Journal of Cryptographic Engineering (2025) 15:16
https://doi.org/10.1007/s13389-025-00380-w

RESEARCH ARTICLE

Fast polynomial inversion algorithms for the post-quantum cryptography

Eun-Young Seo[1] · Young-Sik Kim[1] · Jong-Seon No[2]

Received: 14 October 2022 / Accepted: 11 August 2025

Abstract
Several cryptosystems suggested for the post-quantum cryptography candidates, including Falcon, BIKE, and NTRU, are defined in a polynomial ring. They must derive the inverse polynomial of any given polynomial for generating a public key. This process consumes considerable processing time; therefore, reducing the time to derive the inverse polynomial significantly improves many cryptosystems' performance. In this paper, we primarily suggest two polynomial inversion algorithms, combined-variable-time and combined-constant-time algorithms, based on the modification of the extended Euclidean algorithm. The combined-variable-time algorithm shows how to calculate the inverse polynomial by introducing the combined matrix fast, which is generated by merging several steps of the polynomial operations. In cryptosystems, to defend against side-channel attacks, the implementation with constant running time is essential in preventing information leakage. Thus, we propose the combined-constant-time polynomial inversion algorithm, which expends less running time than the conventional NTRU inversion algorithm. For binary polynomial inversion, the proposed combined-variable-time algorithm is 1.95 times faster than the variable-time algorithm used in the previous NTRU (Silverman Almost inverses and fast NTRU key creation, NTRU Tech Report, no. 014v1, Mar. 15, 1999), and the combined-constant-time algorithms are 1.43 times faster than the reference constant-time algorithms submitted to round 3 of the NIST PQC standardization, respectively. For ternary polynomial inversion, the proposed combined-variable-time and combined-constant-time algorithms are 1.59 and 1.29 times faster than the corresponding reference algorithms.

Keywords Key encapsulation mechanism (KEM) · Lattice-based cryptography · NTRU · Inverse polynomial · Post-quantum cryptography (PQC) · Public key encryption · Side-channel attack

1 Introduction

The National Institute of Standards and Technology (NIST) recently announced the first post-quantum cryptography (PQC) standard algorithms and then proceeded to the fourth round of standardization of PQC for developing cryptosystems that can be secure, even in the quantum computing era. To implement current standard public key cryptosystems like RSA (Rivest-Shamir-Adleman), as described in [2], modular operations involving very large numbers are required for integer multiplication and inversion. However, many PQC candidates are often defined on the polynomial ring. Consequently, they frequently involve calculating the multiplication and inversion of polynomials within a specific ring. Among them, the NTRU [3] is a key encapsulation mechanism (KEM) cryptography that has garnered significant attention because of its short key length and relatively simple implementation. The most time-consuming task in the NTRU is to derive the inverse polynomial for key generation. In addition, the FALCON [4], one of the post-quantum digital signature schemes selected by NIST, is created based on the NTRU polynomials and needs to find the inverse polynomial. Moreover, the BIKE [5] and the NTRU Prime [6], alternative post-quantum KEM algorithms considered in the

Y.-S. Kim: These authors contributed equally to this work.

✉ Young-Sik Kim
ysk@dgist.ac.kr

Eun-Young Seo
eyseo_dgist@dgist.ac.kr

Jong-Seon No
jsno@snu.ac.kr

1 Department of Electrical Engineering and Computer Science, DGIST, Daegu, Republic of Korea

2 The Department of Electrical and Computer Engineering, INMC, Seoul National University, Seoul, Republic of Korea

Published online: 22 August 2025

third round of NIST PQC Standardization, also include the inverse polynomial calculation process. This paper proposes new efficient algorithms to calculate the inverse polynomial with variable-time and constant-time versions.

For two positive integers a and b that are coprime, the multiplicative inverse of b modulo a can be computed using either the extended Euclidean algorithm [7–9] or Fermat's Little Theorem [10, 11]. In this work, we focus primarily on the extended Euclidean algorithm, as our goal is to find the inverse polynomial in a polynomial ring. Its direct implementation requires non-trivial division; hence, Montgomery modular multiplication is utilized for efficient hardware implementations [12–14]. By changing integers to polynomials, the extended Euclidean algorithm can be modified to an algorithm that determines the inverse of a polynomial for a given modulus polynomial. Implementing hardware for determining the inverse of the polynomial defined over the binary field was proposed in [15]. Schroeppel [16] introduced the Almost Inverse algorithm, which Silverman [1] applied to NTRU. Additionally, several fast algorithms have been proposed for obtaining the inverse polynomial in NTRU by adapting the extended Euclidean algorithm [8, 17]. In [18], a constant-time version of the Almost Inverse algorithm was derived, and an hardware implementation was proposed to execute instructions that can be performed simultaneously in parallel, aiming to increase the speed of the algorithm. One way to modify it is to utilize a cyclic matrix constructed from the coefficients of a given polynomial. Since coefficients of the polynomial resulting from the multiplication of two polynomials in the polynomial ring can be derived from the convolution of their coefficients, computing the inverse of the cyclic matrix of the polynomial makes it possible to calculate its inverse polynomial. In [19, 20], and [21], Gaussian elimination is the primary tool for deriving an inverse matrix, and the determinant of the matrix is calculated to check the existence of the inverse. Also, the Number Theoretic Transform (NTT) [22] is a powerful tool that calculates polynomial multiplication and division in quasi-polynomial complexity. Since the FALCON [4] satisfies the modulo condition that is required by the NTT, the polynomial division in the public key generation is implemented by the NTT.

In implementing each component of a cryptosystem, protecting the secret keys and messages from side-channel attacks is as crucial as processing with high speed. Secret parameters can be exposed to the attacker when the processing time is not constant because branch operations depend on the input data. NIST also has recommended providing constant-time implementations of the suggested post-quantum crypto algorithms as an elementary countermeasure for side-channel analysis. Therefore, techniques related to constant-time implementation have garnered attention from modular inversion of integers [23] to modular inversion of polynomials [17]. The algorithm for determining an inverse polynomial in NTRU is also based on the constant-time implementation [1, 17, 24]. Because Bernstein and Yang's algorithm [17] is known as faster than any other previously suggested algorithms [25, 26], the NTRU inversion algorithm submitted to the NIST round 3 is implemented with [17]. Bernstein and Yang's algorithm [17] keeps track of constant terms of polynomials and reduces the degree of polynomials by applying a variant of the extended Euclidean algorithm. They formulated the constant-time inverse polynomial calculation algorithm, and the inverse polynomial was derived from fixed $2N + 1$ iterations, where N is the maximum degree of the polynomials. Recently, in [27], FPGA implementations of the BIKE are proposed, and Bernstein and Yang's algorithm [17] is applied to the polynomial inversion. The authors of [27] optimized the hardware to implement the inversion algorithm efficiently and used the same number of iterations with [17]. In addition, Drucker et al. [28] propose an optimized method for computing inverse polynomials in BIKE, which operates over polynomials with binary coefficients. Their method enhances the efficiency of an inversion technique structurally similar to the Itoh-Tsuji Inversion (ITI) algorithm [29].

This study proposes two algorithms that improve the efficiency of computing inverse polynomials through modifications to the extended Euclidean algorithm, with simulation results demonstrated within the NTRU cryptographic framework. The first algorithm focuses on speeding up the algorithm in cases where the calculation time of each inverse polynomial varies. The information leakage can be prevented even in variable-time algorithms by carefully implementing it without the branch operation and carrying out further dummy operations even after the polynomial inverse calculation is completed. The second algorithm runs in constant time and reduces the operation time compared to the original NTRU algorithm by simultaneously handling multiple coefficients. We analyze how the algorithms work fast by trading-off complexity and memory space.

This paper is organized as follows. In Sect. 2, basic preliminaries for the subsequent discussion are presented. In Sect. 3, we explain how to make the algorithm fast by tracking additional information and merging matrices by suggesting the combined-variable-time algorithm. In Sect. 4, we suggest a constant-time algorithm with $N+1$ iterations, which reduces running time effectively by combining several steps and consuming additional memory resources. In Sect. 5, we present the numerical simulation results for performance of both combined-variable-time and combined-constant-time algorithms. Finally, we conclude this paper in Sect. 6.

2 Preliminaries

2.1 Basic notations

Matrix is denoted by a boldfaced capital letter, for example, $\mathbf{A}$. The matrix multiplication is denoted by $\cdot$, similar to polynomial and integer multiplications. $\mathbf{Z}_p$ means a finite subset of integer $\mathbf{Z}$, generated by the modulo p operation such as $\mathbf{Z}_p = \{0, 1, 2, \cdots, p-1\}$. $\mathbf{Z}_p[x]$ denotes a set of polynomials with coefficients in $\mathbf{Z}_p$. $\mathbf{Z}_p[x]/g(x)$ is a subset of $\mathbf{Z}_p[x]$, and its elements are obtained by modulo $g(x)$ operation to a polynomial of $\mathbf{Z}_p[x]$. We define the degree of a polynomial as the highest order of the term with a non-zero coefficient. When a polynomial, $f(x) = \sum_{i=0}^{N} f_i \cdot x^i \in \mathbf{Z}_p[x]/g(x)$, has a non-zero coefficient f_N, the degree of $f(x)$ is N, denoted by $\deg f$. When the degree of $g(x)$ is N, the degree of an element of $\mathbf{Z}_p[x]/g(x)$ is less than N. For the given polynomials $f_1(x) = \sum_{i=0}^{N} a_i \cdot x^i$ and $f_2(x) = \sum_{i=0}^{N} b_i \cdot x^i$, addition is represented by $f_1(x) + f_2(x) = \sum_{i=0}^{N}(a_i + b_i) \cdot x^i$ and multiplication is represented by $f_1(x) \cdot f_2(x) = \sum_{j=0}^{N} \sum_{i=0}^{N}(a_i \cdot b_j) \cdot x^{i+j}$ with modulo p and $g(x)$ operations. For NTRU, $g(x)$ is defined as $g(x) = 1 + x + \ldots + x^N$. However, the proposed algorithm can be generally applied to calculate inverse polynomials when $g(x)$ and $f(x)$ have no common factor.

