A Lightweight Implementation of NTRUEncrypt for 8-bit AVR Microcontrollers

Hao Cheng, Johann Großschädl, Peter B. Rønne, and Peter Y. A. Ryan

SnT and CSC, University of Luxembourg
6, Avenue de la Fonte, L–4364 Esch-sur-Alzette, Luxembourg
hao.cheng.001@student.uni.lu, johann.groszschaedl@uni.lu
peter.roenne@uni.lu, 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 a serious contender in NIST's ongoing Post-Quantum Cryptography (PQC) standardization project. An important criterion for the assessment of candidates is their computational cost in various hardware and software environments. This paper contributes to the evaluation of NTRUEncrypt on the ATmega class of AVR microcontrollers, which belongs to the most popular 8-bit platforms in the embedded domain. More concretely, we present AvrNtru, a carefully-optimized implementation of NTRUEncrypt that we developed from scratch with the goal of achieving high performance and resistance to timing attacks. AvrNtru complies with version 3.3 of the EESS#1 specification and supports recent product-form parameter sets like `ees443ep1`, `ees587ep1`, and `ees743ep1`. A full encryption operation (including mask generation and blinding-polynomial generation) using the `ees443ep1` parameters takes 834,272 clock cycles on an ATmega1281 microcontroller; the decryption is slightly more costly and has an execution time of 1,061,683 cycles. When choosing the `ees743ep1` parameters to achieve a 256-bit security level, 1,539,829 clock cycles are cost for encryption and 2,103,228 clock cycles for decryption. We achieved these results thanks to a novel *hybrid* technique for multiplication in truncated polynomial rings where one of the operands is a sparse ternary polynomial in product form. Our hybrid technique is inspired by Gura et al's hybrid method for multiple-precision integer multiplication (CHES 2004) and takes advantage of the large register file of the AVR architecture to minimize the number of load instructions. A constant-time multiplication in the ring specified by the `ees443ep1` parameters requires only 210,827 cycles, which sets a new speed record for the arithmetic component of a lattice-based cryptosystem on an 8-bit microcontroller.

Keywords: Post-Quantum Cryptography, NTRU, Polynomial Arithmetic, Product-Form Polynomials, Constant-Time Implementation

1 Introduction

NTRU (short for N-th degree truncated polynomial ring [41]) 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-1990's [27]. The security of the NTRU encryption scheme rests on the hardness of the Closest Vector Problem (CVP) and Shortest Vector Problem (SVP) in a special class of lattices known as NTRU lattices [6, 13, 44]. Similar to RSA [43], NTRU can be used to design both encryption and signature schemes [24]. However, NTRU has two notable features that distinguish it from RSA and other classical public-key cryptosystems, including those based on elliptic curves [8]. First, NTRU seems robust to

attacks using Shor's algorithm [45] on a quantum computer and can therefore be considered for deployment in a post-quantum world [5, 6]. Second, the main arithmetic operation of NTRU is multiplication ("convolution") of polynomials of degree 438 (for 128-bit security [12]) with small integer coefficients, which is much less costly than a modular exponentiation performed on 3072-bit integers (required for RSA with a security level of 128 bits) or a scalar multiplication in a 256-bit elliptic-curve group. Furthermore, the computational cost of exponentiation and scalar multiplication generally increases with the third power of the group order (i.e. the complexity is bounded by $\mathcal{O}(n^3)$ for n-bit operands), while the complexity of an ordinary multiplication of two n-th degree polynomials is $\mathcal{O}(n^2)$ [27]. These features make NTRU well suited for resource-limited devices such as smart cards, wireless sensor nodes, and RFID tags [2, 4, 9, 18, 20].

In the past 20 years, the security of NTRU in general, and issues related to its parameterization in particular, has been a topic of controversial debate. The original NTRU proposal from [27] lacks a formal security proof, which means its security can only be conjectured on basis of the hardness of the best known attack [6]. Soon after the first presentation of the NTRU algorithm, it became apparent that the NTRU key recovery problem can be embedded into a certain class of lattices as an SVP and, therefore, lattice reduction techniques can be applied to obtain the private key [13]. The to-date most efficient cryptanalytic attack against NTRU combines such lattice reduction with meet-in-the-middle strategies and has an asymptotic complexity that grows exponentially with the security level in bits [31]. Nonetheless, the lack of a strong security guarantee is often perceived as a significant drawback of NTRU, especially when compared with more recent instances of lattice-based cryptography for which a reduction proof to a hard lattice problem exists, e.g. ring-LWE [41]. Besides the analysis of the raw NTRU algorithm, also the security of NTRU-based encryption and digital signature schemes has attracted a lot of interest [24]. In the former case (i.e. NTRUEncrypt), the security situation seems (relatively) stable, apart from discussions about the correct parameterization to minimize decryption failures and to improve the resistance against lattice reduction and meet-in-the-middle attacks [33, 26]. Unfortunately, the history of signature algorithms based on the NTRU lattice is far less glorious and includes a few break-and-repair cycles. In short, the very first NTRU-based signature scheme, called NSS [28], as well as its successor NTRUSign [23] were badly broken [19, 40], and the same happened with some ad-hoc fixes [16]. Very recently, two new proposals appeared in the literature; one hides the information leaked by standard NTRUSign signatures via Gaussian noise [1], while the other uses rejection sampling to ensure that a transcript of signatures contains no information about the private basis [25].

In 2011, Stehlé and Steinfeld [47] introduced a variant of NTRUEncrypt and demonstrated that it is provably secure based on the hardness of certain lattice problems in a natural class of ideal lattices, corresponding to the ideals in the ring $\mathcal{R} = \mathbb{Z}[x]/(x^n + 1)$ where n is a power of two. The security proof includes a worst-to-average-case reduction, which means the scheme is secure as long as the underlying lattice problems are hard in the worst case. Stehlé and Steinfeld generate the polynomials the secret key consists of via rejection sampling from a discrete Gaussian distribution over $\mathcal{R}$ because, in this way, the corresponding public key becomes statistically close to uniform. However, these modifications along with the fact that, according to [10], a ring of degree $n = 2048$ is needed for 144-bit security[1], make the Stehlé-Steinfeld variant significantly slower and also

[1]The Stehlé-Steinfeld variant requires n to be a power of two. For comparison, in the classical NTRU scheme, a ring of degree $N = 439$ suffices for 128-bit security.

larger (in terms of key and ciphertext size, respectively) than the classical NTRU scheme; see e.g. [10] for concrete results. Recently, another tweak of the NTRU cryptosystem, called NTRU Prime [7], was introduced that operates in a field of the form $(\mathbb{Z}/q\mathbb{Z})[x]/(x^p - x - 1)$ where p and q are prime. Such fields have less structure than conventional NTRU rings, which, as argued in [7], can help to reduce the number of potential attack avenues.

Besides security aspects, also the efficiency of NTRU in hard- and software has been very actively researched in recent years. The relevant literature covers a wide spectrum ranging from high-speed software implementation for graphics processing units [22] over optimizations for 32-bit ARM processors [4] down to low-power VLSI designs for sensor nodes and RFID tags [2, 18]. Moreover, the vulnerability against timing analysis [46], simple and differential power analysis [36], as well as fault attacks [35] has been studied and several countermeasures have been proposed. However, to the best of our knowledge, the literature contains only three papers on efficient NTRU implementation for 8-bit processors (namely [9, 15, 38]), which is somewhat surprising since NTRU is known to be "light-weight" and well suited for resource-restricted environments. In all three papers an 8-bit AVR microcontroller [3] was used as target platform. Driessen et al. [15] introduced an optimized implementation of the NTRUSign signature scheme and evaluated its performance at a security level of 80 bits. Unfortunately, NTRUSign is nowadays considered completely broken [40, 16]. Monteverde [38] implemented NTRUEncrypt in ANSI C99 and simulated the execution time of key generation, encryption and decryption with two sets of parameters, one for "moderate" security and the other for "standard" security. However, these parameters stem from a time when the real impact of hybrid attacks [31] was underestimated, which means the parameter sets used by Monteverde provide less security than originally expected. Finally, Boorghany et al. [9] developed NTRU software for ARM and AVR processors as part of an effort to evaluate lattice-based authentication protocols. They used the latest parameters for the 128-bit security level and reported that, on an ATmega64 processor with 4 kB of RAM, NTRU encryption with secure padding is about 1.5 times faster than ring-LWE encryption, while the decryption times are similar. The source code of both NTRU and ring-LWE was written in C and does not contain any hand-crafted Assembly fragments for performance-critical operations. It should also be mentioned that neither Boorghany et al. nor Monteverde [38] attempted to protect their implementation against side-channel attacks.

1.1 Motivation

The current status of research on efficient NTRU implementation for resource-restricted microcontrollers leaves a number of questions open. In particular, the existing literature does not disclose how one can optimize NTRU's polynomial arithmetic at the Assembly level to reach peak performance[2]. Furthermore, the impact of countermeasures against side-channel attacks such as timing attacks has not been adequately analyzed in the literature. Filling these white spots in the research landscape is important for two reasons; the first is related to the ever-increasing proliferation of resource-constrained devices, while the second is due to the growing demand for quantum-secure public-key cryptography.

