AvrNtru: Lightweight NTRU-based Post-Quantum Cryptography for 8-bit AVR Microcontrollers

Hao Cheng, Johann Großschädl, Peter B. Rønne, and Peter Y. A. Ryan
DCS and SnT, University of Luxembourg, L-4364 Esch-sur-Alzette, Luxembourg
{hao.cheng, johann.groszschaedl, peter.roenne, peter.ryan}@uni.lu

Abstract*—Introduced in 1996, NTRUEncrypt is not only one of the earliest but also one of the most scrutinized lattice-based cryptosystems and expected to remain secure in the upcoming era of quantum computing. Furthermore, NTRUEncrypt offers some efficiency benefits over "pre-quantum" cryptosystems like RSA or ECC since the low-level arithmetic operations are less computation-intensive and, thus, more suitable for constrained devices. In this paper we present AvrNtru, a highly-optimized implementation of NTRUEncrypt for 8-bit AVR microcontrollers that we developed from scratch to reach high performance and resistance to timing attacks. AvrNtru complies with the EESS #1 v3.1 specification and supports product-form parameter sets such as `ees443ep1`, `ees587ep1`, and `ees743ep1`. An entire encryption (including mask generation and blinding-polynomial generation) using the `ees443ep1` parameters requires 847973 clock cycles on an ATmega1281 microcontroller; the decryption is more costly and has an execution time of 1051871 cycles. We achieved these results with the help of a novel hybrid technique for multiplication in a truncated polynomial ring, whereby one of the operands is a sparse ternary polynomial in product form and the other an arbitrary element of the ring. A constant-time multiplication in the ring given by the `ees443ep1` parameters takes only 192577 cycles, which sets a new speed record for the arithmetic part of a lattice-based cryptosystem on AVR.

***Index Terms*—Post-quantum cryptography, Polynomial arithmetic, Product-form polynomials, Constant-time implementation**

I. Introduction

NTRU is the collective name for a family of lattice-based public-key cryptosystems that has its origins in an encryption algorithm proposed in the mid-90's [1]. The security of the NTRU encryption scheme rests upon the intractability of the Closest Vector Problem (CVP) and Shortest Vector Problem (SVP) in a special family of lattices. Like RSA, NTRU can provide both encryption and signatures. However, NTRU has two notable features that distinguish it from RSA and various other public-key cryptosystems, including schemes based on elliptic curves. First, NTRU seems resistant against attacks using Shor's algorithm on a powerful quantum computer and can, hence, be considered for deployment in a post-quantum world. Second, the major arithmetic operation of NTRU is multiplication ("convolution") of polynomials of degree 438 (for 128 bits of security [2]) with small coefficients, which is clearly less costly than a modular exponentiation performed on 3072-bit integers (required for RSA with a security level of roughly 128 bits) or a scalar multiplication in an elliptic curve group of order 256 bits. This feature makes NTRU well suited for many kinds of resource-restricted devices such as smart cards, wireless sensor nodes, and RFID tags.

In this paper we present a highly-optimized software implementation of the ring (i.e. polynomial) arithmetic operations of NTRUEncrypt for the 8-bit AVR platform that achieves record-setting execution times. Our software avoids any form of key-dependent control flow (e.g. conditional branches) in order to withstand timing-based side-channel attacks. Nonetheless, our polynomial arithmetic is significantly faster than that of existing NTRU implementations and outperforms the arithmetic component of most other lattice-based encryption schemes on 8-bit microcontrollers reported in the literature (e.g. [3], [4]). Achieving both high speed *and* high resistance against timing attacks far from trivial and was only possible by devising sophisticated optimization techniques, which are described in the following sections. Our main contribution is a novel and very efficient "hybrid" technique for convolution in a truncated polynomial ring $\mathcal{R}$, where one of the operands is a sparse ternary polynomial (i.e. a polynomial that consists of very few non-0 coefficients, which are either -1 or 1).

Our hybrid technique is inspired by Gura et al.'s classical hybrid method for multiple-precision integer multiplication from CHES 2004 [5] and aims to exploit the relatively large register space of the AVR architecture in order to minimize the execution time. We describe the application of our hybrid technique to a special convolution algorithm that requires the ternary polynomial to be in *product form* [6] (i.e. have the form $a(x) = a_1(x) \star a_2(x) + a_3(x)$ where $a_1(x)$, $a_2(x)$, and $a_3(x)$ are sparse) and introduce an optimized constant-time implementation in assembly language. Product-form polynomials were originally presented in [6] and make it possible to significantly reduce the computational cost of the convolution without compromising the security of NTRU. However, even though the advantages of product-form polynomials are well known since almost 20 years, they have been rarely used in practice because product-form convolution turned out to be very hard to implement in a timing-attack-resistant way. The present paper introduces a solution to this almost 20-year-old cryptographic-engineering challenge and finally demonstrates that product-form polynomials are useful in practice.

