Efficient Ring-LWE Encryption on 8-bit AVR Processors

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

[1] University of Luxembourg
6, rue Richard Coudenhove-Kalergi, L–1359 Luxembourg
{zhe.liu,johann.groszschaedl}@uni.lu
[2] Pusan National University
San-30, Jangjeon-Dong, Geumjeong-Gu, Busan 609–735, Korea
{hwajeong,howonkim}@pusan.ac.kr
[3] Katholieke Universiteit Leuven
Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium
{sujoy.sinharoy,ingrid.verbauwhede}@esat.kuleuven.be

Abstract. Public-key cryptography based on the "ring-variant" of the Learning with Errors (ring-LWE) problem is both efficient and believed to remain secure in a post-quantum world. In this paper, we introduce a carefully-optimized implementation of a ring-LWE encryption scheme for 8-bit AVR processors like the ATxmega128. Our research contributions include several optimizations for the Number Theoretic Transform (NTT) used for polynomial multiplication. More concretely, we describe the Move-and-Add (MA) and the Shift-Add-Multiply-Subtract-Subtract (SAMS2) technique to speed up the performance-critical multiplication and modular reduction of coefficients, respectively. We take advantage of incompletely-reduced intermediate results to minimize the total number of reduction operations and use a refined memory-access pattern to decrease the RAM footprint of an NTT multiplication. Furthermore, we propose a byte-wise scanning strategy to improve the performance of a discrete Gaussian sampler based on the Knuth-Yao random walk algorithm. For medium-term security, our ring-LWE implementation needs 590 k, 672 k, and 276 k clock cycles for key-generation, encryption, and decryption, respectively. On the other hand, for long-term security, the execution time of key-generation, encryption, and decryption amount to 2.2 M, 2.6 M, and 686 k cycles, respectively. These results set new speed records for ring-LWE encryption on an 8-bit processor and outperform related RSA and ECC implementations by an order of magnitude.

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

1 Introduction

The vast majority of today's widely-used public-key cryptosystems is based on integer factorization and discrete logarithm problems, which are believed to be intractable with current computing technology. However, these hard problems can be solved by using Shor's algorithm [32] (or a variant of it) on a quantum computer. Lattice-based cryptography is often considered a premier candidate for realizing post-quantum cryptosystems [28]. Its security relies on worst-case computational assumptions in lattices that will remain hard even for quantum computers. In the recent past, a large body of research has been devoted to the efficient implementation of lattice-based cryptosystems, whereby resource-constrained environments received particular attention (see e.g. [5, 10, 22]). This is much owed to the fact that the Internet is currently in the midst of a transition from a network connecting commodity computers (i.e. PCs and notebooks) to a network of smart objects ("things"). Even today, there are significantly more non-traditional computing devices connected to the Internet than conventional computers [14]. Among the smart devices that are populating the Internet are various kinds of sensors, actuators, meters, consumer electronics, medical monitors, household appliances, vehicles, and even items of clothing. Many of these devices are very restricted in terms of computing power, memory capacity, and energy supply. For example, a typical wireless sensor node, like the widely-used MICAz mote, features an 8-bit AVR ATmega processor clocked at 8 MHz and a few kB of RAM. However, in order to enable such devices to communicate in a secure way, they need to be capable of executing public-key cryptography as otherwise end-to-end authentication and end-to-end key exchange would not be possible. Implementing public-key algorithms on an 8-bit processor poses quite a challenge, not only for RSA and ECC, but also post-quantum techniques like lattice-based cryptography. This raises the question of how well the "cryptosystems of the future" are suited for the "Internet of the future," i.e. the so-called "Internet of Things (IoT)," and one aspect of this question is the performance of lattice-based cryptosystems on 8-bit platforms such as AVR [2].