The Internet of Things (IoT) [48] represents the next phase of the evolution of the Internet into a network that is able to integrate the physical world into the virtual world. In the near future, the IoT is expected to encompass billions of sensors, actuators, and

[2]Driessen et al's multiplication of truncated polynomials for NTRUSign is written in Assembly language, but they adopted Karatsuba's technique, while we exploit the product form of sparse ternary polynomials described in [30] in our work.

numerous other "smart" devices, many of which are battery-powered and, therefore, highly constrained in computational resources (e.g. wireless sensor nodes). RSA and Elliptic Curve Cryptography (ECC) pose a heavy burden on the scarce resources of such IoT devices, especially in terms of energy consumption. A carefully-optimized implementation of NTRU could make public-key cryptography feasible on devices on which RSA or ECC is too battery-demanding. Therefore, progress in the area of efficient implementation of NTRU (and other efficient lattice-based cryptosystems) for 8-bit processors carries the potential to advance the whole field of IoT security.

In recent years, the interest in post-quantum cryptography based on lattice problems has been steadily growing. NTRU has often served as a benchmark to assess the practicality and efficiency of new lattice-based cryptosystems [10]. In general, a new proposal is deemed "efficient" if it is competitive with NTRU in terms of execution time and certain other criteria such as the size of keys and ciphertexts (resp. signatures). Unfortunately, real-world NTRU benchmarks do (currently) not exist for 8-bit processors due to the lack of optimized software implementations using state-of-the-art parameters. Filling this gap can make a valuable contribution to standardization activities in the area of post-quantum cryptography such as the one initiated by the NIST in February 2016.

1.2 Contributions

We describe a carefully-optimized implementation of the ring (i.e. polynomial) arithmetic operations of NTRUEncrypt for 8-bit AVR processors that achieves record-setting execution times. Our software avoids any form of key-dependent control flow (e.g. conditional branches) and is, hence, resistant to timing-based side-channel attacks. Yet, our polynomial arithmetic is significantly faster than that of existing NTRU implementations and outperforms the "core" arithmetic of all other lattice-based encryption schemes on 8-bit processors reported in the literature (e.g. [37, 42]). Achieving high speed *and* high security (i.e. resistance against timing attacks) is everything else than trivial and required us to devise some sophisticated optimization techniques, which we will describe in detail in the next sections. The research contribution of this paper is threefold and can be summarized as follows.

- We introduce a new "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 consisting of very few non-zero 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 [21] and aims to reduce the number of load and store instructions compared to the standard product-scanning approach. We describe the application of our hybrid technique to a special convolution algorithm that requires the ternary polynomial to be in *product form* [30] (i.e. have the form $a(x) = a_1(x) \star a_2(x) + a_3(x)$ where $a_1(x)$, $a_2(x)$, $a_3(x)$ are sparse) and present an optimized constant-time implementation in AVR assembly language.

- We discuss software optimization techniques for some of the "auxiliary" functions used in NTRUEncrypt, most notably the so-called Blinding Polynomial Generation Method (BPGM) and Mask Generation Function (MGF) [12]. Both are based on a hash function (e.g. SHA-256 [39]) and impact not only the performance but also the security (including resistance to timing attacks) of NTRUEncrypt. We describe an assembler implementation of the compression function of SHA-256 that comes

with optimizations similar to those of Cheng et al. for SHA-512 [11] and achieves a good trade-off between execution time and (binary) code size.

- We provide a detailed analysis of the execution time, RAM consumption, and code size of AvrNtru for two security levels, using an 8-bit ATmega1281 [3] microcontroller as evaluation platform. These benchmarking results were collected with the product-form parameter sets `ees443ep1` and `ees743ep1` [12], which target 128 and 256 bits of (pre-quantum) security, respectively. We assess the contribution of the convolution, MGF, and BPGF to the overall execution time of NTRUEncrypt and give new insights into their relative cost when they are implemented to withstand timing attacks. Finally, we compare the results of AvrNtru with that of previous implementations of NTRU variants, other lattice-based and classic cryptosystems.

2 A Brief Review of NTRUEncrypt

In this section we first set some basic notation and then give a brief description of the key generation, encryption, and decryption operations.

Notation and Parameters

Before NTRUEncrypt or any other NTRU-based cryptosystem can actually be used, the two involved parties need to agree on a common set of domain parameters. Such domain parameters always include the triple (N, p, q), which defines the underlying algebraic structures [26]. N is the so-called *degree parameter* and typically taken to be a prime between 400 and 800. Currently, $N = 439$ is recommended to achieve a security level of 128 bits [12], whereas in the past $N = 397$ was deemed sufficient for this level [33]. The degree parameter determines the quotient ring $\mathcal{R} = \mathbb{Z}[x]/(x^N - 1)$ in which all arithmetic operations take place [27, 24]. Every element of $\mathcal{R}$ has a unique representative in the form of a polynomial in x of degree $N - 1$ with coefficients from $\mathbb{Z}$. 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[3], which can be performed very efficiently compared to the multiplication in more general polynomial quotient rings (see Sect. 3 for details on implementation aspects).

The parameters p and q are called *small modulus* and *large modulus*, respectively [24]. Common choices for q are a power of two, which is mainly motivated by the need for fast arithmetic modulo q. The most recent parameters sets in [12] use $q = 2^{11} = 2048$ across all security levels, but in the past smaller values like $2^7 = 128$ (for 80-bit security) or $2^8 = 256$ (for higher security levels) were recommended[4] [4]. On the other hand, the small modulus p is either a positive integer or alternatively a polynomial and must be relatively prime to q in the ring $\mathcal{R}$. When both p and q are integers, this requirement translates into $\gcd(p, q) = 1$. Small values of p (in relation to q) decrease the probability of decryption failures and reduce the storage requirements for the private key [26]. The parameters given in [12] fix the value of p to 3; alternative choices used in the past include $p = 2$ (which requires q to be prime [33]) and the polynomial $p = x + 2$ [30].

[3] Thus, 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.

[4] Such updates of parameter sets were necessary due to the discovery of new attacks [31]. Unfortunately, these differences in parameterization, especially with respect to N and q, make it very difficult to compare our results with that of past work.

As we will see below, the multiplications in $\mathcal{R}$ executed during encryption and decryption involve a reduction of all coefficients modulo q so that the final result is an element of the quotient ring $\mathcal{R}_q = (\mathbb{Z}/q\mathbb{Z})[x]/(x^N - 1)$. This ring can naturally be identified with a subset of the ring $\mathcal{R}$ and any polynomial $a(x) \in \mathcal{R}$ can simply be "reduced" to become an element of $\mathcal{R}_q$ by reducing all its N coefficients modulo q. The equivalent 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 $a'(x) \in \mathcal{R}$ satisfying $a'(x) \bmod q = a(x)$ whose coefficients a'_i lie in $[-q/2, q/2 - 1]$. Public keys in NTRUEncrypt are elements of $\mathcal{R}_q$, while private keys are "small" polynomials (i.e. polynomials with coefficients of roughly the same magnitude as p). In this paper, we only consider private keys of the form $f(x) = 1 + pF(x)$ [30], where $F(x)$ is a *ternary polynomial* with coefficients in $\{-1, 0, 1\}$. We define $\mathcal{T}$ as the set of all such ternary polynomials having a degree of $N - 1$. Furthermore, we define $\mathcal{T}(d_1, d_2)$ for positive integers d_1 and d_2 as the 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 [29]. Plaintexts (after padding and formatting) are represented as elements of $\mathcal{T}$, whereas ciphertexts are elements of $\mathcal{R}_q$.

The literature mentions a few additional parameters, including d_f, d_g, and d_r, which specify the number of ± 1 coefficients of ternary polynomials, f, g, r respectively, which will be generated randomly during the key generation and encryption, see below. To simplify matters in this paper, we take $d = d_f = d_g = d_r$ and we call d the *weight parameter*. The parameter sets given in [12] fix the weight parameter to $d = \lfloor N/3 \rceil$ in order to maximize the size of the key space [26].

Key Generation

The key generation procedure gets a quadruple of the form (N, p, q, d) that meets all guidelines mentioned above as input. Our description of the key generation given below is mainly based on [44, Sect. 3.1]. In order to obtain a key pair for NTRUEncrypt, the following steps shall be carried out.

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

As already stated before, we only consider private keys of the special form $f(x) = 1 + pF(x)$ where $F(x)$ is a ternary polynomial in this paper. Such keys were first suggested in [30] and allow for a number of optimizations, which are not possible when $f(x)$ is a small polynomial of more general form. In our case (i.e. $p = 3$), the coefficients f_i of $f(x)$ are in $\{-3, 0, 3\}$ for $1 \leq i \leq N - 1$, while $f_0 \in \{-2, 1, 4\}$, and consequently $f(x) \equiv 1 \bmod p$. However, an arbitrary polynomial does not have this property and requires the key generation function to be implemented as described in the first NTRU paper from 1998

[27]. In this original NTRU proposal, $f(x)$ is a ring element that is invertible modulo q and modulo p. The key generation involves two inversions of $f(x)$, one modulo q as specified above (Step 3) and one modulo p, whereby the result of the latter is then used as operand of a convolution in the decryption. This extra inversion and convolution has a significant impact on the performance of key generation and decryption, respectively [30]. Fortunately, it is possible to avoid both when $f(x) \equiv 1 \bmod p$ since, in this case, the inverse of $f(x)$ modulo p is simply 1.

