# Efficiency Improvement for NTRU

Johannes Buchmann, Martin Döring*, and Richard Lindner

Technische Universität Darmstadt
Department of Computer Science
Hochschulstraße 10, 64289 Darmstadt, Germany
{buchmann,doering,rlindner}@cdc.informatik.tu-darmstadt.de

**Abstract.** The NTRU encryption scheme is an interesting alternative to well-established encryption schemes such as RSA, ElGamal, and ECIES. The security of NTRU relies on the hardness of computing short lattice vectors and thus is a promising candidate for being quantum computer resistant. There has been extensive research on efficient implementation of the NTRU encryption scheme. In this paper, we present a new algorithm for enhancing the performance of NTRU. The proposed method is between 15% and 22% faster on average than the best previously known method. We also present a highly efficient implementation of NTRU within the Java Cryptography Architecture.

**Keywords:** NTRU, efficiency improvement, implementation.

## 1 Introduction

Encryption schemes commonly used today are RSA [13], ElGamal [4], and ECIES [11]. The security of those schemes relies on the difficulty of factoring large composite integers or computing discrete logarithms. However, it is unclear whether these computational problems remain intractable in the future. For example, Shor [18] showed that quantum computers can be used to factor integers and to compute discrete logarithms in the relevant groups in polynomial time. Also, in the past thirty years there has been significant progress in solving the integer factorization and discrete logarithm problems using classical computers [15,16,3,1]. It is therefore necessary to develop alternative encryption schemes which do not rely on the difficulty of factoring or computing discrete logarithms and which are considered secure even against quantum computer attacks.

A promising candidate for such a quantum secure encryption scheme is the lattice-based public-key cryptosystem NTRU [7] in its NAEP/SVES-3 variant [9,10]. The cryptosystem is patented by NTRU Cryptosystems, Inc., a company founded in 1996 by J. Hoffstein, J. Pipher, and J. H. Silverman. SVES-3 is currently undergoing a standardization process and will presumably be included in the upcoming IEEE standard 1363.1 [6]. We refer to the SVES-3 variant proposed in the draft standard as *NTRUSVES*.

* Author supported by SicAri, a project funded by the German Ministry for Education and Research (BMBF). See http://www.sicari.de.

*Our contribution.* We propose a new algorithm for fast multiplication of NTRU polynomials which improves on [6] and [14]. Depending on the used parameters, our algorithm achieves an average-case speedup between 21% and 35% compared to [6] and between 15% to 22% compared to [14]. The proposed algorithm also is very space efficient.

In addition, we provide a highly efficient Java implementation of NTRUSVES according to draft version 8 of IEEE P1363.1 which incorporates our proposed multiplication algorithm. The implementation is provided within the Java Cryptography Architecture (JCA, [19,20]) and will be part of the Java Cryptographic Service Provider FlexiProvider [5].

*Related work.* IEEE P1363.1 [6] proposes an algorithm for fast multiplication of NTRU polynomials which is due to Bailey et al. [2]. Lee et al. [14] present an improved sliding window multiplication algorithm. The authors state that using their algorithm, the NTRU encryption and decryption operations can be sped up by up to 32% compared to Bailey et al.'s algorithm. However, this seems to be a best-case estimate. Our experiments show that the average-case speedup is between 7% and 16%, depending on the used parameter set.

*Organization.* The paper is organized as follows: Section 2 gives a brief mathematical description of NTRU and NAEP/SVES-3. In Section 3, we describe our new multiplication algorithm and compare it with the algorithms of Bailey et al. and Lee et al. Section 4 provides details of our NTRUSVES implementation as well as measurement results of the implementation. Section 5 concludes the paper.

## 2 Mathematical background

In this section, we give a brief mathematical description of NTRU in the binary and the product form variant. We also illustrate the *NTRU Asymmetric Encryption Padding (NAEP)* scheme in its most common instantiation: the *Shortest Vector Encryption Scheme, third revision (SVES-3)* [9,10].

### 2.1 NTRU

The main parameters of NTRU are defined in Table 1. The stated security requirements are taken from IEEE P1361.1-D9 [6], which is the latest draft of this standard to date.

Define the ring of convolution modular polynomials

$$R := \mathbb{Z}[X] \ / \ (X^N - 1)$$

using the main parameter $N$. All computations in this section are performed in $R$.

Let $D(d)$ denote the set of binary polynomials of degree less than $N$ with hamming weight $d$. Two space parameters $d_F, d_g \in \mathbb{N}$ define the private key spaces $D(d_F), D(d_g)$.

| Parameter | Description | Security requirements |
|---|---|---|
| $N \in \mathbb{N}$ | Dim parameter | $N$ has to be prime. |
| $p \in \mathbb{N}$ | Small modulus | $p$ has to be equal to 2. |
| $q \in \mathbb{N}$ | Big modulus | $q \neq N$ has to be prime and large enough to prevent decryption errors (see *correctness*). |

**Table 1.** NTRU main parameters

*Key pair generation.* Choose uniformly at random the binary polynomials $F \in D(d_F)$ and $g \in D(d_g)$. Compute $f := 1 + pF$. Check whether $f$ is invertible in $R$ modulo $q$ and denote the inverse $f_q^{-1}$. If the inverse does not exist, start over. Otherwise, compute the polynomial

$$h := f_q^{-1} pg \pmod{q}.$$

The private key is $f$, the public key is $h$.

