High-Performance Ideal Lattice-Based Cryptography on 8-bit ATxmega Microcontrollers

Extended Version

Thomas Pöppelmann, Tobias Oder, and Tim Güneysu

Horst Görtz Institute for IT-Security, Ruhr-University Bochum, Germany*
{thomas.poeppelmann,tobias.oder,tim.gueneysu}@rub.de

Abstract. Over the last years lattice-based cryptography has received much attention due to versatile average-case problems like Ring-LWE or Ring-SIS that appear to be intractable by quantum computers. But despite of promising constructions, only few results have been published on implementation issues on very constrained platforms. In this work we therefore study and compare implementations of Ring-LWE encryption and the bimodal lattice signature scheme (BLISS) on an 8-bit Atmel ATxmega128 microcontroller. Since the number theoretic transform (NTT) is one of the core components in implementations of lattice-based cryptosystems, we review the application of the NTT in previous implementations and present an improved approach that significantly lowers the runtime for polynomial multiplication. Our implementation of Ring-LWE encryption takes 27 ms for encryption and 6.7 ms for decryption. To compute a BLISS signature, our software takes 329 ms and 88 ms for verification. These results outperform implementations on similar platforms and underline the feasibility of lattice-based cryptography on constrained devices.

Keywords: Ideal lattices, NTT, RLWE, BLISS, ATxmega

1 Introduction

RSA and ECC-based schemes are the most popular asymmetric cryptosystems to date, being deployed in billions of security systems and applications. Despite of their predominance, they are known to be susceptible to attacks using quantum computers [53] on which significant resources are spent to boost their further development [50]. Additionally, RSA and ECC have been shown to be quite inefficient on very small and constrained devices like 8-bit AVR microcontrollers [27, 32]. A possible alternative are asymmetric cryptosystems based on hard problems in ideal lattices. The special algebraic structure of ideal lattices [40] defined in $\mathcal{R} = \mathbb{Z}_q[\mathbf{x}]/\langle x^n + 1\rangle$ allows a significant reduction of key and ciphertext sizes and enables efficient arithmetic using the number theoretic transform (NTT)[1] [5, 44, 55]. To realize lattice-based public key encryption several proposals exists (see [10] for a comparison) like classical NTRU [31] (defined in $\mathbb{Z}_q[\mathbf{x}]/\langle x^n - 1\rangle$), provably secure NTRU [54] (defined in $\mathbb{Z}_q[\mathbf{x}]/\langle x^n + 1\rangle$), or a scheme based on the ring learning with errors (RLWE) problem [35, 40] (from now on referred to as RLWEenc). From an implementation perspective, the RLWEenc scheme is currently one of the best-studied lattice-based public key encryption schemes (see [7, 11, 13, 38, 48, 49, 51]) and is similar to a recently proposed key exchange protocol [8, 46]. Concerning signature schemes, several proposals exist like GLP [25] (derived from [39]), BG [1], PASSSign [30], a modified NTRU signature scheme [42], or a signature scheme derived from a recently proposed IBE scheme [20]. However, so far the Bimodal Lattice Signature Scheme (BLISS) [18] seems superior in terms of signature size, performance, and security. Despite their popularity, implementation efforts so far mainly led to very efficient hardware designs for RLWEenc [11, 49, 51] and BLISS [47] and fast software on 32-bit microcontrollers [45] and 64-bit microprocessors [18, 26] but only few works cover constrained 8-bit architectures [6, 7, 38]. Additionally, current works usually rely on the straightforward Cooley-Tukey radix-2 decimation-in-time

* This work was partially funded by the European Union H2020 SAFEcrypto project (grant no. 644729), European Union H2020 PQCRYPTO project (grant no. 645622), German Research Foundation (DFG), and DFG Research Training Group GRK 1817/1.

[1] The NTT can be described as Fast Fourier Transform over $\mathbb{Z}_q$.

This article is based on an earlier article: High-Performance Ideal Lattice-Based Cryptography on 8-bit ATxmega Microcontrollers, Thomas Pöppelmann, Tobias Oder, Tim Güneysu, Latincrypt 2015

algorithm (e.g., [7, 13, 47, 51]) to implement the NTT and thus to realize polynomial multiplication $\mathbf{c} = \mathbf{a} \cdot \mathbf{b}$ for $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathcal{R}$ as $\mathbf{c} = \mathsf{INTT}(\mathsf{NTT}(\mathbf{a}) \circ \mathsf{NTT}(\mathbf{b}))$. However, by taking a closer look at works on the implementation [12, 16] of the highly related fast Fourier transform (FFT) it becomes evident that the sole focus on Cooley-Tukey radix-2 decimation-in-time algorithms prevents further optimizations of the NTT, especially given the constraints of an 8-bit architecture.

Contribution. The contribution of this work is twofold. We first review different approaches and varieties of NTT algorithms and then adapt and optimize these algorithms for the polynomial multiplication use-case prevalent in ideal lattice-based cryptography. Improvements compared to previous work are mainly achieved by merging certain operations into the NTT itself (multiplication by n^{-1}, powers of ψ and ψ^{-1}) and by removing the expensive bit-reversal step. In the second part we provide an efficient implementation of these NTT algorithms on the 8-bit AVR/ATxmega architecture. By using this optimized NTT we achieve high performance for RLWEenc and BLISS. Our work shows that lattice-based cryptography can be used to realize the two most basic asymmetric primitives (public key encryption and signatures) on very constrained devices and with high performance. To the best of our knowledge, we provide the smallest implementation of BLISS on the AVR architecture in terms of flash consumption and also fastest implementation in terms of runtime since one signature computation requires only 329 ms and verification requires 88 ms. By performing encryption in 27 ms and decryption in 6.7 ms, our implementation of RLWEenc also outperforms previous work on AVRs. To allow third-party evaluation of our results, source code and documentation is available on our website[2].

Outline. In Section 2 we review the number theoretic transform (NTT), RLWEenc, and BLISS. We then show NTT algorithms that are better suited for polynomial multiplication in Section 3. Our AVR ATxmega128 implementation is described in detail in Section 4 and we discuss our results in Section 5.

2 Background

In this section we introduce the NTT and explicitly describe its application in the algorithms of the RLWEenc public key encryption scheme and the BLISS signature scheme.

2.1 The Number Theoretic Transform and Negacyclic Convolutions

The number theoretic transform (NTT) [5, 44, 55] is similar to the discrete Fourier transform (DFT) but all complex roots of unity are exchanged for integer roots of unity and arithmetic is also carried out modulo an integer q in the field $GF(q)$[3]. Main applications of the NTT, besides ideal lattice-based cryptography, are integer multiplication (e.g., Schönhage and Strassen [52]) and signal processing [5]. The forward transformation $\tilde{\mathbf{a}} = \mathrm{NTT}(\mathbf{a})$ of a length n sequence $(\mathbf{a}[0], \ldots, \mathbf{a}[n-1])$ to $(\tilde{\mathbf{a}}[0], \ldots, \tilde{\mathbf{a}}[n-1])$ with elements in $\mathbb{Z}_q$ is defined as $\tilde{\mathbf{a}}[i] = \sum_{j=0}^{n-1} \mathbf{a}[j]\omega^{ij} \bmod q,\ i = 0, 1, \ldots, n-1$ where ω is an n-th primitive root of unity. The inverse transform $\mathbf{a} = \mathrm{INTT}(\tilde{\mathbf{a}})$ is defined as $\mathbf{a}[i] = n^{-1} \sum_{j=0}^{n-1} \tilde{\mathbf{a}}[j]\omega^{-ij} \bmod q,\ i = 0, 1, \ldots, n-1$ where ω is exchanged by ω^{-1} and the final result scaled by n^{-1}. For an n-th primitive root of unity ω_n it holds that $\omega_n^n = 1 \bmod q$, $\omega_n^{n/2} = -1 \bmod q$, $\omega_{\frac{n}{2}} = \omega_n^2$, and $\omega_n^i \neq 1 \bmod q$ for any $i = 1, \ldots, n-1$.

The main operation in ideal lattice-based cryptography is polynomial multiplication[4]. Schemes are usually defined in $\mathcal{R} = \mathbb{Z}_q[\mathbf{x}]/\langle x^n + 1 \rangle$ with modulus $x^n + 1$ where n is a power of two and one can make use of the *negacyclic convolution* property of the NTT that allows carrying out a

[2] See http://www.sha.rub.de/research/projects/lattice/

[3] Actually, this is overly restrictive and the NTT is also defined for certain composite numbers (n has to divide $p-1$ for every prime factor p of q). However, for the given target parameter sets common in lattice-based cryptography we can restrict ourselves to prime moduli and refer to [44] for further information on composite moduli NTTs.

[4] Similar to exponentiation being the main operation of RSA or point multiplication being the main operation of ECC.

polynomial multiplication in $\mathbb{Z}_q[\mathbf{x}]/\langle x^n+1\rangle$ using length-n transforms and no zero padding. When $\mathbf{a}=(\mathbf{a}[0],\ldots\mathbf{a}[n-1])$ and $\mathbf{b}=(\mathbf{b}[0],\ldots\mathbf{b}[n-1])$ are polynomials of length n with elements in $\mathbb{Z}_q$, ω be a primitive n-th root of unity in $\mathbb{Z}_q$ and $\psi^2=\omega$, then we define $\mathbf{d}=(\mathbf{d}[0],\ldots\mathbf{d}[n-1])$ as the negative wrapped convolution of $\mathbf{a}$ and $\mathbf{b}$ so that $\mathbf{d}=\mathbf{a}\cdot\mathbf{b} \mod x^n+1$. We then define $\bar{\mathbf{a}}=\mathsf{PowMul}_\psi(\mathbf{a})=(\mathbf{a}[0],\psi\mathbf{a}[1],\ldots,\psi^{n-1}\mathbf{a}[n-1])$ as well as the inverse multiplication by powers of ψ^{-1} denoted as $\mathbf{a}=\mathsf{PowMul}_{\psi^{-1}}(\bar{\mathbf{a}})$. Then it holds that $\mathbf{d}=\mathsf{PowMul}_{\psi^{-1}}(\mathsf{INTT}(\mathsf{NTT}(\mathsf{PowMul}_\psi(\mathbf{a})\circ\mathsf{NTT}(\mathsf{PowMul}_\psi(\mathbf{b})))))$ [15, 16, 55], where $\circ$ denotes point-wise multiplication. For simplicity, we do not always explicitly apply PowMul_ψ or $\mathsf{PowMul}_{\psi^{-1}}$ when it is clear from the context that a negacyclic convolution is computed.

2.2 The RLWEenc Cryptosystem

The semantically secure public key encryption scheme RLWEenc was proposed in [35, 40, 41] and is also used as a building block in the identity-based encryption scheme (IBE) by Ducas, Lyubashevsky, and Prest [20]. We provide the key generation procedure $\mathsf{RLWEenc}_{\text{GEN}}$ in Algorithm 1, the encryption procedure $\mathsf{RLWEenc}_{\text{ENC}}$ in Algorithm 2, and the decryption procedure $\mathsf{RLWEenc}_{\text{DEC}}$ in Algorithm 3. All algorithms explicitly use calls to the NTT and function names used later on during the evaluation of our implementation (see Section 5). The exact placement of NTT transformations is slightly changed compared to [51], which saved one transformation compared to [48], as $\mathbf{c}_2$ is not transmitted in NTT form and thus removal of least significant bits is still possible (see [20, 48]).