The inverse of $f(x)$ modulo q (Step 3) can be found using a variant of the Euclidean algorithm, provided it exists. However, when the domain parameters are properly chosen (such as those in [26]), the probability that both $f(x)$ and $g(x)$ are invertible modulo q is very high; consequently, the steps 1 to 5 will be executed only once in most runs of the key generation function. Note that the polynomial $g(x)$ is intentionally chosen from $\mathcal{T}(d+1, d)$ and not $\mathcal{T}(d, d)$ since the elements of the latter set never have inverses in $\mathcal{R}_q$ [29]. The multiplication in Step 6 represents the computation of a *convolution product*, which consists of a polynomial multiplication modulo $x^N - 1$ (yielding a polynomial of degree $N-1$ as result), followed by a reduction of the coefficients modulo q. We use the star symbol $\star$ for this operation to distinguish it from a conventional polynomial multiplication that returns a product of degree $2N-2$.

Encryption

The encryption function takes as input the domain parameters, described above, the message M to be encrypted and the public key $h(x)$ of the receiving party as input. Like any other public-key encryption scheme, NTRUEncrypt requires the message to be properly padded and formatted to prevent certain attacks. The standardized NTRUEncrypt scheme in [12] defines several supporting functions to transfer a message M into a ternary polynomial $m(x) \in \mathcal{T}$ that can serve as plaintext in the encryption operation outlined below. We will not explain these functions further since this paper focuses on efficient polynomial arithmetic. In order to obtain the ciphertext $c(x)$ of a message M to be encrypted under the public key $h(x)$, the following steps shall be carried out.

1. Represent the message M as a ternary polynomial $m(x) \in \mathcal{T}$.
2. Generate a (pseudo)random blinding polynomial $r(x) \in \mathcal{T}(d, d)$.
3. Compute $c(x) = p\,h(x) \star r(x) + m(x) \bmod q$.
4. Output $c(x)$ as ciphertext.

Note that, in practice, the polynomial $m(x)$ representing the (padded and formatted) plaintext needs to have a certain minimal number coefficients equal to 1, -1, and 0 (see e.g. [26] for more details). The encryption is a randomized process because it involves a so-called *blinding polynomial* $r(x)$ that is chosen uniformly at random from the set $\mathcal{T}(d, d)$.[5] In terms of arithmetic operations (Step 3), the encryption mainly consists of the computation of the convolution product of $h(x)$ and $r(x)$, followed by a multiplication of all N coefficients by $p = 3$. Then, the plaintext polynomial $m(x)$ is added to the product and the coefficients are reduced modulo q. The ciphertext $c(x)$ is an element of $\mathcal{R}_q$.

[5] The standardized NTRUEncrypt scheme [12] generates $r(x)$ by hashing the message together with an object-ID, a random salt, and a part of the public key $h(x)$.

Decryption

The receiving party needs the private key $f(x)$ corresponding to the public key used for encryption to decrypt the ciphertext $c(x)$. In order to recover the message M, the following steps shall be carried out.

1. Compute $a(x) = c(x) \star f(x) \bmod q = p\,c(x) \star F(x) + c(x) \bmod q$.
2. Compute $a'(x) = \text{center-lift}(a(x))$.
3. Compute $b(x) = a'(x) \bmod p$.
4. Compute $m(x) = \text{center-lift}(b(x))$.
5. Convert $m(x)$ to message M and output it.

The decryption process is deterministic, but depending on the parameterization, it can happen that the obtained plaintext is not completely correct (we will discuss decryption failures in more detail below). Step 1 contains the main arithmetic operation of decryption, namely the computation of the convolution product of the ciphertext $c(x)$ and the private key $f(x)$. Substitution of $c(x)$ by $p\,h(x) \star r(x) + m(x) \equiv p\,f(x)^{-1} \star g(x) \star r(x) + m(x) \bmod q$ combined with the fact that $f(x)^{-1} \star f(x) \equiv 1 \bmod q$ leads to the following equation.

$$a(x) \equiv c(x) \star f(x) \equiv p\,g(x) \star r(x) + m(x) \star f(x) \bmod q \tag{1}$$

In order to explain why (and how) the decryption works, let us first ignore the reduction modulo q and assume Equation (1) is an exact equality in $\mathcal{R}$ instead of a congruence relation. Under this assumption, it is fairly straightforward to see that $a(x) \equiv m(x) \bmod p$ because all coefficients of the term $p\,g(x) \star r(x)$ are multiples of p, which means they vanish modulo p, and $f(x) \equiv 1 \bmod p$ due to the special form of $f(x)$. Now we have to analyze under which conditions the assumption made before is true so that Equation (1) holds in $\mathcal{R}$ and not just in $\mathcal{R}_q$. For this purpose, it is very important to recall that all four involved polynomials have small coefficients. Concretely, the coefficients of $g(x)$, $m(x)$, and $r(x)$ lie in the interval $[-1, 1]$ and those of $f(x)$ in $[-3, 4]$. When both $g(x)$ and $r(x)$ have d coefficients equal to 1 and also d coefficients equal to -1, then the largest possible coefficient of the convolution product $g(x) \star r(x)$ is $2d$ [29]. The maximum value of the coefficients of $m(x) \star f(x)$ is very similar (see [29] for an in-depth explanation). For well-chosen parameters, there is a very high probability (or even certainty) that all coefficients of $p\,g(x) \star r(x) + m(x) \star f(x)$ lie in the interval $[-q/2, q/2 - 1]$ [27]. If this is the case, a reduction modulo q and subsequent center-lift (Step 2) does not change the actual value of any of these coefficients, which means the decryption process will succeed in recovering the plaintext polynomial $m(x)$ by reducing $a'(x)$ modulo p as described above. The final step is a center-lift to obtain $m(x)$ with coefficients in $\{-1, 0, 1\}$.

A decryption failure occurs when a validly encrypted plaintext is not recovered correctly, which is extremely unlikely with well-chosen parameters. Before going into detail, let us recap that, in order for the decryption to succeed, the center-lift of the convolution product $a(x) = c(x) \star f(x) \bmod q$ needs to equal $p\,g(x) \star r(x) + m(x) \star f(x)$ exactly and not just modulo q. As noted above, this is the case when all coefficients of $p\,g(x) \star r(x) + m(x) \star f(x)$ lie in the interval $[-q/2, q/2 - 1]$. On the other hand, if one or more coefficients are outside this interval, the decryption fails as these coefficients can only be recovered modulo q and not with their original magnitude. This is because the out-of-range coordinates of $p\,g(x) \star r(x) + m(x) \star f(x)$ appear "wrapped around" modulo

q in the polynomial $a'(x)$ obtained in Step 2, whereas all coordinates of magnitude less than $q/2$ appear unchanged in $a'(x)$. Any wrap error in $a'(x)$ propagates to the polynomial $m(x)$ computed in Step 4, which means the affected coefficients do not match with those of the original plaintext polynomial used for encryption since $\gcd(q, p) = 1$. More concretely, wrapped coefficients are off by a multiple of q modulo p (see [32] for a detailed discussion).

Decryption failures threaten the security of NTRUEncrypt because they can leak information about the secret key $f(x)$ to an attacker [32]. However, when using state-of-the-art parameter sets, the probability of a decryption failure is vanishingly low. For example, the parameter set `ees439ep1` from [12], which is assumed to provide a security level of (at least) 128 bits, features a decryption failure rate of 2^{-195} [26], i.e. one out of 2^{195} ciphertexts does not get decrypted correctly. On the other hand, the parameter set `ees743ep1` (targeting 256 bits of security) comes with a slightly higher probability for a decryption failure to occur, namely 2^{-112} [26]. However, it is possible to entirely prevent decryption failures by choosing the parameter set to satisfy $q > (6d + 1)p$ [29]. Indeed, as confirmed in [44, p. 43], decryption never fails with the `ees743ep1` parameters when q is increased from 2048 to 4096. Unfortunately, increasing q reduces the resistance of NTRUEncrypt against lattice reduction attacks and increases the size of ciphertexts and public keys. This explains why the parameter sets given in [12] aim for a compromise: instead of completely eliminating decryption failures, the parameters were generated to have a non-zero (but very small) failure probability in order to make them more "implementation-friendly." In the case of `ees439ep1`, an attacker would need to carry out 2^{195} encryptions before she can expect to find a message $m(x)$ and blinding polynomial $r(x)$ that triggers a decryption failure, which is much more costly than other forms of attack. The situation is different for the `ees743ep1` parameter set since the 2^{112} encryption operations needed to find a pair $m(x), r(x)$ that gets decrypted incorrectly are clearly below the (nominal) cost of breaking a cryptosystem aimed at 256 bits of security. Fortunately, the standardized version of NTRUEncrypt [12] allows the decrypting party to recover $r(x)$ from $c(x)$ and re-compute the ciphertext as in Step 3 of the encryption operation[6], which makes it possible to detect a failure by simply comparing the re-computed ciphertext with the received one.