*Encryption.* Encode a message $M$ into a binary polynomial $m$. Randomly choose a binary blinding polynomial $r$. The ciphertext is the polynomial

$$e := m + rh \pmod{q}.$$

*Decryption.* Let $e$ be the ciphertext. Compute

$$\begin{aligned} a &:= fe = fm + pgr \pmod{q} \\ &\overset{(\star)}{=} m + p(Fm + gr) \pmod{q}. \end{aligned}$$

Reduce the coefficients of $a$ into the interval $[0, q)$. Compute the polynomial

$$m := a \pmod{p}$$

and decode the message $M$.

*Correctness.* The correctness of the decryption operation rests upon the following easy-to-prove lemma:

**Lemma 1.** *Let $b \in D(d)$ and $r \in R$ be arbitrary. Then it holds that*

$$\|br\|_\infty \leq d\|r\|_\infty,$$

where the max-norm on $R$ is defined as

$$\left\| \sum_{i=0}^{N-1} r_i X^i \right\|_\infty := \max_{i=0,\ldots,N-1} \{|r_i|\}.$$

Decryption works if equality $(\star)$ holds over $R$ without taking both sides modulo $q$. By Lemma 1 this is guaranteed by choosing $d_F$ and $d_g$ such that

$$1 + p(d_F + d_g) < q.$$

NTRU can be used with two additional space parameters $d_m, d_r \in \mathbb{Z}$. The message $M$ is then encoded into a polynomial $m \in D(d_m)$ and the blinding polynomial $r$ is chosen from $D(d_r)$. Using these additional parameters makes NTRU more efficient and the constraint on the parameters is relaxed to

$$1 + p(\min\{d_F, d_m\} + \min\{d_g, d_r\}) < q.$$

*Arithmetic.* All arithmetic operations described in the preceding paragraphs are performed in the ring $R$. After all computations, the coefficients of the polynomials are reduced modulo $q$, except once during decryption, where they are reduced modulo $p$. As shown in the previous paragraph, this modulo $p$ step may be preceded by a reduction modulo $q$ if the space parameters are chosen appropriately. Thus, all arithmetic operations can be performed modulo $q$, that is, in the finite ring $R_q := \mathbb{Z}[X] \,/\, (q, X^N - 1)$ instead of $R$.

### 2.2 Product form variant

There is a more efficient variant of NTRU, called the *product form variant*. In the product form variant, the binary polynomials $F$ and $r$ are replaced by so-called *product form polynomials*. Product form polynomials are of the form $f_1 f_2 + f_3$, where $f_1$, $f_2$, and $f_3$ are very sparse binary polynomials. In this section, we describe the differences to the regular, sometimes called *binary* variant.

*Parameters.* Choose $N, p, q$ as before. The space parameters are $d_{f_1}$, $d_{f_2}$, $d_{f_3}$, $d_g \in \mathbb{N}$.

*Key pair generation.* Randomly choose $f_i \in D(d_{f_i})$ for $1 \leq i \leq 3$ and compute $F := f_1 f_2 + f_3$. The remaining steps are as before.

*Correctness.* In the product form variant, it is still needed that

$$a = fm + pgr = m + p(f_1 f_2 m + f_3 m + gr) \text{ in } R,$$

to avoid decryption failures. This is guaranteed if

$$1 + p(\min\{d_{f_1}, d_{f_2}\} + d_{f_3} + d_g) < q.$$

### 2.3 NAEP/SVES-3

NAEP/SVES-3 is a scheme based on NTRU that is provably secure against adaptive chosen ciphertext attacks in the random oracle model, similar to OAEP+ for RSA. The description of NAEP/SVES-3 can be found in Appendix B.

## 3 Pattern multiplication

We propose a new algorithm for the multiplication of elements of $R_q$ with binary polynomials which is up to 35% faster than the algorithm proposed by IEEE P1363.1-D8 and up to 22% faster than the algorithm of Lee et al. [14]. We call the new algorithm *pattern multiplication*. The algorithm is described in the following sections.

### 3.1 Basic idea

Throughout the paper, we identify polynomials $a(X) = \sum_{i=0}^{N-1} a_i X^i \in R_q$ with their coefficient vector $(a_0, \ldots, a_{N-1})$. The product $ab$ of two polynomials $a, b \in R$ can be represented by the convolution operation $c = a * b$, which is given by the equation

$$c_k = \sum_{i+j \equiv k \pmod N} a_i b_j$$

for $k = 0, \ldots, N-1$. This operation generally requires $N^2$ integer multiplications. However, if the polynomial $b$ is binary, the multiplication can be performed much faster. Note that the convolution operation $c = a * b$ can also be written as a vector-matrix multiplication:

$$c = (b_0, \ldots, b_{N-1}) \cdot \begin{pmatrix} a_0 & \ldots & a_{N-2} & a_{N-1} \\ a_{N-1} & \ldots & a_{N-3} & a_{N-2} \\ \vdots & \ddots & \vdots & \vdots \\ a_1 & \ldots & a_{N-1} & a_0 \end{pmatrix} = (b_0, \ldots, b_{N-1}) \cdot \begin{pmatrix} a \\ Xa \\ X^2 a \\ \vdots \\ X^{N-1} a \end{pmatrix}.$$

The last equality holds since multiplication of a polynomial $a$ with a monomial $X^i$ in $R$ corresponds to a rotation to the right of the coefficient vector of $a$ by $i$ positions.

