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 special coefficient-storage method to decrease the RAM footprint of NTT multiplications. In addition, 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 use a special coefficient-storage method 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 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].

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: $a \leftarrow \mathrm{BitReverse}(a)$
 2: for $i$ from 2 by $2i$ to $n$ do
 3:     $\omega_i \leftarrow \omega_n^{n/i}$, $\omega \leftarrow 1$
 4:     for $j$ from 0 by 1 to $i/2-1$ do
 5:         for $k$ from 0 by $i$ to $n-1$ do
 6:             $U \leftarrow a[k+j]$
 7:             $V \leftarrow \omega \cdot a[k+j+i/2]$
 8:             $a[k+j] \leftarrow U+V$
 9:             $a[k+j+i/2] \leftarrow U-V$
10:         end for
11:         $\omega \leftarrow \omega \cdot \omega_i$
12:     end for
13: end for
14: return $a$
```

As can be seen from Algorithm 1, an iterative NTT consists of three nested loops. The outermost loop (i-loop, line 2 to 13) starts with $i = 2$ and increases i by doubling it in each iteration. When $i = n$, the loop terminates, and so the overall number of iterations is only $\log_2(n)$. In each iteration, the value of the so-called twiddle factor ω_i is computed via an exponentiation $\omega_i = \omega_n^{n/i}$, while the value of ω is initialized with 1. Compared to the i-loop, the j-loop (i.e. line 4 to 12) executes more iterations, whereby the actual number of iterations can be seen as a sum of a geometric progression for 2^i with i starting from 0 and having a maximum value of $\log_2(n-1)$. Thus, the j-loop is iterated $n-1$ times and, in each iteration, the twiddle factor ω is updated by performing a coefficient modular multiplication (line 11). Apparently, the innermost loop (i.e. the k-loop, line 5 to 10) consumes the majority of the total execution time of the NTT algorithm since it is iterated roughly $\frac{n}{2} \cdot \log_2(n)$ times. In each iteration of the innermost loop, the two coefficients $a[i+j]$ and $a[i+j+i/2]$ are loaded from memory into registers, and then $a[i+j+i/2]$ is multiplied by the twiddle factor ω. Thereafter, the values 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 approaches for sampling from a discrete Gaussian distribution, among which we chose the algorithm of Knuth and Yao [17]. This algorithm stores the probabilities of the sample points and performs a random walk by following a binary tree, known as the Discrete Distribution Generating (DDG) tree [17, 30, 13]. Such 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 $s$
$d \leftarrow 0$
for $col$ from 0 by 1 to $MAXCOL$ do
    $d \leftarrow 2d + (r\&1)$
    $r \leftarrow r \gg 1$
    for $row$ from $MAXROW$ by $-1$ to 0 do
        $d \leftarrow d - P_{mat}[row][col]$
        if $d = -1$ then
            if $(r\&1) = 1$ then
                return $q - row$
            else
                return $row$
            end if
        end if
    end for
end for
return 0
```

A low-level implementation of the Knuth-Yao random walk along the DDG tree was described by Roy et al. [30] and is shown in Algorithm 2. The random walk reads the probability bits of the sample points from a matrix, called 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}, respectively. The two loop boundaries $MAXCOL$ and $MAXROW$ represent the overall number of columns and rows of P_{mat}. Before starting the random walk, a counter d has to be 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. We refer to [30] for a more-detailed description. Faster versions of the Knuth-Yao random walk algorithm using small lookup tables were presented in [29, 10].

2.5 Parameter Selection

Our implementation of ring-LWE encryption adopts the parameter set (n, q, σ) with $(256, 7681, 11.31/\sqrt{2\pi})$ and $(512, 12289, 12.18/\sqrt{2\pi})$ to match the common 128 and 256-bit security levels, respectively. The discrete Gaussian sampler is limited to 12σ to have a high-precision statistical difference from the theoretical distribution, which is less than 2^{-90}. These parameter sets were also used by most of the previous hardware (e.g. [15, 29]) and software implementations (e.g. [5, 6, 10]), which facilitates a comparison with these works.

