KB SRAM. It is a powerful and popular processor with an AES crypto-accelerator and can be used in a wide range of applications, such as industrial, hand-held battery applications as well as some medical devices. The implementation is written using a mixed ANSI C and Assembly languages. In particular, the main structure of ring-LWE scheme and interface are written in C while the modular operations are implemented in Assembly. We complied our implementation with speed optimization option -O3 on the latest version of Atmel Studio. In order to obtain accurate timings, we ran each operation at least 1000 times and calculated the average cycle count for one operation.

5.2 Experimental Results

Table 1 summarizes the execution time of the main components of ring-LWE encryption schemes (including the NTT, the Knuth-Yao sampler, key-generation, encryption as well as decryption operation) for both of medium-term and long-term security levels. As mentioned before, for each security level, our ring-LWE encryption scheme contains two implementations for each arithmetic operation, one of which is optimized for speed and the other optimized for memory (i.e. memory efficient). The high-speed (HS) oriented implementation makes full use of all the optimization techniques described in Section 3 (except Subsection 3.6) and Section 4 and the data is kept in RAM. On the other hand, the memory-efficient (ME) oriented implementation uses the refined memory-access scheme in 3.6 in all basic operations and store the pre-computed tables into flash ROM.

Table 1: Execution time of main components of the ring-LWE encryption scheme (in clock cycles)

Implementation	NTT	KY	`Gen`	`Enc`	`Dec`
HS-256	169,423	26,763	590,774	666,671	299,538
ME-256	351,346	39,027	1,245,934	1,623,876	562,902
HS-512	484,680	321,977	2,303,241	2,721,372	700,999
ME-512	890,294	344,855	3,745,173	4,433,556	1,450,713

As shown in the Table 1, the NTT operation only requires 169423 clock cycles for HS-256 implementation, however, the execution time increases sharply to 484680 cycles for HS-512 implementation, which is 2.86 times slower. The Knuth-Yao sampler for HS-256 requires an average of 26763, while 321,977 cycles are needed for HS-512. Both of these observations can be explained by the fact that the length of coefficients for HS-512 is twice as HS-256 and the reduction operation with $p = 12889$ is much slower than with $p = 7681$ on an 8-bit AVR processor and AES accelerator can boost performance of random number generations. It is also interesting to compare the performance of HS oriented with ME oriented implementations for ring-LWE encryption scheme. Taking HS-256 as an example, the key generation, encryption and decryption require an execution time of roughly $590K$, $666K$ and $300K$, respectively, which is twice faster than the ME-256 implementation. Apparently, this is mainly because the refined memory-access scheme has been applied into all basic operations, including coefficient addition, subtraction, multiplication up to the NTT, the Knuth-Yao sampler and each component requires more execution time to catch the coefficients from memory into registers and store the data back into memory.

Table 2: Memory requirements of key generation, encryption as well as decryption (in bytes)

Implementation	`Gen`	`Enc`	`Dec`	`Total`
RAM/ROM (HS-256)	1,585/8,512	2,609/9,284	1,585/5,880	2,609/13,776
RAM/ROM (ME-256)	1,297/8,400	2,129/8,448	1,297/5,840	2,129/13,226
RAM/ROM (HS-512)	3,121/11,348	6,193/14,188	3,121/7,506	6,193/19,198
RAM/ROM (ME-512)	2,737/10,816	4,529/12,316	2,737/8,590	4,529/17,550

Table 2 lists the RAM and ROM requirements of key-generation, encryption and decryption. For the whole ring-LWE encryption scheme implementation, the HS-256 requires roughly 2.6K RAM and 13.7K ROM, while the ME-256 needs 2.1K RAM and 13.2K ROM. Both of the RAM requirements increase approximately 130% when comparing HS-512 and ME-512 with HS-256 and ME-256. Thanks to the proposed refined memory-access scheme in Subsection 3.6, the ME oriented implementations could save 19% and 21% RAM requirements while consuming roughly the same ROM as HS oriented implementations for both of medium-term and long-term security levels [4].