In the proposed algorithm, the 'swap operation' between two polynomials $f(x)$ and $g(x)$ means that $t(x) \leftarrow f(x)$, $f(x) \leftarrow g(x)$, and $g(x) \leftarrow t(x)$, in this order. That is, the notations of $f(x)$ and $g(x)$ are exchanged after the operation. We utilize bitwise operators such as ($|$, $\&$, $\bar{\cdot}$, $\oplus$), which denote (OR, AND, NOT, XOR) operations, respectively. For example, when we have $a = 1$ and $b = 0$, $(a \mid b)$ is one and $(a \& b)$ is zero. Furthermore, $\overline{a}$ is zero. These operators are applied to only the binary variables in this paper.

2.2 Variable-time inverse polynomial calculation algorithm in NTRU

For a polynomial, $f(x) \in \mathbf{Z}_p[x]/g(x)$, if an inverse polynomial of $f(x)$ exists in the given polynomial ring, it is denoted by $f^{-1}(x)$ and satisfies that

$$f(x) \cdot f^{-1}(x) = 1 \mod g(x).$$

Silverman proposed the inverse polynomial calculation algorithm as in Algorithm 1. For $g(x)$ and $f(x)$ in $\mathbf{Z}_p[x]/g(x)$ which have no common factor, we can find two polynomials $l_1(x), l_2(x) \in \mathbf{Z}_p[x]/g(x)$ such as

$$g(x) \cdot l_1(x) + f(x) \cdot l_2(x) = \gamma \mod g(x), \tag{1}$$

by using the polynomial version of the extended Euclidean algorithm. Here, $\gamma^{-1} \cdot l_2(x)$ corresponds to $f^{-1}(x) \mod g(x)$

Algorithm 1 Variable-time modular inverse of polynomial over $\mathbf{Z}_p$ [1]

```
Require: $f(x)$, a prime $p$
Ensure: $b(x) = f^{-1}(x)$ in $\mathbf{Z}_p[x]/g(x)$
 1: $k \leftarrow 0, b(x) = 0, c(x) = 1$
 2: while $f_0 = 0$ do
 3:     $f(x) \leftarrow f(x)/x, b(x) \leftarrow x \times b(x)$
 4: end while
 5: while true do
 6:     while $g_0 = 0$ do
 7:         $g(x) \leftarrow g(x)/x, c(x) \leftarrow x \times c(x), k \leftarrow k + 1$
 8:     end while
 9:     if $\deg(g) = 0$ then
10:         $b(x) \leftarrow b(x)/g_0 \pmod p$
11:         $b(x) \leftarrow x^{N-k} b(x) \pmod{g(x)}$
12:         break
13:     else
14:         if $\deg(g) < \deg(f)$ then
15:             swap $f$ and $g$ and swap $b$ and $c$
16:         end if
17:     end if
18:     $u \leftarrow g_0 f_0^{-1} \pmod p$
19:     $g(x) \leftarrow g(x) - u \times f(x) \pmod p$
20:     $b(x) \leftarrow b(x) - u \times c(x) \pmod p$
21: end while
```

because of $f(x) \cdot \gamma^{-1} \cdot l_2(x) = 1 \bmod g(x)$. Our goal is to find $l_2(x)$ satisfying (1), which can be done as follows [1]. Let $\mathbf{M} = \begin{pmatrix} l_1(x) & l_2(x) \\ * & * \end{pmatrix}$ be a matrix satisfying

$$\mathbf{M} \cdot \begin{pmatrix} g(x) \\ f(x) \end{pmatrix} = \begin{pmatrix} \gamma \\ * \end{pmatrix}, \tag{2}$$

where $*$ denotes any polynomial in $\mathbf{Z}_p[x]/g(x)$, and γ is a non-zero element of $\mathbf{Z}_p$; then, $f^{-1}(x)$ is equal to $\gamma^{-1} \cdot l_2(x)$. Thus, $l_2(x)$ can be extracted by multiplying $\mathbf{M}$ as

$$\mathbf{M} \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} l_2(x) \\ * \end{pmatrix}. \tag{3}$$

Then, we can notice that even if we exchange $g(x)$ and $f(x)$ in (2), we can obtain the same $f^{-1}(x)$ by exchanging zero and one in (3).

The role of the matrix $\mathbf{M}$ is to generate a polynomial of degree zero with multiplication and addition of two given polynomials, $f(x)$ and $g(x)$ as in Algorithm 1, [1, 26]. Following the concept of the extended Euclidean algorithm, we are required to generate a new lower-degree polynomial from the subtraction of two polynomials $g(x)$ and $f(x)$ multiplied with proper monomials, to determine the GCD of $g(x)$ and $f(x)$. This procedure can be expressed by matrix multiplication, where the matrices are chosen among the following;

1. $\begin{pmatrix} x^{-1} & 0 \\ 0 & 1 \end{pmatrix}$: Reduce polynomial degree until it has a non-zero constant (Step 4 in Algorithm 1).

2. $\begin{pmatrix} c_1 & c_2 \\ 0 & 1 \end{pmatrix}$: Update the first polynomial $g(x)$ by subtracting the second polynomial $f(x)$ where $c_1 = f_0$ and $c_2 = -g_0$ (Steps 14 and 15 in Algorithm 1).
3. $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$: Swap two polynomials $g(x)$ and $f(x)$ if $\deg g(x) < \deg f(x)$ (Step 11 in Algorithm 1).

According to Algorithm 1, we can apply the above three matrices depending on the degree and coefficients of two intermediate polynomials. The matrix multiplication of properly chosen matrices generates the polynomial inversion matrix, $\mathbf{M} = \mathbf{H}_r \cdot \ldots \cdot \mathbf{H}_2 \cdot \mathbf{H}_1$, where each $\mathbf{H}_i$ corresponds to one of the above three matrices for $1 \le i \le r$.

For example, the way of selecting $\mathbf{H}_i$ depends on the coefficients of polynomials, $f^{[i]}(x)$ and $g^{[i]}(x)$, where $f^{[i]}(x) = \sum_{j=0}^{N} f_j^{[i]} \cdot x^j$ and $g^{[i]}(x) = \sum_{j=0}^{N} g_j^{[i]} \cdot x^j$. If $\deg f^{[i]}(x) > \deg g^{[i]}(x)$, we can exchange $f^{[i]}(x)$ and $g^{[i]}(x)$ to each other. First, by comparing constant terms in each polynomial, we multiply the following matrices sequentially to the polynomials, $g^{[i]}(x)$ and $f^{[i]}(x)$, as follows:

$$\begin{pmatrix} x^{-1} & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} c_1^{[i]} & c_2^{[i]} \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} g^{[i]}(x) \\ f^{[i]}(x) \end{pmatrix} = \begin{pmatrix} g^{[i+1]}(x) \\ f^{[i+1]}(x) \end{pmatrix}.$$

Polynomial $g^{[i+1]}(x)$ is equal to $x^{-1} \cdot (f_0^{[i]} \cdot g^{[i]}(x) - g_0^{[i]} \cdot f^{[i]}(x))$, where $\deg g^{[i]}(x) > \deg g^{[i+1]}(x)$. However, $f^{[i+1]}(x)$ has the same degree as $f^{[i]}(x)$ because a non-zero constant multiplication does not alter the degree of the polynomial. If $\deg f^{[i+1]}(x) \ge \deg g^{[i+1]}(x)$, $g^{[i+1]}(x)$ swaps with $f^{[i+1]}(x)$ by multiplying the matrix $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$.

In this paper, we propose a new method of determining the matrix $\mathbf{M}$, which enables speeding up the calculation of an inverse polynomial by combining several steps.