The introduction of the Learning With Errors (LWE) problem [28] and its ring variant (i.e. ring-LWE) [21] opened up a way to build efficient lattice-based public-key cryptosystems. The first practical evaluations of LWE and ring-LWE encryption were presented by Göttert et al. at CHES 2012 [15]. According to their results, the ring-LWE encryption scheme is at least four times faster and requires less memory than the encryption scheme based on the standard LWE problem. A large variety of subsequent hardware and software implementations of ring-LWE-based public-key encryption or digital signature schemes improved performance and memory footprint [22, 10, 5, 6, 25]. Oder et al. [22] introduced an efficient implementation of Bimodal Lattice Signature Schemes (BLISS) on a 32-bit ARM Cortex-M4F processor; the most optimized variant of their software needs 6 M cycles for signing, 1 M cycles for verification, and 368 M cycles for key generation, respectively, at a medium-term security level. Recently, De Clercq et al. [10] described a ring-LWE encryption scheme on exactly the same ARM platform and reported an execution time of 121 k cycles per encryption

and 43.3 k cycles per decryption for medium-term security, which increases to 261 k cycles (encryption) and roughly 96.5 k cycles (decryption) when long-term security is desired. The first implementation of a lattice-based cryptosystem on an 8-bit processor was published by Boorghany et al. in 2014 [5, 6]. They evaluated four lattice-based authentication protocols on both an 8-bit AVR and a 32-bit ARM processor. On the 8-bit platform (i.e. AVR), their implementation of the Fast Fourier Transform (FFT) needs 755 k and 2.2 M cycles for medium and long-term security, respectively. Thanks to the efficiency of the polynomial multiplication and the Gaussian sampler function, their LWE-based encryption scheme achieves an execution time of 2.8 M cycles for key generation, 3 M cycles for encryption, as well as 1.4 M cycles for decryption, all at a medium-term security level. Very recently, Pöppelmann et al. [25] compared implementations of ring-LWE encryption and the Bimodal Lattice Signature Scheme (BLISS) on an 8-bit ATxmega128 processor. For medium-term security, they reported 1.3 M cycles for ring-LWE encryption and 381 k cycles for decryption, respectively.

1.1 Research Contributions

This paper continues the line of research on the efficient implementation of the ring-LWE encryption scheme on 8-bit AVR processors. Our core contributions are several optimizations to reduce the execution time and RAM requirements of ring-LWE encryption, decryption, and key generation. More specifically, the contributions of this paper can be summarized as follows.

1. The efficiency of coefficient modular multiplication is crucial for high-speed NTT operations. We present the Move-and-Add (MA) method to perform the coefficient multiplication and the Shift-Add-Multiply-Subtract-Subtract (SAMS2) technique to accelerate the reduction operation. The former aims at reducing the number of `add` instructions by rescheduling the order of the byte multiplications, whereas the latter replaces expensive `MUL` instructions by cheaper shifts and additions.
2. In the NTT computations, the vast majority of execution time is spent on performing modular reduction since it is the most frequent operation in the innermost loop. We exploit the idea of incomplete modular arithmetic (see e.g. [34]), which means we allow (i.e. tolerate) incompletely reduced intermediate results for the coefficients and perform the reduction operation in a "lazy" fashion. Our experimental results show that this approach decreases the overall number of modular reductions by 6% on average.
3. The intermediate coefficients during the computation of an NTT require a considerable amount of RAM. We propose a refined memory access scheme that enables us to make full use of the allocated space. For example, when the coefficients are 13 bits long, we keep 16 coefficients in 26 bytes, and save in this way up to 19% RAM compared to the straightforward approach.
4. To increase efficiency of our discrete Gaussian sampler based on the Knuth-Yao random walk algorithm [17], we propose a byte-scanning technique to minimize execution time.

On basis of these optimizations, we present a total of four implementations of a ring-LWE encryption scheme for the 8-bit AVR platform (e.g. AT(x)mega microcontrollers); two at the medium-term security level and two for long-term security. For each of these two security levels, we developed both a High-Speed (HS) and a Memory-Efficient (ME) variant. For medium-term security, the HS implementation requires roughly 590 k, 672 k, and 276 k clock cycles to perform a key-generation, encryption, and decryption, respectively. Alternatively, at the long-term security level, the speed-optimized key-generation, encryption, and decryption take 2.2 M, 2.6 M, and 686 k clock cycles, respectively. Both our HS and ME implementation significantly improve the speed records for ring-LWE encryption on an 8-bit AVR processor. Furthermore, it should be noted that all optimizations described in this paper can also be used to speed up LWE-based signature schemes (e.g. [24]) on the AVR platform.