3 Implementation of the Polynomial Arithmetic

As mentioned in the last section, the computation of a convolution product of elements of the quotient ring $\mathcal{R} = \mathbb{Z}[x]/(x^N - 1)$ differs from a conventional multiplication of two polynomials in that it involves a reduction of the product modulo $x^N - 1$ [27, 29]. Since $x^N \equiv 1 \bmod x^N - 1$, this reduction is relatively cheap and just consists of a substitution of all powers x^{k+N} with $0 \leq k < N - 1$ by x^k, which means the exponents of the powers of x have to be reduced modulo N. In other words, the higher $N - 1$ coefficients of the product are additively "wrapped" into the lower $N - 1$ coefficients. Figure 1 depicts the computation of the convolution product $p(x) = u(x) \star v(x)$ for $N = 5$, i.e. the operands $u(x) = u_4x^4 + \cdots u_1x + u_0$ and $v(x) = v_4x^4 + \cdots + v_1x + v_0$ have degree $N - 1 = 4$.

To discuss the convolution $p(x) = u(x) \star v(x)$ in more general form, let $u(x)$, $v(x)$ be elements of $\mathcal{R} = \mathbb{Z}[x]/(x^N - 1)$ represented by polynomials of degree $N - 1$, i.e. we have

[6] In the standardized NTRUEncrypt scheme, the plaintext polynomial $m(x)$ consists of the actual message and a random padding $b(x)$, called *salt* in [12]. The blinding polynomial $r(x)$ is generated by hashing $m(x)$ together with $b(x)$, an object-ID, and a part of the decrypting party's public key $h(x)$. Upon receipt of the ciphertext, the decrypting party recovers the plaintext $m(x)$ and extracts $b(x)$, which allows her to compute $r(x)$ in exactly the same way as the encrypting party did.

u_4	u_3	u_2	u_1	u_0
v_4	v_3	v_2	v_1	v_0
u_4v_0	u_3v_0	u_2v_0	u_1v_0	u_0v_0
u_3v_1	u_2v_1	u_1v_1	u_0v_1	u_4v_1
u_2v_2	u_1v_2	u_0v_2	u_4v_2	u_3v_2
u_1v_3	u_0v_3	u_4v_3	u_3v_3	u_2v_3
u_0v_4	u_4v_4	u_3v_4	u_2v_4	u_1v_4
p_4	p_3	p_2	p_1	p_0

Figure 1: Convolution $p(x) = u(x) \star v(x)$ in the ring $\mathcal{R} = \mathbb{Z}[x]/(x^5 - 1)$.

$u(x) = u_{N-1}x^{N-1} + \cdots + u_1x + u_0$ and $v(x) = v_{N-1}x^{N-1} + \cdots + v_1x + v_0$. Then, the convolution product $p(x) = u(x) \star v(x) = u(x)v(x) \bmod x^N - 1$ is given as

$$
\begin{aligned}
p(x) \;=\; u(x)v(x) \bmod x^N - 1 \;&=\; \left(\sum_{i=0}^{N-1} u_i x^i\right)\left(\sum_{j=0}^{N-1} v_j x^j\right) \bmod x^N - 1 \\
&= \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} u_i v_j x^{i+j} \bmod x^N - 1 \;=\; \sum_{k=0}^{2N-2}\left(\sum_{i+j=k} u_i v_j\right)x^k \bmod x^N - 1 \\
&= \sum_{k=0}^{N-1}\left(\sum_{i+j=k} u_i v_j\right)x^k + \sum_{k=N}^{2N-2}\left(\sum_{i+j=k} u_i v_j\right)x^k \bmod x^N - 1 \\
&= \sum_{k=0}^{N-1}\left(\sum_{i+j=k} u_i v_j\right)x^k + \sum_{k=0}^{N-2}\left(\sum_{i+j=k+N} u_i v_j\right)x^{k+N} \bmod x^N - 1 \\
&= \sum_{k=0}^{N-1}\left(\sum_{i+j=k} u_i v_j\right)x^k + \sum_{k=0}^{N-2}\left(\sum_{i+j=k+N} u_i v_j\right)x^k \;=\; \sum_{k=0}^{N-1}\left(\sum_{i+j\equiv k \bmod N} u_i v_j\right)x^k \\
&= \sum_{k=0}^{N-1} p_k x^k \quad \text{with} \quad p_k \;=\; \sum_{i+j\equiv k \bmod N} u_i v_j \qquad (2)
\end{aligned}
$$

Each coefficient p_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 p_k in Eq. (2) can also be expressed in the following way.

$$
p_k \;=\; \sum_{i+j\equiv k \bmod N} u_i v_j \;=\; \sum_{i=0}^{N-1} u_i v_{(k-i \bmod N)} \;=\; \sum_{j=0}^{N-1} u_{(k-j \bmod N)} v_j \qquad (3)
$$

Equation (3) makes it clear that the computation of p_k consists of the addition of N coefficient-products of the form u_iv_j. Consequently, a straightforward algorithm for convolution in the ring $\mathcal{R} = \mathbb{Z}[x]/(x^N - 1)$ has a complexity of $\mathcal{O}(N^2)$, exactly as the conventional multiplication of two polynomials of degree $N - 1$ using e.g. the operand-scanning or product-scanning technique. Advanced multiplication techniques like Karatsuba's algorithm (which can be applied to convolution as well [14, 34]) allow one to reduce the complexity to $\mathcal{O}(N \log N)$. Also the multiplication technique for sparse product-form polynomials we describe below has linearithmetic complexity and is very well suited for implementation on small microcontrollers.

From an arithmetic point of view, the most time-consuming operation of NTRU encryption is the computation of the convolution product $h(x) \star r(x) \bmod q$, whereby the coefficient of $h(x)$ are randomly distributed modulo q and $r(x)$ is a ternary polynomial with a certain minimum number of non-zero coefficients (specified by the weight parameter d_r). The decrypting party has to compute two polynomial convolutions, namely $c(x) \star f(x) \bmod q$ to obtain the message and, thereafter, $h(x) \star r(x) \bmod q$ to ensure that the message has been correctly recovered (i.e. that the decryption did not fail). As explained in the previous section, the polynomial $c(x)$, which represents the ciphertext, 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. Hence, the convolutions carried out in both encryption and decryption are simply multiplications of a polynomial with coefficients in the range of $[0, q-1]$ by a ternary polynomial, both of degree up to $N-1$. The fact that one of the operands is a ternary polynomial implies that the computation of the convolution product essentially boils down to additions and subtractions of coefficients modulo q. Consequently, only `add` and `sub` instructions need to be executed, both of which have normally a latency of a single clock cycle, even on a small 8-bit microcontroller. This is a significant advantage of NTRU over other lattice-based cryptosystems whose core arithmetic operations are Number-Theoretic Transforms (NTTs), most notably cryptosystems based on the ring-variant of the Learning With Errors (LWE) problem. The computation of an NTT (and inverse NTT) involves multiplications of coefficients, which requires the execution of `mul` instructions. On most embedded platforms, the `mul` instruction does not execute in a single cycle but has a latency of several clock cycles and also consumes much more power (and, hence, much more energy) than a simple single-cycle `add` or `sub` instructions.

The computational cost of the convolution product amounts to dN additions (resp. subtractions) of coefficients bounded by q, whereby d is the number of non-zero coefficients of the ternary polynomial $r(x)$. To maximize efficiency, it is tempting to make $r(x)$ as sparse as possible. However, as explained in the previous section, the number of non-zero coefficients of $r(x)$ must not be below a certain limit (specified by the weight parameter d_r) as otherwise the search space for $r(x)$ would become too small to guarantee the desired level of security. Nonetheless, it is possible to significantly reduce the computational cost without compromising security by taking $r(x)$ to be a *product of polynomials with few non-zero coefficients* as originally proposed in [30]. In this case we can write $r(x) = r_1(x)r_2(x)$, where r_1 and r_2 are ternary polynomials with d_1 and d_2 non-zero coefficients, respectively. Then $r(x)$ will have approximately $d_1 d_2$ non-zero coefficients. In practice, $r(x)$ will have a few coefficients outside the range $[-1, 0, 1]$, but that will not affect matters very much. It is important to notice that the computation of the product

$$h(x) \star r(x) = (h(x) \star r_1(x)) \star r_2(x) \tag{4}$$

requires only $(d_1+d_2)N$ coefficient additions/subtractions, which means the computational complexity is proportional to the sum of d_1 and d_2. On the other hand, the search space for the pair of polynomials (r_1, r_2) is proportional to the product of the r_1 search space and the r_2 search space (see [30] for a more detailed treatment). In summary, using a 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 $d_1 d_2$. A similar optimization is possible for the ternary polynomial $F(x)$ of the private key $f(x) = 1 + pF(x)$. For example, one might take $F(x)$ to have the form $F(x) = f_1(x) \star f_2(x) + f_3(x)$ where f_1, f_2, and f_3 are sparse polynomials as suggested in [30]. In this case, the private key $f(x)$ becomes

$$f(x) = 1 + p\left(f_1(x) \star f_2(x) + f_3(x)\right). \tag{5}$$

When f_1, f_2, and f_3 are sparse ternary polynomials then the convolution of $c(x)$ by $f(x)$ can be optimized to reach very high speed in software, even on small microcontrollers. Furthermore, when the target platform does not have a data cache (which is actually the case for virtually all 8 and 16-bit microcontrollers and also for many 32-bit models, e.g. most members of the ARM Cortex-M series), it is possible to implement the convolution to have constant execution time, which means the execution time only depends on the number of non-zero coefficients of $f(x)$ but not the coefficients themselves. Our software implementation represents the ciphertext-polynomial $c(x)$ as an array of words of type `uint16_t`, similar to [4]. 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-zero elements. This representation of f_1 has two advantages, namely (i) it is easy to load the corresponding coefficients of $c(x)$ by simply adding the offset to the start address of the array in which $c(x)$ is stored, and (ii) the polynomial f_1 does not consume much space in RAM since we only need to consider the non-zero coefficients.