We present a detailed analysis of the execution time, RAM consumption, and code size of an assembler implementation of NTRUEncrypt, which we call AvrNtru, for two levels of security, using an 8-bit ATmega1281 microcontroller as evaluation platform. The benchmarking results were collected with the two product-form parameter sets `ees443ep1` and `ees743ep1`, which target 128 and 256 bits of pre-quantum security, respectively [2]. Both parameters are used in some real-world applications, for example the embedded SSL/TLS software library WolfSSL [7]. We evaluate the contribution of the convolution to the overall execution of an encryption and decryption, and provide some new insights to its relative cost when NTRUEncrypt is implemented such that it resists timing attacks. Finally, we compare the benchmarking results of AvrNtru with that of some published implementations of NTRU variants and other cryptosystems on AVR.

II. Overview of NTRUEncrypt

A triple of the form (N, p, q) specifies the basic algebraic structures of NTRUEncrypt [8]. N is called *degree parameter* and determines the quotient ring $\mathcal{R} = \mathbb{Z}[x]/(x^N - 1)$ in which all arithmetic operations take place [1], [9]. Since the "modulus polynomial" that defines $\mathcal{R}$ is simply $x^N - 1$, the multiplication of two polynomials from $\mathcal{R}$ corresponds to the cyclic convolution of their coefficients[1], which can be computed very efficiently compared to the multiplication in more general polynomial quotient rings (see Section IV for details regarding implementation). The parameter p and q are called *small* and *large modulus*, respectively [9]. For example, the parameter sets in [2] use $q = 2^{11} = 2048$ and $p = 3$ across all security levels.

As we will explain below, the multiplications in $\mathcal{R}$ carried out in encryption and decryption involve a reduction of the coefficients modulo q to get, as final result, an element of the quotient ring $\mathcal{R}_q = (\mathbb{Z}/q\mathbb{Z})[x]/(x^N - 1)$. This ring can be identified with a subset of the ring $\mathcal{R}$, and any polynomial $a(x) \in \mathcal{R}$ can be straightforwardly "reduced" to become an element of $\mathcal{R}_q$ by reducing all of its N coefficients modulo q. The corresponding operation in the other direction, used to "lift" an element $a(x)$ from $\mathcal{R}_q$ to $\mathcal{R}$, is a *center-lift* and returns the (unique) polynomial $a'(x) \in \mathcal{R}$ that (i) satisfies $a'(x) \bmod q = a(x)$ and (ii) has coefficients a'_i lying in the range of $[-q/2, q/2 - 1]$. Public keys in NTRUEncrypt are elements of $\mathcal{R}_q$, while private keys are "small" polynomials (i.e. polynomials with coefficients of approximately the same magnitude as p). In this paper, we only consider private keys of the form $f(x) = 1 + pF(x)$ [6], where $F(x)$ is a *ternary polynomial*, i.e. its coefficients are in $\{-1, 0, 1\}$. We define $\mathcal{T}$ as the set of all such ternary polynomials with a degree of $N - 1$. Further, we define $\mathcal{T}(d_1, d_2)$ for positive integers d_1, d_2 as a subset of $\mathcal{T}$ that contains all ternary polynomials which have d_1 coefficients equal to $+1$, d_2 coefficients equal to -1, and the remaining $N - d_1 - d_2$ equal to 0 [10]. The parameters in [2] have a weight parameter $d = \lfloor N/3 \rceil$ for the ternary polynomials to maximize the size of the key space [8]. Plaintexts (after some padding and formatting) are given as elements of $\mathcal{T}$, while ciphertexts are elements of $\mathcal{R}_q$.

Keypair Generation

The generation of a key pair consists of the following steps:

1) Generate a random ternary polynomial $F(x) \in \mathcal{T}(d, d)$.
2) Compute $f(x) = 1 + pF(x)$.
3) Compute the inverse $f(x)^{-1} \bmod q$. When $f(x)$ has no inverse modulo q, then go to Step 1.
4) Generate a second random ternary polynomial $g(x) \in \mathcal{T}(d + 1, d)$. Redo this step when $g(x)$ is not invertible modulo q.
5) Compute $h(x) = f(x)^{-1} \star g(x) \bmod q$.
6) Output $f(x)$ as private key and $h(x)$ as public key.

Encryption

The following steps allow one to obtain the ciphertext $c(x)$ of a message M encrypted under public key $h(x)$:

1) Encode the message M by using a random salt b into a ternary polynomial $m(x) \in \mathcal{T}$.
2) Generate a random blinding polynomial $r(x) \in \mathcal{T}(d, d)$ based on M, b, and $h(x)$.
3) Compute $R(x) = p\, h(x) \star r(x) \bmod q$ and generate a ternary mask polynomial $v(x) \in \mathcal{T}$ based on $R(x)$.
4) Compute $m'(x) = \text{center-lift}(m(x) + v(x) \bmod p)$.
5) Compute $c(x) = R(x) + m'(x) \bmod q$.
6) Output $c(x)$ as ciphertext.