1.2 Paper Outline

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

2 Background

2.1 The Ring-LWE Encryption Scheme

The encryption schemes used in this paper are based on the ring version of the Learning With Errors (i.e. ring-LWE) problem. The more general form of this problem, i.e. the LWE problem, is parameterized by a dimension $n \geq 1$, a modulus q, and an error distribution. This error distribution is generally taken as a discrete Gaussian distribution $\mathcal{X}_\sigma$ with standard deviation σ and mean 0 so as to achieve the best entropy/standard deviation ratio [11]. In the literature, the LWE problem is, in general, defined as follows: 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 must be kept as a secret. The LWE distribution $A_{s,\mathcal{X}}$ is defined over $\mathbb{Z}_q^n \times \mathbb{Z}_q$ and comprises 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 with a polynomial number of $(\mathbf{a}, t)$ pairs sampled from $A_{s,\mathcal{X}}$ and his task is to try to find the secret polynomial $\mathbf{s}$. Similarly, in the *decision* version of the LWE problem, the attacker attempts 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, Lyubashevsky et al. [21] 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. As the name suggests, in the the ring-LWE problem, the elements a, s, and t are polynomials in the ring R_q. Lyubashevsky et al.'s ring-LWE encryption scheme was later optimized by Roy et al. [29] with the aim of reducing the cost of the polynomial arithmetic. In their scheme, the polynomial arithmetic carried out during a decryption operation requires only one Number Theoretic Transform (NTT) operation. Besides this computational optimization, Roy et al.'s scheme performs sampling from the discrete Gaussian distribution using a Knuth-Yao sampler. In the remainder of this section, we will first describe the major steps of Roy et al.'s version of the encryption scheme and thereafter we will recap the mathematical concepts of the NTT and the Knuth-Yao sampling.

2.2 Key Generation, Encryption, and Decryption

In the following, we describe the steps used in the encryption scheme proposed by Roy et al. [29]. We denote the NTT of a polynomial a by $\tilde{a}$.

- 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:
$$\tilde{r_1} = \mathrm{NTT}(r_1),\ \tilde{r_2} = \mathrm{NTT}(r_2)$$
and then an operation $\tilde{p} = \tilde{r_1} - \tilde{a} \cdot \tilde{r_2} \in R_q$ is performed. The public key is the polynomial pair $(\tilde{a}, \tilde{p})$ and the private key is the polynomial $\tilde{r_2}$.
- 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 the message M by $q/2$. Thereafter, three error polynomials $e_1, e_2, e_3 \in R_q$ are sampled from $\mathcal{X}_\sigma$. The ciphertext can be obtained as a set of two polynomials $(\tilde{C_1}, \tilde{C_2})$:
$$(\tilde{C_1}, \tilde{C_2}) = (\tilde{a} \cdot \tilde{e_1} + \tilde{e_2}, \tilde{p} \cdot \tilde{e_1} + \mathrm{NTT}(e_3 + M'))$$
- Decryption stage **Dec($\tilde{C_1}$, $\tilde{C_2}$, $\tilde{r_2}$)**: One inverse NTT has to be performed to recover M':
$$M' = \mathrm{INTT}(\tilde{r_2} \cdot \tilde{C_1} + \tilde{C_2})$$
and then a decoder is used to recover the original message M from M'.

2.3 Number Theoretic Transform

Our implementation adopts the Number Theoretic Transform (NTT) [8] to perform the required polynomial multiplications. An NTT can be seen as a variant of the Fast Fourier Transform (FFT) that operates in a finite ring $\mathbb{Z}_q$. Instead of using complex roots of unity, an 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

Algorithm 1. Iterative Number Theoretic Transform

Input: Polynomial $a(x) \in \mathbb{Z}_q[x]$ of degree $n-1$, primitive n-th root of unity $\omega \in \mathbb{Z}_q$
Output: Polynomial $a(x) = \mathrm{NTT}(a) \in \mathbb{Z}_q[x]$

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

ω_n denotes a primitive n-th root of unity. Algorithm 1 shows the iterative form of the NTT algorithm, which is taken from Cormen et al. [8].