When using a straightforward multiplication technique for polynomials then the sparse nature of $f_1(x)$ causes most of partial products to be zero. So rather than employing a traditional polynomial multiplication algorithm that wastes a lot of time by computing zero terms, we scan the array $F1$ containing the offsets of the non-zero coefficients of f_1 and calculate only those partial-product terms which are be non-zero. A particular non-zero coefficient will appear in N partial-product terms. Our optimized multiplication algorithm begins by zero-initializing an array of coefficients that will hold the product $p(x)$. To aid the explanation, let us assume that array `A` contains N coefficients in the range $[0, q-1]$ and array `B` contains the indices j of the coefficients of a trinary polynomial $b(x)$ that are $+1$. To get the result $p(x) = p(x) + a(x)b(x) \bmod x^N - 1$, we have to calculate for each coefficient P[k] of p(x) the sum of all coefficient products of the form A[i]*B[j] for which i+j is congruent to k mod N. Taking the least-significant coefficient P[0] as example, we have to sum up all coefficients A[N-j], except when j = 0 we have to use A[0] instead of A[N]. This can be performed with a simple loop that calculates for each index j in array B the address of the corresponding coefficient A[N-j] and stores it in array B (i.e. the index j is replaced by the address of A[N-j]). However, when array B contains index j = 0, we store the address of A[0] instead of the address of A[N].

Since the coefficients of polynomial b(x) are only 0 or 1, the polynomial multiplication $r(X) = r(X) + a(X) * b(X) \bmod X^N - 1$ boils down to the addition of coefficients, which is done in a nested loop. In each iteration of the outer loop, we load eight coefficients P[k] from p(x) and add the corresponding coefficients from a(x), starting with the least significant coefficient P[0]. For P[0], we just have to add up all the coefficients A[N-j] for which we have already computed the addresses in the first loop above; these addresses are stored in array B. However, we do this via an "operand-scanning" approach, i.e. in the first iteration of the inner loop, we add A[N-j] to P[0], A[N-j+1] to P[1], and so forth, until we finally add A[N-j+7] to P[7]. We access array A through the 16-bit X pointer, i.e. at the beginning of the inner loop, we load the current element of array B to the (XH:XL) registers. The coefficients A[N-j], A[N-j+1], .., A[N-j+7] can be loaded very efficiently from array A thanks to the automatic post-increment addressing mode of AVR. At the end of the inner loop, we write the address contained in the (XH:XL) register pair back to array B from where we loaded it. However, since X got incremented by 2 with every loading of a coefficient, it may happen that X exceeds the address of A[N-1], in which case we have to subtract 2N from X so that X points to an element of array A between A[0] and A[N-1].

4 Implementation of the Auxiliary Functions

4.1 Data Types and Conversions

Three data types are used in the standardized NTRUEncrypt in order to be well suited for the various arithmetic operations and auxiliary functions: (i) `octet string` is an ordered array of bytes such as the seed of a hash function and the transmitted data, that is, plaintext strings, public key strings, etc; (ii) `ring element` is an element of $\mathcal{R}_q$ described in Sect. 2; (iii) `ternary ring element` is a ternary polynomial $a(x) \in \mathcal{T}$ whose coefficients are all in the set $\{-1, 0, 1\}$ (-1 is represented as 2 in practice). The way to store both `ring element` and `ternary ring element` is by using an array consisting of their ordered coefficients, except for the sparse ternary polynomials such as the blinding polynomial $r(x)$ and private key $F(x)$. A sparse polynomial is a special `ternary ring element` where the indices are stored instead of each coefficient.

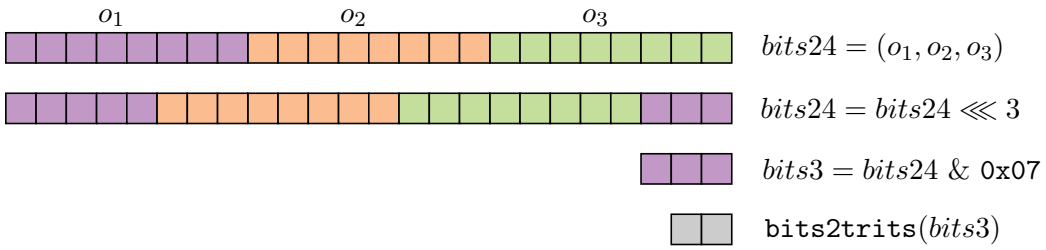

Figure 2: Single iteration of nested loop for converting three octets to sixteen trits

Accordingly, there are some conversions between two different data types during the encryption and decryption, and our implementations for these conversions focus on less code size and faster speed and meanwhile take constant-time executions into account. For instance, the conversion from `octet string` to `ternary ring element` transforms every three-bit of the octet string to two-trit until meeting the required number of trits. This conversion logically divides the whole octet string into every three octets, because three-octet (i.e. 24 bits) properly generate 16 trits. The full conversion is made up of many sub-conversions, where each sub-conversion works for converting one three-octet group. In each sub-conversion, we apply a nested loop with the same left rotation to replace eight different right shifts. As shown in Figure 2, at the beginning of sub-conversion, we first left-rotate $bits24$ by 3 bits and then do an `AND` operation between $bits24$ with `0x07` instead of directly right-shifting $bits24$ by 21 bits to obtain $bits3$. The advantage is that for getting the remaining seven $bits3$ we just repeat the same operations rather than to right-shift $bits24$ by different quantities i.e. 18 bits, 15 bits, ... 3 bits, respectively. Apart from that, we create a look-up table in flash ROM for rapidly converting three bits to two trits, i.e. `bits2trits` method in Figure 2.

In addition, Figure 3 illustrates our optimized conversion from `ring element` to `octet string`. The large modulus q is 2048 in all the supported parameter sets, so each ring element occupies 11 bits. This conversion could be intrinsically regarded as removing the unused bit space and compacting the actual memory usage. We show the transformation of the first three `ring elements` as an example in Figure 3, where we develop an optimized right rotation method to simplify operations for a single ring element. The rotation quantity i initialed as three and typically increases three in each iteration, while once it is more than eight, it will be subtracted eight at the next iteration. Meanwhile, the same ring element will be operated again, e.g. the third ring element is operated twice in

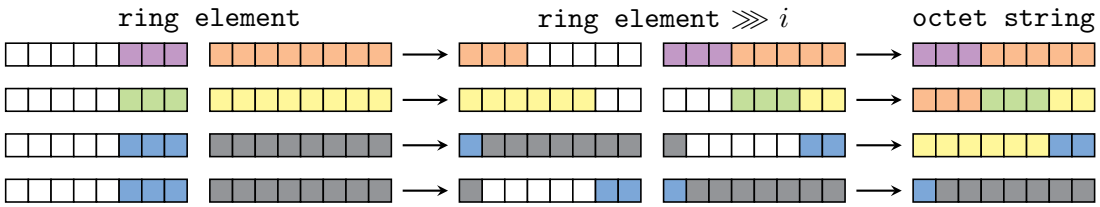

Figure 3: Convert first three ring elements in the conversion from `ring element` to `octet string`

lines 3 and 4. In each iteration, the method generates one valid octet by an `OR` operation between the lower byte of the rotated ring element and the remaining bits, and stores the new remaining bits i.e. higher byte of the rotated ring element for the next iteration. Unfortunately, sometimes not all the bits of the lower byte of rotated ring element are valid, e.g. the most significant bit of the lower byte of the rotated ring element in the third line. To block these invalid bits during the conversion, we utilize the related mask values and store mask values in a look-up table in advance.

Furthermore, there is no extra code necessary for converting the last incomplete block in some of the Assembler conversions. Instead, we reuse the Assembler code of dealing with standard blocks by setting the flag to distinguish these two different cases. Moreover, the execution time of these conversions only depends on the related public parameters of the parameter set and is independent of any sensitive information, i.e. plaintext and private key, etc.

4.2 Hash Function

Hash functions play the core role in the Index Generation Function (IGF) and Mask Generation Function (MGF). In the chosen parameter set `ees443ep1` and `ees743ep1`, SHA-256 is recommended to provide the desired security level [12]. Our highly-optimized SHA-256 implementation for 8-bit AVR adopts the optimization techniques which are originally designed for SHA-512 described in [11]. Similarly, we developed an AVR assembler compression function plus the other components written in C language. Aiming at the 32-bit operand instead of the 64-bit operand, we reasonably modify the optimization techniques in [11], whereby they are correctly employed in SHA-256. Two optimization strategies for the compression function of SHA-256 are utilized here. The first one is that inside four sigma operations, we minimize the overall number of bitwise rotations and "merge" the bytewise rotations with XORs. The other one is that we "duplicate" the eight working variables and intensively use the indirect addressing mode with displacement of the AVR architecture, which helps us avoid the word-wise rotation in each iteration. Both of these two strategies contribute to accelerate the SHA-256, and the latter technique also efficiently reduce the code size. Furthermore, we carefully developed the padding process to make our SHA-256 implementation have a constant running time to hash the same numbers of the 64-byte blocks.

