Sliding Window Method for NTRU*

Mun-Kyu Lee[1,**], Jung Woo Kim[2], Jeong Eun Song[1], and Kunsoo Park[2]

[1] School of Computer Science and Engineering,
Inha University, Incheon 402-751, Korea
mklee@inha.ac.kr
[2] School of Computer Science and Engineering,
Seoul National University, Seoul 151-742, Korea

Abstract. The NTRU cryptosystem is a ring-based public key system using hard problems over lattices. There has been an extensive research on efficient implementation of NTRU operations, including recent results such as Bailey et al.'s software implementation over a resource-constrained device and Gaubatz et al.'s hardware implementation using only 3,000 gates. In this paper, we present a new algorithm to improve further the performance of NTRU. We speed up the encryption and decryption operations of NTRU up to 32% using some temporary memory, and if we can use precomputation, then the speed-up becomes up to 37%. Our method is based on the observation that specific sub-operations are repeated frequently in the underlying polynomial operations of NTRU.

1 Introduction

The NTRU cryptosystem [1] is a public key cryptosystem over polynomial rings, whose security is based on hard problems over lattices. After the introduction of NTRU encryption, a digital signature scheme using NTRU lattices, which is called NTRUsign [2], was also proposed. Since Coppersmith and Shamir [3] presented an attack against NTRU using lattice basis reduction algorithms, there have been various attempts to break NTRUencrypt [4] and NTRUsign [5,6,7]. However, none of these attacks revealed any significant weakness in the lattice problems used for NTRU [8].

On the other hand, there has been an extensive research on the efficient implementation of NTRU. Hoffstein and Silverman [9,10] proposed to use special forms of polynomials to reduce the amount of computation in NTRU while preserving its security, and Bailey et al. [11] showed that NTRU can be efficiently implemented over resource-constrained devices. Recently, Gaubatz, Kaps and Sunar [12] presented a hardware implementation of NTRU using no more than 3,000 gates, showing it is possible to use public key cryptography on sensor nodes. Now NTRU is being considered for the IEEE P1363.1 standard [13].

* This work was supported by grant No.R01-2006-000-10957-0 from the Basic Research Program of the Korea Science & Engineering Foundation.

** Corresponding author.

J. Katz and M. Yung (Eds.): ACNS 2007, LNCS 4521, pp. 432–442, 2007.

In this paper, we present a method to improve further the performance of NTRU. We speed up the encryption and decryption operations of NTRU by 25 to 37%, based on the observation that specific patterns are repeated frequently in a convolution operation, which is a dominant polynomial operation of NTRU. Our contributions are as follows:

- We propose an efficient method to find such patterns, which resembles the sliding window method for exponentiation. Hence we name our method a *sliding window method for NTRU*. We show that our method is optimal in the sense that it can find the maximum number of these patterns within a given window size.
- We propose a new convolution algorithm that improves on the algorithm given in [11]. According to our experiments, the new algorithm accelerates the encryption and decryption operations of NTRU up to 32% using some temporary memory. If we can use off-line precomputation, then the speed-up becomes up to 37%.

2 Preliminaries

2.1 Convolution

Let Z be the set of integers. The polynomial ring over Z, denoted by $Z[X]$, is the set of all polynomials with coefficients in Z. We work in the quotient ring $R = Z[X]/(X^N - 1)$. An element $a \in R$ can be written as a polynomial or a vector,

$$a(X) = \sum_{i=0}^{N-1} a_i X^i = [a_0, a_1, \ldots, a_{N-1}].$$

Then multiplication of $a \in R$ and $b \in R$ can be represented as the convolution product c, which is given by $c(X) = a(X) * b(X)$ with

$$c_k = \sum_{i=0}^{k} a_i b_{k-i} + \sum_{i=k+1}^{N-1} a_i b_{N+k-i} = \sum_{i+j \equiv k \pmod N} a_i b_j,$$

since $X^N \equiv 1 \bmod (X^N - 1)$.

In principle, this operation requires N^2 integer multiplications. However, for a typical product used by NTRU, either a or b has small coefficients, so the computation of $a * b$ can be done very fast.