2.3 Constant-time inverse polynomial calculation algorithm in NTRU

The NTRU inverse algorithm submitted to the third round of the NIST PQC standardization is a constant-time algorithm, which is based on the Bernstein and Yang's algorithm [17]. Algorithm 2 shows how to derive the inverse polynomial over $\mathbf{Z}_3[x]/g(x)$ in the optimized implementation of NTRU. The constant-time algorithm has fixed $2N+1$ iterations and reduces just one degree of one polynomial among two given polynomials at each iteration. Bernstein and Yang's paper introduced the divstep function that reduces the given polynomials, $g(x)$ and $f(x)$, such as

Algorithm 2 Constant-time inverse polynomial calculation algorithm in NTRU[17]

```
Require: $f(x) = \sum_{i=0}^{N} f_i \cdot x^i$, $g(x) = \sum_{i=0}^{N} g_i \cdot x^i$
Ensure: $v(x) = f^{-1}(x)$ in $\mathbf{Z}_3[x]/g(x)$
1: $v(x) = \sum_{i=0}^{N} v_i \cdot x^i = 0$, $w(x) = \sum_{i=0}^{N} w_i \cdot x^i = 1$, delta $\leftarrow 1$
2: for $i = 0, i \le 2N-1, i \leftarrow i+1$ do
3:     for $j = N, j > 0, j \leftarrow j-1$ do
4:         $v_j \leftarrow v_{j-1}$
5:     end for
6:     $v_0 \leftarrow 0$
7:     sign$\leftarrow 2 \cdot g_0 \cdot f_0 \mod 3$
8:     swap$\leftarrow$ MSB$((-\delta) \,\&\, (-f_0))$
9:     delta $\leftarrow$ delta $\oplus$ (swap $\&$(delta $\oplus$ $(-$delta$)$)) + 1
10: end for
11: for $j = 0, j \le N, j \leftarrow j+1$ do
12:     $f_j \leftarrow f_j \oplus$ (swap $\&(f_j \oplus g_j)$)
13:     $g_j \leftarrow g_j \oplus$ (swap $\&(f_j \oplus g_j)$)
14:     $v_j \leftarrow v_j \oplus$ (swap $\&(v_j \oplus w_j)$)
15:     $w_j \leftarrow w_j \oplus$ (swap $\&(v_j \oplus w_j)$)
16: end for
17: for $j = 0, j \le N, j \leftarrow j+1$ do
18:     $f_j \leftarrow f_j + (\text{sign} \cdot g_j) \mod 3$
19:     $w_j \leftarrow w_j + (\text{sign} \cdot v_j) \mod 3$
20: end for
21: for $j = 0, j \le N-1, j \leftarrow j+1$ do
22:     $f_j \leftarrow f_{j+1}$
23: end for
24: $f_N \leftarrow 0$
25: sign$\leftarrow g_0$
```

$$\text{divstep}(\delta, g(x), f(x)) = \begin{cases} (1-\delta, f(x), (f(0)\cdot g(x) - g(0)\cdot f(x))/x), & \text{if } \delta > 0 \text{ and } f(0) \neq 0 \\ (1+\delta, g(x), (g(0)\cdot f(x) - f(0) g(x))/x), & \text{otherwise,} \end{cases}$$

where the initial value of δ is equal to one, assuming that the degree of $g(x)$ is not smaller than the degree of $f(x)$. Iterative application of the divstep function derives the matrix, $\mathbf{M}$, in (2). For example, when $\delta > 0$ and $f(0)$ is not equal to zero, the reduced polynomials are generated by the matrix multiplication,

$$\begin{pmatrix} 0 & 1 \\ f(0)\cdot x^{-1} & -g(0)\cdot x^{-1} \end{pmatrix} \cdot \begin{pmatrix} g(x) \\ f(x) \end{pmatrix}.$$

Within $2N+1$ number of iterations, the right side matrix of (2) becomes $\begin{pmatrix} \gamma \\ 0 \end{pmatrix}$.

They compare constant terms of the given polynomials and generate a new polynomial similar to steps 14–15 in Algorithm 1. Unlike Algorithm 1, the constant-time algorithm runs while maintaining the number of iterations even though it obtains the inverse polynomial in the middle of the iterations. To ensure that executing additional iterations does not change the result, they store the information of decremented degrees and use it to swap the polynomials. Extending the algorithm, we suggest the fast constant-time algorithm that runs $N+1$ iterations by simultaneously processing constant and first-order terms in one iteration.

3 Fast variable-time inverse polynomial calculation algorithm

This section introduces a combined matrix to calculate the matrix **M** and explains how the combined-variable-time algorithm achieves the fast running time. Although the proposed algorithm spends a different amount of time calculating the inverse polynomial depending on the given polynomial, its running time is faster than the constant-time algorithm in the average sense, which helps apply the side-channel attack risk-free environment. This section comprises two parts; one introduces the combined matrix, and the other suggests Algorithm 4, which is derived by merging the combined matrices for upgrading performance.

3.1 Combined matrix

We propose generating new polynomials with the combined matrix from the given polynomials, which considers three steps simultaneously. The primary concept of the proposed algorithm is to derive a polynomial with a non-zero constant term by manipulating two given polynomials. The other polynomial is determined by the polynomial having a smaller degree among two given polynomials. Then, it becomes necessary to predict in advance the degree and coefficient of the polynomial obtained when we follow Algorithm 1. We describe the procedure of making the integrated multiplication matrix as follows.

0) To shift the coefficients to lower degree positions when the polynomials have zero constant terms, we multiply $\mathbf{M}_0$ with the initial input polynomials, $g^{[0]}(x)$ and $f^{[0]}(x)$. When we define B_g and B_f as the lowest degree with the non-zero coefficient of the polynomials $g^{[0]}(x)$ and $f^{[0]}(x)$, $\mathbf{M}_0$ is represented as

$$\mathbf{M}_0 = \begin{pmatrix} x^{-B_g} & 0 \\ 0 & x^{-B_f} \end{pmatrix}.$$

1) In Algorithm 1, we can consider three processes at the same time if we preset some parameters derived from the two intermediate polynomials $g^{[i]}(x)$ and $f^{[i]}(x)$. First, we define an indicator variable η_i to denote the degree information of intermediate polynomials $g^{[i]}(x)$ and $f^{[i]}(x)$ such as

$$\eta_i = \begin{cases} 1, & \text{if} \deg g^{[i]}(x) \geq \deg f^{[i]}(x) \\ 0, & \text{otherwise.} \end{cases} \tag{4}$$

When η_i is one, we swap polynomials $g^{[i]}(x)$ and $f^{[i]}(x)$, which can be represented as

$$\begin{pmatrix} 1-\eta_i & \eta_i \\ \eta_i & 1-\eta_i \end{pmatrix} \begin{pmatrix} g^{[i]}(x) \\ f^{[i]}(x) \end{pmatrix}.$$

2) Second, we can find B_i which is the lowest order of $f_0^{[i]} g^{[i]}(x) - g_0^{[i]} f^{[i]}(x)$ and multiply $\begin{pmatrix} U_i & 0 \\ 0 & 1 \end{pmatrix}$ to intermediate polynomials, where $U_i = 1/x^{-B_i}$.
3) By combining these processes, we can obtain a single matrix as

$$\begin{aligned}\mathbf{M}_i &= \begin{pmatrix} U_i & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} c_1^{[i]} & c_2^{[i]} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1-\eta_i & \eta_i \\ \eta_i & 1-\eta_i \end{pmatrix} \\ &= \begin{pmatrix} U_i(c_1^{[i]}(1-\eta_i) + c_2^{[i]}\eta_i) & U_i(c_1^{[i]}\eta_i + c_2^{[i]}(1-\eta_i)) \\ \eta_i & 1-\eta_i \end{pmatrix},\end{aligned}$$

where $c_1^{[i]}$ is defined by $(f_0^{[i]})^{1-\eta_i} \cdot (-g_0^{[i]})^{\eta_i}$ and $c_2^{[i]}$ is defined by $(f_0^{[i]})^{\eta_i} \cdot (-g_0^{[i]})^{1-\eta_i}$. Then, the matrix can be simplified as

$$\mathbf{M}_i = \begin{pmatrix} f_0^{[i]} \cdot U_i & -g_0^{[i]} \cdot U_i \\ \eta_i & 1-\eta_i \end{pmatrix}, \ i > 0, \tag{5}$$

which reduces each iteration in Algorithm 1 into one matrix, called the combined matrix. With the combined matrix, $\mathbf{P}_i$ is calculated as

$$\mathbf{P}_{i+1} = \begin{pmatrix} g^{[i+1]}(x) \\ f^{[i+1]}(x) \end{pmatrix} = \mathbf{M}_i \cdot \mathbf{P}_i, \ i \geq 0.$$

Multiplying $\mathbf{M}_i$ to $\mathbf{P}_i$ has an effect on reducing the degree of polynomials such as $\deg g^{[i]}(x) + \deg f^{[i]}(x) - B_i \geq \deg g^{[i+1]}(x) + \deg f^{[i+1]}(x)$. As the step progresses, the degrees of the resulting polynomials become smaller, and the algorithm stops whenever the degree of $g^{[i]}(x)$ becomes zero. If the inverse of $f^{[0]}(x)$ exists, the algorithm completes with a non-zero constant γ for $g^{[i]}(x)$.

At each step, we obtain the matrix $\mathbf{Q}_i$ with $\mathbf{Q}_0 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$, by multiplying the combined matrix $\mathbf{M}_i$. Then, at the end of the i-th step, we have

$$\mathbf{Q}_{i+1} = \mathbf{M}_i \cdot \mathbf{M}_{i-1} \cdots \mathbf{M}_1 \cdot \mathbf{M}_0 \cdot \mathbf{Q}_0.$$

If the iteration stops at the i-th step, the inverse polynomial is directly formulated from the second row polynomial of $\mathbf{Q}_i$ multiplied by γ^{-1}. The overall procedure is simply explained in Algorithm 3, which stems from Algorithm 1. Algorithm 3 itself does not seem to give many advantages in the running time compared to Algorithm 1. However, the framework developed to represent Algorithm 3 will be the basis of deriving Algorithm 4 in the following subsection.