As specified in Algorithm 1, the iterative NTT algorithm consists of three nested loops. The outermost loop (i-loop, line $2 \sim 13$) starts from $i = 2$ and increases i by doubling it in each iteration. When $i = n$, the loop terminates, i.e. the overall number of iterations is only $log_2(n)$. In each iteration, the value of the twiddle factor ω_i is computed by executing a power operation $\omega_i = \omega_n^{n/i}$, and the value of ω is initialized by 1. Compared to the i-loop, the j-loop (line $4 \sim 12$) 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 11. Apparently, the innermost loop (k-loop, line $5 \sim 10$) occupies most part of the execution time of the NTT algorithm since it is executed roughly $\frac{n}{2} log_2 n$ times. In each iteration of the innermost loop (line $6 \sim 9$), two coefficients $a[i+j]$ and $a[i+j+i/2]$ are loaded from memory into registers, and then $a[i+j+i/2]$ are multiplied by the twiddle factor ω. Thereafter, the value of $a[k+j]$ and $a[k+j+i/2]$ are updated and stored in memory.

2.4 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 [17, 30, 13]. A DDG tree efficiently counts the visited non-zero nodes to find the sample based on probability.

Algorithm 2. Low-level implementation of Knuth-Yao sampling [30]

Input: Probability matrix P_{mat}, random number r, modulus q
Output: Sample value s

```
1:  for col from 0 by 1 to MAXCOL do
2:      d ← 2d + (r&1)
3:      r ← r ≫ 1
4:      for row from MAXROW by −1 to 0 do
5:          d ← 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:             end if
12:         end if
13:     end for
14: end for
15: return 0
```

A low-level implementation of the Knuth-Yao random walk along the DDG tree was proposed in [30], which is shown in Algorithm 2. The random walk reads the probability bits of the sample points from a matrix known as the probability matrix (P_{mat}). The i-th row of P_{mat} is the probability of the sample point $|i|$. The algorithm uses two loops with counters *col* and *row* to read the bits from the columns and rows of P_{mat}. The two loop boundaries *MAXCOL* and *MAXROW* represent the number of columns and rows of P_{mat}. Before starting the random walk, a counter d is initialized to zero. Whenever a new column of P_{mat} is to be read, the counter d is updated using a random bit r. During the random walk, the visited column of P_{mat} is scanned bit-by-bit, and each non-zero bit in the column decrements the value of d. When d becomes negative for the first time, the random walk stops and the value of the *row* counter is taken as the magnitude of the sample. Now another random bit is generated to determine the sign of the sample. For a more detailed description of the Knuth-Yao random walk, interested readers may follow [30]. Faster versions of the Knuth-Yao random walk were presented in [29, 10] using small lookup tables.

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, 29] and software implementations, e.g., [5, 6, 10]. This also helps us to compare our work with previous work.

3 Optimization Techniques for NTT Computation

3.1 Look-Up Table for Twiddle Factors

In each iteration of the j-loop, a new twiddle factor ω (line 11 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 $n - 1$. A straightforward computation of ω on-the-fly involves memory-access of ω and ω_i and a modular integer multiplication. Hence for an NTT computation, a significant portion of the computation time is spent on calculating the twiddle factors. On the other hand, storing all the intermediate twiddle factors ω and ω_i in the RAM is extremely expensive considering the fact that an 8-bit AVR processor has only several kilo bytes of RAM.

However, both of the computations of the power of ω_n in i-loop and twiddle factor $\omega \leftarrow \omega \cdot \omega_i$ in j-loop can be considered as fixed costs. Based on this observation, our solution is to store all the twiddle factors ω in a ROM. This is similar to the technique used in [26] for a hardware implementation. More specifically, we pre-compute the twiddle factor "off-line" and store them in a look-up table in the flash ROM. We only need to transfer the twiddle factor that is required for the current iteration of the j-loop from ROM to RAM. This requires only two bytes of the RAM.

3.2 Algorithmic Optimization

We follow the parameter sets from [29] 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 [24, 29]. 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 [3], merging the scaling operation by n^{-1} with the chain of multiplications during the post processing operation in the inverse NTT [29].