3 Optimization Techniques for NTT Computation

3.1 Look-Up Table for Twiddle Factors

In each iteration of the j-loop of Algorithm 1, a new twiddle factor ω is computed through a modular multiplication (line 11). The total number of times a new ω is obtained in an NTT operation amounts to $n-1$. A straightforward computation of ω on-the-fly involves two memory accesses (to load ω and ω_i) and a modular multiplication. Hence, in an NTT computation, a considerable portion of the execution time is spent with calculating the twiddle factors. On the other hand, storing all intermediate twiddle factors ω and ω_i in memory is also problematic due to the fact that a standard 8-bit AVR processor features only a few kB of RAM.

Both the computation of the (n/i)-th power of ω_n in the i-loop (line 3) and the modular multiplication $\omega \cdot \omega_i$ in the j-loop (line 11) can be considered as fixed costs. Based on this observation, our solution is to store all the twiddle factors ω in ROM (resp. flash), very similar to the approach used in [26] for a hardware implementation. More concretely, we pre-compute the twiddle factors "off-line" and store them in a look-up table in ROM or flash so that we need to transfer only the twiddle factor that is required for the current iteration of the j-loop from ROM to RAM. In this way, only two bytes in RAM are needed.

3.2 Algorithmic Optimizations

The parameter sets of our ring-LWE encryption scheme given in the previous section use a modulus q that is prime and satisfies $q \equiv 1 \bmod 2n$ [29]. In such a setting, a polynomial multiplication can be carried out efficiently with only n-point NTTs (resp. INTT) due to special technique known as negative wrapped convolution, which has used before in a number of 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 sets is mainly driven by the fact that our target platform, an 8-bit AVR processor, is severely limited in resources, and performing $2n$-point NTTs (resp. a $2n$-point INTT) during a polynomial multiplication would increase the execution time and RAM requirements. Besides the application of the negative wrapped convolution, we adopt some other optimization techniques that were firstly proposed for hardware designs, but are suitable to improve the execution time of a software implementation too. These optimizations include the interchanging of the j and k-loops in the NTT algorithm [3], and the merging of the scaling operation by n^{-1} with the chain of multiplications performed during the post-processing operation in the inverse NTT [29].

3.3 Fast Coefficient Multiplication

During an NTT computation, $\frac{n}{2} \cdot \log_2(n)$ coefficient multiplications are carried out in the nested loops. Therefore, an efficient implementation of the coefficient

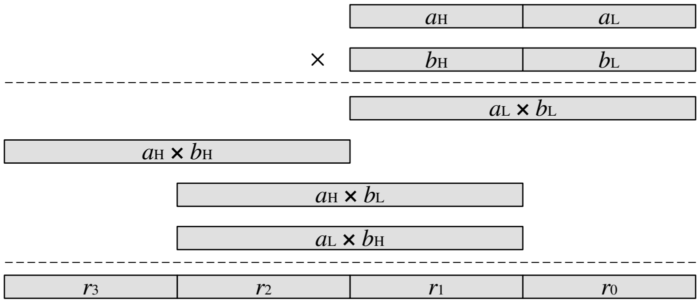

Fig. 1. The Move-and-Add (MA) coefficient multiplication

multiplication operation is essential to achieve fast execution time. Due to the parameter sets we use, the coefficients are 13 or 14 bits long, and, thus, can be stored in two 8-bit registers. Inspired by the hybrid method for multi-precision multiplication [16], we propose a "Move-and-Add" (MA) technique to perform the coefficient multiplications. The MA multiplication technique, illustrated in Fig. 1, aims at minimizing the total number of `adc` instructions. First, the two coefficients a and b are loaded from RAM and stored in two registers. We then 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 $r0$ and $r1$ with help of the `movw` instruction [2]. Next, we form the product $a_H \cdot b_H$ and move the result to registers $r2$ and $r3$. Thereafter, we multiply byte a_H by byte b_L, add the resulting 16-bit product $a_L \cdot b_H$ to the register pair $r1, r2$, and propagate a potential carry from the last addition to $r3$. Finally, we perform the byte multiplication of a_L by b_H in the same way as before, i.e. the product is added to the registers $r1, r2$ and the carry bit is propagated into $r3$. In summary, the execution of our MA method for fast coefficient multiplication involves four `mul`, two `movw`, and a total of six `add` or `adc` instructions.