If the polynomial $b$ is binary, the multiplication with $a$ therefore amounts to adding polynomials of the form $X^i a$, which can in turn be computed as rotations of the coefficient vector of $a$, for all $i$ such that $b_i = 1$. So, only additions over $\mathbb{Z}$ are necessary in order to compute the product $ab$ in $R$. If $b$ has hamming weight $d$, the product can be computed with $dN$ additions over $\mathbb{Z}$. The product in $R_q$ is computed identically, only that the coefficients of $c$ are additionally reduced modulo $q$ once at the end of the multiplication. This idea is due to Bailey et al. [2] and the corresponding multiplication algorithm is incorporated into the IEEE P1363.1 draft standard.

In the following, we denote binary polynomials as bit strings. It is possible to reduce the number of additions needed to compute the product $ab$ by using bit patterns of the binary polynomial $b$. We consider patterns containing two 1s. Consider the binary polynomial $b$ in Figure 1.

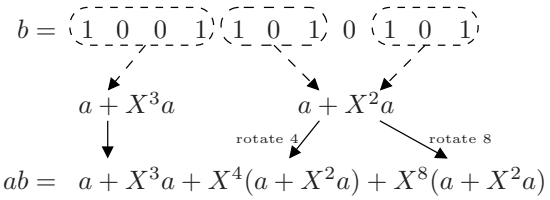


**Fig. 1.** Multiplication of $a, b$ using pairs as pattern.

The bit pattern 101 occurs twice. By computing $a + X^2 a$ once and storing it in a lookup table, the number of additions needed to compute the product $ab$ can be reduced by $N$. More general, it is possible to reduce the number of additions needed to compute $ab$ whenever a bit pattern occurs more than once in $b$. It is thus desirable to choose patterns in a way that maximizes the number of pattern occurrences and to efficiently identify the patterns in $b$.

### 3.2 The proposed algorithm

In this section, we describe our proposed algorithms for finding bit patterns of a binary polynomial $b$ and for computing the product of $b$ with arbitrary polynomials $a \in R_q$ using these patterns.

*Pattern finding.* A binary polynomial $b$ of hamming weight $d$ is represented by the sequence $b_0, \ldots, b_{d-1}$ of its non-zero coefficient locations. The polynomial is traversed once in reverse order, starting at $b_{d-1}$. Every pair of coefficient locations $(b_i, b_{i-1})$ represents a bit pattern of length $b_i - b_{i-1}$. The coefficient location $b_i$ is stored in a corresponding list and $i$ is decreased by 2. A possibly remaining single coefficient location (in case that $d$ is odd) is stored separately. The description of the algorithm can be found in Algorithm 1.

**Algorithm 1** Pattern finding

**System Parameters:** integer $N$
**Input:** a binary polynomial $b$ given as the sequence $b_0, \ldots, b_{d-1}$ of its non-zero coefficient locations
**Output:** a sequence of arrays $(L_0, \ldots, L_{N-d+1})$ of bit pattern locations of $a$

1: create empty arrays $(L_0, \ldots, L_{N-d+1})$ ▷ holds the result
2: set $index \leftarrow d - 1$ ▷ start at highest index of $b$
3: **while** $index > 0$ **do** ▷ as long as 2 or more coefficients remain
4: set $dist \leftarrow b_{index} - b_{index-1}$ ▷ compute pattern size
5: append $b_{index}$ to $L_{dist}$ ▷ append start index to corresponding array
6: set $index \leftarrow index - 2$ ▷ jump to next pair of coefficients
7: **if** $index = 0$ **then** ▷ if a single coefficient remains
8: append $b_0$ to $L_0$ ▷ append it to $L_0$
9: **return** $(L_0, \ldots, L_{N-d+1})$ ▷ return result

The algorithm essentially requires $\lfloor d/2 \rfloor$ subtractions over $\mathbb{Z}$ and memory for storing $\lceil d/2 \rceil$ integers from the interval $[0, N)$.

*Pattern multiplication* In the following, we describe our proposed algorithm for computing the product $ab$ of an arbitrary polynomial $a \in R_q$ and a binary polynomial $b$ given as the sequence of arrays $(L_0, \ldots, L_{N-d+1})$ of bit pattern locations as computed by Algorithm 1.

Each non-empty list $L_i, i > 0$ represents a bit pattern of $b$. For each such $L_i$, the corresponding pattern $P = a + X^i a$ is computed. This pattern is then successively rotated by the elements of $L_i$ and added to the result polynomial. A possibly remaining single coefficient is treated separately without computing a pattern. The detailed description of the algorithm can be found in Algorithm 2.

**Algorithm 2** Pattern multiplication

**System Parameters:** integers $N$, $q$
**Input:** a polynomial $a = (a_0, \ldots, a_{N-1}) \in R_q$, a sequence of arrays $(L_0, \ldots, L_{N-d+1})$ of pattern locations of a binary polynomial $b$
**Output:** $c = a * b$

create empty coefficient array $c = (c_0, \ldots, c_{N-1})$ ▷ holds the result
create empty coefficient array $P = (P_0, \ldots, P_{N-1})$ ▷ holds a pattern

**for all** $i > 0$ such that $L_i$ is not empty **do** ▷ process patterns
    **for** $j$ from 0 to $N - 1$ **do** ▷ compute pattern $P = a + X^i a$
        set $P_j \leftarrow a_j + a_{i+j \pmod N}$
    let $d_i$ denote the size of $L_i$ ▷ get number of occurrences of this pattern
    **for** $j$ from 0 to $d_i - 1$ **do** ▷ multiply using the pattern
        **for** $k$ from 0 to $N - 1$ **do**
            $c_{L_i[j]+k \pmod N} \leftarrow c_{L_i[j]+k \pmod N} + P_k$