Algorithm 3 Variable-time inverse polynomial calculation algorithm

```
Require: $\mathbf{M}_0, \mathbf{P}_0 = \begin{pmatrix} g^{[0]}(x), \\ f^{[0]}(x) \end{pmatrix}, \mathbf{Q}_0 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$
Ensure: $(\mathbf{P}_i(1,1))^{-1} \cdot \mathbf{Q}_i(1,1) = f^{-1}(x)$ in $\mathbf{Z}_p[x]/g(x)$
1: $\mathbf{P}_1 = \mathbf{M}_0 \cdot \mathbf{P}_0$
2: $\mathbf{Q}_1 = \mathbf{M}_0 \cdot \mathbf{Q}_0$
3: $i \leftarrow 1$
4: while $\deg g^{[i]}(x) > 0$ do
5:     Update $U_i$ from $f_0^{[i]} g^{[i]}(x) - g_0^{[i]} f^{[i]}(x)$
6:     Update $\eta_i$ from (4)
7:     Generate $\mathbf{M}_i$ from (5)
8:     $\mathbf{P}_{i+1} \leftarrow \mathbf{M}_i \cdot \mathbf{P}_i$
9:     $\mathbf{Q}_{i+1} \leftarrow \mathbf{M}_i \cdot \mathbf{Q}_i$
10:    $i \leftarrow i + 1$
11: end while
```

3.2 Combining multiple steps

Based on the combined matrix in Algorithm 3, we consider the method of combining multiple steps and develop Algorithm 4, combined-variable-time algorithm, in this subsection. We explain how Algorithm 4 runs much faster than Algorithm 3.

First, it is possible to combine $\mathbf{M}_i$ with $\mathbf{M}_{i+1}$ by computing $\mathbf{M}_{i+1} \cdot \mathbf{M}_i$ and apply them in a single step instead of executing them in two separate steps. To calculate $\mathbf{M}_{i+1} \cdot \mathbf{M}_i$, assuming that η_i and B_i are known, we need to understand how η_{i+1} and B_{i+1} are determined specifically. Depending on whether η_i is 0 or 1, the polynomial that needs to be checked to calculate B_{i+1} changes. Additionally, the polynomials that need to be compared to determine η_{i+1} also vary depending on the value of η_i. Algorithm 4 computes the elements of $\mathbf{M}_{i+1} \cdot \mathbf{M}_i$ at once for odd values of i. If multiplying only $\mathbf{M}_i$ at the end still results in finding the inverse, then $\mathbf{M}_{i+1}$ becomes the identity matrix and does not affect the final computation of the correct inverse polynomial.

When η_i is one, deg $f^{[i]}(x)$ is lower than or equal to the deg $g^{[i]}(x)$. Then, to evaluate B_{i+1}, we should check the lowest degree with the non-zero coefficient of the polynomial

$$f_0^{[i]} \cdot g^{[i+1]}(x) - g_0^{[i+1]} \cdot f^{[i]}(x), \tag{6}$$

because $f^{[i+1]}(x)$ is equal to $f^{[i]}(x)$. The η_{i+1} is determined as

$$\eta_{i+1} = \begin{cases} 1, & \deg g^{[i+1]}(x) \geq \deg f^{[i]}(x) \\ 0, & \text{otherwise.} \end{cases} \tag{7}$$

When η_i is zero, deg $f^{[i]}(x)$ is higher than deg $g^{[i]}(x)$. Then, to evaluate B_{i+1}, we should check the lowest degree with a non-zero coefficient of the polynomial

$$g_0^{[i]} \cdot g^{[i+1]}(x) - g_0^{[i+1]} \cdot g^{[i]}(x), \tag{8}$$

because $f^{[i+1]}(x)$ is equal to $g^{[i]}(x)$. Then η_{i+1} is determined as

$$\eta_{i+1} = \begin{cases} 1, & \deg g^{[i+1]}(x) \geq \deg g^{[i]}(x) \\ 0, & \text{otherwise.} \end{cases} \tag{9}$$

From the relation between η_i and η_{i+1}, as explained above, we can express $\mathbf{M}_{i+1}$ as follows:

$$\mathbf{M}_{i+1} = \begin{pmatrix} {g_0^{[i]}}^{1-\eta_i} \cdot {f_0^{[i]}}^{\eta_i} \cdot U_{i+1} & -g_0^{[i+1]} \cdot U_{i+1} \\ \eta_{i+1} & 1 - \eta_{i+1} \end{pmatrix},$$

where $U_{i+1} = x^{-B_{i+1}}$. Then, each element of $\mathbf{M}_{i+1} \cdot \mathbf{M}_i$ is derived using $g_0^{[i]}$, $f^{[i]}(x)$, $g_0^{[i+1]}$, η_i, η_{i+1}, U_i, and U_{i+1}. For example, the (1, 1) element of $\mathbf{M}_{i+1} \cdot \mathbf{M}_i$ is given as

$${g_0^{[i]}}^{1-\eta_i} \cdot {f_0^{[i]}}^{1+\eta_i} \cdot U_i \cdot U_{i+1} - g_0^{[i+1]} \cdot U_{i+1} \cdot \eta_i \tag{10}$$

and the (1, 2) element of $\mathbf{M}_{i+1} \cdot \mathbf{M}_i$ is given as

$$-{g_0^{[i]}}^{2-\eta_i} \cdot {f_0^{[i]}}^{\eta_i} \cdot U_i \cdot U_{i+1} - g_0^{[i+1]} \cdot U_{i+1} \cdot (1 - \eta_i). \tag{11}$$

From (10) and (11), when η_i is equal to one, $g^{[i+2]}(x)$ polynomial can be updated as

$$\begin{aligned} g^{[i+2]}(x) =& ({f_0^{[i]}}^2 \cdot g^{[i]}(x) - g_0^{[i]} \cdot f_0^{[i]} \cdot f^{[i]}(x)) \qquad (12) \\ & \cdot U_i \cdot U_{i+1} - g_0^{[i+1]} \cdot g^{[i]}(x) \cdot U_{i+1}, \qquad (13) \end{aligned}$$

where $g_0^{[i+1]}$ is calculated as $f_0^{[i]} \cdot g_{B_i}^{[i]} - g_0^{[i]} \cdot f_{B_i}^{[i]}$.

Algorithm 4 demonstrates these processes. Algorithm 4 updates $g^{[i+2]}(x)$ from $g^{[i]}(x)$ according to (13), while Algorithm 3 executes step 8 twice to update $g^{[i+1]}(x)$ and $g^{[i+2]}(x)$. More specifically, in Algorithm 3, we multiply $f_0^{[i]}$ to each coefficient of the first row polynomial of $\mathbf{P}_i$ and shift the polynomial with the amount of B_i. Then, we multiply $f_0^{[i+1]}$ to each coefficient of the first row polynomial of $\mathbf{P}_{i+1}$ and shift the polynomial with the amount of B_{i+1}. However, in Algorithm 4, we calculate ${g_0^{[i]}}^{1-\eta_i} \cdot {f_0^{[i]}}^{1+\eta_i}$ and multiply it by the first row polynomial of $\mathbf{P}_i$. Then, shift the polynomial with the amount of $B_i + B_{i+1}$ at once. The shifting algorithm spends the same amount of time regardless of how far it moves. Though we have additional second terms in (10) and (11), which depend on η_i value, from the above way, we can explain how the proposed method required fewer multiplications and shifting operations than the previous method. Although a few procedures are required to multiply $\mathbf{M}_i$ to obtain η_{i+1}, U_{i+1}, and $g_0^{[i+1]}$, some memory access time can be saved by multiplying $\mathbf{M}_{i+1} \cdot \mathbf{M}_i$ to $\mathbf{P}_i$ and $\mathbf{Q}_i$, instead of separate execution. Similarly, the combination of n matrices together is also possible. We will show that Algorithm 4 runs much faster than Algorithm 3 through simulation results.

Second, in a parallel programmable environment, it is possible to enhance the effect of using combined matrices. After thread-1 calculates parameters necessary for $\mathbf{M}_{i+1} \cdot \mathbf{M}_i$, it passes them to thread-2 to update $\mathbf{Q}_i$. While thread-2 updates $\mathbf{Q}_i$, thread-1 determines the next parameters. Using parallel processing resources like in [18] allows us to speed up the algorithm, although this comes with the trade-off of increased complexity and additional memory usage.