3.4 Fast Reduction of Coefficient-Products

In an NTT computation, the majority of execution time is spent on performing modular reduction of coefficient products since this operation is costly and has to be carried out in the innermost k-loop. Thus, an efficient reduction operation is a perquisite for a high-speed implementation of the NTT algorithm.

We propose a special Shift-Add-Multiply-Subtract-Subtract (SAMS2) technique to perform the reduction operation modulo $q = 7681$ and $q = 12289$. The main idea is as follows. Let z be a product of two coefficients, i.e. $z = a \cdot b$. To obtain $z \bmod q$, we estimate the quotient $t = z/q$, and then subtract $t \cdot q$ from z, i.e. we compute $z - t \cdot q$. This result may not be fully reduced, which means it can be necessary to do a correction, i.e. to do a few final subtractions. Since $2^{13} \equiv 2^9 - 1 \bmod 7681$, it is not difficult to see that t can be approximated via $(z \gg 13) + (z \gg 17) + (z \gg 21)$. Consequently, the modular reduction involves

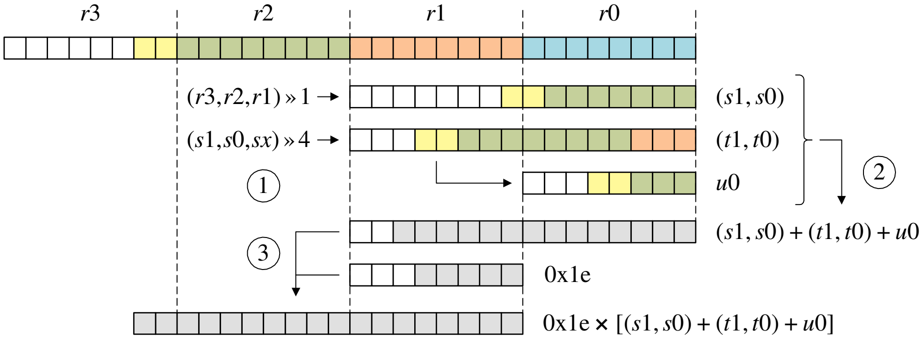

Fig. 2. The first three steps of the SAMS2 method for reduction modulo 7681. ①: shifting; ②: addition; ③: multiplication

four different basic operations, namely, Shifting → Addition → Multiplication → Subtraction → Subtraction (SAMS2). As illustrated in Fig. 2, we keep the product z in the four 8-bit registers $r3$, $r2$, $r1$, and $r0$, all of which are marked by different colors. A fully colored register means that all its bits are occupied (e.g. $r0$–$r2$), whereas white squares indicate that the corresponding bits of the register are empty (e.g. $r3$). The reduction modulo 7681 is done as follows:

1. Shifting: We first shift the three bytes in the registers $(r3, r2, r1)$ one bit to the right and store the result in the registers $(s1, s0, sx)$, whereby only the former two are shown in Fig. 2. Then, we right-shift $(s1, s0, sx)$ by four bits and write the result (which consists of only two bytes) to the two registers $(t1, t0)$. The content of register $u0$ is the same as that of $t1$.
2. Addition: We compute the sum of $(s1, s0)$, $(t1, t0)$, and $u0$. Apparently, this sum is always less than 16 bits long and can be stored in two registers.
3. Multiplication: Now, we multiply the sum from Step 2 by the constant 0x1e (i.e. the upper byte of 7681), which is a (16×8)-bit multiplication.
4. Subtraction: In this step, we subtract both the sum obtained in Step 2 and the product computed in Step 3 from z. The product from Step 3 has to be aligned as shown in Fig. 2, i.e. it must be left-shifted by eight bits.
5. Subtraction. The result from Step 4 may be larger than $q = 7681$; thus, we need to do a correction by subtracting the modulus q at most twice.

Besides achieving fast execution time, the SAMS2 method is also economic in register usage; it occupies only 14 out of the 32 available registers so that no push/pop instructions are required at the beginning/end of a function call. The SAMS2 technique can be easily adapted for the modulus $q = 12289$.