5.3 Comparison with Related Work

Table 3 compares software implementations of lattice-based cryptosystems on different processors. For the 8-bit AVR platform, both the work [7, 8] and our implementation adopt the same parameter sets as we mentioned in Subsection 2.5. The fastest published ring-LWE implementation belongs to [7]. Compared to [7], our NTT transform and Gaussian sampler are 4.5 and 1.9 times faster, respectively. For encryption scheme, Boorghany et al. reported execution time of $2.77 \cdot 10^6$,

Table 3: Performance comparison of software implementation of lattice-based cryptosystems on different processors.

Implementations	`NTT/FFT`	`Sampling`	`Gen`	`Enc`	`Dec`
Implementations on desktop processors, e.g., Core 2 Duo:					
Göttert et al. [16] (256)	N/A	N/A	9,300,000	4,560,000	1,710,000
Göttert et al. [16] (512)	N/A	N/A	13,590,000	9,180,000	3,540,000
Implementations on 32-bit ARM processors, e.g., Cortex-M4F:					
DeClercq et al. [11] (256)	31,583	7,296	117,009	121,166	43,324
DeClercq et al. [11] (512)	71,090	14,592	252,002	261,939	96,520
Oder et al. [24] (512)	122,619	935,936	N/A	N/A	N/A
Implementations on **8-bit AVR processors**, e.g., ATxmega64, ATxmega128:					
Boorghany et al. [8]	1,216,000	N/A	N/A	5,024,000	2,464,000
Boorghany et al. [7]	754,668	N/A	2,770,592	3,042,675	1,368,969
This work (HS-256)	169,423	26,736	590,774	666,671	299,538
Boorghany et al. [7]	2,207,787	617,600	N/A	N/A	N/A
This work (HS-512)	484,680	321,977	2,303,241	2,721,372	700,999

$3.04 \cdot 10^6$ and $1.37 \cdot 10^6$ clock cycles for key generation, encryption and decryption with medium-term security level. For a comparison, our HS-256 only requires $0.59 \cdot 10^6$, $0.67 \cdot 10^6$ and $0.30 \cdot 10^6$ cycles, which is more than 4.5 times faster. The significant progress achieved is mainly due to the proposed optimizations for speeding up the NTT multiplication and Gaussian sampling computations.

In order to show a comparison of the ring-LWE based encryption scheme with some traditional encryption schemes, we compare our ring-LWE implementation with the state-of-the-art RSA and ECC implementations on 8-bit AVR platform in Table 4. The fastest published RSA implementation belongs to [21], the authors reported an execution time of $76.58 \cdot 10^6$ clock cycles for RSA decryption with 80-bit security level [5]. For a comparison, our HS-256 only requires 299538 cycles, which is more than 2556 times faster even with a higher 128-bit security level. They are a few software ECC implementations on 8-bit AVR processors. In prime fields, Hutter and Schwabe in their cryptographic library NaCl [18] reported execution time of 22954657 (HS version) and 28043124 (ME version) clock cycles for single scalar multiplication using the Curve25519 [6]. Similarly, MoTE-ECC [22] requires roughly 9420788 and 21118778 clock cycles for fixed point and random point scalar multiplications for the same security level. In binary fields, Aranha et al. in [1] achieved an execution time of 5898240 clock cycles for a full scalar multiplication over $\mathbb{F}_{2^{233}}$. The widely used Elliptic Curve Integrated Encryption Scheme (ECIES) is based on scalar multiplication, namely, the encryption requires two scalar multiplications, one with fixed point and the other with random point while the decryption needs one scalar multiplication with random point. For a comparison, our ring-LWE encryption scheme (HS oriented) is at least one order of magnitude faster than the ECC

[5] To the best of our knowledge, no RSA implementation with 128-bit security level exists on 8-bit AVR processors, thus, we use 80-bit security for a comparison.

work in [18, 22, 1]. The detailed comparison can be found in Table 4. These research results also show that the ring-LWE encryption is advantageous to traditional PKC for resource-constraint microncontrollers in case of performance.