**if** $L_0$ is not empty **then** ▷ treat possibly remaining single coefficient
    **for** $k$ from 0 to $N - 1$ **do**
        $c_{L_0[0]+k \pmod N} \leftarrow c_{L_0[0]+k \pmod N} + a_k$

**for** $i$ from 0 to $N - 1$ **do** ▷ reduce coefficients
    set $c_i \leftarrow c_i \pmod q$

**return** $c$ ▷ return result

The algorithm requires at most $dN$ additions over $\mathbb{Z}$ in the case that no bit pattern occurs more than once in $b$. For each pattern occurring $d_i > 1$ times, the number of additions is reduced by $(d_i - 1)N$. Additionally, $N$ reductions modulo $q$ are performed. The algorithm requires memory for storing two polynomials (the result polynomial and a pattern polynomial).

### 3.3 Comparison and discussion

The multiplication algorithm of Bailey et al. does not consider bit patterns of the binary polynomial $b$. So, for every non-zero coefficient of $b$, $N$ additions over $\mathbb{Z}$ have to be performed.

The algorithm of Lee et al. uses bit patterns consisting of two neighboring 1s. We say that such a pattern has length $l$ if the two 1s are separated by $l - 1$ 0s. The algorithm only considers patterns of length less than or equal to a parameter

$w$ which is chosen as $w = 5$ for the proposed parameter sets. For each pattern length $l = 1, \ldots, w$, the polynomial $a + X^l a$ is computed and stored in a lookup table. The non-zero coefficients not belonging to any such pattern are treated as in the algorithm of Bailey et al. Binary polynomials are represented as bit strings.

Our algorithm also uses bit patterns consisting of two neighboring 1s, but the patterns can be of arbitrary length, and only the patterns actually occurring in $b$ are considered. We omit the precomputation step and compute the polynomials $a + X^l a$ when needed. We also represent binary polynomials as the array of their non-zero coefficient locations, in accordance with the IEEE P1363.1 proposal. It shows that pattern finding can be performed much easier and faster in this representation.

As already outlined by Lee et al., considering patterns consisting of more than two 1s does not achieve any notable speedup because the probability that these patterns occur more than once in $b$ is very low. Also, an adjusted pattern multiplication algorithm for the product-form variant does not achieve any notable speedup for the same reasons.

Finally, we would like to remark that the precomputation scenario presented by Lee et al. is not always applicable to NTRU. During encryption, it applies only when sending many messages to a single receiver. During decryption, it only applies to one of the two multiplications involved. We therefore propose to use a hybrid solution between the approach of Lee et al. and the one we present in this paper.

### 3.4 Measurement results

In this section, we state the results of the performance measurements of the multiplication algorithms of Bailey et al., Lee et al. [14], and our proposed algorithm. The measurement results are summarized in table 2. Column "Parameter

| *Parameter set* | $t_{Bailey}$ | $t_{Lee}$ | $t_{pattern}$ |
|---|---|---|---|
| ees251ep6 | 0.14 ms | 0.13 ms (+ 7%) | 0.11 ms (+21%) |
| ees347ep2 | 0.25 ms | 0.23 ms (+ 8%) | 0.19 ms (+24%) |
| ees397ep1 | 0.32 ms | 0.29 ms (+ 9%) | 0.23 ms (+28%) |
| ees491ep1 | 0.48 ms | 0.42 ms (+12%) | 0.34 ms (+29%) |
| ees587ep1 | 0.68 ms | 0.59 ms (+13%) | 0.46 ms (+32%) |
| ees787ep1 | 1.18 ms | 0.99 ms (+16%) | 0.77 ms (+35%) |

**Table 2.** Time measurement results of the different multiplication algorithms

set" denotes the used parameter set. Column "$t_{Bailey}$" denotes the multiplication algorithm of Bailey et al., column "$t_{Lee}$" denotes the algorithm of Lee et al., and column "$t_{pattern}$" denotes our proposed pattern multiplication algorithm. The

stated times are average times taken over 50000 multiplications of randomly chosen polynomials for each parameter set.

For the algorithm of Lee et al. and our proposed algorithm, the pattern finding and precomputation steps are taken into account. For these two algorithms, the speedup relative to Bailey et al.'s algorithm is given in addition to the absolute times.

The experiments were made using a computer equipped with a Pentium M 1.6 GHz CPU, 512 MB of RAM and running Microsoft Windows XP. The code was compiled with JDK 1.3 and run under JRE 1.6.

## 4 NTRUSVES implementation

In this section, we provide details of our NTRUSVES implementation. First, we describe the instantiation of SVES-3 given in IEEE P1363.1. Afterwards, we describe the supported parameters, the format of the keys, and the encoding format of polynomials and keys.

### 4.1 Instantiation

IEEE P1363.1 proposes concrete instantiations of the hash functions $G$ and $H$ used in the NAEP/SVES-3 scheme. The hash function $G$ is called *Blinding Value Generation Method (BVGM)* (in draft 8) or *Blinding Polynomial Generation Method (BPGM)* (in draft 9). We decide to use the latter notation for the rest of the paper. The BPGM itself uses a so-called *Seed Expansion Function (SEF)* (draft 8) or *Index Generation Function (IGF)* (draft 9), which in turn uses a hash function. Again, we use the latter name for the rest of the paper.

The draft standard proposes two different BPGM instantiations. The first one (LBP-BPGM1) is used to generate a binary blinding polynomial, the second one (LBP-BPGM2) produces a product form blinding polynomial. Both use the same IGF (IGF-MGF1). The underlying hash function is either SHA-1 or SHA-256 for the proposed parameter sets (see Section 4.2).