2.2 The NTRU Public-Key Cryptosystem

In this section, we briefly review the NTRU cryptosystem. While there are several variants of NTRU, an improved version given in [9,11] can be described as follows:

- NTRU has three public parameters (N, p, q), where $gcd(p, q) = 1$ and $p \ll q$.
- Coefficients of polynomials are reduced mod p or q.
- The inverse of polynomial f mod q, denoted by f^{-1} mod q, is defined as the polynomial satisfying $f * f^{-1} \equiv 1 \bmod q$.

The working draft of IEEE P1363.1 standard [13] presents a few typical parameter sets for NTRU, one of which is $(N, p, q) = (251, 2, 197)$.

Key Generation. Randomly choose polynomials $F, g \in R$ with small coefficients. Then compute $f := 1 + pF$ and $h := pf^{-1} * g \bmod q$, where mod q means that every coefficient in a polynomial is reduced mod q. The private key is the polynomial f and the public key is the polynomial h.

Encryption. Let m be the polynomial representing a message. Then randomly choose a polynomial r of degree $N - 1$ with small coefficients, and compute the ciphertext $e := r * h + m \bmod q$.

Decryption. In order to decrypt e, first compute $a := e * f \bmod q$, choosing the coefficients of a to satisfy $A \leq a_i < A + q$. The value of A is fixed and is determined by a simple formula depending on the other parameters. Then recover the plaintext m as $m := a \bmod p$.

Why Decryption Works. The polynomial a satisfies

$$\begin{aligned} a &\equiv e * f \bmod q \\ &\equiv (r * h + m) * f \bmod q \qquad (\text{since } e \equiv r * h + m) \\ &\equiv pr * g + m * f \bmod q \quad (\text{since } h * f \equiv pg * f^{-1} * f \equiv pg) \end{aligned}$$

Consider the last polynomial $pr * g + m * f$. By an appropriate choice of parameters, one can adjust its coefficients to lie in an interval of length less than q. Hence we can recover

$$a = pr * g + m * f = pr * g + m * (1 + pF)$$

exactly, not merely modulo q. In other words, $m \equiv a \bmod p$.

2.3 Fast Convolution

The most time consuming part of NTRU encryption is computation of the convolution product $r(X) * h(X) \bmod q$. Similarly, the most time consuming part of NTRU decryption is computation of $e(X) * f(X) \bmod q$, and thus $e(X) * F(X) \bmod q$, since $e(X) * f(X) \equiv e(X) + pe(X) * F(X)$.

Note that while the coefficients in polynomials $h(X)$ and $e(X)$ are almost randomly distributed modulo q, we can control the forms of $r(X)$ and $F(X)$. Thus, $r(X)$ and $F(X)$ are usually selected to have binary coefficients, i.e., 0 or 1, so that coefficients may be computed without any multiplication. For example, if $r(X)$ is a binary polynomial with Hamming weight $HW(r)$, i.e., with $HW(r)$ ones, computation of the product $r(X) * h(X) \bmod q$ requires approximately $HW(r) \times N$ operations, where each operation is an addition plus a reduction modulo q. Therefore, if we can use $r(X)$ and $F(X)$ with low Hamming weights, the encryption and decryption procedures become very efficient. In [13], appropriate values for $HW(r)$ and $HW(F)$ are given according to the choice of public parameters (N, p, q).[1]

[1] Note that too small values of $HW(r)$ and $HW(F)$ may compromise security, since the sizes of spaces for r and f become very small.

Algorithm 1. Fast Convolution Algorithm (reproduced from [11])

Input: b an array of d locations for '1' representing the polynomial $a(X)$; $c(X)$ the polynomial; N the number of coefficients in $a(X)$, $c(X)$.
Output: t the array where $t(X) = a(X) * c(X)$.

```
 1: for 0 ≤ j < 2N do
 2:     t_j ← 0
 3: end for
 4: for 0 ≤ j < d do
 5:     for 0 ≤ k < N do
 6:         t_{k+b[j]} ← t_{k+b[j]} + c_k
 7:     end for
 8: end for
 9: for 0 ≤ j < N do
10:     t_j ← (t_j + t_{j+N}) mod q
11: end for
```

Bailey et al. [11] presents an efficient convolution algorithm under the assumption that one of the two input polynomials has binary coefficients. Algorithm 1 shows its simplified form, where $c(X) \in R$ is a general polynomial and $a(X) \in R$ is a binary polynomial with $HW(a) = d$. That is, $a(X)$ represents $r(X)$ and $F(X)$ in NTRU.