The encryption is a randomized process since it involves a *blinding polynomial* $r(x)$ that is chosen uniformly at random from the set $\mathcal{T}(d, d)$ by the Blinding Polynomial Generation Method (BPGM) [2]. In essence, the BPGM generates $r(x)$ by hashing the message along with an object-ID, the random salt b, and (a part of) the public key $h(x)$. The mask $v(x)$ is generated with a so-called Mask Generation Function (MGF) that gets (among other things) $R(x)$ as input [2].

Decryption

The receiver requires the private key $f(x)$ to decrypt the ciphertext $c(x)$. In essence, the message M can be recovered (and its validity verified) by executing the following steps:

1) Compute $a(x) = c(x) \star f(x) \bmod q = p\, c(x) \star F(x) + c(x) \bmod q$ and obtain $a'(x) = \text{center-lift}(a(x))$.
2) Compute $m'(x) = \text{center-lift}(a'(x) \bmod p)$.
3) Compute $R(x) = c(x) - m'(x) \bmod q$, and generate a ternary mask polynomial $v(x) \in \mathcal{T}$ based on $R(x)$.
4) Compute $m(x) = \text{center-lift}(m'(x) - v(x) \bmod p)$.
5) Decode $m(x)$ to message M and salt b.
6) Generate a blinding polynomial $r(x) \in \mathcal{T}(d, d)$ based on M, b, and $h(x)$.
7) Verify the validity of M by checking if $R(x)$ is equal to $p\, h(x) \star r(x)$. Equality means that M is valid.
8) Output M (if it is valid) or an error message.

III. Polynomial Arithmetic for NTRU

As already noted above, the computation of a convolution of elements of the quotient ring $\mathcal{R} = \mathbb{Z}[x]/(x^N - 1)$ differs from an ordinary multiplication of two polynomials because it involves a reduction of the product modulo $x^N - 1$. Since $x^N \equiv 1 \bmod x^N - 1$, this reduction is relatively inexpensive and can be simply carried out by a substitution of all powers x^{k+N} for $0 \leq k < N - 1$ by x^k, i.e. the exponents of the powers of x must be reduced modulo N. In other words, the higher $N - 1$ coefficients of the product (which has degree $2N - 2$) are "wrapped" into the lower $N - 1$ coefficients.

[1]Hence, the quotient ring $\mathcal{R} = \mathbb{Z}[x]/(x^N - 1)$ is often referred to as ring of convolution polynomials or convolution polynomial ring in the literature.

To discuss the convolution $w(x) = u(x) \star v(x)$ in general form, let $u(x)$ and $v(x)$ be elements of $\mathcal{R} = \mathbb{Z}[x]/(x^N - 1)$ represented by polynomials of degree $N-1$, i.e. they are given as $u(x) = u_{N-1}x^{N-1} + \cdots + u_1x + u_0$ and $v(x) = v_{N-1}x^{N-1} + \cdots + v_1x + v_0$. Then, the so-called convolution product $w(x) = u(x) \star v(x) = u(x)v(x) \bmod x^N - 1$ can be computed according to the following equation.

$$\begin{aligned}
w(x) &= u(x)v(x) \bmod x^N - 1 = \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} u_iv_jx^{i+j} \bmod x^N - 1 \\
&= \sum_{k=0}^{N-1}\Bigg(\sum_{i+j=k} u_iv_j\Bigg)x^k + \sum_{k=N}^{2N-2}\Bigg(\sum_{i+j=k} u_iv_j\Bigg)x^k \bmod x^N - 1 \\
&= \sum_{k=0}^{N-1}\Bigg(\sum_{i+j=k} u_iv_j\Bigg)x^k + \sum_{k=0}^{N-2}\Bigg(\sum_{i+j=k+N} u_iv_j\Bigg)x^{k+N} \bmod x^N - 1 \\
&= \sum_{k=0}^{N-1}\Bigg(\sum_{i+j \equiv k \bmod N} u_iv_j\Bigg)x^k = \sum_{k=0}^{N-1} w_kx^k. \qquad (1)
\end{aligned}$$

Each coefficient w_k is the sum of the coefficient-products u_iv_j over all i and j between 0 and $N-1$ satisfying the condition $i + j \equiv k \bmod N$. The sum for w_k in Equation (1) can also be expressed in the following way

$$\begin{aligned}
w_k &= \sum_{i+j \equiv k \bmod N} u_iv_j \\
&= \sum_{i=0}^{N-1} u_iv_{(k-i \bmod N)} = \sum_{j=0}^{N-1} u_{(k-j \bmod N)}v_j. \qquad (2)
\end{aligned}$$

According to Equation (2), the computation of w_k consists of an addition of N coefficient-products of the form u_iv_j. As a consequence, an "ordinary" algorithm for convolution in the ring $\mathcal{R} = \mathbb{Z}[x]/(x^N - 1)$ has complexity $\mathcal{O}(N^2)$, exactly as the conventional multiplication of two polynomials of degree $N-1$ using e.g. operand-scanning. Advanced multiplication techniques like Karatsuba's algorithm (which can be applied to the convolution as well [11], [12]) allow one to reduce the complexity down to $\mathcal{O}(N^{\log_2 3})$. The multiplication technique for sparse product-form polynomials we describe below has a complexity of $\mathcal{O}(N\sqrt{N}) = \mathcal{O}(N^{3/2})$ and is highly suitable for implementation on small microcontrollers.