4.3 Index Generation Function (IGF)

NTRUEncrypt makes use of the Index Generation Function IGF-2 to generate the indices for a sparse ternary polynomial in $\mathcal{T}(d, d)$. IGF-2 is a function for uniformly sampling a fixed size subset of $[0, N-1]$, and Algorithm 1 shows our IGF-2 implementation executed in constant time for the same parameter set. We change the logic of the IGF-2 loop from dealing with each *c-bit* operand to each octet in the hash-generated string buffer.

Algorithm 1 Index Generation Function (IGF-2)

Input: $seed \in \{0,1\}^*$, minimal number of hash function calls for IGF-2 $mincalls$, degree parameter N, number bits in candidate for deriving an index *c-bit*, number of generated indices dr, limit for no bias in IGF-2 $limit$

Output: unsigned integer array $indices[dr]$

1: $Z \leftarrow \texttt{Hash}(seed)$
2: $buf \leftarrow \texttt{Hash}(Z, 0), \texttt{Hash}(Z, 1), \cdots \texttt{Hash}(Z, mincalls - 1)$
3: $indices \leftarrow \emptyset$, $needed \leftarrow$ *c-bit*, $n \leftarrow 0$, $i \leftarrow 0$, $t \leftarrow 0$
4: **for each** octet O in buf **do**
5: **if** $needed \leqslant 8$ **then**
6: $t \leftarrow n|(O \gg (8 - needed))$, $generated \leftarrow$ **True**
7: $O \leftarrow O\&(\texttt{0xFF} \gg (8 - needed))$
8: $needed \leftarrow$ *c-bit* $+ needed - 8$
9: $n \leftarrow O \ll needed$
10: **else**
11: $needed \leftarrow needed - 8$
12: $n \leftarrow n|(O \ll needed)$, $generated \leftarrow$ **False**
13: **end if**
14: $index \leftarrow t \ mod \ N$
15: $flag \leftarrow (t < limit) \wedge (i < dr) \wedge generated \wedge (index \notin indices)$
16: $indices[i] \leftarrow index$
17: $i \leftarrow i + flag$
18: **end for**
19: **return** $indices$

Although the "if-else" statement (i.e. line 5 to 13) may lead to the different execution times, the choice of this statement is deterministic in each iteration. The choice is only decided by the public parameter *c-bits*, so that the constant execution time for the whole IGF-2 function is guaranteed. By setting the flag variable combining the same operations in each iteration, we removed other indeterminate conditional branches (line 14 to 17). Furthermore, in the practical implementation for recording the generated index (i.e. line 17), we make use of a buffer to mark whether this generated index has been recorded before. This buffer is initially set as all zeros, and the value for recording the generated index has an addition with the flag variable in each iteration. Only when all the conditions are satisfied, the counter i will increase. Otherwise, the temporary generated index just refreshes the present indices array but is not stored. At the end, in case that all the octets are not enough to generate the required indices, i.e. finally i is less than dr, IGF-2 will return an error code and then the whole encryption or decryption will be executed again.

4.4 Blinding Polynomial Generation Method (BPGM)

The Blinding Polynomial Generation Method (BPGM) is used to generate a blinding polynomial $r(x) \in \mathcal{T}(d, d)$ from a combined octet string, in order to provide the awareness for plaintext [12]. As can be seen from Algorithm 2, BPGM contains two parts where the former one constructs the seed *sData* (an octet string) while the latter part utilizes IGF-2 to produce the indices of $r(x)$. Due to the different length plaintexts, we developed a constant-time *memcpy* function (used in line 2) to ensure a fixed execution time of the whole BPGM.

Algorithm 2 Blinding Polynomial Generation Method (BPGM)

Input: pseudo random number b, plaintext M, public key H, object identifier OID, security length $secLen$, indices number of blinding polynomial dr(or dr_1, dr_2, dr_3)
Output: blinding polynomial $r(x) \in \mathcal{T}$

1: $hTrunc \leftarrow$ first $secLen$ octets of H
2: construct $sData \leftarrow (OID \parallel M \parallel b \parallel hTrunc)$
3: **if** parameter set uses $product\ form$ **then**
4: $r_1, r_2, r_3 \leftarrow$`IGF-2(`$sData$`, (`dr_1`, `dr_2`, `dr_3`))`
5: $r \leftarrow r_1 \cdot r_2 + r_3$
6: **else**
7: $r \leftarrow$`IGF-2(`$sData$`, `dr`)`
8: **end if**
9: **return** r

Algorithm 3 Mask Generation Function (MGF-TP-1)

Input: $seed \in \{0,1\}^*$, minimal number of hash function calls for MGF-TP-1 $mincalls$, degree parameter N
Output: mask polynomial $v(x) \in \mathcal{T}$

1: $Z \leftarrow$ `Hash`$(seed)$
2: $buf \leftarrow$ `Hash`$(Z, 0)$, `Hash`$(Z, 1)$, $\cdots$ `Hash`$(Z, mincalls - 1)$
3: $v \leftarrow 0 \in \mathcal{T}$, $i \leftarrow 0$
4: **for each** octet O in buf **do**
5: $v[i], v[i{+}1], \ldots v[i{+}4] =$ `octet2trits(`O`)` ▷ $v[i]$ is the i-th coefficient of $v(x)$
6: $i \leftarrow i + 5 \times ((O < 3^5) \wedge (i < N))$
7: **end for**
8: **return** v

4.5 Mask Generation Function (MGF)

Mask Generation Function (MGF) firstly calls hash functions for producing an octet string, then generates a fixed-length mask polynomial $v(x)$ based on this octet string. The standardized NTRUEncrypt only permits using MGF-TP-1 illustrated in Algorithm 3. We slightly modify MGF-TP-1 to execute it in the worst case all the time, i.e. dealing with all the octets in the hash-generated string buffer. In each iteration, only the fact when the octet O is less than 3^5 and meanwhile $v(x)$ is not entirely generated, will lead to an increase of counter i. Obviously, except for a highly optimized hash function, the most "expensive" operation in the remaining part is converting a single octet to five ternary coefficients (at line 5). Theoretically, the method for this conversion is to divide the octet by 3 for five times, yet the method doesn't have a fixed cost for different octets. Our Assembler solution is to pre-compute the corresponding four least significant trits "off-line" for all 81 possibilities and store them in a look-up table in flash ROM, therefore we just need to determine the most significant trit of each input octet. Since each trit occupies two bits, four trits cost precisely one byte in the flash so that our look-up table is compact to save the storage. We adopt the constant time subtraction to determine the most significant trit, i.e. subtracting the 3^4 and $2 \cdot 3^4$ from the octet respectively and calculating the most significant trit according to the carry flag in the Status Register(SREG). Similarly, if $v(x)$ is not completely generated at the end, the MGF-TP-1 will return an error code and then the whole encryption or decryption will restart.

5 Results and Comparison

5.1 Experimental Platform

We compiled our software in Atmel Studio v7.0 which has an extension providing the 8-bit AVR GNU toolchain containing avr-gcc version 5.4.0. All the execution times reported in this section were measured by the support of the cycle-accurate instruction set simulator of Atmel Studio under -O_2 optimization level, whereby the ATmega1281 processor is our target device. This processor could be clocked with a maximal 16 MHz frequency and features 128 kB program memory, 8 kB SRAM. We provide two versions of AvrNtru, using both `ees443ep1` and `ees743ep1` parameter sets, where one is written entirely in C language (hereinafter called C version implementation), and the other one is written in a mix of C language and AVR Assembly language (hereinafter called Assembler version implementation). Both of them are developed based on the optimization techniques described in Sect. 3 and 4. More specifically, for the Assembler version, most functions of our software are written in C language, while we developed the polynomial arithmetics of the multiplication as well as the compression function of SHA256 in Assembly language to speed up the implementation. Additionally, the conversions for different data types and the modulo functions are also written in Assembly language to ensure a constant execution time. Due to the compiling rules of avr-gcc 5.4.0, although applied with optimization techniques, the "pure" C version can't guarantee an absolutely constant execution time. But our Assembler version is not affected by this issue, and it is constant-time and independent of any sensitive information, i.e. plaintext and private key, etc.

5.2 Experimental Results

Table 1 specifies the execution time of major operations (i.e. polynomial multiplication, BPGM, MGF, encryption, and decryption) in our two AvrNtru versions. Due to twice using polynomial multiplications in the decryption process (explained in Sect. 2), the decryption generally costs more time than the encryption. For the `ees443ep1` parameter set, compared with the "pure" C version, our Assembler version costs only 834 k cycles for the encryption and 1061 k cycles for the decryption, which is roughly 4.2 and 3.5 times faster respectively. Additionally, attributed to the optimized polynomial arithmetics and highly-optimized SHA256 hash function, the polynomial multiplication reduced 25.4% of the execution time while the time consumption of BPGM and MGF sharply decreased 82.4% and 84.3% respectively. When choosing the `ees743ep1` parameter, compared to the C version, the Assembler version reduced 75.2% of the encryption time and 66.9% of the decryption time. Table 1 also shows that the time consumption of these three major operations i.e. polynomial multiplication, BPGM, and MGF, actually occupies more than 85% execution time of the encryption and decryption.