In Algorithm 1, line 6 repeats dN times and requires two additions each time, i.e., one for the addition of c_k and the other for the computation of index $k+b[j]$. On the other hand, line 10 repeats N times and it requires two additions and one modular reduction each time. Therefore the total number of operations of Algorithm 1 is exactly $2(d+1)N$ additions and N modular reductions.

3 Sliding Window Method for NTRU

The speed of a convolution operation, and thus the performance of NTRU encryption and decryption, can be improved significantly if some memory is available. In this section, we show the motivation for our work, and give an improved convolution algorithm, which we call the *sliding window method for NTRU*. The new algorithm is based on the observation that for a binary polynomial $a(X) \in R$ which is produced by randomly selecting $HW(a)$ ones out of N possible positions, the distribution of ones has some desirable properties.

3.1 Basic Idea

We begin by examining the structure of a convolution operation. Note that multiplication of $a \in R$ and $c \in R$ can be represented as the convolution product $t \in R$:

$$t_k = \sum_{i=0}^{k} a_i c_{k-i} + \sum_{i=k+1}^{N-1} a_i c_{N+k-i} = \sum_{i+j \equiv k \pmod N} a_i c_j.$$

Because $t \in R$ can also be written as a vector, this operation can be rewritten as the following matrix form:

$$t(X) = a(X) * c(X) = \begin{pmatrix} a_0 & a_{N-1} & a_{N-2} & \cdots & a_2 & a_1 \\ a_1 & a_0 & a_{N-1} & \cdots & a_3 & a_2 \\ a_2 & a_1 & a_0 & \cdots & a_4 & a_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{N-1} & a_{N-2} & a_{N-3} & \cdots & a_1 & a_0 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{N-1} \end{pmatrix}.$$

We can observe that each row in the above $N \times N$ matrix is produced by rotating the previous row to right by one position.

Now we give a small example with a binary polynomial $a(X) = X + X^2 + X^5 + X^6 + X^8 + X^9$ with $N = 10, d = 6$. For simplicity, we will write a binary polynomial as a bit string throughout this section. Thus $a(X)$ will be written as 0110011011. Then $a(X) * c(X)$ can be written as

$$t(X) = a(X) * c(X) = \begin{pmatrix} 0\,1\,1\,0\,1\,1\,0\,0\,1\,1 \\ 1\,0\,1\,1\,0\,1\,1\,0\,0\,1 \\ 1\,1\,0\,1\,1\,0\,1\,1\,0\,0 \\ 0\,1\,1\,0\,1\,1\,0\,1\,1\,0 \\ 0\,0\,1\,1\,0\,1\,1\,0\,1\,1 \\ 1\,0\,0\,1\,1\,0\,1\,1\,0\,1 \\ 1\,1\,0\,0\,1\,1\,0\,1\,1\,0 \\ 0\,1\,1\,0\,0\,1\,1\,0\,1\,1 \\ 1\,0\,1\,1\,0\,0\,1\,1\,0\,1 \\ 1\,1\,0\,1\,1\,0\,0\,1\,1\,0 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \\ c_8 \\ c_9 \end{pmatrix}.$$

Hence t_0 will be computed as $t_0 = c_1 + c_2 + c_4 + c_5 + c_8 + c_9$, which requires six additions.[2] Among these additions, now we concentrate on the term $c_1 + c_2$. We can see that this term also occurs in the computation of t_3 and t_7. This is because the pattern '11' is repeated three times in the binary representation of $a(X)$. To be more precise, $c_1 + c_2$ in the computation of t_0, t_3, t_7 corresponds to $\underline{11}_3$, $\underline{11}_1$, $\underline{11}_2$ in $a(X) = 0\underline{11}_1 00\underline{11}_2 0\underline{11}_3$, respectively. Since the term $c_1 + c_2$ occurs three times, we can compute this term only once, store it in a look-up table, and reuse it when it is required, which can reduce the number of additions by two. This reduction can also be applied to other terms related to the pattern '11'. For example, the term $c_2 + c_3$ occurs in the computation of t_1, t_4 and t_8, the term $c_3 + c_4$ in t_2, t_5 and t_9, the term $c_4 + c_5$ in t_3, t_6 and t_0, and so on. Thus the overall savings by the pattern '11' becomes $2 \times N = 20$.