The most costly arithmetic operation of NTRU encryption is clearly the computation of the convolution product of the form $h(x) \star r(x) \bmod q$, where the coefficients of $h(x)$ are randomly distributed modulo q and $r(x)$ is a ternary polynomial with a certain minimum number of non-0 coefficients (specified by weight parameter d_r). The decrypting party has to compute two convolutions, namely $c(x) \star f(x) \bmod q$ to obtain the message and, then, $h(x) \star r(x) \bmod q$ to ensure that the message was correctly recovered (i.e. the decrypted message is valid). As explained in the previous section, the ciphertext $c(x)$ is an element of $\mathcal{R}_q$ and $f(x)$ has the special form $f(x) = 1 + pF(x)$, where $F(x)$ is a ternary polynomial [6]. Thus, the convolutions performed in both the encryption and the decryption are multiplications of a polynomial with coefficients in the range of $[0, q-1]$ by a ternary polynomial (both of a degree of up to $N-1$). The fact that one of the operands is a ternary polynomial means that the computation of the convolution product essentially boils down to additions and subtractions of coefficients modulo q. Hence, only two basic arithmetic instructions, namely `add` and `sub`, need to be executed, both of which usually have a latency of a single cycle, even on 8-bit microcontrollers. This is a big advantage of NTRU over lattice-based cryptosystems whose arithmetic portion mainly consists of the Number-Theoretic Transform (NTT), most notably Ring-LWE schemes. The computation of an NTT involves multiplications of coefficients, which, in turn, requires the execution of `mul` instructions. However, on most embedded platforms, the `mul` instruction takes several cycles and also consumes more power (and, therefore, more energy) than a single-cycle `add` or `sub` instruction.

IV. Product-Form Convolution

The computation of the convolution $h(x) \star r(x)$ amounts to d_rN additions (resp. subtractions) of coefficients bounded by q, whereby d_r is the number of non-0 coefficients of the ternary polynomial $r(x)$. To maximize efficiency, it seems tempting to make $r(x)$ as sparse as possible. However, as we briefly stated in Section II, the number of non-0 coefficients of $r(x)$ must not be below a certain limit (specified by the weight parameter d_r, see [2]) as otherwise the search space for $r(x)$ can become too small to guarantee the desired level of security. Nonetheless, it is possible to significantly reduce the cost without compromising security by taking $r(x)$ to be a *product of polynomials with few non-0 coefficients*, as was first proposed in [6]. This means $r(x) = r_1(x)r_2(x)$, where $r_1(x)$ and $r_2(x)$ are two ternary polynomials having d_1 and d_2 non-0 coefficients, respectively. Since the terms of $r_1(x)$ and $r_2(x)$ cross-multiply, the product $r(x)$ can be assumed to have roughly d_1d_2 non-0 coefficients. In practice, $r(x)$ will have a few coefficients outside the range $\{-1, 0, 1\}$, but this does not affect the implementation. It is important to notice that the computation of the product

$$h(x) \star r(x) = (h(x) \star r_1(x)) \star r_2(x) \qquad (3)$$

takes $(d_1 + d_2)N$ coefficient additions/subtractions, i.e. the overall computational complexity is proportional to the sum of d_1 and d_2 [6]. On the other hand, the search space for the pair of polynomials $(r_1(x), r_2(x))$ is, in fact, proportional to the product of the $r_1(x)$ search space and $r_2(x)$ search space (see [6] for more detailed discussion). In summary, using the product $r(x) = r_1(x)r_2(x)$ requires computation proportional to the sum $d_1 + d_2$, while giving security proportional to the product d_1d_2. A very similar optimization is possible for the ternary polynomial $F(x)$ that is the main component of the private key $f(x) = 1 + pF(x)$. For example, one could take $F(x)$ to have the form $F(x) = f_1(x) \star f_2(x) + f_3(x)$ where $f_1(x)$, $f_2(x)$, and $f_3(x)$ are sparse polynomials [6]. In this case, the private key $f(x)$ becomes

$$f(x) = 1 + p\left(f_1(x) \star f_2(x) + f_3(x)\right). \qquad (4)$$

Thanks to the sparsity of f_1, f_2, and f_3, the convolution of $c(x)$ by $f(x)$ can be optimized to achieve high speed in software, even on small microcontrollers. Furthermore, when the target platform does not have a data cache (which is the case for virtually all 8 and 16-bit microcontrollers, and also for numerous 32-bit models, e.g. ARM Cortex-M series), it is possible to implement the product-form convolution to have *constant execution time*, i.e. the execution time only depends on the number of the non-0 coefficients of $f(x)$, but not on their value (i.e. whether they are -1 or $+1$). Our software implementation represents the ciphertext-polynomial $c(x)$ as an array of words of type `uint16_t`, similar to [13]. On the other hand, the ternary polynomials f_1, f_2, and f_3 are not stored in the form N-element arrays, but represented as arrays that contain the indices of the non-0 elements. This representation of f_i has two benefits, namely (i) it is easy to load the corresponding coefficients of $c(x)$ by simply adding the index to the start address of the array in which $c(x)$ is stored, and (ii) the polynomial f_i does not need much space in RAM since only non-0 coefficients are considered.