Table 4: Comparison of Ring-LWE encryption schemes with RSA and ECC on 8-bit AVR processors (`RAM` and `ROM` in bytes, `Enc` and `Dec` in clock cycles)

Implementation	`PKC`	`RAM`	`ROM`	`Enc`	`Dec`
[21]	RSA-1024	N/A	N/A	N/A	$75.68 \cdot 10^6$
[18] (HS)	ECC-255	681	28,883	45,909,314	22,954,657
[18] (ME)	ECC-255	922	17,373	56,086,248	28,043,124
[22]	ECC-256	556	14,700	30,539,566	21,118,778
[1]	ECC-233	3,700	38,600	11,796,480	5,898,240
This work (HS)	LWE-256	2,609	13,776	666,671	299,538
This work (ME)	LWE-256	2,129	13,226	1,623,876	562,902

6 Conclusion

This paper presented several optimizations for efficiently implementing ring-LWE encryption scheme on 8-bit AVR platform. In particular, we propose three optimizations to accelerate the execution time and a refined memory-access scheme to reduce the RAM requirements of the NTT-based polynomial multiplication. A combination of these optimizations results in a very efficient NTT computation, which is 4.5 times faster than the previous work on the same platform. We also report high-speed optimized and memory-efficient optimized encryption scheme implementations for both medium-term and long-term security levels, the former of which outperforms the previous best result by a factor of roughly 4.5. 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 resource-constraint 8-bit AVR processors.

References

1. D. F. Aranha, R. Dahab, J. C. López, and L. B. Oliveira. Efficient implementation of elliptic curve cryptography in wireless sensors. *Advances in Mathematics of Communications*, 4(2):169–187, May 2010.
2. TrueRandom. TrueRandom library for Arduino. User Guide, available for download at `http://code.google.com/p/tinkerit/wiki/TrueRandom`, 2010.
3. Atmel Corporation. 8-bit ARV® Instruction Set. User Guide, available for download at `http://www.atmel.com/dyn/resources/prod_documents/doc0856.pdf`, July 2010.
4. AVR-Crypto-Lib. User Guide, available for download at `http://http://avrcryptolib.das-labor.org/trac`, 2012.
5. A. Aysu, C. Patterson, and P. Schaumont. Low-cost and Area-efficient FPGA Implementations of Lattice-based Cryptography. In *HOST*, pages 81–86. IEEE, 2013.

6. D. J. Bernstein. Curve25519: New Diffie-Hellman speed records. In M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, editors, *Public Key Cryptography — PKC 2006*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer Verlag, 2006.
7. 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`.
8. 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.
9. 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.
10. T. Cormen, C. Leiserson, and R. Rivest. *Introduction To Algorithms.* `http://staff.ustc.edu.cn/$\sim$csli/graduate/algorithms/book6/toc.htm`.
11. 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.
12. L. Ducas. Lattice based signatures: Attacks, analysis and optimization. Ph.D Thesis, 2013. `http://cseweb.ucsd.edu~lducas/Thesis/index.html`.
13. N. C. Dwarakanath and S. D. Galbraith. Sampling from Discrete Gaussians for Lattice-based Cryptography on a Constrained Device. *Applicable Algebra in Engineering, Comm. and Computing*, pages 159–180, 2014.
14. D. Evans. The Internet of things: How the next evolution of the Internet is changing everything. Cisco IBSG white paper, available for download at `http://www.cisco.com/web/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf`, apr 2011.
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. N. Göttert, T. Feller, M. Schneider, J. Buchmann, and S. Huss. On the design of hardware building blocks for modern lattice-based encryption schemes. In *Cryptographic Hardware and Embedded Systems–CHES 2012*, pages 512–529. Springer, 2012.
17. N. Gura, A. Patel, A. S. Wander, H. Eberle, and S. Chang Shantz. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems — CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 119–132. Springer Verlag, 2004.
18. M. Hutter and P. Schwabe. NaCl on 8-bit avr microcontrollers. In A. Youssef, A. Nitaj, and A. E. Hassanien, editors, *Progress in Cryptology — AFRICACRYPT 2013*, volume 7918 of *Lecture Notes in Computer Science*, pages 156–172. Springer Verlag, 2013.
19. D. E. Knuth and A. C. Yao. The complexity of nonuniform random number generation. *Algorithms and Complexity: New Directions and Recent Results*, pages 357–428, 1976.
20. Z. Liu and J. Großschädl. New speed records for montgomery modular multiplication on 8-bit AVR microcontrollers. In D. Pointcheval and D. Vergnaud, editors, *The 7th International Conference on Cryptology in Africa — AFRICACRYPT 2014*, volume 8469 of *Lecture Notes in Computer Science*, pages 215–234. Springer Verlag, 2014.
21. Z. Liu, J. Großschädl, and I. Kizhvatov. Efficient and side-channel resistant RSA implementation for 8-bit AVR microcontrollers. In *Proceedings of the 1st International Workshop on the Security of the Internet of Things (SECIOT 2010)*. IEEE Computer Society Press, 2010.
22. Z. Liu, E. Wenger, and J. Großschädl. MoTE-ECC: Energy-scalable elliptic curve cryptography for wireless sensor networks. In I. Boureanu, P. Owezarski, and S. Vau-