Note that the above idea can be applied to other patterns such as '101', '1001', '111', etc., if only we can find these patterns in the binary representation of the polynomial so that these patterns may not share '1's. The following lemma shows a general rule for the relation between pattern occurrences and the amount of computation.

[2] For the sake of convenience in explanation, we do not consider the cost for index computation and reduction mod q here.

Lemma 1. *If a pattern containing n '1's, occurs m times in the binary representation of a polynomial with N coefficients, then we can reduce the number of integer additions by $N(m-1)(n-1)$ at the cost of memory to store N intermediate integers.*

Proof. It is straightforward since the number of integer additions related to such a pattern is reduced to $N(m+n-1)$ from Nmn. □

Therefore, patterns $p_1, p_2, \ldots, p_l$ can be used to reduce the number of integer additions by

$$N \sum_{i=1}^{l} (m_i - 1)(n_i - 1), \tag{1}$$

where p_i contains n_i '1's and it occurs m_i times. (Note that either a pattern with a single '1' or a pattern that occurs just once does not introduce any speed-up.) Thus a method to maximize (1) is crucial for fast NTRU computation. For example, for a string 01101101100, using a pattern '11' such that 0<u>11</u>0<u>11</u>0<u>11</u>00 will provide more saving than using '101' such that 01<u>101</u><u>101</u>100.

3.2 Finding Patterns

In this subsection, we present an efficient pattern-finding algorithm and analyze its performance. Our algorithm is based on the following facts, which will be justified throughout this subsection:

- It is sufficient to consider only the patterns that have a few (or no) zeros between two ones, i.e., '11', '101', '1001', and so on.
- There is an efficient greedy method to find such patterns, i.e., we just scan the given bit string once, marking the positions of pattern occurrences. Actually, we can show this approach is optimal.

Now we examine the first claim. Lemma 2 gives a clue to the question, "which pattern do we have to try to find?" First, the *distance* between two bit positions is defined as the difference of their indices. For example, in a string 1001, the distance between two '1's is 3.

Lemma 2. *Consider a task that chooses a bit according to a distribution where the probability that 1 is selected is p. We repeat this task independently to choose coefficients of a binary polynomial. Let Z be the distance between two neighboring occurrences of 1's. Then $Pr[Z > d] = (1-p)^d$.*

Proof. First, fix a specific nonzero position. Since we assume the independence between coefficients, we can see that $Pr[Z = t] = (1-p)^{t-1}p$. Therefore, we obtain

$$Pr[Z > d] = \sum_{i=d}^{\infty} \{(1-p)^i p\} = \frac{(1-p)^d p}{1-(1-p)} = (1-p)^d.$$

□

According to [13], binary polynomials $F(X)$ and $r(X)$ are randomly selected such that dF and dr coefficients are equal to 1, respectively, and the remaining coefficients equal to 0. While this situation is not exactly the same as the

assumption in Lemma 2, the approximation $p \approx dF/N$ or $p \approx dr/N$ shows a similar behavior to a real distribution, as shown in Table 1. Although the values given in this table are experimental results according to the method of [13], they are almost the same as the values estimated from $Pr[Z = d] = (1-p)^{d-1}p$, where $p = dF/N = dr/N$.

Table 1. Distribution of distances d between two neighboring 1's in $F(X)$ and $r(X)$

parameter set	$(N, dF = dr)$	$d=1$	$d=2$	$d=3$	$d=4$	$d=5$	$d=6$
ees251ep6	$(251, 48)$	0.191	0.156	0.126	0.101	0.083	0.066
ees347ep2	$(347, 66)$	0.191	0.155	0.125	0.102	0.082	0.067
ees397ep1	$(397, 74)$	0.186	0.152	0.124	0.101	0.082	0.067
ees491ep1	$(491, 91)$	0.186	0.152	0.123	0.101	0.082	0.067
ees587ep1	$(587, 108)$	0.184	0.151	0.123	0.101	0.082	0.067
ees787ep1	$(787, 140)$	0.177	0.147	0.120	0.099	0.081	0.067

Lemma 2 and Table 1 show that for practical parameter sets given in [13], the distance between two neighboring 1's is less than or equal to 5 with probability about 2/3. Thus we can expect that patterns such as '11', '101', '1001', '10001' and '100001' should cover a fairly large portion of $F(X)$ and $r(X)$, and provide considerable speed-up. We define these patterns as *simple patterns* with length 2 through 6, respectively.[3]

Now what we have to do is to develop an efficient method that finds the patterns '11', '101', '1001', and so on. We define the window size w, and find only the simple patterns that have up to $w-2$ zeros between two ones. Hence there could be *separated* ones that cannot be paired with neighboring ones. For notational convenience, let p_0 be a separated '1', and let p_1 = '11', p_2 = '101', p_3 = '1001', and so on.