Algorithm 4 Combined-variable-time inverse polynomial calculation algorithm

```
Require: $\mathbf{M}_0, \mathbf{P}_0 = \begin{pmatrix} g^{[0]}(x) \\ f^{[0]}(x) \end{pmatrix}, \mathbf{Q}_0 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$
Ensure: $f^{-1}(x)$ in $\mathbf{Z}_p[x]/g(x)$
 1: $\mathbf{P}_1 = \mathbf{M}_0 \cdot \mathbf{P}_0$
 2: $\mathbf{Q}_1 = \mathbf{M}_0 \cdot \mathbf{Q}_0$
 3: $i \leftarrow 1$
 4: while deg $g^{[i]}(x) > 0$& deg $f^{[i]}(x) > 0$ do
 5:     Update $U_i$ from $f_0^{[i]} g^{[i]}(x) - g_0^{[i]} f^{[i]}(x)$
 6:     Update $\eta_i$ from (4)
 7:     if $\eta_i = 1$ then
 8:         Update $U_{i+1}$ from (6)
 9:         Update $\eta_{i+1}$ from (7)
10:     end if
11:     if $\eta_i = 0$ then
12:         Update $U_{i+1}$ from (8)
13:         Update $\eta_{i+1}$ from (9)
14:     end if
15:     Generate $\mathbf{M}_{i+1} \cdot \mathbf{M}_i$
16:     $\mathbf{P}_{i+2} \leftarrow \mathbf{M}_{i+1} \cdot \mathbf{M}_i \cdot \mathbf{P}_i$
17:     $\mathbf{Q}_{i+2} \leftarrow \mathbf{M}_{i+1} \cdot \mathbf{M}_i \cdot \mathbf{Q}_i$
18:     $i \leftarrow i + 2$
19: end while
20: if deg $g^{[i]}(x) = 0$ then
21:     $f^{-1}(x) = (\mathbf{P}_i(1,1))^{-1} \cdot \mathbf{Q}_i(1,1)$
22: end if
23: if deg $f^{[i]}(x) = 0$ then
24:     $f^{-1}(x) = (\mathbf{P}_i(2,1))^{-1} \cdot \mathbf{Q}_i(2,1)$
25: end if
```

4 Combined-constant-time inverse polynomial calculation algorithm

This section proposes a combined-constant-time inverse polynomial calculation algorithm with $N+1$ iterations which is half of $2N+1$ iterations of NTRU KEM, where N is the degree of the modular polynomial, $g(x)$, in the given polynomial ring. In this section, we use the notation, deg $f^{[i]}(x)$ and deg $g^{[i]}(x)$, to represent the maximum possible degrees of the polynomials instead of the actual highest orders of them. At beginning, deg $f^{[0]}(x)$ and deg $g^{[0]}(x)$ are fixed by N, and each of them can be reduced by zero, one, or two, as the iteration step goes on. When we consider summation of degrees of polynomials, deg $f^{[i]}(x) +$ deg $g^{[i]}(x)$, we reduce this by two such as deg $f^{[i+1]}(x) +$ deg $g^{[i+1]}(x) =$ deg $f^{[i]}(x) +$ deg $g^{[i]}(x) - 2$ through each iteration. This is achieved by checking constant and first-order terms of $f^{[i]}(x)$ and $g^{[i]}(x)$ simultaneously in the construction of the multiplication matrix at each iteration step. We construct Table 1 to illustrate how the degrees of the input polynomials are reduced by two in each iteration, considering the constants and first-order coefficients of the given polynomials. In the first iteration, we generate the polynomials $g^{[1]}(x)$ and $f^{[1]}(x)$ according to the rule specified in case $2-3-i$, which is outlined in the subsequent subsect. 4.1. Specifically, $g^{[1]}(x) = g^{[0]}(x)$ and $f^{[1]}(x) = x^{-2}(g^{[0]}(x) - f^{[0]})$. Then, we observe that the degree of $f^{[1]}(x)$ is at least two degrees lower than the degree of $f^{[0]}(x)$. In the second iteration, we assign $x^{-2}(g^{[1]}(x) - f^{[1]}(x))$ to $g^{[2]}(x)$. Therefore, the degree of $g^{[2]}(x)$ is at least two degrees lower than the degree of $g^{[1]}(x)$. The goal is to make $g^{[i]}(x)$ become one and $f^{[i]}(x)$ become zero after performing this process five times. The updating rule is decided by the constants, first-order coefficients, and the degrees of polynomials. However, our multiplication and summation operations are consistent in each iteration without conditional statements, making the algorithm work in constant time. In the following subsections, we explain specific algorithms for the general q-ary coefficient case and binary coefficient case. To derive the general form of the multiplication matrix, $\mathbf{M}_i$, we considered eight sub-cases based on the conditions of the coefficients of $g^{[i]}(x)$ and $f^{[i]}(x)$. In Sect. 4.1, we defined new important parameters and detailed how the reduction procedure was handled in each sub-case.

Table 1 An example for the binary case

i	$g^{[i]}(x)$	$f^{[i]}(x)$	case
0	$1+x+x^2+x^3+x^4$	$1+x+x^4$	2-3-i
1	$1+x+x^2+x^3+x^4$	$1+x$	1-3
2	$1+x+x^2$	$1+x$	2-3-i
3	$1+x+x^2$	1	1-3
4	1	1	2-3-i
5	1	0	

4.1 Q-ary case

In this subsection, we show how to determine i-th iteration's multiplication matrix, $\mathbf{M}_i$, where $0 \leq i \leq N$, when the coefficients of the polynomials are in $\mathbf{Z}_q$. In the i-th iteration, we reduce $g^{[i]}(x)$ or $f^{[i]}(x)$ by degree two to finish the algorithm within $N+1$ iterations, because we assume the degrees of $g^{[0]}(x)$ and $f^{[0]}(x)$ both to be N. For example, without any summation of polynomials, we can reduce the degree of $g^{[i]}(x)$(or $f^{[i]}(x)$) by two after multiplying x^{-2} when its constant and first-order terms are zero. For a few cases, it is impossible to reduce the degrees of the given polynomial by two, where we simultaneously reduce the degree of each polynomial by one. We keep track of the degree information of the polynomials and define indicator variable δ_i as

$$\delta_i = \begin{cases} 1, & \deg g^{[i]}(x) = \deg f^{[i]}(x) \\ 0, & \deg g^{[i]}(x) > \deg f^{[i]}(x). \end{cases} \tag{14}$$

After $N+1$ iterations, the resulting polynomials have only constant terms because degrees are reduced by two at each step.

In each iteration, we locate the lower degree polynomial to the second row to finish the algorithm with a non-zero first-row polynomial, which is essential to obtain the right inverse polynomial from (2). If the degree of the first-row polynomial becomes smaller than the degree of the second-row polynomial, we swap two polynomials. Then, the degree of the first-row polynomial will always be greater than or equal to the degree of the second-row polynomial, which explains the condition of δ_i. Using δ_i, we decide which polynomial's degree should be reduced first.

Based on these rules, the multiplication matrix $\mathbf{M}_i$ is derived separately in each case. $\mathbf{M}_i$ has a different form depending on the value of δ_i because the polynomial is taken to the next iteration and the arrangement of the polynomials is affected by δ_i. Furthermore, constant and first-order coefficients of $g^{[i]}(x)$ and $f^{[i]}(x)$ should be considered to determine the matrix $\mathbf{M}_i$. We define coefficient matrix $\mathbf{C}_i$ as

$$\mathbf{C}_i = \begin{pmatrix} a_0 & a_1 \\ b_0 & b_1 \end{pmatrix},$$

where $g^{[i]}(x) = \sum_{i=0}^{N} a_i \cdot x^i$ and $f^{[i]}(x) = \sum_{i=0}^{N} b_i \cdot x^i$. Note that $\deg g^{[0]}(x) = \deg f^{[0]}(x) = N$ and $\delta_0 = 1$. Moreover, we define an indicator variable μ_i such as

$$\mu_i = \begin{cases} 1, & a_0 b_1 \neq a_1 b_0 \\ 0, & a_0 b_1 = a_1 b_0, \end{cases} \tag{15}$$

which is considered in one of the cases, where we need to reduce the degree of the second-row polynomial with summation and a_0 is not zero. From the value of δ_i, we divide it into two cases as 1) and 2) as below. Furthermore, from the conditions of the coefficients, the following sub-cases are given as follows;

1) $\delta_i = 0$: In this case, because $\deg g^{[i]}(x) > \deg f^{[i]}(x)$, if it is possible, the degree of $g^{[i]}(x)$ is reduced first and the position of two polynomials is not altered.

1-1) $(a_0, a_1) = (0, 0)$: In this sub-case, the degree of $g^{[i]}(x)$ can be reduced by two after multiplying by

$$\mathbf{M}_i = \begin{pmatrix} x^{-2} & 0 \\ 0 & 1 \end{pmatrix}.$$

Then, the degrees of the resulting polynomials are $\deg g^{[i+1]}(x) = \deg g^{[i]}(x) - 2$ and $\deg f^{[i+1]}(x) = \deg f^{[i]}(x)$.

1-2) $(a_0, a_1) \neq (0, 0)$ and $(b_0, b_1) = (0, 0)$: In this sub-case, the degree of $f^{[i]}(x)$ can be reduced by two multiplying by

$$\mathbf{M}_i = \begin{pmatrix} 1 & 0 \\ 0 & x^{-2} \end{pmatrix}.$$

Then, the degrees of the resulting polynomials are $\deg g^{[i+1]}(x) = \deg g^{[i]}(x)$ and $\deg f^{[i+1]}(x) = \deg f^{[i]}(x) - 2$. When $\mathbf{C}_i = \mathbf{0}$, we follow the sub-case 1-1), which is the policy for convenience, and changing the rule does not affect obtaining the correct answer at the end of the algorithm.