The input $(ID||m||b)$ to the BPGM can be extended to $(ID||m||b||hTrunc)$, where $hTrunc$ are some bits of the encoded public key $h$. Although this option is not used with the proposed parameter sets (i.e., the length of $hTrunc$ is 0), it is supported by our implementation (see also Section 4.3).

The function $H$ is called *Mask Generation Function (MGF)* and uses a hash function. The draft standard proposes one instantiation (MGF1) which uses either SHA-1 or SHA-256 as hash function.

We do not describe the BPGM, IGF, and MGF algorithms in this paper, but rather refer the reader to [6]. Our implementation follows the description of the algorithms of draft 8 precisely.

### 4.2 Parameters

Our implementation supports all recommended parameter sets of draft version 9 of IEEE P1363.1 (see Annex A.5 of the draft standard). For each choice of the

main parameter $N \in \{251, 347, 397, 491, 587, 787\}$, there is a binary and a product form parameter set. The parameter choices correspond to bit security levels of 80, 112, 128, 160, 192, and 256 bits, respectively. Each parameter set is chosen to maximize efficiency for the selected security level.

### 4.3 Key pairs

The name of the parameter set used to generate the keys is stored both in the public and in the private key.

*Public key.* The public key is the polynomial $h = f_q^{-1} pg \pmod q$.

*Private key.* Differing from the draft standard, we do not store the polynomial $f$ as the private key. Instead, the pair of polynomials $(F, g)$ is stored, where $F$ either is a binary or a product form polynomial, and $g$ is a binary polynomial. On the one hand, this speeds up decryption (see Section 4.4) and reduces the size of the encoded private key (see Section 4.6). On the other hand, the public polynomial $h$ is needed to generate the input to the Blinding Polynomial Generation Method (see Section 4.1), so it must be possible to reconstruct $h$ from the private key.

### 4.4 Decryption

The central decryption operation is the computation of the polynomial

$$a = fe \pmod q,$$

where $f = 1 + pF$ is the private polynomial. Since in our implementation, the (binary or product form) polynomial $F$ is stored in the private key (see Section 4.3), this computation is performed as

$$a = e + peF \pmod q,$$

using the efficient multiplication algorithms described in Section 4.5 for the computation of $eF$.

### 4.5 Efficient multiplication

We employ the pattern multiplication algorithm proposed in this paper to compute the product of polynomials in $R_q = \mathbb{Z}[X]/(q, X^N - 1)$ with binary polynomials. For the product form variant, the algorithm described in Section 6.2.6 of the IEEE P1363.1 draft is used.

### 4.6 Encoding of polynomials and keys

Several steps of the encryption and decryption processes require the encoding of polynomials as (and the decoding from) octet strings. Additionally, in order to make the keys usable by public key infrastructures, they have to be encoded as well. In the following sections, we describe the encoding format of polynomials and keys.

*Binary polynomials.* Sparse binary polynomials are stored as a sorted array of the degrees of the monomials having a non-zero coefficient. The degrees are encoded in descending order. Each degree is an integer in the interval $[0, N-1]$, which is encoded as an octet string (byte array) of length $\lceil \log_{256}(N-1) \rceil$ in big endian byte order. Non-sparse binary polynomials are encoded using the BRE2OSP primitive described in Section 7.7.1 of IEEE P1363.1-D8.

*Product form polynomials.* A product form polynomial $f = f_1 f_2 + f_3$ consists of three sparse binary polynomials with the same number of non-zero coefficients. Product form polynomials are encoded as the concatenation of the encodings of $f_1$, $f_2$, and $f_3$ (see preceding paragraph).

*Other ring elements.* Since all ring computations are performed modulo $q$, ring elements are stored as their coefficient vector with coefficients reduced modulo $q$. The ring elements are encoded using the RE2OSP primitive described in Section 7.5.1 of IEEE P1363.1-D8.

*NTRUSVES keys.* NTRUSVES keys are encoded into ASN.1 structures in order to be used with public key infrastructures. The polynomials are encoded as octet strings as described in the preceding sections. The ASN.1 definitions of the NTRUSVES public and private key can be found in Appendix C.

### 4.7 Measurement results

In this section, we state the experimental results of the measurements of our NTRUSVES implementation. We provide time measurements as well as key sizes for all parameter sets proposed by IEEE P1363.1-D9. In Appendix D, we provide similar results for the RSA PKCS #1 v2.1 encryption scheme andcompare the complexity of the two encryption schemes based on these experiments.

The measurement results of our NTRUSVES implementation are summarized in table 3. Column "Parameter set" denotes the used parameter set. The first six parameter sets are binary parameter sets, the other six sets are product form parameter sets. Column "$k$" denotes the bit security level of NTRUSVES with the given parameter set. The estimates are taken from IEEE P1363.1-D9. Columns "$s_{privKey}$" and "$s_{pubKey}$" denote the size of the DER-encoded private key and public key ASN.1 structures, respectively (see Section 4.6). Columns "$t_{kpg}$", "$t_{enc}$", and "$t_{dec}$" denote the time measurement results for key pair generation, encryption, and decryption, respectively.

For the binary parameters sets, the pattern multiplication algorithm proposed in this paper has been used. For each parameter set, 500 key pairs were generated. For each key pair, 2000 random messages of random length between 1 and the maximal possible length were encrypted and decrypted.