Algorithm 1 RLWEenc Key Generation

Precondition: Access to global constant $\tilde{\mathbf{a}}=\mathsf{NTT}(\mathbf{a})$
1: **function** $\mathsf{RLWEenc}_{\text{GEN}}()$
2: $\quad\tilde{\mathbf{r}}_1\leftarrow\mathsf{NTT}(\mathsf{SampleGauss}_\sigma())$
3: $\quad\tilde{\mathbf{r}}_2\leftarrow\mathsf{NTT}(\mathsf{SampleGauss}_\sigma())$
4: $\quad\tilde{\mathbf{p}}=\tilde{\mathbf{r}}_1-\tilde{\mathbf{a}}\circ\tilde{\mathbf{r}}_2$
5: $\quad$**return** $(pk,sk)=(\tilde{\mathbf{p}},\tilde{\mathbf{r}}_2)$
6: **end function**

Algorithm 2 RLWEenc Encryption

Precondition: Access to global constant $\tilde{\mathbf{a}}=\mathsf{NTT}(\mathbf{a})$
1: **function** $\mathsf{RLWEenc}_{\text{ENC}}(\tilde{\mathbf{a}},\tilde{\mathbf{p}},\mu\in\{0,1\}^n)$
2: $\quad\tilde{\mathbf{e}}_1=\mathsf{NTT}(\mathsf{SampleGauss}_\sigma())$
3: $\quad\tilde{\mathbf{e}}_2=\mathsf{NTT}(\mathsf{SampleGauss}_\sigma())$
4: $\quad\mathbf{c}_1=\tilde{\mathbf{a}}\circ\tilde{\mathbf{e}}_1+\tilde{\mathbf{e}}_2$
5: $\quad\tilde{\mathbf{h}}_2=\tilde{\mathbf{p}}\circ\tilde{\mathbf{e}}_1$
6: $\quad\mathbf{e}_3\leftarrow\mathsf{SampleGauss}_\sigma()$
7: $\quad\mathbf{c}_2=\mathsf{INTT}(\tilde{\mathbf{h}}_2)+\mathbf{e}_3+\mathsf{Encode}(m)$
8: $\quad$**return** $(\mathbf{c}_1,\mathbf{c}_2)$
9: **end function**

Algorithm 3 RLWEenc Decryption

1: **function** $\mathsf{RLWEenc}_{\text{DEC}}(\mathbf{c}=[\mathbf{c}_1,\mathbf{c}_2],\tilde{\mathbf{r}}_2)$
2: $\quad$**return** $\mathsf{Decode}(\mathsf{INTT}(\mathbf{c}_1\circ\tilde{\mathbf{r}}_2)+\mathbf{c}_2)$.
3: **end function**

The main idea of the scheme is that during encryption the n-bit encoded message $\bar{\mathbf{m}}=\mathsf{Encode}(m)$ is added to $\mathbf{p}\mathbf{e}_1+\mathbf{e}_3$ (in NTT notation $\mathsf{INTT}(\tilde{\mathbf{h}_2})+\mathbf{e}_3$) which is uniformly random and thus hides the message. Decryption is only possible with knowledge of $\mathbf{r}_2$ since otherwise the large term $\mathbf{a}\mathbf{e}_1\mathbf{r}_2$ cannot be eliminated when computing $\mathbf{c}_1\mathbf{r}_2+\mathbf{c}_2$. The encoding of the message of length n is necessary as the noise term $\mathbf{e}_1\mathbf{r}_1+\mathbf{e}_2\mathbf{r}_2+\mathbf{e}_3$ is still present after calculating $\mathbf{c}_1\mathbf{r}_2+\mathbf{c}_2$ and would prohibit the retrieval of the binary message after decryption. With the simple threshold encoding $\texttt{encode}(m)=\frac{q-1}{2}m$ the value $\frac{q-1}{2}$ is assigned only to each binary one of the string m. The corresponding decoding function needs to test whether a received coefficient $z\in[0..q-1]$ is in the interval $\frac{q-1}{4}\leq z<3\frac{q-1}{4}$ which is interpreted as one and zero otherwise. As a consequence, the maximum error added to each coefficient must not be

larger than $\lfloor \frac{q}{4} \rfloor$ in order to decrypt correctly. The probability for decryption errors is mainly determined by the tailcut τ and the standard deviation σ of the polynomials $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3 \leftarrow D_{\mathbb{Z}^n,\sigma}$, which follow a small discrete Gaussian distribution (sampled by $\mathsf{SampleGauss}_\sigma$). In this context, decreasing s reduces the error probability but also negatively affects the security of the scheme [24,35]. Increasing q on the other hand increases the key size, ciphertext expansion, and reduces performance (on certain devices). To support the NTT, Göttert et al. [24] proposed parameter sets (n, q, s) where $\sigma = s/\sqrt{2\pi}$ denoted as RLWEenc-Ia $(256, 7681, 11.31)$ and RLWEenc-IIa $(512, 12289, 12.18)$. Lindner and Peikert [35] originally proposed the parameter sets RLWEenc-Ib $(192, 4093, 8.87)$, RLWEenc-IIb $(256, 4093, 8.35)$ and RLWEenc-IIIb $(320, 4093, 8.00)$. The security levels of RLWEenc-Ia and RLWEenc-IIb are roughly comparable and RLWEenc-IIb provides 105.5 bits of pre-quantum security[5], according to a refined security analysis by Liu and Nguyen [36] for standard LWE and the original parameter sets. The RLWEenc-IIa parameter set uses a larger dimension n and should thus achieve even higher security than the 156.9 bits obtained by Liu and Nguyen for RLWEenc-IIIb. For the IBE scheme in [20] the parameters $n = 512$, $q \approx 2^{23}$ and a trinary error/noise distribution are used.

2.3 The BLISS Cryptosystem

In this work we only consider the efficient ring-based instantiation of BLISS [18]. We recall the key generation procedure $\text{BLISS}_{\text{GEN}}$ in Algorithm 4, the signing procedure $\text{BLISS}_{\text{SIGN}}$ in Algorithm 5, and the verification procedure $\text{BLISS}_{\text{VER}}$ in Algorithm 6. Key generation requires uniform sampling of sparse and small polynomials $\mathbf{f}, \mathbf{g}$, rejection sampling $(N_\kappa(\mathbf{S}))$, and computation of an inverse. To sign a message, two masking polynomials $\mathbf{y}_1, \mathbf{y}_2 \leftarrow D_{\mathbb{Z}^n,\sigma}$ are sampled from a discrete Gaussian distribution using the $\mathsf{SampleGauss}_\sigma$ function. The computation of $\mathbf{a}\mathbf{y}_1$ is performed using the NTT and the compressed $\mathbf{u}$ is then hashed together with the message μ by Hash. The binary string c' is used by GenerateC to generate a sparse polynomial $\mathbf{c}$. Polynomials $\mathbf{y}_1, \mathbf{y}_2$ then hide the secret key which is multiplied with the sparse polynomials using the SparseMul function. This function exploits that only κ coefficients in $\mathbf{c}$ are set and only $d_1 + d_2$ coefficients in $\mathbf{s}_1$ and $\mathbf{s}_2$. After a rejection sampling and compression step the signature $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$ is returned. The verification procedure $\text{BLISS}_{\text{VER}}$ in Algorithm 6 just checks norms of signature components and compares the hash output with $\mathbf{c}$ in the signature.

In this work we focus on the 128-bit pre-quantum secure BLISS-I parameter set which uses $n = 512$ and $q = 12289$ (same base parameters as RLWEenc-IIa). The density of the secret key is $\delta_1 = 0.3$ and $\delta_2 = 0$, the standard deviation of the coefficients of $\mathbf{y}_1$ and $\mathbf{y}_2$ is $\sigma = 215.73$ and the repetition rate is 1.6. The number of dropped bits in $\mathbf{z}_2$ is $d = 10$, $\kappa = 23$, and $p = \lfloor 2q/2^d \rfloor$. The final size of the signature is 5,600 bits with Huffman encoding and approx. 7,680 bits without Huffman encoding.

3 Faster NTTs for Lattice-Based Cryptography

In this section we examine fast algorithms for the computation of the number theoretic transform (NTT) and show techniques to speed up polynomial multiplication for lattice-based cryptography[6]. The most straightforward implementation of the NTT is a Cooley-Tukey radix-2 decimation-in-time (DIT) approach [14] that requires a bit-reversal step as the algorithm takes bit-reversed input and produces naturally ordered output (from now on referred to as $\text{NTT}^{CT}_{bo \rightarrow no}$). To compute the NTT as defined in Section 2.1 the $\text{NTT}^{CT}_{bo \rightarrow no}$ algorithm applies the Cooley-Tukey (CT) butterfly, which

[5] Up to our knowledge, all security evaluations of RLWEenc (and also BLISS) only consider best known attacks executed on a classical computer. The security levels are thus denoted as *pre-quantum*. A security assessment that considers quantum computers is certainly necessary but is not in the scope of this paper.

[6] Most of the techniques discussed in this section have already been proposed in the context of the fast Fourier transform (FFT). However, they have not yet been considered to speed up ideal lattice-based cryptography (at least not in works like [7, 13, 47, 51]). Moreover, some optimizations and techniques are mutually exclusive and a careful selection and balancing has to be made.

Algorithm 4 BLISS Key Generation

function $\text{BLISS}_{\text{GEN}}()$
  Choose $\mathbf{f}, \mathbf{g}$ as uniform polynomials with exactly $d_1 = \lceil \delta_1 n \rceil$ entries in $\{\pm 1\}$ and $d_2 = \lceil \delta_2 n \rceil$ entries in $\{\pm 2\}$
  $\mathbf{S} = (\mathbf{s}_1, \mathbf{s}_2)^t \leftarrow (\mathbf{f}, 2\mathbf{g}+1)^t$
  if $N_\kappa(\mathbf{S}) \geq C^2 \cdot 5 \cdot (\lceil \delta_1 n \rceil + 4\lceil \delta_2 n \rceil) \cdot \kappa$ **then restart**
  $\mathbf{a}_q = (2\mathbf{g}+1)/\mathbf{f} \bmod q$ (**restart** if $\mathbf{f}$ is not invertible)
  Return$(pk = \mathbf{A}, sk = \mathbf{S})$ where $\mathbf{A} = (\tilde{\mathbf{a}}_1 = \mathsf{NTT}(\mathbf{a}_q), q-2) \bmod 2q$
end function

Algorithm 5 BLISS Signing

function $\text{BLISS}_{\text{SIGN}}(\mu \in \{0,1\}^*, pk{=}\mathbf{A}, sk{=}\mathbf{S})$
  $\mathbf{y}_1 \leftarrow \mathsf{SampleGauss}_\sigma()$
  $\mathbf{y}_2 \leftarrow \mathsf{SampleGauss}_\sigma()$
  $\mathbf{u} = 2\zeta \cdot \mathsf{INTT}(\tilde{\mathbf{a}}_1 \circ \mathsf{NTT}(\mathbf{y}_1)) + \mathbf{y}_2 \bmod 2q$
  $c' \leftarrow \mathsf{Hash}(\lfloor \mathbf{u} \rceil_d \bmod p, \mu)$
  $\mathbf{c} \leftarrow \mathsf{GenerateC}(c')$
  Choose a random bit b
  $\mathbf{z}_1 \leftarrow \mathbf{y}_1 + (-1)^b \mathsf{SparseMul}(\mathbf{s}_1, \mathbf{c})$
  $\mathbf{z}_2 \leftarrow \mathbf{y}_2 + (-1)^b \mathsf{SparseMul}(\mathbf{s}_2, \mathbf{c})$
  Continue with probability
    $1 \Big/ \left(M \exp\left(-\frac{\|\mathbf{Sc}\|^2}{2\sigma^2} \right) \cosh\left(\frac{\langle \mathbf{z}, \mathbf{Sc} \rangle}{\sigma^2} \right) \right)$
    otherwise **restart**
  $\mathbf{z}_2^\dagger \leftarrow (\lfloor \mathbf{u} \rceil_d - \lfloor \mathbf{u} - \mathbf{z}_2 \rceil_d) \bmod p$
  Return $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$
end function