1-3) $(a_0, a_1) \neq (0, 0)$ and $b_0 \neq 0$: In this sub-case, $f^{[i]}(x)$ is taken to the next iteration such as $f^{[i+1]}(x) = f^{[i]}(x)$, and the degree of the first-row polynomial can be reduced by two from summing of the polynomials using the non-zero coefficient b_0. The polynomial, $b_0 \cdot g^{[i]}(x) - a_0 \cdot f^{[i]}(x)$, has a zero constant term, and its degree does not increase, compared to the degree of $g^{[i]}(x)$ because of $\delta_i = 0$. Let $h^{[i]}(x) = x^{-1} \cdot (b_0 \cdot g^{[i]}(x) - a_0 \cdot f^{[i]}(x))$, which has $\deg h^{[i]}(x) = \deg g^{[i]}(x) - 1$. Additionally, the degree of $f^{[i]}(x)$ is at most $\deg g^{[i]}(x) - 1$ because δ_i is zero. Then, we generate $g^{[i+1]}(x)$ such as $g^{[i+1]}(x) = x^{-1} \cdot (b_0 \cdot h^{[i]}(x) - (a_1 b_0 - a_0 b_1) \cdot f^{[i]}(x))$, where its degree is smaller than the degree of $h^{[i]}(x)$ by one. From this procedure, we have $\deg g^{[i+1]}(x) = \deg g^{[i]}(x) - 2$ and $\deg f^{[i+1]}(x) = \deg f^{[i]}(x)$ and the multiplication matrix $\mathbf{M}_i$ is derived as

$$\begin{aligned} \mathbf{M}_i &= \begin{pmatrix} b_0 x^{-1} & -(a_1 b_0 - a_0 b_1) x^{-1} \\ 0 & 1 \end{pmatrix} \\ &\quad \times \begin{pmatrix} b_0 x^{-1} & -a_0 x^{-1} \\ 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} b_0^2 x^{-2} & -(a_1 b_0 - a_0 b_1) x^{-1} - a_0 b_0 x^{-2} \\ 0 & 1 \end{pmatrix}. \end{aligned}$$

1-4) $(a_0, a_1) \neq (0, 0)$ and $b_0 = 0$ and $b_1 \neq 0$: This sub-case should be separated from the sub-cases 1-2) and 1-3) because b_1 is not zero and b_0 is zero. To reduce the degree of the first-row polynomial, the constant term of the second-row polynomial should not be zero, which is satisfied by multiplying x^{-1} to $f^{[i]}(x)$.

From setting $f^{[i+1]}(x) = x^{-1} \cdot f^{[i]}(x)$, the degree of the second-row polynomial is reduced by one. With $f^{[i+1]}(x)$ having a non-zero constant term, the degree of the first-row polynomial can be reduced by one using $g^{[i+1]}(x) = (b_1 \cdot g^{[i]}(x) - a_0 \cdot f^{[i+1]}(x)) \cdot x^{-1}$. We can check that the subtraction of $a_0 \cdot f^{[i+1]}(x)$ from $g^{[i]}(x)$ does not increase the degree of the polynomial because δ_i is zero. Then, from the subtraction and multiplication of x^{-1}, the degree of the first-row polynomial is reduced by one. The multiplication matrix representing this procedure is given as

$$\mathbf{M}_i = \begin{pmatrix} b_1 x^{-1} & -a_0 x^{-2} \\ 0 & x^{-1} \end{pmatrix}.$$

Since we have $\deg f^{[i+1]}(x) = \deg f^{[i]}(x) - 1$ and $\deg g^{[i+1]}(x) = \deg g^{[i]}(x) - 1$, two degrees are reduced in this sub-case.

2) $\delta_i = 1$: In this case, because $\deg g^{[i]}(x) = \deg f^{[i]}(x)$, if possible, the degree of $f^{[i]}(x)$ should be reduced first. When it is inevitable to reduce the degree of $g^{[i]}(x)$, we swap the positions of the polynomials to avoid the second-row polynomial having a larger degree than the first-row polynomial.

2-1) $(a_0, a_1) = (0, 0)$: In this sub-case, we reduce the degree of $g^{[i]}(x)$ by two after multiplying x^{-2}. Then, we exchange the first and second-row polynomials for locating the lower degree in the second row. As we use

$$\mathbf{M}_i = \begin{pmatrix} 0 & 1 \\ x^{-2} & 0 \end{pmatrix},$$

we have $g^{[i+1]}(x) = f^{[i]}(x)$ and $f^{[i+1]}(x) = x^{-2} \cdot g^{[i]}(x)$ with $\deg f^{[i+1]}(x) = \deg g^{[i]}(x) - 2$ and $\deg g^{[i+1]}(x) = \deg f^{[i]}(x)$.

2-2) $(a_0, a_1) \neq (0, 0)$ and $(b_0, b_1) = (0, 0)$: This sub-case gives us a similar multiplication matrix with the sub-case 1-2). In a situation where the degree of $f^{[i]}(x)$ needs to be reduced, there is no need to move the position of $f^{[i]}(x)$ regardless of δ_i, because $f^{[i]}(x)$ is in the second row.

2-3-i) $(b_0, b_1) \neq (0, 0)$ and $a_0 \neq 0$ and $\mu_i = 0$: Contrary to the sub-case 1-3), the sub-case 2-3) is divided into two sub-cases depending on the value of μ_i. In this sub-case, it is possible to reduce the degree of the second-row polynomial by two because a_0 is not zero and μ_i is zero. The polynomial, $h^{[i]}(x) = b_0 \cdot g^{[i]}(x) - a_0 \cdot f^{[i]}(x)$, has a zero constant term, and the first order coefficient is $b_0 \cdot a_1 - a_0 \cdot b_1$, which is also zero because μ_i is zero. Then, we assign $x^{-2} \cdot h^{[i]}(x)$ to $f^{[i+1]}(x)$ and take $g^{[i]}(x)$ into $g^{[i+1]}(x)$. From this procedure, we have $\deg g^{[i+1]}(x) = \deg g^{[i]}(x)$ and $\deg f^{[i+1]}(x) = \deg f^{[i]}(x) - 2$, and the multiplication matrix $\mathbf{M}_i$ can be derived as

$$\mathbf{M}_i = \begin{pmatrix} 1 & 0 \\ b_0 x^{-2} & -a_0 x^{-2} \end{pmatrix}.$$

2-3-ii) $(b_0, b_1) \neq (0, 0)$ and $a_0 \neq 0$ and $\mu_i = 1$: Contrary to the above sub-case 2-3-i), it is impossible to reduce the degree of the second-row polynomial by two because δ_i and μ_i are one. After reducing the degree of the second-row polynomial by one, we adopt it to reduce the degree of the first-row polynomial by one. $f^{[i+1]}(x)$ is derived as $x^{-1} \cdot (b_0 \cdot g^{[i]}(x) - a_0 \cdot f^{[i]}(x))$, which has $\deg f^{[i+1]}(x) = \deg f^{[i]}(x) - 1$. Then, we calculate $g^{[i+1]}(x)$ as

$$\begin{aligned} g^{[i+1]}(x) =& x^{-1} \cdot ((a_1 \cdot b_0 - a_0 \cdot b_1) \cdot g^{[i]}(x) \\ & - a_0 \cdot f^{[i+1]}(x)). \end{aligned}$$

Hence, the multiplication matrix $\mathbf{M}_i$ is given as

$$\begin{aligned} \mathbf{M}_i &= \begin{pmatrix} (a_1 b_0 - a_0 b_1) x^{-1} & -a_0 x^{-1} \\ 0 & 1 \end{pmatrix} \\ &\cdot \begin{pmatrix} 1 & 0 \\ b_0 x^{-1} & -a_0 x^{-1} \end{pmatrix} \\ &= \begin{pmatrix} (a_1 b_0 - a_0 b_1) x^{-1} - a_0 b_0 x^{-2} & a_0^2 x^{-2} \\ b_0 x^{-1} & -a_0 x^{-1} \end{pmatrix} \end{aligned}$$

and we have $\deg g^{[i+1]}(x) = \deg g^{[i]}(x) - 1$ and $\deg f^{[i+1]}(x) = \deg f^{[i]}(x) - 1$.

2-4) $(b_0, b_1) \neq (0, 0)$ and $a_0 = 0$ and $a_1 \neq 0$: In this sub-case, the degree of the first-row polynomial is reduced by one because a_0 is zero. To reduce the degree of the second-row polynomial, the constant term of the first-row polynomial should not be zero, which is satisfied by multiplying x^{-1} to $g^{[i]}(x)$. From setting $g^{[i+1]}(x) = x^{-1} \cdot g^{[i]}(x)$ having a non-zero constant term, the degree of the first-row polynomial is reduced by one. The degree of the second-row polynomial should be reduced by one from $f^{[i+1]}(x) = (b_0 \cdot g^{[i+1]}(x) - a_1 \cdot f^{[i]}(x)) \cdot x^{-1}$. We can check that the addition of $b_0 \cdot g^{[i+1]}(x)$ to $f^{[i]}(x)$ does not increase the degree of the polynomial $f^{[i]}(x)$ because δ_i is one. And, from the summation and multiplication of x^{-1}, the degree of the second-row polynomial is reduced by one such as $\deg f^{[i+1]}(x) = \deg f^{[i]}(x) - 1$. The multiplication matrix representing this procedure is given as

$$\mathbf{M}_i = \begin{pmatrix} x^{-1} & 0 \\ b_0 x^{-2} & -a_1 x^{-1} \end{pmatrix}.$$