As for the RAM footprint, the maximal RAM usage happens in the polynomial multiplication, where it needs three arrays, and each array contains around N integers. Due to an extra temporary array stored in the RAM during the second polynomial multiplication in decryption, for the case of `ees443ep1`, the RAM requirement of decryption is around 1 k bytes more than that of encryption. As shown in Table 2, both RAM footprint and code size of our Assembler version is less than in the C version. In the Assembler version of using `ees443ep1`, although encryption requires 7243 bytes for code size and decryption requires 8028 bytes, the code size of the total NTRUEncrypt just amounts to 9123 bytes due to plenty of reuse of the same components. And the code size between two implementations providing different security levels is quite close, while RAM footprints increase a lot with

Table 1: Execution time (in clock cycles) of major operations in our two versions of AvrNtru for two security levels

Operation	C language (ees443ep1)	C+Asm (ees443ep1)	C language (ees743ep1)	C+Asm (ees743ep1)
Polynomial Mul.	282,606	210,827	705,719	542,187
BPGM	1,760,184	308,801	2,556,062	492,940
MGF	1,206,720	189,525	1,891,114	293,138
Encryption	3,495,776	834,272	5,564,496	1,539,829
Decryption	3,791,657	1,061,683	6,347,688	2,103,228

Table 2: RAM footprint (in bytes) and code size (in bytes) of our two versions of AvrNtru for two security levels

Operation	C language (ees443ep1)		C+Asm (ees443ep1)		C language (ees743ep1)		C+Asm (ees743ep1)	
	RAM	Code	RAM	Code	RAM	Code	RAM	Code
Encryption	3,266	8,622	2,894	7,243	5,178	8,652	4,806	7,288
Decryption	4,233	9,328	3,895	8,028	6,745	9,358	6,407	8,048
Total	4,233	10,994	3,895	9,123	6,745	11,026	6,407	9,740

the higher security level.

5.3 Comparison with Related Work

Table 3 compares software implementations of NTRUEncrypt on different microprocessors. For the 8-bit AVR platform, the implementation in [9] and our AvrNtru implementation both achieve the 128-bit security level. Compared with results in [9], our work costs only 834 k and 1061 k cycles for the encryption and the decryption, which is around 2.4 and 1.3 times faster, respectively. It is remarkable that even compared to the implementation on 32-bit ARM (i.e. Cortex M0 processor) in [20], which uses the same parameter set `ees443ep1` but is not constant-time, our implementation is just slightly less efficient than theirs. Guillen et al. also illustrated a constant-time implementation using the same `ees443ep1` parameters for 32-bit ARM in [20], and it requires roughly 700 k cycles for

Table 3: Performance comparison with previous software implementations of NTRUEncrypt, RSA, ECC, and Ring-LWE encryption on constrained devices. Results are given in clock cycles.

Implementation	Algorithm	Security	Platform	Enc	Dec
This work	NTRUEnc.	128-bit	ATmega1281	834,272	1,061,183
This work	NTRUEnc.	256-bit	ATmega1281	1,539,829	2,103,228
Boorghany et al.[9]	NTRUEnc.	128-bit	ATmega64	2,008,000	1,392,000
Boorghany et al.[9]	NTRUEnc.	128-bit	ARM7TDMI	693,720	998,760
Guillen et al.[20]	NTRUEnc.	128-bit	Cortex M0	588,044	950,371
Guillen et al.[20]	NTRUEnc.	192-bit	Cortex M0	1,040,538	1,634,821
Guillen et al.[20]	NTRUEnc.	256-bit	Cortex M0	1,411,557	2,377,054
Gura et al. [21]	RSA-1024	80-bit	ATmega128	3,440,000	87,920,000
Düll et al. [17]	ECC-255	128-bit	ATmega2560	27,800,794	23,900,397
Liu et al. [37]	Ring-LWE	128-bit	ATxmega128	671,628	275,646

the encryption and 1200 k cycles for the decryption. Most notably, the decryption in our implementation of 256-bit security requires less time than that on 32-bit ARM in [20], while the encryption costs a bit more than that in [20].

In Table 3, apart from NTRUEncrypt software implementations, we also list the fastest 8-bit AVR implementations of both traditional public-key encryption algorithms (RSA and ECC) and another popular lattice-based encryption algorithm (Ring-LWE) to compare with our AvrNtru. We choose implementations offering 128-bit security (except RSA-1024 offering 80-bit security) for comparison. Our AvrNtru outperforms the RSA implementation, achieving 82.8 times faster decryption, even though RSA-1024 cannot match the same security level. As for another widely-used ECC public-key encryption scheme, it requires 27.8 M cycles and 23.9 M cycles for the encryption and decryption, respectively. The to-date fastest 8-bit AVR Ring-LWE encryption implementation is reported in [37], which shows a better performance compared to NTRUEncrypt. However, most of the present Ring-LWE implementations incl. [37] only consist of the polynomial arithmetic without including the secure padding and the other auxiliary functions, which means they would have more execution time in practise. Furthermore, most known high-performance Ring-LWE implementations incl. [37] don't have constant time, in other words, they are vulnerable to timing attacks. The results above show that our NTRUEncrypt software has more superior performance than the classic public-key encryption schemes on the resource-limited devices. Compared to the equally efficient Ring-LWE encryption schemes, our AvrNtru not only ensures high efficiency but also provides better practical security against timing attacks.

6 Conclusions

In this paper, we presented a "hybrid" polynomial multiplication and several optimizations for the auxiliary functions, in order to efficiently implement the full NTRUEncrypt on the 8-bit AVR platform. Combined with these optimization techniques, our AvrNtru implementation, besides resisting the timing attack, also reaches the fastest speed among the known NTRUEncrypt software implementations on 8-bit AVR platform. Achieving the 128-bit security level, the software version written completely in C language and compiled with avr-gcc 5.4.0, costs 3495 k cycles for encryption and 3791 k cycles for decryption. Our AvrNtru written in the mix of C language and Assembly language greatly improved the efficiency, and requires only 834 k cycles for encryption and 1061 k cycles for decryption, that is 4.2 times and 3.5 times faster. Moreover, compared with the known most efficient 8-bit AVR NTRUEncrypt software implementation in [9], our AvrNtru reduced 58.5% and 23.7% of the execution time for encryption and decryption, respectively. More notably, our software for providing a 256-bit security only requires 1,539,829 clock cycles for encryption and 2,103,228 clock cycles for decryption, where the decryption is even faster than the known most efficient NTRUEncrypt implementation on 32-bit ARM in [20]. The final comparison between our software with three other public-key encryption schemes, proves that our AvrNtru, is an excellent candidate of asymmetric encryption for 8-bit AVR processors. All of these shows that NTRUEncrypt is a feasible option for high-speed asymmetric cryptography on resource-constrained devices, especially considering it could resist the timing attack and quantum cryptanalytic attacks.

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

References

[1] C. Aguilar Melchor, X. Boyen, J.-C. Deneuville, and P. Gaborit. Sealing the leak on classical NTRU signatures. In M. Mosca, editor, *Post-Quantum Cryptography — PQCrypto 2014*, volume 8772 of *Lecture Notes in Computer Science*, pages 1–21. Springer Verlag, 2014.

[2] A. C. Atici, L. Batina, J. Fan, I. Verbauwhede, and S. B. Örs Yalçin. Low-cost implementations of NTRU for pervasive security. In *Proceedings of the 19th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP 2008)*, pages 79–84. IEEE Computer Society Press, 2008.

[3] Atmel Corporation. 8-bit AVR® Instruction Set. User Guide, available for download at `http://www.atmel.com/dyn/resources/prod_documents/doc0856.pdf`, July 2008.

[4] D. V. Bailey, D. Coffin, A. J. Elbirt, J. H. Silverman, and A. D. Woodbury. NTRU in constrained devices. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 262–272. Springer Verlag, 2001.

[5] P. S. Barreto, F. P. Biasi, R. Dahab, J. C. López-Hernández, E. M. de Morais, A. D. Salina de Oliveira, G. C. Pereira, and J. E. Ricardini. A panorama of post-quantum cryptography. In Ç. K. Koç, editor, *Open Problems in Mathematics and Computational Science*, pages 387–439. Springer Verlag, 2014.

[6] D. J. Bernstein, J. Buchmann, and E. Dahmen, editors. *Post-Quantum Cryptography*. Springer Verlag, 2009.

[7] D. J. Bernstein, C. Chuengsatiansup, T. Lange, and C. van Vredendaal. NTRU prime. Cryptology ePrint Archive, Report 2016/461, 2016. Available for download at `http://eprint.iacr.org`.

[8] I. F. Blake, G. Seroussi, and N. P. Smart. *Elliptic Curves in Cryptography*, volume 265 of *London Mathematical Society Lecture Notes Series*. Cambridge University Press, 1999.

[9] A. Boorghany, S. Bayat Sarmadi, and R. Jalili. On constrained implementation of lattice-based cryptographic primitives and schemes on smart cards. *ACM Transactions on Embedded Computing Systems*, 14(3):42, May 2015.

[10] D. Cabarcas, P. Weiden, and J. A. Buchmann. On the efficiency of provably secure NTRU. In M. Mosca, editor, *Post-Quantum Cryptography — PQCrypto 2014*, volume 8772 of *Lecture Notes in Computer Science*, pages 22–39. Springer Verlag, 2014.

[11] H. Cheng, D. Dinu, and J. Großschädl. Efficient implementation of the sha-512 hash function for 8-bit avr microcontrollers. In J.-L. Lanet and C. Toma, editors, *Innovative Security Solutions for Information Technology and Communications*, pages 273–287. Springer Verlag, 2019.