denay, editors, *The 12th International Conference on Applied Cryptography and Network Security — ACNS 2014*, volume 8479 of *Lecture Notes in Computer Science*, pages 361–379. Springer Verlag, 2014.

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. 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.
25. D. A. Osvik, J. W. Bos, D. Stefan, and D. Canright. Fast software AES encryption. In *Fast Software Encryption*, pages 75–93. Springer, 2010.
26. T. Pöppelmann, L. Ducas, and T. Güneysu. Enhanced Lattice-Based Signatures on Reconfigurable Hardware. In *Cryptographic Hardware and Embedded Systems - CHES 2014*, volume 8731, pages 353–370. 2014.
27. T. Pöppelmann and T. Güneysu. Towards Efficient Arithmetic for Lattice-Based Cryptography on Reconfigurable Hardware. In A. Hevia and G. Neven, editors, *Progress in Cryptology - LATINCRYPT 2012*, volume 7533 of *LNCS*, pages 139–158. Springer Berlin, 2012.
28. T. Prescott. Random number generation using aes. Technical report, Atmel, Inc., 2011. Available for download at `http://www.atmel.com/ja/jp/Images/article_random_number.pdf`.
29. 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.
30. S. S. Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede. Compact Ring-LWE Cryptoprocessor. In *Cryptographic Hardware and Embedded Systems - CHES 2014*, volume 8731, pages 371–391. 2014.
31. S. S. Roy, F. Vercauteren, and I. Verbauwhede. High Precision Discrete Gaussian Sampling on FPGAs. *Selected Areas in Cryptography - SAC 2013*, pages 383–401, 2014.
32. H. Seo, J. Kim, J. Choi, T. Park, Z. Liu, and H. Kim. Small private key mqpks on an embedded microprocessor. *Sensors*, 14(3):5441–5458, 2014.
33. P. Shor. Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, pages 124–134, Nov 1994.
34. 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.

A SMAS2 method for $q = 12289$

We propose an optimized SMAS2 reduction technique for performing the mod 12289 operation. This main idea is to first estimate the quotient of $t = \frac{a}{q}$, and then perform the subtraction $a - t \cdot q$, finally, the correct result can be obtained by a correction process with a final subtraction. Observing that $2^{14} = 2^{12} - 1 \bmod 12289$, it is not a trivial task to get the approximating value of t is $(a \gg 14) + (a \gg 16) + (a \gg 18) + (a \gg 20) + (a \gg 22) + (a \gg 24)$. More preciously, the reduction process consists of four different basic operations, namely, Shifting $\rightarrow$ Addition $\rightarrow$ Multiplication $\rightarrow$ Subtraction $\rightarrow$ Subtraction (SAMS2). As shown in Figure 4, we keep the product into four registers $(r3, r2, r1, r0)$, which has been marked by different colors. Each of $r3, r2, r1, r0$ is 8-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 12289 using SAMS2 apporach can be performed as follows:

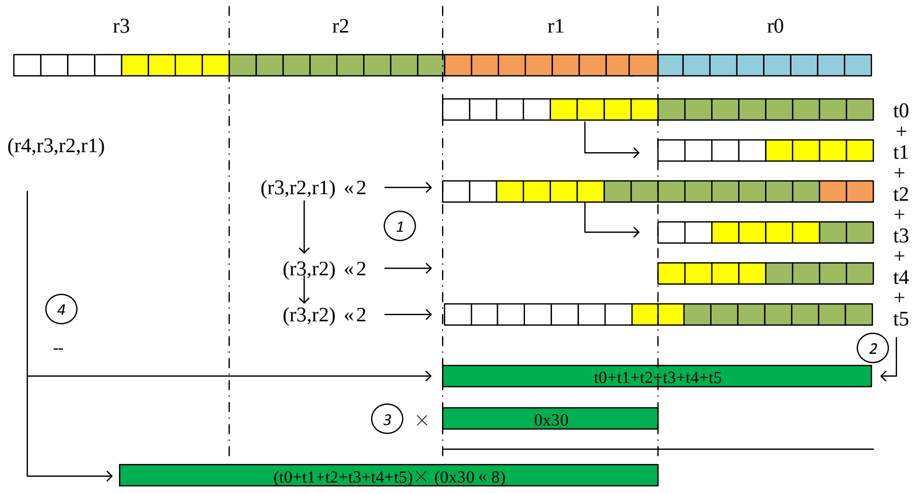

Fig. 4: Fast reduction operation with SMAS2 method for $q = 12289$. ①: shifting; ②: addition; ③: multiplication; ④: subtraction.

1. Shifting. We firstly store $r3, r2$ into $t0$ and $t1$. The, we left shift $r3, r2, r1$ by two bit, and store the intermediate result of $r3, r2$ into $t2$ and $t3$. After that, we left shift by 2-bit to get the results $t4$. Lastly we left shift by 2-bit to get the results $t5$. As shown in the figure, $t0$, $t2$ and $t5$ consist of two temporary registers while $t1$, $t3$ and $t4$ 8-bit long which is stored in one temporary register.
2. Addition. We then perform the addition of $t0+t1+t2+t3+t4+t5$. Apparently, the sum result is less than 16-bit long, which can be kept into two registers.
3. Multiplication. The third step is to multiply the higher byte of `0x30` (i.e. the higher byte of 12289) by $(t0 + t1 + t2 + t3 + t4 + t5)$, which is a 16×8-bit multiplication.

4. Subtraction. Thereafter, we subtract $t0 + t1 + t2 + t3 + t4 + t5$ and the product obtained from Step 3 from r.
5. Subtraction. However, the result we get in step 4 may still be lager than $p = 12289$, thus, we do the correction by conducting one additional SMAS2 and subtracting the modulus p at most twice.

B Refined memory-access scheme for $q = 12289$

As shown in Figure 5, each row represents two bytes and each block represents 1-bit. The first 7 coefficients from a_0 to a_6 are stored into the rows in normal way. Instead of allocating more memory space for 8th coefficients, we make full use of the remaining empty space for storing them. More specifically, the 14-bit is stored into the empty space of the rows of a_0, a_1 ..., a_6. We marked the memory space for storing coefficients a_7 with green color. Loading the coefficients from memory into registers is not a trivial task. $a_0 \sim a_6$ can be obtained by a `AND` operation with `0x3fff`, while a_7 requires to perform more bit rotation operations and memory accesses.

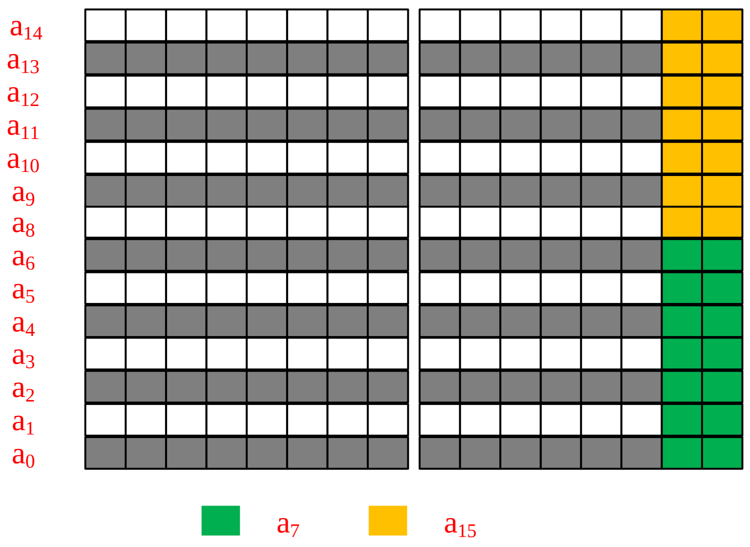

Fig. 5: Reducing the RAM consumption with refined memory-access scheme for $q = 12289$