Through $N+1$ iterations, the combined-constant-time inverse polynomial calculation algorithm can be implemented with $\mathbf{M}_i$. Each iteration comprises two steps. At first, from (2), for given polynomials $g^{[i]}(x)$ and $f^{[i]}(x)$, the multiplication matrix $\mathbf{M}_i$ is calculated, and we store

$$\mathbf{M}_i \cdot \begin{pmatrix} g^{[i]}(x) \\ f^{[i]}(x) \end{pmatrix}$$

to derive $\mathbf{M}_{i+1}$ at the next iteration. From (3), we multiply $\mathbf{M}_i$ by the matrix $\mathbf{M}_{i-1} \cdots \mathbf{M}_0 \cdot \begin{pmatrix} 0 \\ \gamma^{-1} \end{pmatrix}$ to obtain the inverse polynomial at the end of the algorithm. Iterating this procedure $N+1$ times, we obtain the $f^{-1}(x)$ polynomial at the first row of the matrix

$$\mathbf{M}_{N-1} \cdots \mathbf{M}_0 \cdot \begin{pmatrix} 0 \\ \gamma^{-1} \end{pmatrix} = \begin{pmatrix} f^{-1}(x) \\ * \end{pmatrix}$$

because we have

$$\mathbf{M}_{N-1} \cdots \mathbf{M}_0 \cdot \begin{pmatrix} g(x) \\ f(x) \end{pmatrix} = \begin{pmatrix} \gamma \\ 0 \end{pmatrix}.$$

However, if we multiply different $\mathbf{M}_i$ conditioned on the coefficients and δ_i as calculated from the above procedure, information leakage can cause the side channel attack; hence, we design the combined matrix $\mathbf{M}_i$ as

$$\begin{pmatrix} m_1 + m_2 x^{-1} + m_3 x^{-2} & m_4 + m_5 x^{-1} + m_6 x^{-2} \\ m_7 x^{-1} + m_8 x^{-2} & m_9 + m_{10} x^{-1} + m_{11} x^{-2} \end{pmatrix}. \quad (16)$$

This makes it possible to implement the proposed algorithm without a conditional statement. To clarify m_k, $1 \le k \le 11$, we define new parameters α_0, α_1, β_0, and β_1 satisfying that

$$\alpha_i = \begin{cases} 0, & a_i = 0 \\ 1, & a_i \neq 0, \end{cases} \quad \beta_i = \begin{cases} 0, & b_i = 0 \\ 1, & b_i \neq 0, \quad i = 0 \text{ or } 1. \end{cases} \quad (17)$$

Then, the conditions of the sub-cases we introduced can be expressed in Table 2.

When we analyze the (1, 1) element of $\mathbf{M}_i$, it is included in $\{0, 1, x^{-1}, b_1 x^{-1}, x^{-2}, b_0^2 x^{-2}, (a_1 b_0 - a_0 b_1) x^{-1} - a_0 b_0 x^{-2}\}$, which explains why the element is expressed as $m_1 + m_2 x^{-1} + m_3 x^{-2}$. Then, from Table 3, m_1 is derived as

$$m_1 = ((\alpha_0 \mid \alpha_1) \& \overline{\beta_0} \& \overline{\beta_1}) + (\delta_i \& (\beta_0 \mid \beta_1) \& \alpha_0 \& \overline{\mu_i})$$

because the (1, 1) element of M_i is one solely in the sub-cases 1-2) and 2-3-i). Two terms comprising m_1 cannot be one simultaneously, and m_1 is one if the sub-cases happen. Similarly, we formulate all of m_k, $1 \le k \le 11$, in Table 3.

To update δ_i, we keep tracking the degrees of each polynomial, which vary differently depending on the sub-cases.

Table 2 The conditions of cases are expressed by the variables and coefficients

sub-case	condition(0 or 1)	notation
1-1)	$\overline{\delta_i} \& \overline{\alpha_0} \& \overline{\alpha_1}$	t_1
1-2) or 2-2)	$(\alpha_0 \mid \alpha_1) \& \overline{\beta_0} \& \overline{\beta_1}$	t_2
1-3)	$\overline{\delta_i} \& (\alpha_0 \mid \alpha_1) \& \beta_0$	t_3
1-4)	$\overline{\delta_i} \& (\alpha_0 \mid \alpha_1) \& \overline{\beta_0} \& \beta_1$	t_4
2-1)	$\delta_i \& \overline{\alpha_0} \& \overline{\alpha_1}$	t_5
2-3-i)	$\delta_i \& (\beta_0 \mid \beta_1) \& \alpha_0 \& \overline{\mu_i}$	t_6
2-3-ii)	$\delta_i \& (\beta_0 \mid \beta_1) \& \alpha_0 \cdot \mu_i$	t_7
2-4)	$\delta_i \& (\beta_0 \mid \beta_1) \& \overline{\alpha_0} \& \alpha_1$	t_8

Table 3 m_k, $1 \le k \le 11$, is derived for the q-ary case

m_k	value
m_1	$t_2 + t_6$
m_2	$b_1 \cdot t_4 + (a_1 b_0 - a_0 b_1) \cdot t_7 + t_8$
m_3	$t_1 + b_0^2 \cdot t_3 + (-a_0 b_0) \cdot t_7$
m_4	t_5
m_5	$(a_0 b_1 - a_1 b_0) \cdot t_3$
m_6	$(-a_0 b_0) \cdot t_3 + (-a_0) \cdot t_4 + a_0^2 \cdot t_7$
m_7	$b_0 \cdot t_7$
m_8	$t_5 + b_0 \cdot t_6 + b_0 \cdot t_8$
m_9	$t_1 + t_3$
m_{10}	$t_4 + (-a_0) \cdot t_7 + (-a_1) \cdot t_8$
m_{11}	$(-a_0) \cdot t_6 + t_2$

Table 4 $\deg g_i(x)$ and $\deg f_i(x)$ are updated regarding to each case

sub-case	$\deg g^{[i+1]}(x)$	$\deg f^{[i+1]}(x)$
1-1)	$\deg g^{[i]}(x) - 2$	$\deg f^{[i]}(x)$
1-2) or 2-2)	$\deg g^{[i]}(x)$	$\deg f^{[i]}(x) - 2$
1-3)	$\deg g^{[i]}(x) - 2$	$\deg f^{[i]}(x)$
1-4)	$\deg g^{[i]}(x) - 1$	$\deg f^{[i]}(x) - 1$
2-1)	$\deg f^{[i]}(x)$	$\deg g^{[i]}(x) - 2$
2-3-i)	$\deg g^{[i]}(x)$	$\deg f^{[i]}(x) - 2$
2-3-ii)	$\deg g^{[i]}(x) - 1$	$\deg f^{[i]}(x) - 1$
2-4)	$\deg g^{[i]}(x) - 1$	$\deg f^{[i]}(x) - 1$

For each case, we summarize the variation of the degree in Table 4. Then, without using a conditional statement, we implement the updating algorithm of $\deg g^{[i+1]}(x)$ such as

$$\begin{aligned} \deg g^{[i+1]}(x) &= l_1 \cdot \deg g^{[i]}(x) \\ &\quad + l_2 \cdot \deg f^{[i]}(x) - l_3 - 2 \cdot l_4, \end{aligned}$$

where $l_1 = \overline{t_5}$, $l_2 = t_5$, $l_3 = (t_4 \mid t_7 \mid t_8)$, and $l_4 = (t_1 \mid t_3)$. These l_i, $1 \le i \le 4$, are binary values, and they decide the operation of each term. Then l_1 is derived by checking that the polynomials are swapped in the sub-case 2-1). Moreover, l_2 is exclusively one with l_1 because either of l_1 or l_2 should be selected at each iteration. For l_3 and l_4, we combine the sub-cases that have the patterns with the OR operation. Similarly, the updating algorithm of deg $f^{[i+1]}(x)$ is given as

$$\deg f^{[i+1]}(x) = l_5 \cdot \deg g^{[i]}(x) + l_6 \cdot \deg f^{[i]}(x) - l_7 - 2 \cdot l_8,$$

where $l_5 = l_2, l_6 = l_1, l_7 = l_3$, and $l_8 = (t_5 \mid t_6 \mid t_2)$. Using this algorithm, after calculating the degrees of polynomials, we update δ_i.

Through this process, it is confirmed that the previous Algorithm 2, which reduced one degree in one iteration, can be transformed into the Algorithm 5 that reduces two degrees in one iteration. The x^{-2} term in equation (16) shifts coefficients of a polynomial by two orders at once. Additionally, the x^{-1} term in equation (16) combines the summation of two polynomials whose coefficients are shifted by one order into a constant multiplication of a polynomial shifted by one order. Since shifting coefficients by one order consumes the same amount of time as shifting coefficients by two orders, the algorithm of combining these two steps saves running time in this aspect.

The primary difference between the constant-time algorithm and the variable-time algorithm in Sect. 3 is that the operation time is fixed for any chosen input polynomial $f^{[0]}(x)$. As expressed in Algorithm 5, we obtain the inverse polynomial after $N + 1$ iterations, which is achieved by the suggested matrix $\mathbf{M}_i$ in this subsection.

Algorithm 5 Combined-constant-time inverse polynomial calculation algorithm

```
Require: $\mathbf{P}_0 = \begin{pmatrix} g^{[0]}(x) \\ f^{[0]}(x) \end{pmatrix}, \mathbf{Q}_0 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$
Ensure: $(\mathbf{P}_{N-1}(1,1))^{-1} \cdot \mathbf{Q}_{N-1}(1,1) = f^{-1}(x)$ in $\mathbf{Z}_p[x]/g(x)$
1: for $i = 0, i \le N, i \leftarrow i + 1$ do
2:     $\delta_i \leftarrow$ (0 or 1), from (14)
3:     $\mu_i \leftarrow$ (0 or 1), from (15)
4:     $\alpha_0, \alpha_1, \beta_0$, and, $\beta_1 \leftarrow$ (0 or 1), from (17)
5:     Calculate $t_j$, $1 \le j \le 8$, from Table 2
6:     Calculate $m_k$, $1 \le k \le 11$, from Table 3
7:     Update $\mathbf{M}_i$, from (16)
8:     $\mathbf{P}_{i+1} \leftarrow \mathbf{M}_i \cdot \mathbf{P}_i$
9:     $\mathbf{Q}_{i+1} \leftarrow \mathbf{M}_i \cdot \mathbf{Q}_i$
10:    Update deg $g^{[i+1]}(x)$ and deg $f^{[i+1]}(x)$, from Table 4
11: end for
```

4.2 Binary case

We introduce a few points that can be simplified when the coefficients of polynomials are in $\mathbf{Z}_2$.