Algorithm 6 BLISS Verification

function $\text{BLISS}_{\text{VER}}(\mu \in \{0,1\}^*, pk{=}\mathbf{A}, sk{=}\mathbf{S})$
  if $\|(\mathbf{z}_1 | 2^d \cdot \mathbf{z}_2^\dagger)\|_2 > B_2$ **then** Reject
  if $\|(\mathbf{z}_1 | 2^d \cdot \mathbf{z}_2^\dagger)\|_\infty > B_\infty$ **then** Reject
  $\mathbf{r} \leftarrow \mathsf{INTT}(\tilde{\mathbf{a}}_1 \circ \mathsf{NTT}(\mathbf{z}_1))$
  $c' = \mathsf{Hash}\left(\lfloor 2\zeta \cdot \mathbf{r} + \zeta \cdot q \cdot \mathbf{c} \rceil_d + \mathbf{z}_2^\dagger \bmod p, \mu \right)$
  Accept iff $\mathbf{c} = \mathsf{GenerateC}(c')$
end function

computes $a' \leftarrow a + \omega b$ and $b' \leftarrow a - \omega b$ for some values of $\omega, a, b \in \mathbb{Z}_q$, overall $\frac{n \log_2(n)}{2}$ times. The biggest disadvantage of relying solely on the $\text{NTT}^{CT}_{bo \rightarrow no}$ algorithm is the need for bit-reversal, multiplication by constants, and that it is impossible to merge the final multiplication by powers of ψ^{-1} into the twiddle factors of the inverse NTT (see [51]). With the assumption that twiddle factors (powers of ω) are stored in a table and thus not computed on-the-fly it is possible to further simplify the computation and to remove bit-reversal and to merge certain steps. This assumption makes sense on constrained devices like the ATxmega, which have a rather large read-only flash.

3.1 Merging the Inverse NTT and Multiplication by Powers of ψ^{-1}

In [51] Roy et al. use the standard $\text{NTT}^{CT}_{bo \rightarrow no}$ algorithm for a hardware implementation and show how to merge the multiplication by powers of ψ (see Section 2.1) into the twiddle factors of the forward transformation. However, this approach does not work for the inverse transformation due to the way the computations are performed in the CT butterfly as the multiplication is carried out before the addition. In this section we show that it is possible to merge the multiplication by powers of ψ^{-1} during the inverse transformation using a fast decimation-in-frequency (DIF) algorithm [23]. The DIF NTT algorithm splits the computation into a sub-problem on the even outputs and a sub-problem on the odd outputs of the NTT and has the same complexity as the $\text{NTT}^{CT}_{bo \rightarrow no}$ algorithm. It requires usage of the so-called Gentlemen-Sande (GS) butterfly which computes $a' \leftarrow a + b$ and $b' \leftarrow (a - b)\omega$ for some values of $\omega, a, b \in \mathbb{Z}_q$. Following [12, Section 3.2], where ω_n is an n-th primitive root of unity and by ignoring the multiplication by the scalar n^{-1}, the inverse NTT and application of PowMul_ψ

can be defined as

$$\mathbf{a}[r] = \psi^{-r} \sum_{\ell=0}^{n-1} \mathbf{A}[\ell]\omega_n^{-r\ell} = \psi^{-r} \left(\sum_{\ell=0}^{\frac{n}{2}-1} \mathbf{A}[\ell]\omega_n^{-r\ell} + \sum_{\ell=0}^{\frac{n}{2}-1} \mathbf{A}[\ell+\frac{n}{2}]\omega_n^{-r(\ell+\frac{n}{2})} \right) \tag{1}$$

$$= \psi^{-r} \sum_{\ell=0}^{\frac{n}{2}-1} \left(\mathbf{A}[\ell] + \mathbf{A}[\ell+\frac{n}{2}]\omega_n^{-r\frac{n}{2}} \right) \omega_n^{-r\ell}, r = 0, 1, \ldots, n-1. \tag{2}$$

When r is even this results in

$$\mathbf{a}[2k] = \sum_{\ell=0}^{\frac{n}{2}-1} \left(\mathbf{A}[\ell] + \mathbf{A}[\ell+\frac{n}{2}] \right) \omega_{\frac{n}{2}}^{-k\ell} (\psi^2)^{-k} \tag{3}$$

and for odd r in

$$\mathbf{a}[2k+1] = \sum_{\ell=0}^{\frac{n}{2}-1} \left(\mathbf{A}[\ell] - \mathbf{A}[\ell+\frac{n}{2}]\omega_n^{-\ell} \right) \omega_{\frac{n}{2}}^{-k\ell} \psi^{-(2k+1)} \tag{4}$$

$$= \sum_{\ell=0}^{\frac{n}{2}-1} \left(\left(\mathbf{A}[\ell] - \mathbf{A}[\ell+\frac{n}{2}] \right) \psi^{-1}\omega_n^{-\ell} \right) \omega_{\frac{n}{2}}^{-k\ell} (\psi^2)^{-k}, k = 0, 1, \ldots, \frac{n}{2}-1. \tag{5}$$

The two new half-size sub-problems where ψ is exchanged by ψ^2 can now be again solved using the recursion. As a consequence, when using an in-place radix-2 DIF algorithm it is necessary to multiply all twiddle factors in the first stage by ψ^{-1}, all twiddle factors in the second stage by ψ^{-2} and in general by ψ^{-2^s} for stage $s \in \{0, 1, \ldots, \log_2(n)-1\}$ to merge the multiplication by powers of ψ^{-1} into the inverse NTT (see Figure 1 for an illustration). In case the PowMul_{ψ} or $\mathsf{PowMul}_{\psi^{-1}}$ operation is merged into the NTT computation we denote this by an additional superscript ψ or ψ^{-1}, e.g., as $\mathrm{NTT}_{bo\rightarrow no}^{CT,\psi}$.

3.2 Removing Bit-Reversal

For memory efficient and in-place computation a reordering or so-called bit-reversal step is usually applied before or after an NTT/FFT transformation due to the required reversed input ordering of the $\mathrm{NTT}_{bo\rightarrow no}^{CT}$ algorithm used in works like [7, 13, 47, 51]. However, by manipulation of the standard iterative algorithms and independently of the used butterfly (CT or GS) it is possible to derive natural order to bit-reversed order ($no \rightarrow bo$) as well as bit-reversed to natural order ($bo \rightarrow no$) forward and inverse algorithms. The derivation of FFT algorithms with a desired ordering of inputs and outputs is described in [12] and we followed this description to derive the NTT algorithms $\mathrm{NTT}_{bo\rightarrow no}^{CT}$, $\mathrm{NTT}_{no\rightarrow bo}^{CT}$, $\mathrm{NTT}_{no\rightarrow bo}^{GS}$, and $\mathrm{NTT}_{bo\rightarrow no}^{GS}$, as well es their respective inverse counterparts. It is also possible to construct self-sorting NTTs ($no \rightarrow no$) but in this case the structure becomes irregular and temporary memory is required (see [12]).

3.3 Tuning for Lattice-Based Cryptography

The optimizations discussed in this section so far can be used to generically optimize polynomial multiplication in $\mathbb{Z}_q[\mathbf{x}]/\langle x^n+1\rangle$. However, for lattice-based cryptography there are special conditions that hold for most practical algorithms; in the NTT-enabled algorithms of RLWEenc and BLISS every point-wise multiplication (denoted by $\circ$) is performed with a constant and a variable, usually a randomly sampled polynomial. Thus the most common operation in lattice-based cryptography is *not* simple polynomial multiplication but multiplication of a (usually random) polynomial by a

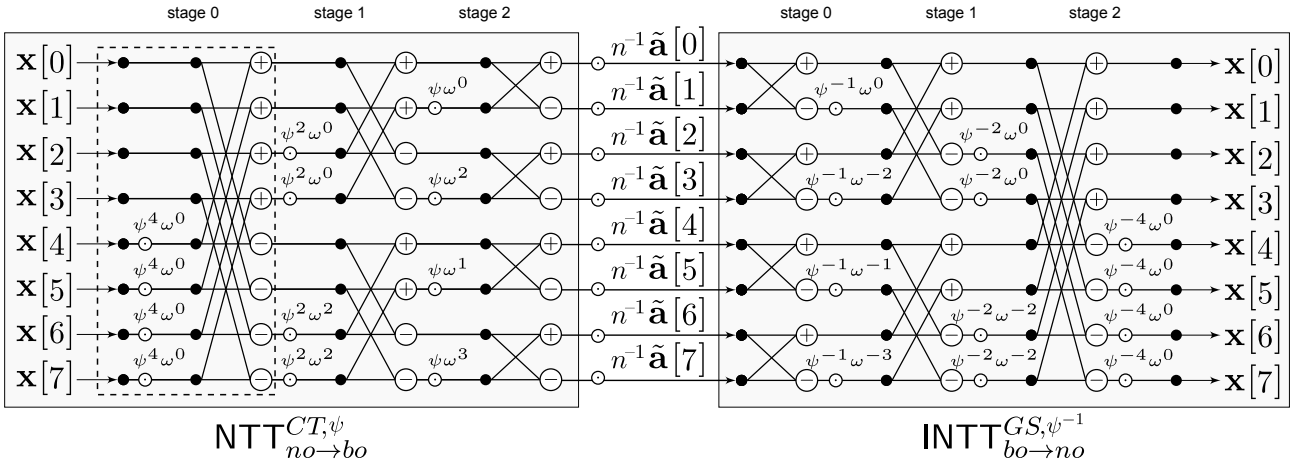

Fig. 1. Signal flow graph for multiplication of a polynomial $\mathbf{x}$ by a pre-transformed polynomial $\tilde{\mathbf{a}} = \mathrm{NTT}^{CT,\psi}_{no\to bo}(\mathbf{a})$, using the $\mathrm{NTT}^{CT,\psi}_{no\to bo}$ and $\mathrm{INTT}^{GS,\psi^{-1}}_{bo\to no}$ algorithms.

constant polynomial (i.e., global constant, or public key). Thus the scaling factor n^{-1} can be multiplied into the pre-computed and pre-transformed constant $\tilde{\mathbf{a}} = n^{-1}\mathrm{NTT}^{CT}_{no\to bo}(\mathbf{a})$. Taking into account that we also want to remove the need for bit-reversal and want to merge the multiplication by powers of ψ into the forward and inverse transformation (as discussed in Section 3.1) we propose to use an $\mathrm{NTT}^{CT,\psi}_{no\to bo}$ for the forward transformation and an $\mathrm{INTT}^{GS,\psi^{-1}}_{bo\to no}$ for the inverse transformation. In this case a polynomial multiplication $\mathbf{c} = \mathbf{a} \cdot \mathbf{e}$ can be implemented without bit-reversal as $\mathbf{c} = \mathrm{INTT}^{GS,\psi^{-1}}_{bo\to no}\left(\mathrm{NTT}^{CT,\psi}_{no\to bo}(\mathbf{a}) \circ \mathrm{NTT}^{CT,\psi}_{no\to bo}(\mathbf{e})\right)$. In Figure 2 we compare the necessary blocks for the straightforward approach and our proposal and provide an example flow diagram for $n = 8$ in Figure 1. For more details, pseudo-code of $\mathrm{NTT}^{CT,\psi}_{no\to bo}$ is provided in Algorithm 7 and pseudo-code of $\mathrm{INTT}^{GS,\psi^{-1}}_{bo\to no}$ is given in Algorithm 8.