The convolution of $c(x)$ by the product-form polynomial $f(x)$ can be carried out by three "sub-convolutions," i.e. we first compute $t_1(x) = c(x) \star f_1(x)$, thereafter we compute $t_2(x) = t_1(x) \star f_1(x)$, and finally $t_3(x) = c(x) \star f_3(x)$. This means a product-form convolution actually consists of three sparse-ternary-polynomial convolutions, which dominate the overall execution time. When using a standard multiplication technique then the sparse nature of $f_i(x)$ causes most of the partial products to be 0. So rather than employing a standard polynomial multiplication algorithm that wastes a lot of time by computing 0-terms, we scan $f_i(x)$ for non-0 coefficients and calculate only those few partial-product terms which are non-0. Any non-0 coefficient of $f_i(x)$ will appear in a total of N partial-product terms.

To aid the explanation of our multiplication method, let us use the notation from the start of Section III, which means $u(x)$ is a polynomial of degree N with coefficients from the range $[0, q-1]$, but $v(x)$ is now a ternary polynomial with $m \approx \sqrt{2N/3}$ non-0 coefficients, half of which are $+1$ and the other half -1. We store the N coefficients of $u(x)$ in the array `u` and the indices j of the non-0 coefficients of $v(x)$ in array `v`. To obtain $w(x) = u(x)v(x) \bmod x^N - 1$, we have to calculate for each coefficient w_k of $w(x)$ the sum of all the coefficient products of the form $u_i v_j$ for which $i + j$ is congruent to $k \bmod N$ (see Equation (2)). Taking the least-significant coefficient w_0 (i.e. $k = 0$) as example, we sum up the coefficients $u_{-j \bmod N}$ for every index j for which v_j is $+1$ (or subtract $u_{-j \bmod N}$ for every index j for which v_j is -1). Our implementation includes a simple pre-computation step that calculates for each index j in array `v` the address of coefficient `u[N-j]` and stores it in a temporary array on the stack. However, when array `v` contains index $j = 0$, we store the address of `u[0]` and not the address of `u[N]`. We can re-use this temporary array for the computation of the next result-coefficient w_1 since, when $k = 1$, the coefficients $u_{1-j \bmod N}$ have to be summed up. The addresses of these coefficients are computed by adding 2 to each element of the temporary array. However, care needs to be taken when one of the addresses in the temporary array reaches the address of `u[N]`; if this happens then $2N$ must be subtracted from the address, i.e. the address of `u[N]` has to be corrected to the address of `u[0]`.

After completion of the "pre-computation" phase with the address calculations described above, the actual computation of the sparse-polynomial convolution is relatively simple. In short, the convolution product $w(x)$ is obtained via a nested loop that consists of an outer loop and two inner loops. The outer loop is iterated N times and produces in each iteration one coefficient w_k of the result $w(x)$, starting with the least-significant coefficient w_0. Each of the inner loops is iterated exactly $m/2$ times, whereby the first inner loop performs additions of coefficients, and the second inner loop executes coefficient-subtractions. Our optimized AVR implementation of the first inner loop comprises four operations. At first, an element of the temporary array (which contains addresses of elements of array `u`) is loaded from RAM and placed in pointer register `X`. Then, the coefficient of $u(x)$ that resides at the address in `X` is loaded and added to a coefficient sum held in two registers. Third, the address in `X` is incremented by 2 (which can be done at no additional cost by using the auto-increment addressing mode) so as to prepare it for the next iteration of the outer loop. Finally, the updated address is written back to the temporary array. The second inner loop performs the same steps, except that the loaded coefficient is *subtracted* from the coefficient sum. In total, Nm additions or subtractions of coefficients are carried out.

An important detail not explained above is the correction of the address computed in the third step of each iteration of the inner loop. Since each inner loop is iterated N times in total, it is inevitable that, after a certain number of outer-loop iterations, the incremented address in `X` will exceed the boundary of array `u`, which means it points to `u[N]`. If this happens, $2N$ must be subtracted from `X` (so that `X` contains the address of `u[0]` instead of `u[N]`) before `X` is written back to the temporary array. This address correction has to be performed in constant time (i.e. must not execute conditional statements) to guarantee the software is able to resist timing attacks. Unfortunately, a constant-time implementation of the address correction is relatively costly as it takes 13 cycles on an AVR microcontroller, which has a significant impact on the overall execution time since it needs to be performed in each iteration of the inner loop. For comparison, the actual coefficient addition or subtraction, including all required load and store instructions, takes only 10 clock cycles. We tackle this problem by adopting the so-called *hybrid* multiplication technique of Gura et al from CHES 2004 [5], which means we take advantage of AVR's (relatively) large register space to reduce the number of address corrections.