The experiments were made using a computer equipped with a Pentium M 1.6 GHz CPU, 512 MB of RAM and running Microsoft Windows XP. The code was compiled with JDK 1.3 and run under JRE 1.6.

| *Parameter set* | $k$ | $s_{privKey}$ | $s_{pubKey}$ | $t_{kpg}$ | $t_{enc}$ | $t_{dec}$ |
|---|---|---|---|---|---|---|
| ees251ep6 | 80 | 218 bytes | 296 bytes | 15.0 ms | 0.2 ms | 0.2 ms |
| ees347ep2 | 112 | 529 bytes | 740 bytes | 26.7 ms | 0.3 ms | 0.4 ms |
| ees397ep1 | 128 | 595 bytes | 840 bytes | 34.8 ms | 0.3 ms | 0.5 ms |
| ees491ep1 | 160 | 723 bytes | 1028 bytes | 51.3 ms | 0.5 ms | 0.7 ms |
| ees587ep1 | 192 | 853 bytes | 1220 bytes | 71.7 ms | 0.6 ms | 1.0 ms |
| ees787ep1 | 256 | 1118 bytes | 1620 bytes | 127.8 ms | 1.0 ms | 1.5 ms |
| ees251ep7 | 80 | 194 bytes | 548 bytes | 14.9 ms | 0.1 ms | 0.2 ms |
| ees347ep3 | 112 | 462 bytes | 740 bytes | 27.7 ms | 0.2 ms | 0.3 ms |
| ees397ep2 | 128 | 518 bytes | 840 bytes | 35.6 ms | 0.2 ms | 0.3 ms |
| ees491ep2 | 160 | 630 bytes | 1028 bytes | 53.6 ms | 0.3 ms | 0.5 ms |
| ees587ep2 | 192 | 738 bytes | 1220 bytes | 74.8 ms | 0.5 ms | 0.7 ms |
| ees787ep2 | 256 | 969 bytes | 1620 bytes | 131.7 ms | 0.7 ms | 1.1 ms |

**Table 3.** NTRUSVES key sizes and time measurement results

## 5 Conclusion

In this paper, we present an efficient multiplication algorithm for NTRU which achieves an average-case speedup between 15% and 22% compared to the previously best-known results. Since the algorithm also is very space efficient, it is especially well-suited for resource-constrained devices. Since NTRU is currently undergoing an IEEE standardization process, it would be reasonable to incorporate our proposed algorithm into the upcoming standard. We also present a highly efficient implementation of NTRUSVES according to the IEEE P1363.1-D8 draft standard as part of a Java Cryptographic Service Provider. The implementation can be used with any application that uses the cryptographic framework provided by Java.

## References


1. K. Aoki, J. Franke, T. Kleinjung, A. K. Lenstra, and D. A. Osvik. A kilobit special number field sieve factorization. Cryptology ePrint Archive, Report 2007/205, 2007. Available at `http://eprint.iacr.org/2007/205`.
2. D. V. Bailey, D. Coffin, A. Elbirt, J. H. Silverman, and A. D. Woodbury. Ntru in constrained devices. In *Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems (CHES '01)*, volume 2162 of *Lecture Notes in Computer Science*, pages 262–272. Springer Verlag, 2001.
3. S. Cavallar, B. Dodson, A. K. Lenstra, W. M. Lioen, P. L. Montgomery, B. Murphy, H. te Riele, K. Aardal, J. Gilchrist, G. Guillerm, P. C. Leyland, J. Marchand, F. Morain, A. Muffett, C. Putnam, C. Putnam, and P. Zimmermann. Factorization of a 512-Bit RSA Modulus. In *Advances in Cryptology – EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 1–18. Springer Verlag, 2000.

4. T. Elgamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In *Advances in Cryptology – CRYPTO '84*, volume 196 of *Lecture Notes in Computer Science*, pages 10–18. Springer Verlag, 1985.
5. The FlexiProvider group at Technische Universität Darmstadt. *FlexiProvider, an open source Java Cryptographic Service Provider*, 2001–2007. Available at `http://www.flexiprovider.de/`.
6. The IEEE P1363 Study Group for Future Public-Key Cryptography Standards. Draft Standard for Public-Key Cryptographic Techniques Based on Hard Problems over Lattices. Available at `http://grouper.ieee.org/groups/1363/lattPK/draft.html`, January 2007.
7. J. Hoffstein, J. Pipher, and J. H. Silverman. NTRU: A Ring-Based Public Key Cryptosystem. In *Proceedings of the Third International Symposium on Algorithmic Number Theory*, volume 1423 of *Lecture Notes in Computer Science*, pages 267–288. Springer Verlag, 1998.
8. R. Housley, W. Polk, W. Ford, and D. Solo. RFC 3280 (Proposed Standard): Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. Available at `http://www.ietf.org/rfc/rfc3280.txt`, April 2002. Updated by RFCs 4325, 4630.
9. N. Howgrave-Graham, J. H. Silverman, A. Singer, and W. Whyte. NAEP: Provable Security in the Presence of Decryption Failures. Cryptology ePrint Archive, Report 2003/172, 2003. Available at `http://eprint.iacr.org/2003/172`.
10. N. Howgrave-Graham, J. H. Silverman, and W. Whyte. Choosing Parameter Sets for NTRUEncrypt with NAEP and SVES-3. In *Topics in Cryptology CT-RSA 2005*, volume 3376 of *Lecture Notes in Computer Science*, pages 118–135. Springer Verlag, 2005.
11. IEEE. IEEE Standard Specifications for Public-Key Cryptography, January 2000. See also "IEEE 1363 Amendment 1: Additional Techniques".
12. RSA Laboratories. PKCS #8: Private-Key Information Syntax Standard (version 1.2). Available at `http://www.rsa.com/rsalabs/node.asp?id=2130`, November 1993.
13. RSA Laboratories. PKCS #1: RSA Cryptography Standard (version 2.1). Available at `http://www.rsa.com/rsalabs/node.asp?id=2125`, June 2002.
14. M.-K. Lee1, J. W. Kim, J. E. Song, and K. Park. Sliding Window Method for NTRU. In *Proceedings of ACNS 2007*, volume 4521 of *Lecture Notes in Computer Science*, pages 432–442. Springer Verlag, 2007.
15. A. K. Lenstra and H. W. Lenstra, Jr., editors. *The development of the number field sieve*, volume 1554 of *Lecture Notes in Mathematics*. Springer Verlag, 1993.
16. A. K. Lenstra and E. R. Verheul. Selecting cryptographic key sizes. *Journal of Cryptology*, 14(4):255–293, 2001.
17. National Institute of Standards and Technology (NIST) Computer Security Resource Center (CSRC). SP 800-57 Part 1, Recommendation for Key Management – Part 1: General (Revised). Available at `http://csrc.nist.gov/CryptoToolkit/tkkeymgmt.html`, March 2007.
18. P. W. Shor. Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In *Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science (FOCS 1994)*, pages 124–134. IEEE Computer Society Press, 1994.
19. Sun Microsystems. *The Java Cryptography Architecture API Specification & Reference*, 2002. Available at `http://java.sun.com/j2se/1.4.2/docs/guide/security/CryptoSpec.html`.