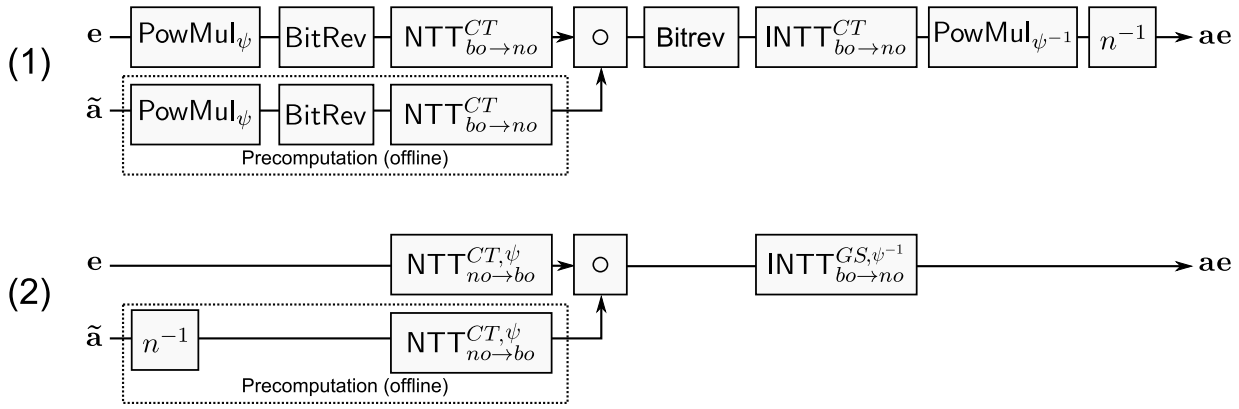

Fig. 2. Comparison of naive implementation of polynomial multiplication by a pre-computed constant using only $\mathrm{NTT}^{CT}_{bo\to no}$ and $\mathrm{INTT}^{CT}_{bo\to no}$ (1) and our proposed approach (2) using $\mathrm{NTT}^{CT,\psi}_{no\to bo}$ and $\mathrm{INTT}^{GS,\psi^{-1}}_{bo\to no}$.

4 Implementation of Lattice-Based Cryptography on ATxmega128

In this section we provide details on our implementation of the NTT as well as RLWEenc, and BLISS on the ATxmega128 (see Appendix A.2 for more information). Note that our implementation does

not take protection against timing side channels into account which is required for most practical and interactive applications. In this work we solely focused on performance and to a lesser extent on a small memory and code footprint.

4.1 Implementation of the NTT

For the use in RLWEenc and BLISS we focus on the optimization of the $\text{NTT}^{CT,\psi}_{no\to bo}$ and $\text{INTT}^{GS,\psi^{-1}}_{bo\to no}$ transformations. We implemented both algorithms in C and optimized modular multiplication using assembly language.

Modular Multiplication. To implement the NTT according to Section 3 a DIT $\text{NTT}^{CT,\psi}_{no\to bo}$ and a DIF $\text{INTT}^{GS,\psi^{-1}}_{bo\to no}$ transformation are required and the most expensive computation in both algorithms is integer multiplication and reduction modulo q ($\frac{n\log_2(n)}{2}$ times per NTT). In [7] Boorghany, Sarmadi, and Jalili report that most of the runtime of their FFT/NTT is spent on the computation of modulo operations. They review modular reduction algorithms for suitability and propose an approximate variant of Barrett reduction [3] that leads to an FFT/NTT that is 1.26 times faster than one using the compiler generated modulo reduction. However, a straightforward implementation using the C-operator % with a constant modulus is quite expensive and requires around 600 cycles in our own experiments due to the generic libc modular reduction (`call __udivmodsi4`). As a consequence, the software of the authors of [7] still consumes approx. $\frac{754668}{\frac{256}{2}\log_2(256)} = 736$ cycles for one FFT/NTT butterfly.

Another approach is a subtract-and-shift algorithm which loads the shifted modulus as constant and the input into a temporary register. It then continues to compare the value in the temporary register to this modulus, subtracts if the input is larger or equal to the modulus and then shifts the modules by one to the right. This continues until the shifted modulus is equal to the original modulus. The pseudo-code for this operation is shown in Figure 3. The biggest improvement in assembly stems from the ability to limit the operations on the active registers. As an example, when the input is 28 bits wide the first comparisons and shifts have to be performed on four registers (`REDUCE28` to `REDUCE25`), but after four iterations all operations (comparison, subtraction, shift) have to be performed only on three registers (from `REDUCE24`), or two registers (from `REDUCE16`). This approach guarantees that one modular multiplication for $q = 7681$ takes *at most* 216 cycles. While we do not take into account constant time operation for side-channel protection, the implementation can be made trivially constant time and then always runs with the worst-case runtime.

An even more efficient implementation has been proposed by Liu et al. [38] who introduce an assembly optimized shifting-addition-multiplication-subtraction-subtraction algorithm (SMAS2). They achieve, for $q = 7681$ case, a modular multiplication in 53 clock cycles which beats the subtract-and-shift approach mentioned above. For all reductions modulo $q = 7681$ and $q = 12289$ we thus rely on the SMAS2 method[7].

Extraction of Stages. As additional optimization we use specific routines for the first and the last stage of each NTT. A common optimization is to recognize that $\omega^0 = 1$ in the first stage of the $\text{NTT}^{CT}_{no\to bo}$ so that only additions are required. As we merge the multiplication by powers of ψ into the NTT this is not the case anymore (see Figure 1). However, it is still beneficial to write a specific loop that performs the first stage of the $\text{NTT}^{CT,\psi}_{no\to bo}$ and the last stage of the $\text{INTT}^{GS,\psi^{-1}}_{bo\to no}$ transformation to achieve less loop overhead (simpler address generation) and less loads and stores.

[7] In an earlier version of this work we relied on the subtract-and-shift method which is clearly inferior compared to SMAS2 by Liu et al. [38].

```
mm12289(a, b)
uint8_t v[4];
c = a*b;
v = 12289 << 14

REDUCE28:
if (c >= v)
      c = c -v
v = v << 1;
```

```
REDUCE27:
if (c >= v)
      c = c -v
v = v << 1;
REDUCE26:
if (c >= v)
      c = c -v
v = v << 1;
```

```
REDUCE25:
if (c >= v)
      c = c -v
v = v << 1;
REDUCE24:
if (c >= v)
      c = c -v
v = v << 1;

...
```

Fig. 3. Efficient prime specific modular multiplication for $q = 7681$ and $q = 12289$.

Usage of Look-up Tables for Narrow Input Distributions. As discussed in Section 3.3 it is common in lattice-based cryptography to apply forward transformations mostly to values sampled from a narrow Gaussian error/noise distribution (other polynomials are usually constants and pre-computed). In this case only a limited number of possible inputs to the butterfly of the first stage of the $\mathrm{NTT}^{CT,\psi}_{no\rightarrow bo}$ transformation exist and it is possible to pre-compute look-up tables to speed-up the modular multiplication. The range of possible input coefficients to the first stage butterfly is rather small, since they are Gaussian distributed and bounded by $[-\tau\sigma, \tau\sigma]$ for standard deviation σ and tail-cut factor τ. Additionally, we only store the result of multiplications of two positive factors. That means that for negative inputs, we invert before the look-up and again after the look-up. The same approach would also work for the binary error distribution used for the IBE scheme in [20] and it would be possible to cover even two or more stages due to the very limited input range.

4.2 Implementation of LP-RLWE

Our implementation of $\mathsf{RLWEenc}_{\mathrm{ENC}}$ and $\mathsf{RLWEenc}_{\mathrm{DEC}}$ of the $\mathsf{RLWEenc}$ scheme as described in Section 2.2 mainly consists of forward and inverse NTT transformations (NTT and INTT), Gaussian sampling ($\mathsf{SampleGauss}_\sigma$), and point-wise multiplication ($\mathsf{Pointwise}$, also denoted as $\circ$). We assume that secret and public keys are stored in the read-only flash memory, but loading from RAM would also be possible, and probably even faster. Due to the usage of the NTT, the $\mathrm{NTT}^{CT,\psi}_{no\rightarrow bo}$ transformation is only applied on the Gaussian distributed polynomials $\mathbf{e}_1, \mathbf{e}_2$. Thus we can optimize the transformation for this input distribution and with either $\sigma = 4.52$ or $\sigma = 4.85$ it is possible to substitute approx. 99.96% of the multiplications in stage one of $\mathrm{NTT}^{CT,\psi}_{no\rightarrow bo}$ by look-ups to a table of 16 entries that requires only 32 bytes of flash memory. For the sampling of the Gaussian distributed polynomials with high precision[8] we use a cumulative distribution table (CDT) [17, 22]. We construct the table M with entries $p_z = \Pr(x \leq z : x \leftarrow D_\sigma)$ for $z \in [0, \tau\sigma]$ with a precision of $\lambda = 128$ bits. The tail-cut factor τ determines the number of lines $|z_t| = \lceil \tau\sigma \rceil$ of the table and reduces the output to the range $x \in \{-\lceil \tau\sigma \rceil, \ldots, \lceil \tau\sigma \rceil\}$. To sample a value we choose a uniformly random y from the interval $[0, 1)$ and a bit b and return the integer $(-1)^b z \in \mathbb{Z}$ such that $y \in [p_{z-1}, p_z)$. Further we store only the positive half of the tables and then sample a sign bit. For this, the probability of sampling zero has been pre-halved when constructing the table. For efficiency reasons we just work with the binary expansion of the fractional part instead of floating point arithmetic as all numbers used are smaller than 1.0. The constant CDF matrix M is stored in the read-only flash memory with $k = \lceil \sigma\tau \rceil$ rows and $l = \lceil \lambda/8 \rceil$ columns. In order to sample a Gaussian distributed value we perform a linear search in the table to obtain z. Another option would be binary search, however, for this table size with $x \leftarrow D_\sigma$ being small, the evaluation can already be stopped after only a few comparisons with high probability. The test if the random y is in the range $[p_{z-1}, p_z)$ is performed in a lazy manner on bytes. In this

[8] It is debatable which precision is really necessary in $\mathsf{RLWEenc}$ and what impact less precision would have on the security of the scheme, e.g., $\lambda = 40$. But as the implementation of the CDT for small standard deviations σ is rather efficient and for better comparison with related work like [6, 7, 13] we chose to implement high precision sampling and set $\lambda = 128$.

terms laziness means that a comparison is finished when the first bit (or byte on an 8-bit architecture) has been found that differs between two values. Thus we do not need to sample the full λ bits of y and obtain the result of the comparisons early. Random numbers are obtained from a pseudo random number generator (PRNG) using the hardware AES-128 engine running in counter mode. The PRNG is seeded by noise from the LSB of the analog digital converter. For the state (key, plaintext) 32 bytes of statically allocated memory are necessary. The final table size is 624 bytes for $q = 7681$ and 660 bytes for $q = 12289$.