3.5 Minimizing the Number of Reduction Operations

Apart from the coefficient multiplications, addition and subtraction operations are also executed in the innermost NTT loop. In general, a coefficient addition

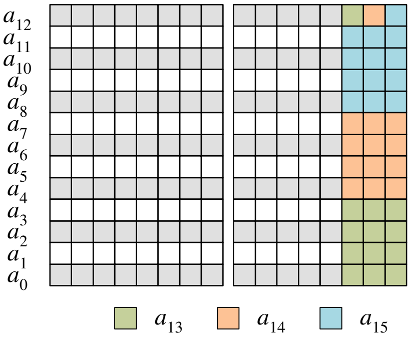

Fig. 3. Refined coefficient storage method to reduce the RAM footprint for $q = 7681$. Each row consists of two bytes, whereby each square represents a single bit.

$r = a + b \bmod q$ is carried out via a "conventional" integer addition of the two operands, followed by a conditional subtraction of the modulus q if the sum is not smaller than q to get a final result in the range of $[0, q - 1]$. A subtraction of coefficients can be performed in a similar fashion.

Inspired by the concept of "incomplete" modular arithmetic, as explained in e.g. [34], our implementation does not make an exact comparison between the sum $s = a + b$ and q, but rather compares s with 2^m where $m = \lceil \log_2(q) \rceil$, the bit-length of q. Taking $q = 7681$ as an example, the incomplete addition works as follows. We first perform a normal coefficient addition and then compare the higher byte of s with 2^5. If this byte is greater than or equal to 2^5 (which also means $r \geq 2^{13}$), then a subtraction of q needs to be performed. However, if the operands a and b are incompletely reduced (i.e. in the range $[0, 2^{13} - 1]$), up to two subtractions of q may be necessary [18]. Our implementation accepts two operands that are at most 13 bits long, but not necessarily smaller than q, and returns a result r for which the same holds. In the last iteration of the outermost loop of the NTT (i.e. when $i = 256$ in our case), a final correction process is performed to bring the result back into the range $[0, q - 1]$. This incomplete reduction technique can be used for coefficient addition, subtraction, as well as multiplication. Our practical results show that this approach allows one to save roughly 6% of the reduction operations, thereby speeding up the NTT.

3.6 Reducing the RAM Consumption

The NTT computation requires to store the coefficients of intermediate results in RAM, which is a precious resource on our target platform. This can pose a problem since the number of coefficients is very large and each coefficient needs two bytes. More specifically, when taking $q = 7681$ for dimension $n = 256$ as an example, each coefficient has a length of 13 bits (i.e. two bytes) and, hence, the intermediate result occupies a total of 512 bytes in RAM. In order to reduce the RAM footprint, we propose a special coefficient storage method (illustrated in Fig. 3) and refined memory-access technique.

Since the three most significant bits are empty when storing a 13-bit coefficient in two bytes, it is possible to accommodate 16 coefficients in 26 bytes in RAM. Each of the rows in Fig. 3 represents two bytes and each square marks a single bit. The first 13 coefficients (i.e. from a_0 to a_{12}) are stored in the rows in a straightforward way. Instead of allocating additional memory space for the 14th, 15th, and 16th coefficient, we make full use of the empty bits in the rows to store them. More precisely, we divide the 14th coefficient (i.e. a_{13} in Fig. 3) into two parts; the first part contains the lower 12 bits of a_{13} and gets placed in the empty space of the rows a_0, a_1, a_2, and a_3, while the second part, i.e. the most significant bit of a_{13}, is stored in the 14th bit of the 13th row. We do the same with the coefficients a_{14} and a_{15}, which are marked by different colors in Fig. 3. Due to this special storage method, the loading of the coefficients from RAM to registers needs some "post processing." The coefficients a_0 to a_{12} can be obtained via an AND operation with 0x1fff, while a_{13}, a_{14}, and a_{15} require to perform several loads and then an "assembling" of the bits to get the actual coefficient. Thanks to this coefficient-storage method, we are able to reduce the RAM requirements by 18.75% for $q = 7681$. A similar approach can be applied to the modulus $q = 12289$.

4 Optimization of the Knuth-Yao Sampler

The Knuth-Yao algorithm [17] requires a probability matrix P_{mat} that contains the probabilities of sampling a random number at a discrete position from the Gaussian distribution. Our Knuth-Yao implementation for AVR mainly adopts the optimizations in [10]; in addition, we propose a byte-wise scanning method to further reduce the execution time.