20. Sun Microsystems. *The Java Cryptography Extension (JCE) Reference Guide*, 2002. Available at `http://java.sun.com/j2se/1.4.2/docs/guide/security/jce/JCERefGuide.html`.

## A Proof of Lemma 1

*Proof.* Let $B = \{i \mid b_i \neq 0\}$ be the set of indices of $b$'s non-zero coefficients. Since $b \in D(d)$, it holds that $|B| = d$. Rewrite the product and use the triangle inequality to obtain

$$\|br\|_\infty = \left\| \sum_{i \in B} r(X) X^i \right\|_\infty \leq \sum_{i \in B} \left\| r(X) X^i \right\|_\infty$$

Note that the max-norm of $r(X)$ on $R$ is not changed by a multiplication with powers of $X$ since this multiplication corresponds to a rotation of the coefficients. Conclude that

$$\|br\|_\infty \leq \sum_{i \in B} \left\| r(X) X^i \right\|_\infty = \sum_{i \in B} \|r(X)\|_\infty = d \, \|r\|_\infty .$$

□

## B NAEP/SVES-3

The scheme uses two hash functions $G$ and $H$. Fix the maximal message bit length *maxLen* and the bit length *bLen* of some random strings. Precompute the internal message bit length

$$nLen := bLen + (\log_2(maxLen) + 1) + maxLen.$$

*Encryption (see Figure 2).* In order to encrypt a message $M$, compute its bit length *MLen* and choose a random string $b$ of length *bLen*. Compute a blinding polynomial $r = G(b||M||ID)$, where *ID* is a number that uniquely identifies the used parameter set.

Pad the message as $(b||MLen||M||00\ldots)$ to obtain a string $M$ of the predefined bit length *nLen*. Compute the exclusive-or of $M$ with $H(rh)$, the image of the product of the blinding polynomial and the public key under the second hash function $H$ to obtain $m$. Encrypt $m$ using the NTRU encryption primitive as described in Section 2.1.

*Decryption (see Figure 3).* Decrypt a ciphertext $e$ with the NTRU decryption primitive as described in Section 2.1 into a polynomial $m$. Compute the difference $rh := e - m$ and the exclusive-or of $e$ with $rh$ to obtain a bit string of length *nLen*. Interpret this bit string as $(b'||MLen'||M'||trunc)$. Check that *trunc* consists only of zeroes and that $MLen'$ is the bit length of $M'$. Compute $r' = G(b'||M'||ID)$ and check whether $r'h$ equals $rh$ which was computed earlier. If all checks are positive, return $M$ as the decrypted message.