4.3 Implementation of BLISS

Besides polynomial arithmetic the most expensive operation for BLISS is the sampling of $\mathbf{y}_1$ and $\mathbf{y}_2$ from a discrete Gaussian distribution ($\mathsf{SampleGauss}_\sigma$). For the rather large standard deviation of $\sigma = 215.73$ (RLWEenc requires only $\sigma = 4.85$) a straightforward CDT sampling approach, even with binary search, would lead to a large table with roughly $\tau\sigma = 2798$ entries of approx. 30 to 40 kilobytes overall (see [6]). Another option for embedded devices would be the Bernoulli approach from [18] implemented in [7] but the reported performance of 13,151,929 cycles to sample one polynomial would cause a massive performance penalty. As a consequence, we implemented the hardware-optimized sampler from [47] on the ATxmega. It uses the convolution property of Gaussians combined with Kullback-Leibler divergence and mainly exploits that it is possible to sample a Gaussian distributed value with variance σ^2 by sampling x_1, x_2 from a Gaussian distribution with smaller standard deviation σ' such that $\sigma'^2 + k^2\sigma'^2 = \sigma^2$ and combining them as $x_1 + kx_2$ (for BLISS-I $k = 11$ and $\sigma' = 19.53$). Additionally, the performance of the σ'-sampler is improved by the use of short-cut intervals where each possible value of the first byte of the uniformly distributed input is assigned to an interval that specifies the range of the possible sampled values. This approach reduces the number of necessary comparisons and nearly compensates for the additional costs incurred by the requirement to sample two values (x_1, x_2 with $\sigma' = 19.53$) instead of one directly (with $\sigma = 215.73$). The sampling is again performed in a lazy manner and we use the same PRNG based on AES-128 as for RLWEenc.

To implement the $\mathrm{NTT}^{CT}_{no\rightarrow bo}$ we did not use a look-up table for the first stage as the input range $[-\sigma\tau, \sigma\tau]$ of $\mathbf{y}_1$ or $\mathbf{z}_1$ is too large. To realize the reduction mod $2q$, we create a second reduction function by simply extending the approach from Figure 3. The difference is that for the modulus $2q$, `v` is loaded with the initial value `12289 << 16` and therefore two additional steps `REDUCE30` and `REDUCE29` have to be inserted. Finally, we exclude the last step that subtracts q to get a result in $[0, 2q]$. For the instantiation of the random oracle (Hash) that is required during signing and verification we have chosen the official AVR implementation of Keccak [4]. From the output of the hash function the sparse polynomial $\mathbf{c}$ with κ coefficients equal to one is generated by the GenerateC (see [19, Section 4.4]) routine. We store only κ indices where a coefficient of $\mathbf{c}$ is one. This reduces the dynamic RAM consumption and allows a more efficient implementation of the multiplication of $\mathbf{c}$ by $\mathbf{s}_1$ and $\mathbf{s}_2$ using the SparseMul routine. By using column-wise multiplication and by ignoring all zero coefficients, the multiplication can be performed more efficiently than with the NTT.

5 Results and Comparison

All implementations are measured on an 8-bit ATxmega128A1 microcontroller running at 32 MHz and featuring 128 Kbytes read-only flash, 8 Kbytes RAM and 2 Kbytes EEPROM. Cycle accurate performance measurement were obtained using two coupled 16-bit timer/counters and dynamic RAM consumption is measured using stack canaries (see Appendix A.2). All public and private keys are assumed to be stored in the flash of the microcontroller and we consider the `.text` + `.data` + `.bootloader` sections to determine the flash memory utilization. For our implementation we used no calls to the standard library, the avr-gcc compiler in version 4.7.0, and the following compiler options (shortened): `-Os -fpack-struct -ffunction-sections -fdata-sections -flto`.

Table 1. Cycle counts and Flash memory consumption in bytes for the implementation of Ring-LWEEncrypt on an 8-bit ATxmega128 microcontroller using the NTT. The stack usage is divided into a fixed amount of memory necessary for plaintext, ciphertext, and additional components (like random number generation) and the dynamic consumption of the encryption and decryption routine. We encrypt a message of n bits.

Operation	(n=256, q=7681)	(n=512, q=12289)
	Cycle counts and stack usage	
$\mathsf{RLWEenc}_{\mathrm{ENC}}$	874,347 (109 bytes)	2,196,945 (102 bytes)
$\mathsf{RLWEenc}_{\mathrm{DEC}}$	215,863 (73 bytes)	600,351 (68 bytes)
$\mathrm{NTT}^{CT,\psi}_{no\rightarrow bo}$	185,360	502,896
$\mathrm{INTT}^{GS,\psi^{-1}}_{bo\rightarrow no}$	168,853	427,827
$\mathsf{SampleGauss}_{\sigma}$	84,001	170,861
PwMulFlash	22,012	53,891
AddEncode	16,884	37,475
Decode	4,407	8,759
	Cycle counts of obsolete functions	
$\mathrm{NTT}^{CT}_{bo\rightarrow no}$	198,491	521,872
BitRev	29,696	75,776
BitrevDual	32,768	79,872
PowMul_{ψ}	35,068	96,603
	Static memory consumption in bytes	
Complete binary	6,668	9,258
RAM	1,088	2,144

LP-RLWE. Detailed cycle counts for the encryption and decryption as well as the most expensive operations are given in Table 1. The costs of the encryption are dominated by the NTT (two calls of $\mathrm{NTT}^{CT,\psi}_{no\rightarrow bo}$ and one call of $\mathrm{INTT}^{GS}_{bo\rightarrow no}$) which requires approx. 62% of the overall cycles for RLWEenc-Ia. The Gaussian sampling requires 29% of the overall cycles which is approx. 328 (RLWEenc-Ia) or 334 (RLWEenc-IIa) cycles per sample. The reason that $\mathrm{NTT}^{CT,\psi}_{no\rightarrow bo}$ is slightly faster than $\mathrm{INTT}^{GS,\psi^{-1}}_{bo\rightarrow no}$ is that we use table look-ups for the first stage of the forward transformation (see Section 4.2). The remaining amount of cycles (9%) is consumed by additions, point-wise multiplications by a constant/key stored in the flash (PwMulFlash), and message encoding (Encode). In Table 1 we also list cycle counts of operations that are now obsolete, especially BitRev and PowMul. For PowMul we assume an implementation where the powers of ψ are computed on-the-fly to save flash memory, otherwise the costs are the same as PwMulFlash. Decryption is extremely simple, fast, and basically calls $\mathrm{INTT}^{GS,\psi^{-1}}_{bo\rightarrow no}$, the decoding and an addition so that roughly 148 decryption operation could be performed per second on the ATxmega128. Note that we also evaluated RLWEenc-Ib, RLWEenc-IIb, RLWEenc-IIIb using classic schoolbook multiplication and Karatsuba multiplication and provide results in Appendix A.3. However, it turned out that these algorithms cannot beat the NTT and might only be advantageous when extremely small code size is required

The NTT can be performed in place so that no additional large temporary memory on the stack is needed. But storing the NTT twiddle factors for forward and inverse transforms in flash consumes $2n$ words $= 4n$ bytes which is around 15% of the allocated flash memory for $q = 7681$ and around 22% for $q = 12289$.

BLISS. In Table 2, we present detailed cycle counts for signing and verifying as well as for the most expensive operations in BLISS-I. Due to the rejection sampling and the chosen parameter set 1.6 signing attempts are required on average to create one signature. One attempt requires 6,381,428 cycles on average and only a small portion of the computation, i.e. the hashing of the message, does not have to be repeated in case of a rejection. During a signing attempt the most expensive operation is the sampling of two Gaussian distributed polynomials which takes $2 \times 1{,}140{,}600 = 2{,}281{,}200$ cycles (36% of the overall cycles). The calls to $\mathrm{NTT}^{CT,\psi}_{no\rightarrow bo}$ and $\mathrm{INTT}^{GS,\psi^{-1}}_{bo\rightarrow no}$ account for 16% of the overall

Table 2. Cycle counts and Flash memory consumption in bytes for the implementation of BLISS on an 8-bit ATxmega128 microcontroller. The stack usage is divided into a fixed amount of memory necessary for for message, signature, and additional components (like random number generation) and the dynamic consumption of the signing and verification routine. We sign a message of n bits.

Operation	(n=512, q=12289)
	Cycle counts and stack usage
$\text{BLISS}_{\text{SIGN}}$	10,537,981 (4,012 bytes)
$\text{BLISS}_{\text{VER}}$	2,814,118 (1,103 bytes)
$\text{NTT}^{CT,\psi}_{no\rightarrow bo}$	521,872
$\text{INTT}^{GS,\psi^{-1}}_{bo\rightarrow no}$	497,815
$\text{SampleGauss}_{\sigma}$	1,140,600
SparseMul	503,627
Hash	1,335,040
GenerateC	4,410
DropBits	11,826
$\cosh\left(\frac{\langle \mathbf{z},\mathbf{Sc}\rangle}{\sigma^2}\right)$	75,601
$M\exp\left(-\frac{\|\mathbf{Sc}\|^2}{2\sigma^2}\right)$	37,389
	Static memory consumption in bytes
Complete binary	18,674
RAM	2,411

cycles of one attempt. In contrast to the RLWEenc implementation we do not use a look-up table for the first stage of $\text{NTT}^{CT,\psi}_{no\rightarrow bo}$. Additionally, we do not implement a separate modulo reduction after the subtraction in the GS butterfly and reduce the result after the multiplication which explains the slightly better performance of $\text{INTT}^{GS,\psi^{-1}}_{bo\rightarrow no}$. Hashing the compressed $\mathbf{u}$ and the message μ is time consuming and accounts for roughly 21% of the overall cycles during one attempt. Savings would be definitely possible by using a different hash function (see [2] for an evaluation of different functions) but Keccak appears to be a conservative choice that matches the 128-bit security target very well. The sparse multiplication takes only 503,627 cycles for one multiplication. This makes it a favorable approach on the ATxmega and overall the sparse multiplication is 3.6 times faster than an NTT multiplication approach that would require one $\text{NTT}^{CT,\psi}_{no\rightarrow bo}$, two $\text{INTT}^{GS,\psi^{-1}}_{bo\rightarrow no}$ and two PwMulFlash calls. The flash memory consumption includes $2n$ words which equals $4n = 2{,}048$ bytes for the NTT twiddle factors and 3,374 bytes for look-up tables of the sampler.

Comparison. A detailed comparison of our implementation with related work that also targets the AVR platform[9] and provides 80 to 128-bits of security is given in Table 3. Our implementation of RLWEenc-Ia encryption outperforms the software from [7] by a factor of 3.5 and results from [6] by a factor of 5.7 in terms of cycle counts. Decryption is 6.3 times and 11.4 times faster, respectively. Compared to the work by Liu et al. [38] we achieve similar results. The authors decided to use a Knuth-Yao Sampler that performs better than our CDT and are faster for the RLWEenc-Ia parameter set but slower for RLWEenc-IIa. For the RLWEenc-IIa we compensate our slower sampler by the faster NTT implementation. Since there is no sampling in the decryption, our decryption is faster for both security levels (39% for RLWEenc-Ia , 17% for RLWEenc-IIa). A faster implementation of our Gaussian sampler, e.g., by also using hash tables and binary search could provide a certain speed up but would also result in more flash consumption.

A comparison between our implementation of BLISS-I and the implementations of [7] and [6] is difficult since the authors implemented the signature as authentication protocol. Therefore they only provide the runtime of a complete protocol run that corresponds to one signing operation and

[9] While the ATxmega128 and ATxmega64 compared to the ATmega64 differ in their operation frequency and some architectural differences cycle counts are mostly comparable.

Table 3. Comparison of our implementations with related work. Cycle counts marked with (*) indicate the runtime of BLISS-I as authentication protocol instead of signature scheme. By AX128 we identify the ATxmega128A1 clocked with 32 MHz, by AX64 the ATxmega64A3 clocked with 32 MHz, by AT64 the ATmega64 clocked with 8 MHz, by AT128 the ATmega128 clocked with 8 MHz, and by AT2560 the ATmega2560 clocked with 16 MHz. Implementations marked with (†) are resistant against timing attacks.