We use a greedy algorithm that scans the input bit string from right to left just once. Algorithm 2 shows this algorithm. The reason why we start from right can be found in the behavior of Algorithm 1 that we use as a basis for our new convolution algorithm. That is, if we examine the construction of a specific t_j, we can see that the coefficients c_k accumulated to t_j are scanned from higher degrees to lower degrees, except one wrap-around at c_{N-1}.

We call our method a *sliding window method for NTRU*, since its bit-scanning behavior resembles that of the well-known sliding window method for exponentiation. The only differences are that there should be only up to two '1's in a single window, and the scanning is done in the opposite direction. It is easy to see that Algorithm 2 requires exactly N bit comparisons regardless of w, and the arrays $b_0, b_1, \ldots, b_{w-1}$ cover all positions of '1's in x.

[3] We need not consider patterns containing more than two '1's, since these patterns occur too rare. For example, for a fixed position of '1', the probability that the next two bits are all '1's, i.e., the probability that it makes a pattern '111' is $p^2 \approx 0.037$ for $N = 251, d = 48$.

Algorithm 2. Finding Simple Patterns with Window Size $\leq w$

Input: x a binary string; N length of x; w window size.
Output: b_0 an array representing the positions for separated '1's; $b_1 \dots b_{w-1}$ arrays representing the positions for $p_1, \dots, p_{w-1}$, respectively.

```
i ← N − 1
while i ≥ 0 do
  if x_i = 1 then
    if p_j = x_{i−j} ... x_i for some j in {1, 2, ..., w − 1}, then
      append i to b_j
      i ← i − (j + 1)
    else
      append i to b_0
      i ← i − w
    end if
  else
    i ← i − 1
  end if
end while
```

The following example illustrates our method for $w = 4, N = 28$:

$$\underline{1}0000\underline{1}\,\underline{1}00\underline{1}000\underline{1}00\underline{1}0\underline{1}0\underline{1}0\underline{11}0000\underline{1}. \tag{2}$$

The arrays b_0 through b_3 will be as follows:

$$b_0 = [27, 5, 0], b_1 = [23], b_2 = [20], b_3 = [16, 9]. \tag{3}$$

Theorem 1. *Algorithm 2 is an optimal algorithm to find the maximum number of simple patterns with length $\leq w$ in a bit string.*

Proof. Note that if there is an interval containing $w - 1$ or more consecutive zeros, then there cannot be any simple pattern with length $\leq w$ that overlaps this interval. Therefore these zero intervals partition the input string x into many segments, and the distance of two neighboring ones that belong to a same segment should always be less than $w-1$. Now we only have to show that within a single segment, the greedy algorithm is optimal, which is straightforward since we can find k simple patterns in a segment with $2k$ or $2k + 1$ ones. □

We remark that although Algorithm 2 is an optimal algorithm for a linear bit string, we may find one more simple pattern by merging the first and last '1's in the string if we can deal with a circular string.

3.3 New Convolution Algorithm

If the positions for simple patterns are given by Algorithm 2, then Algorithm 3 can be used to accelerate a convolution operation. Algorithm 3 can be viewed as an improved version of Algorithm 1, and it is a sliding window method using

Algorithm 3. Sliding Window Method for Fast Convolution

Input: $b_0, \ldots, b_{w-1}$, where b_i is an array of d_i positions of p_i from the polynomial $a(X)$; $c(X)$ the polynomial; N the number of coefficients in $a(X), c(X)$; w window size.

Output: t the array where $t(X) = a(X) * c(X)$.