Applying Gura et al's hybrid method means that, instead of computing one single coefficient w_i of $w(x)$ per iteration of the outer loop and adding (resp. subtracting) only a single coefficient in each iteration of the inner loop(s), we process eight coefficients in each iteration. This implies we have to keep eight coefficient sums in registers, which is no problem since AVR provides 32 general-purpose registers. Thus, the address correction in the inner loop(s) has to be carried out only after addition or subtraction of eight coefficients, which

```c
#define INTMASK(x) (~((x) - 1))

void mul_tern_sparse(uint16_t *w, const uint16_t *u, const
     uint16_t *v, int vlen, int N)
{
  int index[vlen], i, j, k;
  register uint16_t w0, w1, w2, w3, w4, w5, w6, w7;

  for (i = 0; i < vlen; i ++)
    index[i] = INTMASK(v[i] != 0) & (N - v[i]);

  for (i = 0; i < N; i += 8) {
    w0 = w[i  ]; w1 = w[i+1]; w2 = w[i+2]; w3 = w[i+3];
    w4 = w[i+4]; w5 = w[i+5]; w6 = w[i+6]; w7 = w[i+7];
    for (j = 0; j < vlen/2; j ++) {
      k = index[j];
      w0 += u[k  ]; w1 += u[k+1]; w2 += u[k+2]; w3 += u[k+3];
      w4 += u[k+4]; w5 += u[k+5]; w6 += u[k+6]; w7 += u[k+7];
      index[j] = k + 8 - (INTMASK(k + 8 >= N) & N);
    }
    for (j = vlen/2; j < vlen; j ++) {
      k = index[j];
      w0 -= u[k  ]; w1 -= u[k+1]; w2 -= u[k+2]; w3 -= u[k+3];
      w4 -= u[k+4]; w5 -= u[k+5]; w6 -= u[k+6]; w7 -= u[k+7];
      index[j] = k + 8 - (INTMASK(k + 8 >= N) & N);
    }
    w[i  ] = w0; w[i+1] = w1; w[i+2] = w2; w[i+3] = w3;
    w[i+4] = w4; w[i+5] = w5; w[i+6] = w6; w[i+7] = w7;
  }
}
```

Listing 1. Simple C Code for Sparse-Ternary-Polynomial Convolution.

reduces the overall execution time significantly. However, to avoid that the address in X exceeds the bound of array u, we define this array to contain $N + 7$ elements, and initialize u[N] with u[0], u[N+1] with u[1], and so on. Only the combination of the constant-time address correction with the hybrid multiplication technique makes it possible to obtain an implementation that is both fast *and* resistant against timing attacks. Furthermore, our constant-time hybrid convolution technique for sparse ternary polynomials is also quite simple and can be implemented with just 30 lines of ISO C code as shown in Listing 1.

V. Results and Comparison

We integrated the product-form convolution described in the last section into AvrNtru, an optimized implementation of NRTEUEncrypt for the 8-bit AVR platform. AvrNtru is compliant with version 3.1 of the EESS #1 specification and supports product-form parameter sets like ees443ep1 and ees743ep1 [2]. We aimed to achieve high speed, resistance against timing attacks, and high scalability, i.e. the ability to switch the parameter set "on the fly" without recompilation of the source code. AvrNtru was written in ISO C99 and includes hand-optimized assembly code for functions that are either performance-critical or could potentially leak sensitive information in a timing attack. In particular, the product-form convolution, helper functions for e.g. data-type conversions or encoding/decoding of data, and the compression function of SHA256, use optimized assembly code. The hash function SHA256 is an essential part of the BPGM and MGF (see Section II), both of which have a significant impact on the overall performance of NTRUEncrypt. Our implementation of SHA256 adopts very similar optimizations as the SHA512 software for AVR introduced in [14]. The specific device we used to test (resp. benchmark) AvrNtru is an ATmega1281 microcontroller, which features 8 kB RAM and 128 kB flash memory. We compiled our source code with avr-gcc version 5.4.0 under optimization level O2. The compilation produces constant-time executables that take a fixed number of cycles for different inputs (but same parameter set), which confirms that AvrNtru can withstand timing attacks.

TABLE I
Execution time (in clock cycles) of AvrNtru.

Operation	ees443ep1		ees743ep1	
	C	ASM	C	ASM
Ring-Mul	262,916	192,577	695,676	519,746
Encryption	3,476,073	847,973	5,554,461	1,550,538
Decryption	3,756,058	1,051,871	6,272,385	2,080,078

TABLE II
RAM footprint and code size (both in bytes) of AvrNtru.