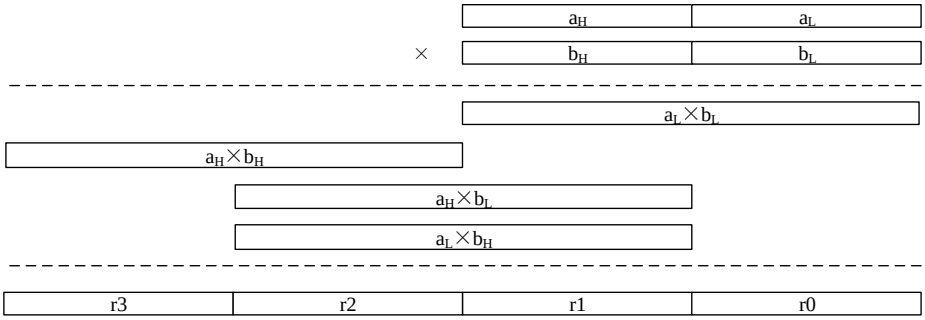

Fig. 1. The MOV-and-ADD coefficient multiplication.

3.3 Efficient Coefficient Multiplication

During an NTT computation, $\frac{n}{2}\log_2 n$ coefficient multiplications are performed in the nested loops. Hence efficient implementation of the coefficient multiplication operation is essential to achieve fast computation time. In our implementation the coefficients are 13-bit or 14-bit long, and thus can be accommodated in two 8-bit registers. Inspired by the hybrid technique for multi-precision multiplication [16], we propose the "MOV-and-ADD" technique for performing coefficient multiplications. The MOV-and-ADD multiplication technique aims at minimizing the number of `adc` instructions. As shown in Figure 1, the two coefficients a and b are fetched from the RAM and then loaded 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 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 a perquisite for high-speed implementation of NTT algorithm.

We propose an optimized SAMS2 reduction technique for performing the $\bmod 7681$ and $\bmod 12289$ 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 have 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.

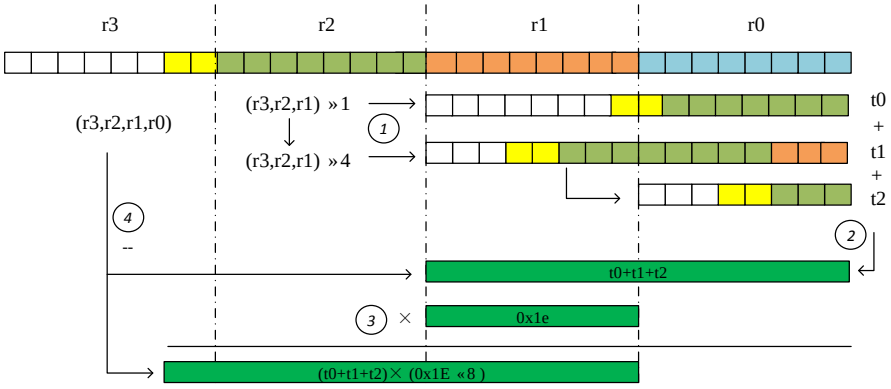

Fig. 2. Fast reduction operation with SAMS2 method for $q = 7681$. ①: shifting; ②: addition; ③: multiplication; ④: subtraction.

Another property of SAMS2 technique is its economic register usage; it occupies only 14 out of the 32 available registers [2] such that no `push`/`pop` instructions are required at the beginning/end of the function. SMAS2 method can also be used for modulus $q = 12289$.

3.5 Reducing 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 $[0, 2^{\lceil log_2 p \rceil}]$ [18]. 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 Reducing the 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 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. Similar memory-access scheme can be used for the modulus $q = 12289$.