1: **for** $0 \le j < w-1$ **do**
2: $\quad c_{j+N} \leftarrow c_j$
3: **end for**
4: **for** $1 \le i \le w-1$ **do**
5: $\quad$ **for** $0 \le j < N$ **do**
6: $\quad\quad T_i[j] \leftarrow c_j + c_{j+i}$
7: $\quad$ **end for**
8: **end for**
9: **for** $0 \le j < 2N$ **do**
10: $\quad t_j \leftarrow 0$
11: **end for**
12: **for** $0 \le j < d_0$ **do**
13: $\quad$ **for** $0 \le k < N$ **do**
14: $\quad\quad t_{k+b_0[j]} \leftarrow t_{k+b_0[j]} + c_k$
15: $\quad$ **end for**
16: **end for**
17: **for** $1 \le i \le w-1$ **do**
18: $\quad$ **for** $0 \le j < d_i$ **do**
19: $\quad\quad$ **for** $0 \le k < N$ **do**
20: $\quad\quad\quad t_{k+b_i[j]} \leftarrow t_{k+b_i[j]} + T_i[k]$
21: $\quad\quad$ **end for**
22: $\quad$ **end for**
23: **end for**
24: **for** $0 \le j < N$ **do**
25: $\quad t_j \leftarrow (t_j + t_{j+N}) \bmod q$
26: **end for**

precomputation tables. Lines 1 through 8 is the precomputation stage, and it requires $2(w-1)N + (w-1)$ integer additions including index computation. On the other hand, lines 9 through 26 is the convolution stage which requires $2(d_0 + d_1 + \cdots + d_{w-1} + 1)N$ additions and N modular reductions. Thus the total amount of computation of Algorithm 3 is $2(d_0 + d_1 + \cdots + d_{w-1} + w)N + (w-1)$ additions and N modular reductions, and the total amount of temporary memory for the precomputation table is $N(w-1)$ integers. Recall that Algorithm 1 requires $2(d+1)N$ additions and N modular reductions. Since $d_0 + d_1 + \cdots + d_{w-1} + w$ is much smaller than $d+1$, we can expect a significant speed-up by Algorithm 3. For example, if we use parameters $N = 251, d = dF = dr = 48$, then $d + 1 = 49$ and $d_0 + d_1 + \cdots + d_{w-1} + w = 42.399, 38.802, 36.748, 35.707, 35.171, 35.050$ for $w = 2, 3, 4, 5, 6, 7$, respectively, according to our experiment. We also see that large values for w are not so attractive since their amount of computation is almost the same as that of smaller w, while the amount of required memory is almost proportional to w. Hence we decide to fix $w = 5$.

4 Experimental Results

Now we present our experimental results for $w = 5$ with various parameter sets given in [13]. Table 2 shows the performance of NTRU encryption and decryption operations over a Pentium IV 3.0GHz CPU with 1.0GB RAM. We used C language and Microsoft Visual Studio .NET 1.0 environment. The third and fourth columns of this table represent the required time to perform an encryption and a decryption, respectively, using the original convolution algorithm (Algorithm 1). The fifth and seventh columns represent the required time when we use the improved algorithm. We can see that the new algorithm accelerates these operations by 25 to 32%. Note that this result is consistent with the values that we can estimate from the analysis given in Section 3.3 assuming a convolution operation consumes most of the computation time for encryption or decryption. That is, for $w = 5$, the gain is estimated as $(49 - 35.707)/49 \approx 27.13\%$.

Note that in some situation, the sender might know the identity of the recipient in advance, or she might send messages frequently to the same recipient. In this case, information related to the recipient's public key can be preprocessed. By setting $a(X) \leftarrow r(X)$ and $c(X) \leftarrow h(X)$, the values $T_i[j]$ in Algorithm 3 can be precomputed, i.e., lines 1 through 8 can be performed off-line. In this case, $T_i[j]$'s are not any more in a temporary memory, but they should be stored in a precomputation table. By this precomputation, we can further reduce the amount of on-line computation, which is given in the sixth column of Table 2. Now the performance gain over Algorithm 1 becomes 33 to 37%. These values are also consistent with the estimation from Section 3.3, i.e., $(49 - 35.707 + 5 - 1)/49 \approx 35.29\%$.