Scheme	Device	Operation	Cycles		OP/s	
RLWEenc-Ia ($n = 256$), **our work**	AX128 (32 MHz)	Enc/Dec	874,347	215,863	36.60	148.24
RLWEenc-IIa ($n = 512$), **our work**	AX128 (32 MHz)	Enc/Dec	2,196,945	600,351	14.57	53.30
RLWEenc-Ia ($n = 256$) [38]	AX128 (32 MHz)	Enc/Dec	666,671	299,538	48	106.83
RLWEenc-IIa ($n = 512$) [38]	AX128 (32 MHz)	Enc/Dec	2,721,372	700,999	11.76	45.65
RLWEenc-Ia ($n = 256$) [7]	AT64 (8 MHz)	Enc/Dec	3,042,675	1,368,969	2.63	5.84
RLWEenc-Ia ($n = 256$) [6]	AX64 (32 MHz)	Enc/Dec	5,024,000	2,464,000	6.37	12.98
BLISS-I, **our work**	AX128 (32 MHz)	Sign/Verify	10,537,981	2,814,118	3.04	11.37
BLISS-I (Bernoulli) [7]	AT64 (8 MHz)	Sign+Verify	42,069,682*		0.19	
BLISS-I (CDT) [6]	AX64 (32 MHz)	Sign+Verify	19,328,000*		1.65	
$\mathrm{NTT}^{CT,\psi}_{no\rightarrow bo}$ ($n = 256$), **our work**	AX128 (32 MHz)	NTT	198,491		161.21	
$\mathrm{NTT}^{CT}_{bo\rightarrow no}$ ($n = 256$) [38]	AX128 (32 MHz)	NTT	169,423		188.88	
$\mathrm{NTT}^{CT}_{bo\rightarrow no}$ ($n = 256$) [7]	AT64 (8 MHz)	NTT	754,668		10.60	
$\mathrm{NTT}^{CT}_{bo\rightarrow no}$ ($n = 256$) [6]	AX64 (32 MHz)	NTT	1,216,000		26.32	
$\mathrm{NTT}^{CT,\psi}_{no\rightarrow bo}$ ($n = 512$), **our work**	AX128 (32 MHz)	NTT	521,872		61.31	
$\mathrm{NTT}^{CT}_{bo\rightarrow no}$ ($n = 512$) [38]	AX128 (32 MHz)	NTT	484,680		66.02	
$\mathrm{NTT}^{CT}_{bo\rightarrow no}$ ($n = 512$) [7]	AT64 (8 MHz)	NTT	2,207,787		3.62	
$\mathrm{NTT}^{CT}_{bo\rightarrow no}$ ($n = 512$) [6]	AX64 (32 MHz)	NTT	2,752,000		11.63	
QC-MDPC [28]	AX128 (32 MHz)	Enc/Dec	26,767,463	86,874,388	1.20	0.36
Ed25519† [32]	AT2560 (16 MHz)	Sign/Verify	23,211,611	32,619,197	0.67	0.49
RSA-1024 [27]	AT128 (8 MHz)	Enc/Dec	3,440,000	87,920,000	2.33	0.09
RSA-1024† [37]	AT128 (8 MHz)	priv. key	75,680,000		0.11	
Curve25519† [21]	AT2560 (16 MHz)	Point mul.	13,900,397		1.15	
ECC-ecp160r1 [27]	AT128 (8 MHz)	Point mul.	6,480,000		1.23	

one verification in our results, but without the expensive hashing as the sparse polynomial $\mathbf{c}$ is not obtained from a random oracle but randomly generated by the other protocol party. However, our implementations of $\mathrm{BLISS}_{\mathrm{SIGN}}$ and $\mathrm{BLISS}_{\mathrm{VER}}$ still require less cycles than the implementation of BLISS-I. The biggest improvement stems from the usage of the KL-convolution sampler from [47] which is superior (1,140,600 cycles per polynomial) compared to the Bernoulli approach (13,151,929 cycles per polynomial [7]) and the straight-forward CDT approach used in [6]. As our implementation of BLISS-I needs 18,674 bytes of flash memory, it is also smaller than the implementation of [6] that requires 66.5 kB of flash memory and the implementation of [7] that needs 25.1 kB of flash memory.

Compared with the 80-bit secure McEliece cryptosystem based on QC-MDPC codes from [28] we get 31 times less cycles for the encryption and even 402 times less cycles for decryption. Our 128-bit secure BLISS-I implementation is 2.2 times faster for signing and 11.6 faster for verification compared to an implementation of the Ed25519 signature scheme for an ATmega2560 [32]. Translating the implementation results for RSA and ECC given in [27] to cycle counts, it turns out that an ECC secp160r1 operation requires 6.5 million cycles. RSA-1024 encryption with public key $e = 2^{16} + 1$ takes 3.4 million cycles [27] and RSA-1024 decryption with Chinese Remainder Theorem (CRT) requires 75.68 million cycles (this number is taken from [37]). A comparison with NTRU implementations is currently not easily possible due to lack of published results for the AVR platform[10].

[10] One exception is a Master thesis by Monteverde [43], but the implemented NTRU251:3 variant is not secure anymore according to recent recommendations in [29].

We also refer to [13] for an implementation of RLWEenc-Ia and RLWEenc-IIa on an ARM Cortex-M4 (32-bit, 168 MHz). However, as the Cortex-M4 is a 32-bit processor a comparison across architectures with different bit-widths is naturally hard.

Outlook. As previously stated, we optimized our implementation for performance and did not consider protection against timing attacks. A natural extension of our work would be a constant time implementation or a at least an implementation whose timing behavior is independent of the secret key or message. While a constant time NTT seems to be relatively straightforward, a constant time discrete Gaussian sampler will probably be more costly than the current implementation due to the high dependence on lazy evaluation (see [8] for an implementation of a constant time CDT sampler). Additionally, we are not aware of cryptanalysis for RLWEenc or BLISS that explicitly takes acceleration of attacks by quantum computers into account. In order to provide a clear picture regarding the security level of these schemes in the presence of quantum computers such analysis seems to be highly desirable.

References

1. Bai, S., Galbraith, S.D.: An improved compression technique for signatures based on learning with errors. In: Benaloh, J. (ed.) Topics in Cryptology - CT-RSA 2014 - The Cryptographer's Track at the RSA Conference 2014, San Francisco, CA, USA, February 25-28, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8366, pp. 28–47. Springer (2014), http://dx.doi.org/10.1007/978-3-319-04852-9_2 1
2. Balasch, J., Ege, B., Eisenbarth, T., Gérard, B., Gong, Z., Güneysu, T., Heyse, S., Kerckhof, S., Koeune, F., Plos, T., Pöppelmann, T., Regazzoni, F., Standaert, F., Assche, G.V., Keer, R.V., van Oldeneel tot Oldenzeel, L., von Maurich, I.: Compact implementation and performance evaluation of hash functions in ATtiny devices. In: Mangard, S. (ed.) Smart Card Research and Advanced Applications - 11th International Conference, CARDIS 2012, Graz, Austria, November 28-30, 2012, Revised Selected Papers. Lecture Notes in Computer Science, vol. 7771, pp. 158–172. Springer (2012), http://dx.doi.org/10.1007/978-3-642-37288-9_11 12
3. Barrett, P.: Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In: Proceedings on Advances in cryptology—CRYPTO '86. pp. 311–323. Springer-Verlag, London, UK (1987), http://dl.acm.org/citation.cfm?id=36664.36688 8
4. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V., Keer, R.V.: Keccak implementation overview (2012), version 3.2, see http://keccak.noekeon.org/Keccak-implementation-3.2.pdf 10
5. Blahut, R.E.: Fast Algorithms for Signal Processing. Cambridge University Press (2010), http://amazon.com/o/ASIN/0521190495/ 1, 2
6. Boorghany, A., Jalili, R.: Implementation and comparison of lattice-based identification protocols on smart cards and microcontrollers. IACR Cryptology ePrint Archive 2014, 78 (2014), http://eprint.iacr.org/2014/078, preliminary version of [7] 1, 9, 10, 12, 13, 14
7. Boorghany, A., Sarmadi, S.B., Jalili, R.: On constrained implementation of lattice-based cryptographic primitives and schemes on smart cards. IACR Cryptology ePrint Archive 2014, 514 (2014), http://eprint.iacr.org/2014/514, successive version of [6] 1, 2, 4, 6, 8, 9, 10, 12, 13, 14
8. Bos, J.W., Costello, C., Naehrig, M., Stebila, D.: Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. IACR Cryptology ePrint Archive 2014, 599 (2014), http://eprint.iacr.org/2014/599 1, 14
9. Brakerski, Z., Langlois, A., Peikert, C., Regev, O., Stehlé, D.: Classical hardness of learning with errors. In: Boneh, D., Roughgarden, T., Feigenbaum, J. (eds.) Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013. pp. 575–584. ACM (2013), http://doi.acm.org/10.1145/2488608.2488680 19
10. Cabarcas, D., Weiden, P., Buchmann, J.: On the efficiency of provably secure NTRU. In: Mosca, M. (ed.) Post-Quantum Cryptography - 6th International Workshop, PQCrypto 2014, Waterloo, ON, Canada, October 1-3, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8772, pp. 22–39. Springer (2014), http://dx.doi.org/10.1007/978-3-319-11659-4_2 1
11. Chen, D.D., Mentens, N., Vercauteren, F., Roy, S.S., Cheung, R.C.C., Pao, D., Verbauwhede, I.: High-speed polynomial multiplication architecture for Ring-LWE and SHE cryptosystems. IEEE Trans. on Circuits and Systems 62-I(1), 157–166 (2015), http://dx.doi.org/10.1109/TCSI.2014.2350431 1
12. Chu, E., George, A.: Inside the FFT Black Box Serial and Parallel Fast Fourier Transform Algorithms. CRC Press, Boca Raton, FL, USA (2000) 2, 5, 6
13. de Clercq, R., Roy, S.S., Vercauteren, F., Verbauwhede, I.: Efficient software implementation of Ring-LWE encryption. IACR Cryptology ePrint Archive 2014, 725 (2014), http://eprint.iacr.org/2014/725 1, 2, 4, 6, 9, 14
14. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex Fourier series. Mathematics of Computation 19, 297–301 (1965) 4