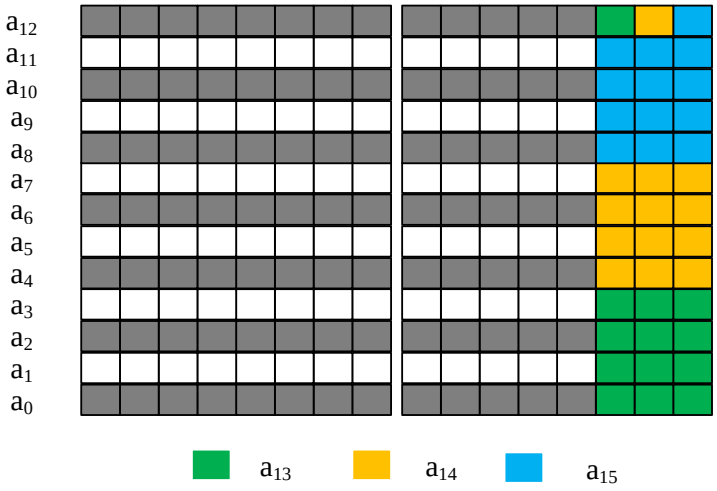

Fig. 3. Reducing the RAM consumption with refined memory-access scheme for $q = 7681$. Each row represents a memory of two bytes, while each block represents one bit.

4 Optimizations of the Knuth-Yao Sampler

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 [10], however, we propose a byte-wise method to further reduce the execution time.

Probability Matrix with Small Memory Footprint. 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 [10]. 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.

Byte-Wise Scanning. The bit-scanning operation as shown in Algorithm 2 (line 7) 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. As shown in Algorithm 3 (line 15 $\sim$ 17), the byte-wise method only requires eight additions, one subtraction

The ROM consumption of probability matrix can be further reduced to 4352 bytes by eliminating consecutive zero bits. In order to make a balance between the execution time and ROM consumption, we decide to use the current version in our implementation.

Algorithm 3. Knuth-Yao Sampling with byte-wise scanning

Input: Probability matrix P_{mat}, random number r, modulus q
Output: Sample value s

```
1: index ← r&255
2: r ← r ≫ 8
3: s ← LUT1[index]
4: if msb(s) = 0 then
5:    if (r&1) = 1 then
6:       return q − s
7:    else
8:       return s
9:    end if
10: end if
11: d ← s&7
12: for col from 8 by 1 to MAXCOL do
13:    d ← 2d + (r&1)
14:    r ← r ≫ 1
15:    for row from MAXROW by −8 to 0 do
16:       if (P_mat[row][col]||P_mat[row − 1][col]||...||P_mat[row − 7][col]) > 0 then
17:          sum = Σ_{i=row}^{row−7} (P_mat[i][col])
18:          d ← d − sum
19:          if d < 0 then
20:             if d = −1 then
21:                for j from row − 7 by 1 to row do
22:                   if P_mat[j][col] = 1 then
23:                      if (r&1) = 1 then
24:                         return q − j
25:                      else
26:                         return j
27:                      end if
28:                   end if
29:                end for
30:             else
31:                for j from row − 7 by 1 to row do
32:                   d ← d + P_mat[j][col]
33:                   if d = −1 then
34:                      if (r&1) = 1 then
35:                         return q − j
36:                      else
37:                         return j
38:                      end if
39:                   end if
40:                end for
41:             end if
42:          end if
43:       end if
44:    end for
45: end for
46: return 0
```

and one conditional branch statements, which saves seven conditional branch statements at the cost of one subtraction.

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 conduct the simple comparison between zero and bit counter (line 18 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 [10] into our byte-wise scanning implementations (shown in lines $2 \sim 9$ 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.

4.1 Pseudo Random Number Generator with AES Accelerator

We choose the PRNG algorithm suggested by [27] that runs the AES block cipher in counter mode, encrypting successive values of an incrementing counter. Atmel's ATxmega128A1 begins to support the AES crypto-accelerator that computes encryptions with reasonable computation overheads (i.e. 375 clock cycles) and small memory footprint for the AES trigger program. This is a significant progress compared to the software implementation of AES on the ATmega128 processor, which requires 1993 clock cycles and 2K program [23] with pre-computations. Another attractive feature of the ATxmega128A1 is the build-in AES accelerator that can be operated independently of the microprocessor, which therefore eliminates the latencies due to the AES encryption [31]. We exploit this idea into our Knuth-Yao sampler implementation. More specifically, we trigger the AES operation immediately after obtaining the output of the AES encryption and then conduct other operations. However, the AVR microprocessor only supports 128-bit AES accelerator. For the case of long-term security level, we choose to use the software version of the 256-bit AES developed by AVR-Crypto-Lib, where the 256-bit AES encryption takes 3521 clock cycles [9].