Operation	ees443ep1				ees743ep1			
	C		ASM		C		ASM	
	RAM	Size	RAM	Size	RAM	Size	RAM	Size
Encrypt.	3,004	7,856	2,989	6,652	4,853	7,856	4,879	6,652
Decrypt.	3,935	8,596	3,934	7,460	6,407	8,596	6,427	7,460
Enc.+Dec.	3,935	10,268	3,934	8,940	6,407	10,268	6,427	8,940

Table I specifies the execution time of AvrNtru for the parameter sets eess443ep1 and eess743ep1, which are aimed at 128 and 256 bits of security, respectively. The table also lists the time for a ring-multiplication (i.e. convolution) alone, which is 192.6 k cycles for the NTRU ring of degree $N = 443$. This is, to our knowledge, the fastest convolution time on AVR ever reported in the literature and also marks a new speed record for the arithmetic part of a lattice-based cryptosystem on an 8-bit microcontroller. We made various experiments with other techniques for ring-multiplication, in particular combinations between multi-level Karatsuba and the hybrid multiplication approach from [5]. The fastest was a variant with four levels of Karatsuba and a hybrid method that processes two coefficients at a time. However, according to our evaluation, this Karatsuba-based multiplication has an execution time of 1.1 M cycles for $N = 443$, which means our product-form convolution almost six times faster. A full encryption takes roughly 848 k clock cycles when using the ees443ep1 parameters; the decryption is 24% slower since it involves a second convolution. The proposed product-form convolution makes the arithmetic part of NTRUEncrypt so fast that the overall execution time is now dominated by the auxiliary functions, most notably MGF and BPGF.

Table II gives the memory (i.e. stack) usage and code size of AvrNtru. When using the ees443ep1 parameters, the assembly-accelerated implementation needs 3.9 kB RAM and occupies 8.9 kB flash memory. The peak RAM consumption in encryption happens during the convolution because three arrays have to be kept in RAM, each consisting of $2N$ bytes (i.e. 2658 bytes in total). Decryption takes more RAM since $R(x)$ (obtained in step 3) has to be temporarily stored on the stack before performing the second convolution (step 7). Due to the fact that encryption and decryption can share various functions, the overall code size of encryption plus decryption is just slightly larger than encryption or decryption alone.

TABLE III
EXECUTION TIME (IN CLOCK CYCLES) OF SOME PRE- AND POST-QUANTUM CRYPTOSYSTEMS ON DIFFERENT MICROCONTROLLERS.

Implementation	Alg.	Security	Processor	Enc.	Dec.
This work	NTRU	128-bit	ATmega1281	847,973	1,051,871
This work	NTRU	256-bit	ATmega1281	1,550,538	2,080,078
Boorghany [15]	NTRU	128-bit	ATmega64	1,390,713	2,008,678
Boorghany [15]	NTRU	128-bit	ARM7TDMI	693,720	998,760
Guillen [16]	NTRU	128-bit	Cortex-M0	588,044	950,371
Guillen [16]	NTRU	192-bit	Cortex-M0	1,040,538	1,634,821
Guillen [16]	NTRU	256-bit	Cortex-M0	1,411,557	2,377,054
Gura [5]	RSA	80-bit	ATmega128	3,440,000	87,920,000
Düll [17]	ECC	128-bit	ATmega2560	13,900,397	13,900,397
Liu [3]	RLWE	128-bit	ATxmega128	796,872	215,031
Liu [3]	RLWE	256-bit	ATxmega128	1,975,806	553,536

Table III compares the execution time of various software implementations of NTRUEncrypt, including AVRNTRU, on 8-bit and 32-bit platforms. Compared to Boorghany et al's implementation [15], which presently holds the speed record for NTRUEncrypt on an AVR microcontroller (for 128 bits of security), AVRNTRU is 1.6 times faster for encryption and 1.9 times faster for decryption. Due to the efficiency of the product-form convolution, AVRNTRU is just a bit slower than most NTRU implementations on more powerful 32-bit platforms and even outperforms Guillen et al's NTRU decryption for ARM at the 256-bit security level. Table III also shows the execution time of some other public-key cryptosystems on AVR. The to-date fastest implementation of variable-base scalar multiplication on Curve25519 was introduced by Düll et al. in [17] and takes $13.9\,\mathrm{M}$ cycles. In comparison, when AVRNTRU is instantiated for 128-bit security, it outperforms Curve25519 by over an order of magnitude. An optimized implementation of another popular lattice-based encryption scheme, namely Ring-LWE, was reported in [3], though this software mainly covers the ring-arithmetic and should not be seen as implementation of a fully-fledged cryptosystem like NTRUEncrypt. AVRNTRU is slower for both the encryption and decryption at the 128-bit security level, mainly because it includes costly non-arithmetic components like BPGF and MGF. When only ring arithmetic is considered, AVRNTRU is faster at both the 128-bit and 256-bit level of security.