15. Crandall, R., Fagin, B.: Discrete weighted transforms and large-integer arithmetic. Mathematics of Computation 62(205), 305–324 (1994) 3
16. Crandall, R., Pomerance, C.: Prime Numbers: A Computational Perspective. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc. (2001) 2, 3
17. Devroye, L.: Non-Uniform Random Variate Generation. Springer-Verlag (1986), http://luc.devroye.org/rnbookindex.html 9
18. Ducas, L., Durmus, A., Lepoint, T., Lyubashevsky, V.: Lattice signatures and bimodal Gaussians. In: Canetti, R., Garay, J.A. (eds.) Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I. Lecture Notes in Computer Science, vol. 8042, pp. 40–56. Springer (2013), http://dx.doi.org/10.1007/978-3-642-40041-4_3, conference version of [19] 1, 4, 10
19. Ducas, L., Durmus, A., Lepoint, T., Lyubashevsky, V.: Lattice signatures and bimodal Gaussians. IACR Cryptology ePrint Archive 2013, 383 (2013), http://eprint.iacr.org/2013/383, full version of [19] 10, 15
20. Ducas, L., Lyubashevsky, V., Prest, T.: Efficient identity-based encryption over NTRU lattices. In: Sarkar, P., Iwata, T. (eds.) Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II. Lecture Notes in Computer Science, vol. 8874, pp. 22–41. Springer (2014), http://dx.doi.org/10.1007/978-3-662-45608-8_2 1, 3, 4, 9
21. Düll, M., Haase, B., Hinterwälder, G., Hutter, M., Paar, C., Sánchez, A.H., Schwabe, P.: High-speed curve25519 on 8-bit, 16-bit and 32-bit microcontrollers. Design, Codes and Cryptography (to appear), document ID: bd41e6b96370dea91c5858f1b809b581, http://cryptojedi.org/papers/#mu25519 13
22. Dwarakanath, N.C., Galbraith, S.D.: Sampling from discrete Gaussians for lattice-based cryptography on a constrained device. Appl. Algebra Eng. Commun. Comput. 25(3), 159–180 (2014), http://dx.doi.org/10.1007/s00200-014-0218-3 9
23. Gentleman, W.M., Sande, G.: Fast fourier transforms: for fun and profit. In: American Federation of Information Processing Societies: Proceedings of the AFIPS '66 Fall Joint Computer Conference, November 7-10, 1966, San Francisco, California, USA. AFIPS Conference Proceedings, vol. 29, pp. 563–578. AFIPS / ACM / Spartan Books, Washington D.C. (1966), http://doi.acm.org/10.1145/1464291.1464352 5
24. Göttert, N., Feller, T., Schneider, M., Buchmann, J., Huss, S.A.: On the design of hardware building blocks for modern lattice-based encryption schemes. In: Prouff, E., Schaumont, P. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7428, pp. 512–529. Springer (2012), http://dx.doi.org/10.1007/978-3-642-33027-8_30 4
25. Güneysu, T., Lyubashevsky, V., Pöppelmann, T.: Practical lattice-based cryptography: A signature scheme for embedded systems. In: Prouff, E., Schaumont, P. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7428, pp. 530–547. Springer (2012), http://dx.doi.org/10.1007/978-3-642-33027-8_31 1
26. Güneysu, T., Oder, T., Pöppelmann, T., Schwabe, P.: Software speed records for lattice-based signatures. In: Gaborit, P. (ed.) Post-Quantum Cryptography - 5th International Workshop, PQCrypto 2013, Limoges, France, June 4-7, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7932, pp. 67–82. Springer (2013), http://dx.doi.org/10.1007/978-3-642-38616-9_5 1
27. Gura, N., Patel, A., Wander, A., Eberle, H., Shantz, S.C.: Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In: Joye, M., Quisquater, J.J. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings. Lecture Notes in Computer Science, vol. 3156, pp. 119–132. Springer (2004), http://dx.doi.org/10.1007/978-3-540-28632-5_9 1, 13, 18
28. Heyse, S., von Maurich, I., Güneysu, T.: Smaller keys for code-based cryptography: QC-MDPC McEliece implementations on embedded devices. In: Bertoni, G., Coron, J. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2013 - 15th International Workshop, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8086, pp. 273–292. Springer (2013), http://dx.doi.org/10.1007/978-3-642-40349-1_16 13
29. Hirschhorn, P.S., Hoffstein, J., Howgrave-Graham, N., Whyte, W.: Choosing NTRUEncrypt parameters in light of combined lattice reduction and MITM approaches. In: Abdalla, M., Pointcheval, D., Fouque, P.A., Vergnaud, D. (eds.) Applied Cryptography and Network Security, 7th International Conference, ACNS 2009, Paris-Rocquencourt, France, June 2-5, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5536, pp. 437–455 (2009), http://dx.doi.org/10.1007/978-3-642-01957-9_27 13
30. Hoffstein, J., Pipher, J., Schanck, J.M., Silverman, J.H., Whyte, W.: Practical signatures from the partial Fourier recovery problem. In: Boureanu, I., Owesarski, P., Vaudenay, S. (eds.) Applied Cryptography and Network Security - 12th International Conference, ACNS 2014, Lausanne, Switzerland, June 10-13, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8479, pp. 476–493. Springer (2014), http://dx.doi.org/10.1007/978-3-319-07536-5_28 1
31. Hoffstein, J., Pipher, J., Silverman, J.H.: NTRU: A ring-based public key cryptosystem. In: Buhler, J. (ed.) Algorithmic Number Theory, Third International Symposium, ANTS-III, Portland, Oregon, USA, June 21-25, 1998, Proceedings. Lecture Notes in Computer Science, vol. 1423, pp. 267–288. Springer (1998), http://dx.doi.org/10.1007/BFb0054868 1

32. Hutter, M., Schwabe, P.: Nacl on 8-bit AVR microcontrollers. In: Youssef, A., Nitaj, A., Hassanien, A.E. (eds.) Progress in Cryptology - AFRICACRYPT 2013, 6th International Conference on Cryptology in Africa, Cairo, Egypt, June 22-24, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7918, pp. 156–172. Springer (2013), http://dx.doi.org/10.1007/978-3-642-38553-7_9 1, 13
33. Hutter, M., Wenger, E.: Fast multi-precision multiplication for public-key cryptography on embedded microprocessors. In: Preneel, B., Takagi, T. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6917, pp. 459–474. Springer (2011), http://dx.doi.org/10.1007/978-3-642-23951-9_30 18
34. Karatsuba, A., Ofman, Y.: Multiplication of multidigit numbers on automata. In: Soviet physics doklady. vol. 7, p. 595 (1963) 19
35. Lindner, R., Peikert, C.: Better key sizes (and attacks) for LWE-based encryption. In: Kiayias, A. (ed.) Topics in Cryptology - CT-RSA 2011 - The Cryptographers' Track at the RSA Conference 2011, San Francisco, CA, USA, February 14-18, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6558, pp. 319–339. Springer (2011), http://dx.doi.org/10.1007/978-3-642-19074-2_21 1, 3, 4
36. Liu, M., Nguyen, P.Q.: Solving BDD by enumeration: An update. In: Dawson, E. (ed.) Topics in Cryptology - CT-RSA 2013 - The Cryptographers' Track at the RSA Conference 2013, San Francisco,CA, USA, February 25-March 1, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7779, pp. 293–309. Springer (2013), http://dx.doi.org/10.1007/978-3-642-36095-4_19 4
37. Liu, Z., Großschädl, J., Kizhvatov, I.: 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) 13
38. Liu, Z., Seo, H., Roy, S.S., Großschädl, J., Kim, H., Verbauwhede, I.: Efficient Ring-LWE encryption on 8-bit AVR processors. IACR Cryptology ePrint Archive 2015, 410 (2015), http://eprint.iacr.org/2015/410, to appear in CHES'15 1, 8, 12, 13
39. Lyubashevsky, V.: Lattice signatures without trapdoors. In: Pointcheval, D., Johansson, T. (eds.) Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7237, pp. 738–755. Springer (2012), http://dx.doi.org/10.1007/978-3-642-29011-4_43 1
40. Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. In: Gilbert, H. (ed.) Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 - June 3, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6110, pp. 1–23. Springer (2010), http://dx.doi.org/10.1007/978-3-642-13190-5_1 1, 3, 16
41. Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings (2010), presentation of [40] given by Chris Peikert at Eurocrypt'10. See http://www.cc.gatech.edu/~cpeikert/pubs/slides-ideal-lwe.pdf. 3
42. Melchor, C.A., Boyen, X., Deneuville, J., Gaborit, P.: Sealing the leak on classical NTRU signatures. In: Mosca, M. (ed.) Post-Quantum Cryptography - 6th International Workshop, PQCrypto 2014, Waterloo, ON, Canada, October 1-3, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8772, pp. 1–21. Springer (2014), http://dx.doi.org/10.1007/978-3-319-11659-4_1 1
43. Monteverde, M.: NTRU software implementation for constrained devices. Master's thesis, Katholieke Universiteit Leuven (2008) 13
44. Nussbaumer, H.J.: Fast Fourier Transform and Convolution Algorithms, Springer Series in Information Sciences, vol. 2. Springer, Berlin, DE (1982) 1, 2
45. Oder, T., Pöppelmann, T., Güneysu, T.: Beyond ECDSA and RSA: lattice-based digital signatures on constrained devices. In: The 51st Annual Design Automation Conference 2014, DAC '14, San Francisco, CA, USA, June 1-5, 2014. pp. 1–6. ACM (2014), http://doi.acm.org/10.1145/2593069.2593098 1
46. Peikert, C.: Lattice cryptography for the Internet. In: Mosca, M. (ed.) Post-Quantum Cryptography - 6th International Workshop, PQCrypto 2014, Waterloo, ON, Canada, October 1-3, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8772, pp. 197–219. Springer (2014), http://dx.doi.org/10.1007/978-3-319-11659-4_12 1
47. Pöppelmann, T., Ducas, L., Güneysu, T.: Enhanced lattice-based signatures on reconfigurable hardware. In: Batina, L., Robshaw, M. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8731, pp. 353–370. Springer (2014), http://dx.doi.org/10.1007/978-3-662-44709-3_20 1, 2, 4, 6, 10, 13
48. Pöppelmann, T., Güneysu, T.: Towards practical lattice-based public-key encryption on reconfigurable hardware. In: Lange, T., Lauter, K.E., Lisonek, P. (eds.) Selected Areas in Cryptography - SAC 2013 - 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8282, pp. 68–85. Springer (2013), http://dx.doi.org/10.1007/978-3-662-43414-7_4 1, 3
49. Pöppelmann, T., Güneysu, T.: Area optimization of lightweight lattice-based encryption on reconfigurable hardware. In: IEEE International Symposium on Circuits and Systemss, ISCAS 2014, Melbourne, Victoria, Australia, June 1-5, 2014. pp. 2796–2799. IEEE (2014), http://dx.doi.org/10.1109/ISCAS.2014.6865754 1, 19, 20
50. Rich, S., Gellman, B.: NSA seeks quantum computer that could crack most codes. The Washington Post (2013), http://wapo.st/19DycJT 1

51. Roy, S.S., Vercauteren, F., Mentens, N., Chen, D.D., Verbauwhede, I.: Compact Ring-LWE cryptoprocessor. In: Batina, L., Robshaw, M. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8731, pp. 371–391. Springer (2014), http://dx.doi.org/10.1007/978-3-662-44709-3_21 1, 2, 3, 4, 5, 6
52. Schönhage, A., Strassen, V.: Schnelle multiplikation grosser zahlen. Computing 7(3), 281–292 (1971) 2
53. Shor, P.: Algorithms for quantum computation: discrete logarithms and factoring. In: Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on. pp. 124–134. IEEE (1994) 1
54. Stehlé, D., Steinfeld, R.: Making NTRU as secure as worst-case problems over ideal lattices. In: Paterson, K.G. (ed.) Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6632, pp. 27–47. Springer (2011), http://dx.doi.org/10.1007/978-3-642-20465-4_4 1
55. Winkler, F.: Polynomial Algorithms in Computer Algebra (Texts and Monographs in Symbolic Computation). Springer, 1 edn. (8 1996), http://amazon.com/o/ASIN/3211827595/ 1, 2, 3

A Appendix

A.1 NTT Algorithms

For a description of $\text{NTT}^{CT,\psi}_{no\rightarrow bo}$ see Algorithm 7 and for $\text{INTT}^{GS,\psi^{-1}}_{bo\rightarrow no}$ see Algorithm 8.