Probability Matrix with Low 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 our AVR processor, we stored each 55-bit column in seven words, where each word is 8 bits long. In this case, only one bit is wasted per column and the probability matrix just occupies 6,104 bytes in total[4].

Byte-Wise Scanning. The bit-scanning operation as specified in Algorithm 2 (line 6) requires to check each bit and decreases the distance (d) whenever the bit is set. Instead of doing this scanning operation at the bit level, we perform it in a byte-wise fashion. As indicated in Algorithm 3 (line 17 to 18), the byte-wise method only requires eight additions, one subtraction and one conditional branch statement, which means it saves seven branches at the (slight) expense of one subtraction.

[4] The ROM footprint of the probability matrix can be further reduced to 4352 bytes by eliminating consecutive zero bits. In order to make a balance between execution time and ROM consumption, we decided to use the original variant in our work.

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

Input: Probability matrix P_{mat}, random number r, modulus q

Output: Sample value s

```
$index \leftarrow r\&255$
$r \leftarrow r \gg 8$
$s \leftarrow \mathrm{LUT1}[index]$
if $\mathrm{MSB}(s) = 0$ then
    if $(r\&1) = 1$ then
        return $q - s$
    else
        return $s$
    end if
end if
$d \leftarrow s\&7$
for $col$ from 8 by 1 to $MAXCOL$ do
    $d \leftarrow 2d + (r\&1)$
    $r \leftarrow r \gg 1$
    for $row$ from $MAXROW$ by $-8$ to 0 do
        if $(P_{mat}[row][col] \parallel P_{mat}[row-1][col] \parallel \ldots \parallel P_{mat}[row-7][col]) > 0$ then
            $sum = \sum_{i=row}^{row-7}(P_{mat}[i][col])$
            $d \leftarrow d - sum$
            if $d < 0$ then
                if $d = -1$ then
                    for $j$ from $row - 7$ by 1 to $row$ do
                        if $P_{mat}[j][col] = 1$ then
                            if $(r\&1) = 1$ then
                                return $q - j$
                            else
                                return $j$
                            end if
                        end if
                    end for
                else
                    for $j$ from $row - 7$ by 1 to $row$ do
                        $d \leftarrow d + P_{mat}[j][col]$
                        if $d = -1$ then
                            if $(r\&1) = 1$ then
                                return $q - j$
                            else
                                return $j$
                            end if
                        end if
                    end for
                end if
            end if
        end if
    end for
end for
return 0
```

Efficient Skipping of All-Zero Bytes. Another issue related to the probability matrix is the occurrence of all-zero bytes. In order to efficiently skip the scanning operations for such bytes, we compare the eight concatenated bits in line 16 of Algorithm 3 with 0. Since these eight bits fit into one byte, a simple byte-comparison allows us to determine whether the bits are 0 or not. In this way, we can save a number of scanning operations and, thereby, speed up the whole sampling process.

Look-Up Table in DDG Tree. We applied the Look-Up Table (LUT)-based approach proposed in [10] to our byte-wise scanning implementation (shown in line 1 to 10 of Algorithm 3). At first, we do the sampling with an 8-bit random number as index to the LUT in the first eight levels for a Gaussian distribution with $\sigma = 11.31/\sqrt{2\pi}$. If the most significant bit of the look-up result is cleared then the algorithm completed the look-up operation successfully. Otherwise, the most significant bit of the look-up result is set, which means a look-up failure has occurred and we proceed with the next level of the sampling. Similarly, a second LUT is used for level 9 to 13 in the same Gaussian distribution.