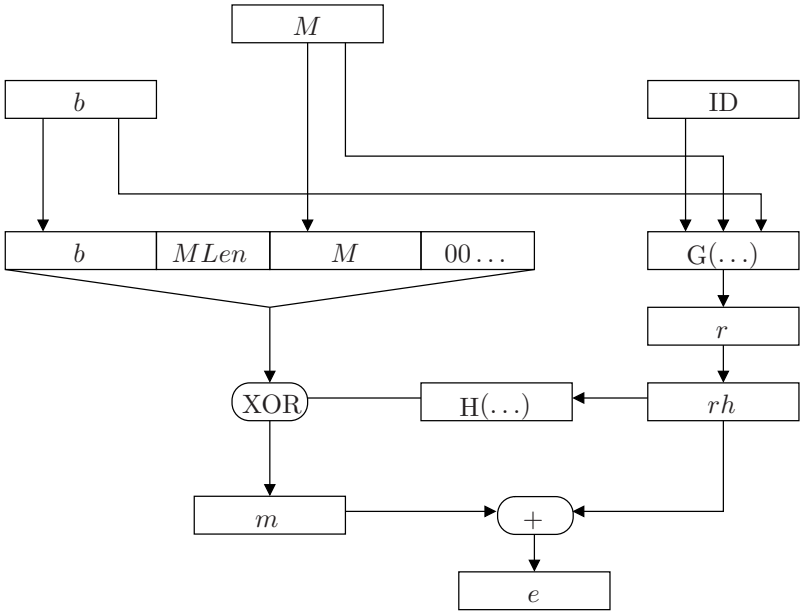


**Fig. 2.** SVES-3 encryption

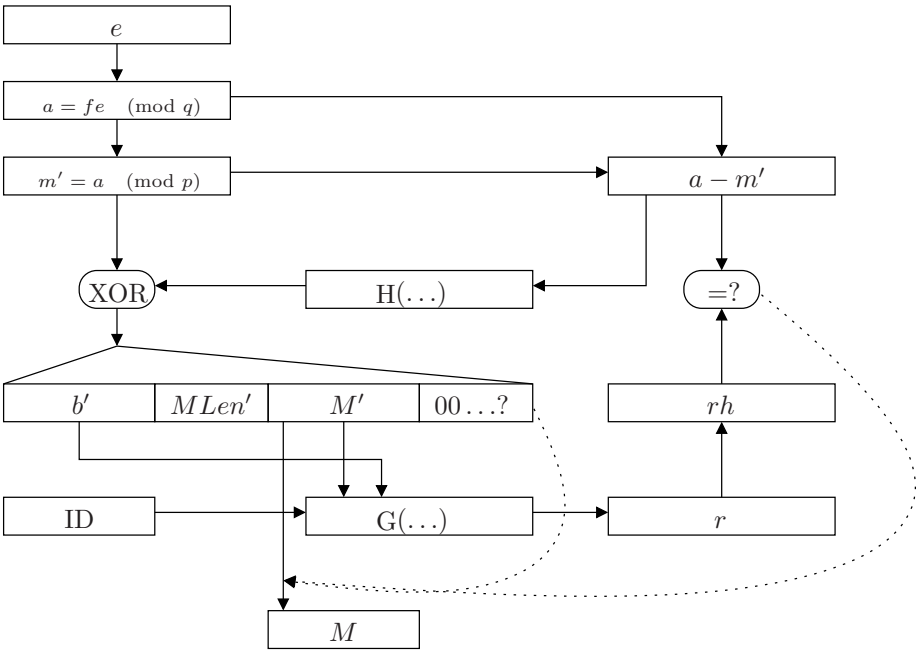


**Fig. 3.** SVES-3 decryption

## C ASN.1 structures

*Public key.* The NTRUSVES public key ASN.1 structure is

```
NTRUSVESPublicKey ::= SEQUENCE {
  paramName     IA5STRING      -- name of the parameter set
  encH          OCTET STRING   -- encoded polynomial h
}
```

The public key structure is embedded into a SubjectPublicKeyInfo structure as defined in RFC 3280 [8].

*Private key.* The NTRUSVES private key ASN.1 structure is

```
NTRUSVESPrivateKey ::= SEQUENCE {
  paramName     IA5STRING      -- name of the parameter set
  encF          OCTET STRING   -- encoded polynomial F
  encG          OCTET STRING   -- encoded polynomial g
}
```

The private key structure is embedded into a PrivateKeyInfo structure as defined in PKCS #8 [12].

## D RSA PKCS #1 v2.1 measurement results and comparison

In this section, we state the results of the measurements of our RSA PKCS #1 v2.1 implementation. The implementation is part of the Java Cryptographic Service Provider FlexiProvider [5]. The implementation uses the built-in modular arithmetic of Java (class `BigInteger`). The results are summarized in table 4.

Column "Key size" denotes the bit size of the modulus. Column "$k$" denotes the bit security level of RSA for the given key size. The estimates are taken from the NIST Key Management Guideline [17]. Columns "$s_{privKey}$" and "$s_{pubKey}$" denote the size of the DER-encoded private key and public key ASN.1 structures, respectively (see Section 4.6). Columns "$t_{kpg}$", "$t_{enc}$", and "$t_{dec}$" denote the time measurement results for key pair generation, encryption, and decryption, respectively.

For each key size, 20 key pairs were generated. The public exponent was chosen as $e = 2^{16} + 1$ for all key sizes and key pairs. For each key pair, 1000 random messages of random length between 1 and the maximal possible length were encrypted and decrypted.

| *Key size* | $k$ | $s_{privKey}$ | $s_{pubKey}$ | $t_{kpg}$ | $t_{enc}$ | $t_{dec}$ |
|---|---|---|---|---|---|---|
| 1024 | 80 | 634 bytes | 162 bytes | 0.9 s | 0.7 ms | 13.2 ms |
| 2048 | 112 | 1218 bytes | 194 bytes | 6.8 s | 2.7 ms | 91.7 ms |
| 3072 | 128 | 1794 bytes | 422 bytes | 27.3 s | 5.9 ms | 294.4 ms |
| 4096 | 144 | 2374 bytes | 550 bytes | 104.1 s | 10.3 ms | 682.5 ms |

**Table 4.** RSA PKCS #1 v2.1 key sizes and time measurement results

**Comparison** The measurement results given in Section 4.7 show that the NTRUSVES key pair generation, encryption and decryption operations are substantially faster than their RSA counterparts for the same security level. This is true also for larger security parameters because the asymptotic complexity of NTRUSVES grows slower in terms of the security parameter than the complexity of RSA.

For the same security level, the size of NTRUSVES private keys is about 1/3 of the size of RSA private keys. NTRUSVES public keys are about twice as large as RSA public keys.