5 Performance Evaluation and Comparison

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 Atmel Studio 6.2. 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	`Key-Gen`	`Enc`	`Dec`
HS-256	193,731	26,763	589,900	671,628	275,646
ME-256	322,288	39,027	1,310,616	1,532,823	673,489
HS-512	441,572	255,218	2,165,239	2,617,459	686,367
ME-512	917,866	300,780	3,738,052	4,270,671	1,444,786

As shown in the Table 1, the NTT operation only requires $193,731$ clock cycles for HS-256 implementation, however, the execution time increases sharply to $441,572$ cycles for HS-512 implementation. The Knuth-Yao sampler for HS-256 requires an average of $26,763$ cycles, while $255,218$ 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 $590K$, $670K$ and $275K$, 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	`Key-Gen`	`Enc`	`Dec`	`Total`
RAM/ROM (HS-256)	1,585/8,884	2,609/8,812	1,585/6,026	2,609/13,604
RAM/ROM (ME-256)	1,297/9,260	2,129/8,536	1,297/6,016	2,129/13,756
RAM/ROM (HS-512)	3,121/12,074	6,193/13,486	3,121/8,512	6,193/18,894
RAM/ROM (ME-512)	2,737/12,106	4,529/12,166	2,737/8,614	4,529/18,010

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 $13.6K$ ROM, while the ME-256 needs $2.1K$ RAM and $13.7K$ ROM. 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 [9].

5.3 Comparison with Related Work

Table 3 compares software implementations of lattice-based cryptosystems on different processors. For the 8-bit AVR platform, the previous work [5, 6, 25] and our implementation adopt the same parameter sets as we mentioned in Subsection 2.5. Compared to the recent work [25], our HS-256 only requires $670K$ and $275K$ cycles for encryptionand decryption, which is roughly 2X and 1.4 faster. The significant progress achieved is mainly due to a combination of algorithmic optimizations and the proposed practical optimization techniques 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 [19], the authors reported an execution time of $76.58 \cdot 10^6$ clock cycles for RSA decryption with 80-bit security level. For a comparison, our HS-256 only requires $275,646$ cycles, which is more than 278 times faster even with a higher 128-bit security

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 comparison.

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

Implementation	NTT/FFT	Sampling	Key-Gen	Enc	Dec
Implementations on high-performance processors, e.g. Core 2 Duo:					
Göttert [15] (256)	n/a	n/a	9,300,000	4,560,000	1,710,000
Göttert [15] (512)	n/a	n/a	13,590,000	9,180,000	3,540,000
Implementations on 32-bit ARM processors, e.g. Cortex-M4F:					
De Clercq [10] (256)	31,583	7,296	117,009	121,166	43,324
De Clercq [10] (512)	71,090	14,592	252,002	261,939	96,520
Oder [22] (512)	122,619	935,936	n/a	n/a	n/a
Implementations on 8-bit AVR processors, e.g. ATxmega64, ATxmega128:					
Boorghany [6] (256)	1,216,000	n/a	n/a	5,024,000	2,464,000
Boorghany [5] (256)	754,668	n/a	2,770,592	3,042,675	1,368,969
Pöppelmann [25] (256)	334,646	n/a	n/a	1,314,977	381,254
This work (HS-256)	193,731	26,763	589,900	671,628	275,646
Boorghany [5] (512)	2,207,787	617,600	n/a	n/a	n/a
Pöppelmann [25] (512)	855,595	n/a	n/a	3,279,142	1,019,350
This work (HS-512)	441,572	255,218	2,165,239	2,617,459	686,367

level. They are a few software ECC implementations on 8-bit AVR processors. For example, Düll et al. in [12] reported execution time of $13,900,397$ (HS version) and $14,146,844$ (ME version) clock cycles for single scalar multiplication using the Curve25519 [4]. 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 work in [12, 20, 1]. These research results also show that the ring-LWE encryption is advantageous to traditional PKC for resource-constraint microncontrollers in case of performance.