4.1 Pseudo-Random Number Generation Using AES Accelerator

Our implementation adopts the PRNG algorithm suggested in [27], which runs the AES block cipher in counter mode, i.e. it encrypts successive values of an incrementing counter. The Atmel ATxmega128A1 microcontroller features an AES crypto-accelerator that allows one to perform encryptions with reasonable computational overhead (375 clock cycles) and small memory footprint for the AES trigger program. This is a significant improvement compared to software implementations of the AES, which require (at least) 1993 cycles per block and occupy some 2 kB program memory on an ATmega128 [23]. Another attractive feature of the ATxmega's built-in AES crypto-accelerator is that it can operate independently of the processor, which allows one to "hide" the latencies due to AES encryption [31]. We exploit this feature in our Knuth-Yao sampler implementation, i.e. we trigger the next AES operation immediately after getting the result of the current one and then proceed with other tasks. Unfortunately, the AES accelerator of the ATxmega128A1 can only support 128-bit keys, which is not sufficient for long-term security. Therefore, we decided to use the software AES from the AVR Crypto Lib [9] for the long-term security level; it requires 3521 clock cycles to encrypt a block under a 256-bit key.

5 Performance Evaluation and Comparison

5.1 Experimental Platform

Our prototyping platform is an Atmel Xplained evaluation board that contains an ATxmega128A1 processor. This processor can be clocked with a maximum frequency of 32 MHz and features 128 kB flashable program memory as well as

8 kB SRAM. It is popular 8-bit processor with an AES crypto-accelerator and can be used in a wide range of applications. Our ring-LWE software is written in a mix of ANSI C and Assembly language. More precisely, while most functions of our ring-LWE encryption scheme are written in ANSI C, we implemented all modular arithmetic operations for the NTT in Assembly language to reduce the execution time. We compiled our software with Atmel Studio 6.2 and applied the speed optimization option O3. In order to obtain accurate timings, we ran each operation at least 1000 times and calculated the average cycle count.

5.2 Experimental Results

Table 1 specifies the execution time of the main components (i.e. the NTT, the Knuth-Yao sampler, key-generation, encryption, and decryption) of our ring-LWE encryption schemes for both medium-term and long-term security. As was mentioned earlier in this paper, we implemented two versions of the arithmetic operations for each security level, one that is optimized for speed and a second aiming at low memory footprint (i.e. memory efficiency). The two high-speed (HS) implementations make full use of the optimization techniques described in Sect. 3 (except Subsect. 3.6) and Sect. 4, whereby all data is kept in RAM. On the other hand, the memory-efficient (ME) implementations use the optimized coefficient storage method and memory access scheme from Subsect. 3.6 for all basic operations and store the pre-computed look-up tables in flash ROM.

Table 1. Execution time (in clock cycles) of the major components of our ring-LWE encryption scheme

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 Table 1, the NTT operation requires only 194 k clock cycles in the case of the HS-256 implementation, but the execution time increases quite sharply to 442 k cycles for the HS-512 variant. The Knuth-Yao sampler for the HS-256 implementation takes an average of about 27 k cycles, while more than 255 k clock cycles are needed for HS-512. The former increase can be explained by the fact that the number of coefficients for HS-512 is doubled compared to HS-256 and the modular reduction operation for $q = 12889$ is more costly than for $q = 7681$. On the other hand, the pseudo-random number generation at the medium-term security level can take advantage of the AES accelerator, whereas for long-term security, the required 256-bit AES operations need to be carried out in software. It is also interesting to compare the execution time of the HS and ME variants at the same security level. Taking HS-256 as an example, the key generation, encryption, and decryption need about 590 k, 672 k, and 276 k cycles, respectively, which is more than twice as fast as the memory-optimized variant ME-256. Apparently, this is primarily because of the more costly access to the coefficients, which slows down all basic operations. More specifically, the coefficient addition, subtraction, as well as multiplication up to the NTT, the Knuth-Yao sampler and each component consume more execution time to load the coefficients from memory and write them back to memory.

Table 2. RAM and ROM requirements (in bytes) of key generation, encryption, and decryption

Implementation	Key-Gen	Enc	Dec	Total
HS-256: RAM/ROM	1,585/8,884	2,609/8,812	1,585/6,026	2,609/13,604
ME-256: RAM/ROM	1,297/9,260	2,129/8,536	1,297/6,016	2,129/13,756
HS-512: RAM/ROM	3,121/12,074	6,193/13,486	3,121/8,512	6,193/18,894
ME-512: RAM/ROM	2,737/12,106	4,529/12,166	2,737/8,614	4,529/18,010

Table 2 shows the RAM and ROM requirements of key-generation, encryption, and decryption. In the case of HS-256, the full ring-LWE implementation requires some 2.6 kB RAM and 13.6 kB ROM, while the ME-256 variant needs roughly 2.1 kB RAM and 13.7 kB ROM. Due to the special coefficient-storage method described in Subsect. 3.6, the memory-optimized implementations save roughly 19% (for medium-term security) and 21% (long-term security) in RAM requirements while consuming approximately the same amount of ROM as the HS implementations.