Table 2. Timings for various NTRU operations with $w = 5$ (μsec)

parameter set	$(N, p, q, dF = dr)$	Using Alg. 1		Using Alg. 3			
		Enc.	Dec.	Enc.1	Enc.2	Dec.	Memory†
ees251ep6	$(251, 2, 197, 48)$	1043	1045	766	683	783	1004
ees347ep2	$(347, 2, 269, 66)$	1937	2001	1380	1260	1392	1388
ees397ep1	$(397, 2, 307, 74)$	2510	2523	1783	1604	1796	1588
ees491ep1	$(491, 2, 367, 91)$	3760	3796	2649	2530	2650	1964
ees587ep1	$(587, 2, 439, 108)$	5327	5379	3685	3517	3742	2348
ees787ep1	$(787, 2, 587, 140)$	9343	9419	6420	6107	6408	3148

†Number of integers to be stored, i.e., $N(w - 1)$

5 Discussion

We proposed a method to speed up NTRU operations by reusing sub-operations that appear frequently. Our experiments show that several kilobytes of memory is sufficient to accelerate NTRU encryption and decryption by 25 to 37%.

The IEEE draft standard [13] proposes to use two classes of polynomials for r or F. The first one is to use binary polynomials with predefined Hamming weight, which is explained in Section 2.3. The second one is to use a

product-form polynomial, i.e., $r = r_1 * r_2 + r_3$ or $F = F_1 * F_2 + F_3$, where $r_1, r_2, r_3, F_1, F_2, F_3$ are binary polynomials with much smaller Hamming weight than those of r and F in the first class. We remark that our sliding window method is applied only to the first case, and it is evaluated to be still slower by about 20–30% than the second case without the window method, according to our analysis. Therefore, it could be an interesting research topic to improve the convolution algorithm for product-form polynomials.

We performed the above experiments and comparison on a Pentium IV processor. However, note that speed-ups are more critical on resource-constrained devices such as a Mica-Z mote with ATmega128 microcontroller. Therefore, implementation over such devices is necessary for complete analysis of our algorithm.

References

1. Hoffstein, J., Pipher, J., Silverman, J.: NTRU: A ring-based public key cryptosystem. In: Algorithmic Number Theory – ANTS III. Volume 1423 of LNCS., Springer (1998) 267–288
2. Hoffstein, J., Howgrave-Graham, N., Pipher, J., Silverman, J., Whyte, W.: NTRUSIGN: Digital signatures using the NTRU lattice. In: CT-RSA 2003. Volume 2612 of LNCS., Springer (2003) 122–140
3. Coppersmith, D., Shamir, A.: Lattice attacks on NTRU. In: Eurocrypt 97. Volume 1233 of LNCS., Springer (1997) 52–61
4. Howgrave-Graham, N., Nguyen, P., Pointcheval, D., Proos, J., Silverman, J., Singer, A., Whyte, W.: The impact of decryption failures on the security of NTRU encryption. In: Crypto 2003. Volume 2729 of LNCS., Springer (2003) 226–246
5. Gentry, C., Jonsson, J., Stern, J., Szydlo, M.: Cryptanalysis of the NTRU signature scheme (NSS) from Eurocrypt 2001. In: Asiacrypt 2001. Volume 2248 of LNCS., Springer (2001) 1–20
6. Gentry, C., Szydlo, M.: Cryptanalysis of the revised NTRU signature scheme. In: Eurocrypt 2002. Volume 2332 of LNCS., Springer (2002) 299–320
7. Nguyen, P., Regev, O.: Learning a parallelepiped: cryptanalysis of GGH and NTRU signatures. In: Eurocrypt 2006. Volume 4004 of LNCS., Springer (2006) 271–288
8. Gama, N., Howgrave-Graham, N., Nguyen, P.: Symplectic lattice reduction and NTRU. In: Eurocrypt 2006. Volume 4004 of LNCS., Springer (2006) 233–253
9. Hoffstein, J., Silverman, J.: Optimizations for NTRU. In: Proceedings of Public-Key Cryptography and Computational Number Theory. (2000)
10. Hoffstein, J., Silverman, J.: Random small Hamming weight products with applications to cryptography. Discrete Applied Mathematics **130** (2003) 37–49
11. Bailey, D.V., Coffin, D., Elbirt, A., Silverman, J.H., Woodbury, A.D.: NTRU in constrained devices. In: Cryptographic Hardware and Embedded Systems – CHES 2001. Volume 2162 of LNCS., Springer (2001) 262–272
12. Gaubatz, G., Kaps, J.P., Sunar, B.: Public key cryptography in sensor networks-revisited. In: ESAS 2004. Volume 3313 of LNCS., Springer (2004) 2–18
13. IEEE P1363.1/D8: Draft standard for public-key cryptographic techniques based on hard problems over lattices (2006)