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

[13] D. Coppersmith and A. Shamir. Lattice attacks on NTRU. In W. Fumy, editor, *Advances in Cryptology — EUROCRYPT '97*, volume 1233 of *Lecture Notes in Computer Science*, pages 52–61. Springer Verlag, 1997.

[14] W. Dai, W. Whyte, and Z. Zhang. Optimizing polynomial convolution for NTRUEncrypt. *IEEE Transactions on Computers*, 67(11):1572–1583, Nov. 2018.

[15] B. Driessen, A. Poschmann, and C. Paar. Comparison of innovative signature algorithms for WSNs. In V. D. Gligor, J.-P. Hubaux, and R. Poovendran, editors, *Proceedings of the 1st ACM Conference on Wireless Network Security (WISEC 2008)*, pages 30–35. ACM Press, 2008.

[16] L. Ducas and P. Q. Nguyen. Learning a zonotope and more: Cryptanalysis of NTRUSign countermeasures. In X. Wang and K. Sako, editors, *Advances in Cryptology — ASIACRYPT*

2012, volume 7658 of *Lecture Notes in Computer Science*, pages 433–450. Springer Verlag, 2012.

[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*, 77(2–3):493–514, Dec. 2015.

[18] G. Gaubatz, J.-P. Kaps, E. Öztürk, and B. Sunar. State of the art in ultra-low power public key cryptography for wireless sensor networks. In *Proceedings of the 3rd IEEE Conference on Pervasive Computing and Communications Workshops (PERCOMW 2005)*, pages 146–150. IEEE Computer Society Press, 2005.

[19] C. Gentry, J. Jonsson, J. Stern, and M. Szydlo. Cryptanalysis of the NTRU signature scheme (NSS) from Eurocrypt 2001. In C. Boyd, editor, *Advances in Cryptology — ASIACRYPT 2001*, volume 2248 of *Lecture Notes in Computer Science*, pages 1–20. Springer Verlag, 2001.

[20] 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 *Proceedings of the 20th Design, Automation and Test in Europe Conference and Exhibition (DATE 2017)*, pages 698–703. IEEE Computer Society Press, 2017.

[21] N. Gura, A. Patel, A. S. Wander, H. Eberle, and S. Chang Shantz. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems — CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 119–132. Springer Verlag, 2004.

[22] J. Hermans, F. Vercauteren, and B. Preneel. Speed records for NTRU. In J. Pieprzyk, editor, *Topics in Cryptology — CT-RSA 2010*, volume 5985 of *Lecture Notes in Computer Science*, pages 73–88. Springer Verlag, 2010.

[23] J. Hoffstein, N. Howgrave-Graham, J. Pipher, J. H. Silverman, and W. Whyte. NTRUSign: Digital signatures using the NTRU lattice. In M. Joye, editor, *Topics in Cryptology — CT-RSA 2003*, volume 2612 of *Lecture Notes in Computer Science*, pages 122–140. Springer Verlag, 2003.

[24] J. Hoffstein, N. Howgrave-Graham, J. Pipher, and W. Whyte. Practical lattice-based cryptography: NTRUEncrypt and NTRUSign. In P. Q. Nguyen and B. Vallée, editors, *The LLL Algorithm: Survey and Applications*, pages 349–390. Springer Verlag, 2010.

[25] J. Hoffstein, J. Pipher, J. M. Schanck, J. H. Silverman, and W. Whyte. Transcript secure signatures based on modular lattices. In M. Mosca, editor, *Post-Quantum Cryptography — PQCrypto 2014*, volume 8772 of *Lecture Notes in Computer Science*, pages 142–159. Springer Verlag, 2014.

[26] 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. Available for download at `http://eprint.iacr.org`.

[27] J. Hoffstein, J. Pipher, and J. H. Silverman. NTRU: A ring-based public key cryptosystem. In J. P. Buhler, editor, *Algorithmic Number Theory, Third International Symposium (ANTS-III)*, volume 1423 of *Lecture Notes in Computer Science*, pages 267–288. Springer Verlag, 1998.

[28] J. Hoffstein, J. Pipher, and J. H. Silverman. NSS: An NTRU lattice-based signature scheme. In B. Pfitzmann, editor, *Advances in Cryptology — EUROCRYPT 2001*, volume 2045 of *Lecture Notes in Computer Science*, pages 211–228. Springer Verlag, 2001.

[29] J. Hoffstein, J. Pipher, and J. H. Silverman. *An Introduction to Mathematical Cryptography*. Undergraduate Texts in Mathematics. Springer Verlag, second edition, 2014.

[30] J. Hoffstein and J. H. Silverman. Optimizations for NTRU. In K. Alster, J. Urbanowicz, and H. C. Williams, editors, *Public-Key Cryptography and Computational Number Theory*, De Gruyter Proceedings in Mathematics, pages 77–88. Walter de Gruyter, 2001.

[31] N. Howgrave-Graham. A hybrid lattice-reduction and meet-in-the-middle attack against NTRU. In A. J. Menezes, editor, *Advances in Cryptology — CRYPTO 2007*, volume 4622 of *Lecture Notes in Computer Science*, pages 150–169. Springer Verlag, 2007.

[32] N. Howgrave-Graham, P. Q. Nguyen, D. Pointcheval, J. Proos, J. H. Silverman, A. Singer, and W. Whyte. The impact of decryption failures on the security of NTRU encryption. In D. Boneh, editor, *Advances in Cryptology — CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 226–246. Springer Verlag, 2003.

[33] N. Howgrave-Graham, J. H. Silverman, and W. Whyte. Choosing parameter sets for NTRUEncrypt with NAEP and SVES-3. In A. J. Menezes, editor, *Topics in Cryptology — CT-RSA 2005*, volume 3376 of *Lecture Notes in Computer Science*, pages 118–135. Springer Verlag, 2005.

[34] A. Hülsing, J. Rijneveld, J. M. Schanck, and P. Schwabe. High-speed key encapsulation from NTRU. In W. Fischer and N. Homma, editors, *Cryptographic Hardware and Embedded Systems — CHES 2017*, volume 10529 of *Lecture Notes in Computer Science*, pages 232–252. Springer Verlag, 2017.

[35] A. A. Kamal and A. M. Youssef. Fault analysis of the NTRUEncrypt cryptosystem. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E94-A(4):1156–1158, Apr. 2011.

[36] M.-K. Lee, J. E. Song, D. Choi, and D.-G. Han. Countermeasures against power analysis attacks for the NTRU public key cryptosystem. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E93-A(1):153–163, Jan. 2010.

[37] Z. Liu, H. Seo, S. S. Roy, J. Großschädl, H. Kim, and I. Verbauwhede. Efficient ring-LWE encryption on 8-bit AVR processors. In T. Güneysu and H. Handschuh, editors, *Cryptographic Hardware and Embedded Systems — CHES 2015*, volume 9293 of *Lecture Notes in Computer Science*, pages 663–682. Springer Verlag, 2015.

[38] M. Monteverde. *NTRU Software Implementation for Constrained Devices*. M.Sc. Thesis, Katholieke Universiteit Leuven, Heverlee, Belgium, 2008.

[39] National Institute of Standards and Technology (NIST). Secure Hash Standard (SHS). Federal Information Processing Standards Publication 180-4, available for download at `http://dx.doi.org/10.6028/NIST.FIPS.180-4`, Aug. 2015.

[40] P. Q. Nguyen and O. Regev. Learning a parallelepiped: Cryptanalysis of GGH and NTRU signatures. In S. Vaudenay, editor, *Advances in Cryptology — EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 271–288. Springer Verlag, 2006.

[41] C. Peikert. A decade of lattice cryptography. Cryptology ePrint Archive, Report 2015/939, 2015. Available for download at `http://eprint.iacr.org`.

[42] T. Pöppelmann, T. Oder, and T. Güneysu. High-performance ideal lattice-based cryptography on 8-bit ATxmega microcontrollers. In K. E. Lauter and F. Rodríguez-Henríquez, editors, *Progress in Cryptology — LATINCRYPT 2015*, volume 9230 of *Lecture Notes in Computer Science*, pages 346–365. Springer Verlag, 2015.

[43] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21(2):120–126, Feb. 1978.

[44] J. M. Schanck. *Practical Lattice Cryptosystems: NTRUEncrypt and NTRUMLS*. M.Sc. Thesis, University of Waterloo, Waterloo, ON, Canada, 2015.

[45] P. W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS '94)*, pages 124–134. IEEE Computer Society Press, 1994.

[46] J. H. Silverman and W. Whyte. Timing attacks on NTRUEncrypt via variation in the number of hash calls. In M. Abe, editor, *Topics in Cryptology — CT-RSA 2007*, volume 4377 of *Lecture Notes in Computer Science*, pages 208–224. Springer Verlag, 2007.

[47] D. Stehlé and R. Steinfeld. Making NTRU as secure as worst-case problems over ideal lattices. In K. G. Paterson, editor, *Advances in Cryptology — EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 27–47. Springer Verlag, 2011.

[48] L. Yan, Y. Zhang, L. T. Yang, and H. Ning. *The Internet of Things: From RFID to the Next-Generation Pervasive Networked Systems*. Auerbach Publications, 2008.