Algorithm 7 Optimized CT Forward NTT

Precondition: Store n powers of ψ in bit-reversed order in psi^*

function $\text{NTT}^{CT,\psi}_{no\rightarrow bo}(\mathbf{a})$
  $m \leftarrow 1$
  $k \leftarrow n/2$
  while $m < n$ **do**
    for $i = 0$ **to** $m - 1$ **do**
      $jFirst \leftarrow 2 \cdot i \cdot k$
      $jLast \leftarrow jFirst + k - 1$
      $\psi_i \leftarrow psi^*[m + i]$
      for $j = jFirst$ **to** $jLast$ **do**
        $l \leftarrow j + k$
        $t \leftarrow \mathbf{a}[j]$
        $u \leftarrow \mathbf{a}[l] \cdot \psi_i$
        $\mathbf{a}[j] \leftarrow t + u \mod q$
        $\mathbf{a}[l] \leftarrow t - u \mod q$
      end for
    end for
    $m \leftarrow m \cdot 2$
    $k \leftarrow n/2$
  end while
  Return a
end function

Algorithm 8 Optimized GS Inverse NTT

Precondition: Store n powers of ψ^{-1} in bit-reversed order in $invpsi^*$

function $\text{INTT}^{GS,\psi^{-1}}_{bo\rightarrow no}(\mathbf{a})$
  $m \leftarrow n/2$
  $k \leftarrow 1$
  while $m > 1$ **do**
    for $i = 0$ **to** $m - 1$ **do**
      $jFirst \leftarrow 2 \cdot i \cdot k$
      $jLast \leftarrow jFirst + k - 1$
      $\psi_i \leftarrow invpsi^*[m + i]$
      for $j = jFirst$ **to** $jLast$ **do**
        $l \leftarrow j + k$
        $t \leftarrow \mathbf{a}[j]$
        $u \leftarrow \mathbf{a}[l]$
        $\mathbf{a}[j] \leftarrow t + u \mod q$
        $\mathbf{a}[l] \leftarrow (t - u) \cdot \psi_i \mod q$
      end for
    end for
    $m \leftarrow m/2$
    $k \leftarrow k \cdot 2$
  end while
  Return a
end function

A.2 The Atmel ATxmega128A1

In this section we briefly introduce the target device and the profiling system for runtime measurements in more detail.

The Target Device.

Our implementation targets the 8-bit Atmel ATxmega128A1 microcontroller on an Xplain evaluation board. The microcontroller is equipped with 128 Kbytes flash for storage of program code and constants, 8 Kbytes of RAM and 2 Kbytes of EEPROM. The maximum operating frequency is 32 MHz

and the ALU is connected to 32 8-bit registers. Some registers have special purposes like R0 and R1 which are connected to an 8×8-bit multiplier or registers R26-31 which are used to address the flash, RAM or EEPROM. The microcontroller supports native AES and DES encryption but no floating point operations. The controller can be programmed directly in assembly or by using other higher level languages, e.g, C/C++ together with the popular `gcc` compiler framework. The ATxmega is compatible[11] with the AVR instruction set which can also be found in smaller devices, e.g., from the ATmega family. Each instruction takes a fixed amount of cycles mainly depending on the type of operation that should be carried out. As an example, the add instruction just needs one clock cycle, the unsigned multiplication mul two, the relative call rcall two cycles and load/store operations targeting the RAM one or two cycles.

Measurement of Performance and Resource Consumption. Atmel has published a cycle accurate simulator for most AVR families. However, the simulator is slow and not optimal to simulate large runtimes typical in public key cryptography. As a consequence, we implemented a profiling framework directly on our development board. For this purpose, we used the microcontroller's capabilities to provide a 32 bit timer/counter by connecting two 16 bit timers over the event system. Thus we are able to provide cycle accurate measurements of execution times. Measurements of flash memory consumption (.text + .data + .bootloader) include program code and constants and are obtained from the `avr-size` utility. Dynamic RAM consumption (stack) which is freed after a function returns is measurement during runtime using a stack canary.

A.3 Implementation of RLWEenc using Schoolbook Multiplication

Ideal lattices are represented by ideals of the ring $\mathbb{Z}_q[\mathbf{x}]/\langle x^n + 1\rangle$ with polynomials of n integer coefficients such that $f(x) = f_0 + f_1 x + f_2 x^2 + \cdots + f_{n-1}x^{n-1} \in \mathbb{Z}_q[x]$. Polynomial multiplication can be computed without expanding a temporary result by considering the special rule that $x^n \equiv -1$. This leads to the classical "schoolbook" multiplication algorithm

$$\mathbf{ab} = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} (-1)^{\left\lfloor \frac{i+j}{n} \right\rfloor} \mathbf{a}[i]\mathbf{b}[j]x^{i+j \bmod n} \bmod q. \tag{6}$$

Advantages of the algorithm are its simplicity, in-place modular reduction and that it does not depend on any parameters (e.g., n or q) of the multiplied polynomials. The time complexity of $\mathcal{O}(n^2)$ is obviously the drawback of this method. However, for constrained platforms with multiple optimizations goals, it was demonstrated (e.g., in [27,33]) that variants of this method can still be very competitive with respect to the more efficient multiplication techniques discussed later on.

Implementation. The implementation of the RLWEenc parameter sets RLWEenc-Ib, RLWEenc-IIb, and RLWEenc-IIIb mainly requires an efficient schoolbook multiplier and the Gaussian sampler discussed in Section 4.2. The core operation is modular multiplication and we realized the modular multiplication and reduction modulo $q = 4093$ (mm4093) in assembly exploiting that $2^{12} \mod 4093 = 3$. Therefore, we can split the $2\log_2 q$ wide output u of the $\log_2 q \times \log_2 q$ multiplier into an upper and a lower part. Thus $u_{25..0} \bmod 4093 \equiv 2^{12}u_{25..12} + u_{11..0} = 3u_{25...12} + u_{11..0} = 2u_{25..12} + u_{25..12} + u_{11..0}$. The maximal size of the result for two modulo q reduced coefficients is 4104 and requires at maximum one additional subsequent subtraction. The multiplication by 3 is realized by shift and addition operations. As we have chosen to represent $\mathbb{Z}_q[\mathbf{x}]/\langle x^n + 1\rangle$ as a polynomial with unsigned coefficients in $[0, q)$ we do not have to deal with more complicated signed arithmetic and reduction. By carefully assigning the available 8-bit registers according to the compiler's calling convention we ensure that no push and pop instructions are necessary in mm4093. The inner loop of the polynomial multiplication routine

[11] There are small differences between AVR families but the core arithmetic relevant for cryptographic operations is basically the same.

Table 4. Cycle counts and flash memory consumption for the implementation of RLWEenc on an 8-bit ATxmega128 microcontroller using the schoolbook algorithm.

Operation	(n=192, q=4093)	(n=256, q=4093)	(n=320, q=4093)
	Cycle counts and stack usage		
encrypt	7,294,976 (446 bytes)	12,289,025 (573 bytes)	21,684,224 (705 bytes)
decrypt	3,205,121 (444 bytes)	5,491,712 (571 bytes)	9,798,655 (703 bytes)
SchoolMul	3,264,511	5,571,584	9,967,615
SampleGauss$_\sigma$	55,296	73,727	89,088
AddEncode	12,773	16,915	20,707
Decode	3,331	4,419	5,507
	Static memory consumption in bytes		
Flash	4,648	5,032	5,418
RAM	824	1,088	1,352

implemented in assembly mainly consist of ld and lpm instructions to read the first coefficient from RAM and the second coefficient from the flash. A subsequent call to mm4093 performs the modular multiplication and a branch determines whether $i + j > n$ so that the result has to be subtracted or just added to the memory address $i + j \mod n$. The implementation of the encryption procedure consists of three calls to the Gaussian sampler, two calls to SchoolMul and the threshold encoding step.

Results. Results for schoolbook multiplication given in Table 4 show that it takes more than seven million cycles for an encryption operation using the smallest parameter set $n = 192$. For decryption it can be seen that approx. 90 percent of the computation time is spent on the two polynomial multiplications (SchoolMul) while all other operations like addition and decoding have only a marginal impact on the runtime. Larger parameter sets turned out to be impractical, e.g., the largest variant for $n = 320$ already consumes more than 21.7/9.8 million cycles for encryption/decryption. The RAM consumption is mainly dominated by the need to be able to store one temporary polynomial which accounts for $2n$ bytes of memory on the stack. Improvements might be possible when q is chosen as a power of two (see [9, 49]) due to even simpler reduction. In our implementation the reduction modulo 4093 accounts for approx. 25 cycles in mm4093 (called $\approx n^2$ times), however, schoolbook multiplication still does not seem to be competitive.

A.4 Implementation of RLWEenc using Karatsuba Multiplication

Polynomial multiplication with improved complexity of $\mathcal{O}(n^{\log(3)})$ operations in $\mathbb{Z}_q$ can be achieved with Karatsuba's method [34]. The idea behind this method is to trade one expensive multiplication with three cheap additions by means of the divide and conquer principle. Our implementation applies the Karatsuba method up to a configurable number of iterations. In practice it is reasonable to limit the recursion to a certain degree in order to avoid too much computations due to the increasing number of additions. The remaining low-degree polynomial multiplications are then performed with the schoolbook algorithm. In case the degree of the input polynomials is not a power of two, we apply zero padding.

Implementation. Implementations of Karatsuba on constrained devices show typically the drawback of a significant memory consumption, especially, with extensive recursions. We initially aim at reducing memory allocation and thus reserve $2n$ words to store the result of a multiplication and $\sum_{i=0}^{r} \frac{n}{2^i}$ words of additional SRAM storage, depending on the number of recursions r. Immediate reallocation of memory is performed after a recursion step has been completed. Since the degree of the polynomial is then twice as large as the degree of the intermediate polynomials in this recursion we can use the

previously disposed memory as a temporary storage for further intermediate polynomials. For the modular multiplications of individual coefficients we apply the assembler implementation mm4093 introduced above. We also measure the impact of changing the parameter q from 4093 to 4096[12]. We implement a mm4096 function to exploit the fact that 4096 is a power of two.

Table 5. Cycle counts and Flash memory consumption in bytes for the implementation of RLWEenc on an 8-bit ATxmega128 microcontroller using the Karatsuba algorithm. As the parameter sets are identical, cycle counts for helper functions like sampling and encoding can be obtained from Table 4.

Operation	(n=192, q=4093)	(n=256, q=4093)	(n=320, q=4093)
	Cycle counts and stack usage		
encrypt	2,858,803 (2,143 bytes)	4,467,168 (2,760 bytes)	6,426,197 (3,392 bytes)
decrypt	1,336,957 (2,139 bytes)	2,108,398 (2,756 bytes)	3,054,861 (3,388 bytes)
	Static memory consumption in bytes		
Flash	5,920	6,304	6,690
RAM	826	1,090	1,354

Table 6. Cycle counts and Flash memory consumption in bytes for the implementation of RLWEenc on an 8-bit ATxmega128 microcontroller using the Karatsuba algorithm with a modified parameter q=4096.

Operation	(n=192, q=4096)	(n=256, q=4096)	(n=320, q=4096)
	Cycle counts and stack usage		
encrypt	2,158,343 (2,128 bytes)	3,240,959 (2,820 bytes)	4,528,167 (3,458 bytes)
decrypt	994,944 (2,180 bytes)	1,508,089 (2,818 bytes)	2,124,268 (3,456 bytes)
	Static memory consumption in bytes		
Flash	4,956	5,340	5,726
RAM	826	1,090	1,354

Results. Table 5 and Table 6 provide cycle counts for a recursive implementation of the Karatsuba algorithm. We experimentally evaluated that six levels of recursion lead to an optimal runtime for all dimensions. With roughly 2.9 million cycles for one encryption and 1.3 million cycles for decryption, this implementation outperforms schoolbook multiplication and is also able to deal with larger values of n. With q=4096 the performance is even better and also the flash memory consumption decreases since we do not need a complex modular reduction function anymore. However, the recursive nature of the Karatsuba algorithm and the need to reduce the polynomial modulo $x^n + 1$ after it has been completely multiplied leads to significant dynamic RAM consumption (stack usage).

[12] Note that the impact on security is currently not clear, but we still see it as an interesting experiment. We also refer to [49] for a description of a hardware implementation of RLWEenc with q being a power of two and a brief discussion on this choice.