6 Conclusion

This paper presented several optimizations to efficiently implement a ring-LWE encryption scheme on 8-bit AVR platform. In particular, we proposed three optimizations to accelerate the execution time and a refined memory-access technique 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 twice as fast as the previous best implementation. We also reported the results we obtained for speed-optimized and a memory-optimized implementations for both medium-term and long-term security levels. All of these achieved results set new speed records for an implementation of a ring-LWE encryption scheme on the 8-bit AVR platform. Finally, a comparison of our

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
Gura et al. [16]	RSA-1024	N/A	N/A	3,440,000	87,920,000
Liu et al. [19]	RSA-1024	N/A	N/A	N/A	75,680,000
Düll et al. [12] (ME)	ECC-255	510	9,912	28,293,688	14,146,844
Düll et al. [12] (HS)	ECC-255	494	17,710	27,800,794	13,900,397
Liu et al. [20]	ECC-256	556	14,700	30,539,566	21,118,778
Aranha et al. [1]	ECC-233	3,700	38,600	11,796,480	5,898,240
This work (HS)	LWE-256	2,609	13,604	671,628	275,646
This work (ME)	LWE-256	2,129	13,756	1,532,823	673,489

implementation with traditional public-key cryptography (i.e. RSA, ECC) sheds some new light on the practical application of ring-LWE in resource-constraint environments.

Acknowledgments

Zhe Liu would like to thank Dr. Frederik Vercauteren and Ruan de Clercq from K.U Leuven for useful discussions and suggestions about this work. Zhe Liu is supported by FNR-AFR project (No.1359142), Luxembourg; Hwajeong Seo is supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIP) (No.10043907, Development of high performance IoT device and Open Platform with Intelligent Software). S. Sinha Roy is supported by Erasmus Mundus PhD Scholarship. This work is supported by the Research Council KU Leuven: TENSE (GOA/11/007), by iMinds, by the Flemish Government, FWO G.0550.12N, G.00130.13N and FWO G.0876.14N, by the Hercules Foundation AKUL/11/19.

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. Atmel Corporation. 8-bit AVR® Instruction Set. User Guide, available for download at `http://www.atmel.com/dyn/resources/prod_documents/doc0856.pdf`, July 2010.
3. 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.
4. 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.

5. A. Boorghany, S. Bayat-Sarmadi, 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.
6. 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.
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. T. Cormen, C. Leiserson, and R. Rivest. *Introduction To Algorithms*. http://staff.ustc.edu.cn/$\sim$csli/graduate/algorithms/book6/toc.htm.
9. Das Labor. AVR Crypto Lib, available for download at http://avrcryptolib.das-labor.org/trac, 2012.
10. 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.
11. L. Ducas. Lattice based signatures: Attacks, analysis and optimization. Ph.D Thesis, 2013. http://cseweb.ucsd.edu~lducas/Thesis/index.html.
12. M. Düll, H. Björn, G. Hinterwälder, M. Hutter, C. Paar, A. H. Sánchez, and P. Schwabe. High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers. http://eprint.iacr.org/2015/343.pdf.
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. In *Cryptographic Hardware and Embedded Systems–CHES 2012*, pages 512–529. Springer, 2012.
16. 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.
17. 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.
18. 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.
19. 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.

20. 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. Vaudenay, 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.
21. 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.
22. 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.
23. D. A. Osvik, J. W. Bos, D. Stefan, and D. Canright. Fast software AES encryption. In *Fast Software Encryption*, pages 75–93. Springer, 2010.
24. 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.
25. T. Pöppelmann, Tobias Oder, and T. Güneysu. Speed Records for Ideal Lattice-Based Cryptography on AVR. In `http://eprint.iacr.org/2015/382.pdf`.
26. 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.
27. 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`.
28. 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.
29. 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.
30. 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.
31. 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.
32. 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.
33. Tinker London. TrueRandom library for Arduino. User Guide, available for download at `http://code.google.com/p/tinkerit/wiki/TrueRandom`, 2010.
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.