5.3 Comparison with Related Work

Table 3 compares software implementations of lattice-based cryptosystems on several different processors. For the 8-bit AVR platform, the implementations in [5, 6, 25] and our software use the same parameter sets as mentioned before in Subsect. 2.5. Compared to the recent work in [25], our HS-256 version requires only 672 k and 276 k cycles for encryption and decryption, which is roughly 2.0 and 1.4 times faster, respectively. This progress is mainly due to a combination of algorithmic optimizations and the proposed low-level techniques to speed up the NTT multiplication and the Gaussian sampling operation.

Table 4 compares the results of our ring-LWE encryption scheme with some classical public-key encryption algorithms, in particular recent RSA and ECC implementations for the 8-bit AVR platform. The to-date fastest RSA software for an AVR microcontroller was reported in [19]; it achieves an execution time of approximately 76.6 M clock cycles for decryption at the 80-bit security level (i.e. 1024-bit modulus)[5]. For comparison, our HS-256 implementation requires

[5] To the best of our knowledge, no RSA implementation providing 128-bit security on an 8-bit processors exists. Thus, we use the 80-bit security level 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

only 276 k cycles for decryption, which is more than 278 times faster despite a much higher (i.e. 128-bit) security level. There exist quite a few ECC software implementations for 8-bit AVR processors. Recently, Düll et al. [12] managed to achieve an execution time of 13.9 M clock cycles (HS version) and 14,1 M cycles (ME version) for a variable-base scalar multiplication on Curve25519 [4]. The widely-used Elliptic Curve Integrated Encryption Scheme (ECIES) is based on scalar multiplications; the encryption involves two scalar multiplications (one with a fixed base point and the other with a random point), while a decryption operation requires a scalar multiplication with a random point. The HS version of our ring-LWE encryption scheme beats any of the ECC implementations in [12, 20, 1] by at least one order of magnitude. Our results also show that ring-LWE encryption is superior to traditional public-key schemes when high speed on resource-constrained devices is desired.

6 Conclusions

This paper presented several optimizations to efficiently implement a ring-LWE encryption scheme on the 8-bit AVR platform. In particular, we proposed three optimizations to accelerate the execution time and a special coefficient-storage technique along with a refined access strategy to reduce the RAM requirements of NTT-based polynomial multiplication. A combination of these optimizations yields a very efficient NTT computation, which is twice as fast as the previous best implementation in the literature. We also reported the results we obtained for a performance-oriented and a memory-efficient implementation at both the

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	Scheme	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

medium-term and long-term security level, respectively. In all four settings, the results we achieved set new speed records for a ring-LWE encryption scheme on an 8-bit AVR processor. Finally, a comparison of our work with RSA and ECC implementations confirms that ring-LWE encryption schemes are a good choice for high-speed public-key cryptography on resource-constrained devices.

Acknowledgments

The authors thank Frederik Vercauteren and Ruan de Clercq from KU Leuven for fruitful discussions and useful suggestions that helped to improve the work described in this paper.

Zhe Liu is supported by the Fonds National de la Recherche (FNR) Luxembourg under AFR grant No. 1359142. Hwajeong Seo is supported by Institute for Information & Communications Technology Promotion (IITP) grant funded by Korea government (MSIP) [No. 10043907, Development of high-performance IoT device and an open platform with intelligent software]. Sujoy Sinha Roy is supported by an Erasmus Mundus Ph.D. 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, FWO G.0876.14N, and 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. `http://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. `http://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. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction To Algorithms.* MIT Press and McGraw-Hill, 1990.
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, ENS Paris and Université Paris Diderot, France, 2013.
12. M. Düll, B. Haase, G. Hinterwälder, M. Hutter, C. Paar, A. H. Sánchez, and P. Schwabe. High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers. Cryptology ePrint Archive, Report 2015/343, 2015. `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)*, 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.