VI. CONCLUSIONS

In this paper, we showed that product-form convolution is an interesting implementation option for NTRU, especially on small microcontrollers without data cache, because it enables high speed *and* resistance against timing-based side-channel attacks. As a consequence, our work refutes the conventional wisdom that product-form polynomials are not very useful in the real world; in particular, it contradicts the claim made in [11] that sparse product-form multiplication is "very hard to implement in constant-time fashion without losing the speed benefits." According to our experiments, the proposed hybrid convolution using product-form polynomials can be executed in about $192.6\,\mathrm{k}$ cycles on AVR (for $N = 443$), which sets a new speed record for the arithmetic part of a lattice-based cryptosystem on an 8-bit device. Due to its efficient product-form convolution, AVRNTRU achieves the fastest execution time of all known NTRUEncrypt software implementations for the AVR platform. Furthermore, AVRNTRU outperforms elliptic curve cryptosystems (using e.g. Curve25519) by more than an order of magnitude. All this makes AVRNTRU well suited for deployment on resource-limited devices that need an efficient public-key encryption scheme with the guarantee of remaining secure in the post-quantum era.

Acknowledgment: This work was supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 779391 (FutureTPM).

REFERENCES

[1] J. Hoffstein, J. Pipher, and J. H. Silverman, "NTRU: A ring-based public key cryptosystem," in *Algorithmic Number Theory Symposium — ANTS*, 1998, pp. 267–288.

[2] Consortium for Efficient Embedded Security, "Efficient embedded security standards (EESS)#1: Implementation aspects of NTRUEncrypt (Version 3.1)," Available at http://github.com/NTRUOpenSourceProject/ntru-crypto/blob/master/doc/EESS1-v3.1.pdf, 2015.

[3] Z. Liu, T. Pöppelmann, T. Oder, H. Seo, S. Sinha Roy, T. Güneysu, J. Großschädl, H. Kim, and I. Verbauwhede, "High-performance ideal lattice-based cryptography on 8-bit avr microcontrollers," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 4, p. 117, 2017.

[4] T. Pöppelmann, T. Oder, and T. Güneysu, "High-performance ideal lattice-based cryptography on 8-bit ATxmega microcontrollers," in *Progress in Cryptology — LATINCRYPT*, 2015, pp. 346–365.

[5] N. Gura, A. Patel, A. S. Wander, H. Eberle, and S. Chang Shantz, "Comparing elliptic curve cryptography and RSA on 8-bit CPUs," in *Cryptographic Hardware and Embedded Systems — CHES*, 2004, pp. 119–132.

[6] J. Hoffstein and J. H. Silverman, "Optimizations for NTRU," in *Public-Key Cryptography and Computational Number Theory*, 2001, pp. 77–88.

[7] WolfSSL Inc., "Quantum-Safe wolfSSL," Available at https://www.wolfssl.com/quantum-safe-wolfssl-2/, 2015.

[8] J. Hoffstein, J. Pipher, J. M. Schanck, J. H. Silverman, W. Whyte, and Z. Zhang, "Choosing parameters for NTRUEncrypt," Cryptology ePrint Archive, Report 2015/708, 2015, https://eprint.iacr.org/2015/708.

[9] J. Hoffstein, N. Howgrave-Graham, J. Pipher, and W. Whyte, "Practical lattice-based cryptography: NTRUEncrypt and NTRUSign," in *The LLL Algorithm: Survey and Applications*. Springer, 2010, pp. 349–390.

[10] J. Hoffstein, J. Pipher, and J. H. Silverman, *An Introduction to Mathematical Cryptography*, 2nd ed., ser. Undergraduate Texts in Mathematics. Springer, 2014.

[11] W. Dai, W. Whyte, and Z. Zhang, "Optimizing polynomial convolution for NTRUEncrypt," *IEEE Transactions on Computers*, vol. 67, no. 11, pp. 1572–1583, 2018.

[12] A. Hülsing, J. Rijneveld, J. M. Schanck, and P. Schwabe, "High-speed key encapsulation from NTRU," in *Cryptographic Hardware and Embedded Systems — CHES*, 2017, pp. 232–252.

[13] D. V. Bailey, D. Coffin, A. J. Elbirt, J. H. Silverman, and A. D. Woodbury, "NTRU in constrained devices," in *Cryptographic Hardware and Embedded Systems — CHES*, 2001, pp. 262–272.

[14] H. Cheng, D. Dinu, and J. Großschädl, "Efficient implementation of the sha-512 hash function for 8-bit avr microcontrollers," in *Innovative Security Solutions for Information Technology and Communications*, 2019, pp. 273–287.

[15] A. Boorghany, S. Bayat Sarmadi, and R. Jalili, "On constrained implementation of lattice-based cryptographic primitives and schemes on smart cards," *ACM Trans. Embed. Comput. Syst.*, vol. 14, no. 3, p. 42, 2015.

[16] O. M. Guillen, T. Pöppelmann, J. M. Bermudo Mera, E. Fuentes Bongenaar, G. Sigl, and M. J. Sepúlveda, "Towards post-quantum security for IoT endpoints with NTRU," in *Design, Automation and Test in Europe — DATE*, 2017, pp. 698–703.

[17] 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," *Designs, Codes and Cryptography*, vol. 77, no. 2–3, pp. 493–514, 2015.