Table 5 m_k, $1 \le k \le 11$, is derived for the binary case

m_k	value
m_1	$t_2 + t_6$
m_2	$t_4 + (a_1 b_0 - a_0 b_1) \cdot t_7 + t_8$
m_3	$t_1 + t_3 + b_0 \cdot t_7$
m_4	t_5
m_5	$(a_0 b_1 - a_1 b_0) \cdot t_3$
m_6	$a_0 \cdot t_3 + a_0 \cdot t_4 + t_7$
m_7	$b_0 \cdot t_7$
m_8	$t_5 + b_0 \cdot t_6 + b_0 \cdot t_8$
m_9	$t_1 + t_3$
m_{10}	$t_4 + t_7 + t_8$
m_{11}	$t_6 + t_2$

Initially, we are not required to calculate γ^{-1} because γ is fixed as one in (2) for the binary case. For a non-zero element in $\mathbf{Z}_3$, the γ^{-1} is equal to γ. Except for these binary and ternary cases, the inverse calculation of γ is required generally. Furthermore, we adopt a_i and b_i instead of defining α_i and β_i because the values are equal for the binary case. Then, m_i can be simplified additionally. For example, m_2 has the term $b_1 \cdot t_4$, which is equal to t_4, because t_4 already contains b_1 condition. When these advantages are considered, values of m_i's are calculated in Table 5.

The degree updating algorithm is similar to the q-ary case because the indicator variables are adopted instead of using the constant coefficients directly in the q-ary case.

5 Simulation result

We simulate the combined-variable-time in Algorithm 4 and combined-constant-time algorithms in Algorithm 5 for the binary and ternary coefficient cases. In addition, we compare the running time of the proposed algorithms with the variable-time algorithm in Algorithm 3 and the optimized constant-time in Algorithm 2, which are included in the NTRU optimized implementation in the third round of NIST PQC Standardization. The variable-time algorithm in Algorithm 3 is another form of Algorithm 1, which is used in the previous NTRU implementation. The simulation is run on an iMac with an M1 chip with eight cores and 16GB memory by compiling in debug mode. After generating 10,000 random polynomials, we measure the average elapsed clock cycle of obtaining the inverse polynomials. The degree of $g(x)$ noted by N, is chosen by NTRU parameters such as 508, 676, 820 for NTRU-HPS, and 700 for NTRU-HRSS.

From Table 6, in the binary case, we check that the proposed combined-variable-time in Algorithm 4 outperforms the variable-time inverse polynomial Algorithm 3

Table 6 Simulation results for the binary case (in clock cycles) with respect to NIST PQC NTRU parameters ($N = 508, 676, 820$ for NTRU-HPS, $N = 700$ for NTRU-HRSS) on M1

N	Algorithm 3 (variable-time)	Algorithm 4 (combined-variable-time)	Algorithm 2 (NTRU constant-time)	Algorithm 5 (combined-constant-time)
508	793	405(x1.96)	3015	2106(x1.43)
676	1391	707(x1.97)	5326	3715(x1.43)
700	1491	758(x1.97)	5726	3982(x1.43)
820	2046	1035(x1.98)	7835	5458(x1.44)

Table 7 Simulation results for the ternary case (in clock cycles) with respect to NIST PQC NTRU parameters ($N = 508, 676, 820$ for NTRU-HPS, $N = 700$ for NTRU-HRSS) on M1

N	Algorithm 3 (variable-time)	Algorithm 4 (combined-variable-time)	Algorithm 2 (NTRU constant-time)	Algorithm 5 (combined-constant-time)
508	2726	1714(x1.59)	5726	4406(x1.30)
676	4815	3059(x1.57)	10105	7771(x1.30)
700	5157	3277(x1.57)	10836	8327(x1.30)
820	7206	4485(x1.61)	14877	11432(x1.30)

in the execution time. The average execution time of the proposed combined-variable-time algorithm is 51% of the variable-time algorithm. Also, the combined-constant-time algorithm is faster than the constant-time NTRU algorithm, where the average execution time of the proposed combined-constant-time algorithm is 69.8% of the constant-time NTRU algorithm, which is achieved by reducing the iteration time and using additional memory space for saving degree information.

The running time of the variable-time algorithms depends on each chosen polynomial, so it is hard to predict the average time. As expected, we verify that the variable-time algorithms give faster results than the constant-time algorithms. By comparing the execution time of the proposed fast combined-variable-time in Algorithm 4 and the constant-time NTRU Algorithm 2, we can check that Algorithm 2 takes seven times more execution time than Algorithm 4 to avoid the risk of revealing information with the constant-time property. This gap in execution time occurs from the difference of the reducing degrees in the variable-time and constant-time algorithms. The variable-time algorithms reduce degrees as many as possible at once, whereas the NTRU constant-time algorithm reduces just one degree at each iteration. The proposed combined-constant-time Algorithm 5 reduces the gap of the execution time with the variable-time algorithms by increasing the number of degrees reduced at one iteration while retaining the constant-time property.

For the ternary case, we verify similar results with the binary case in Table 7. The average execution time of the proposed combined-variable-time in Algorithm 4 is 62.8% of the variable-time Algorithm 3. And, the average execution time of the proposed combined-constant-time in Algorithm 5 is 77% of the NTRU constant-time Algorithm 2. In the ternary case, to calculate the values of m_k for the combined-constant-time algorithm, we need to execute a multiplication operation and a modular operation regarding the ternary coefficients, where an additional overhead leads to an increase in the running time compared to the binary coefficient case.

Additionally, the algorithms are simulated on the ARM Cortex M4 platform STM32F407VG-disc, which incorporates the STM32F407VGT6 microcontroller featuring a 32-bit Arm Cortex-M4 with FPU core that can run at up to 168MHz, along with 1-Mbyte flash memory and 192-Kbyte RAM. Following the generation of 1,000 random polynomials, the average time required to obtain the inverse polynomials is measured in tick counts. Referring to Table 8 for the binary case and Table 9 for the ternary case, it becomes evident that the suggested algorithms demonstrate comparable performance enhancements to those observed in the previous M1 simulation results. In both binary and ternary scenarios, the performance improvement of Algorithm 4 on the Cortex M4 exceeds that of the simulated results on the M1. And, Algorithm 5 performs better on the M1 compared to its simulation results on the Cortex M4. From Table 8, in the binary case for $N = 820$, we check that the average execution time of the proposed combined-variable-time algorithm is 47.8% of the variable-time algorithm. Also, the combined-constant-time algorithm is faster than the constant-time NTRU algorithm, where the average execution time of the proposed combined-constant-time algorithm is 73.4% of the constant-time NTRU algorithm.

Table 8 Simulation results for the binary case (in tick counts) with respect to NIST PQC NTRU parameters ($N = 508, 676, 820$ for NTRU-HPS, $N = 700$ for NTRU-HRSS) on ARM Cortex M4

N	Algorithm 3 (variable-time)	Algorithm 4 (combined-variable-time)	Algorithm 2 (NTRU constant-time)	Algorithm 5 (combined-constant-time)
508	367	170(x2.16)	1321	1029(x1.28)
676	651	298(x2.18)	2485	1801(x1.38)
700	715	322(x2.22)	2699	1940(x1.39)
820	918	439(x2.09)	3608	2649(x1.36)

Table 9 Simulation results for the ternary case (in tick counts) concerning NIST PQC NTRU parameters ($N = 508, 676, 820$ for NTRU-HPS, $N = 700$ for NTRU-HRSS) on ARM Cortex M4

N	Algorithm 3 (variable-time)	Algorithm 4 (combined-variable-time)	Algorithm 2 (NTRU constant-time)	Algorithm 5 (combined-constant-time)
508	843	507(x1.66)	1746	1461(x1.20)
676	1492	897(x1.66)	3238	2568(x1.26)
700	1621	962(x1.69)	3524	2748(x1.28)
820	2259	1258(x1.80)	4715	3678(x1.28)

6 Conclusion

In this study, we proposed two polynomial inversion algorithms, the combined-variable-time, and the combined-constant-time algorithms, defined in a polynomial ring, which can be applied in many areas. Both algorithms exhibited improved performances from the running time perspective compared to the original NTRU variable-time and constant-time algorithms, which were verified using the simulation results. Although variable-time algorithms carry a risk of information leakage, they can be employed within the environment to thwart side-channel attacks by leveraging variable execution times based on internal values, particularly in server systems. The proposed combined-constant-time algorithm can be optimized further by applying 2-bit two's complement representation to perform the multiplication of the ternary coefficients, which is suggested in [30]. In the extension of our work, we may consider combining k steps for $k > 2$ generally. However, it is worth noting that the derivation of elements in the higher dimensional matrix $\mathbf{M}_i$ will become more intricate. Moreover, when the number of coefficients is not a multiple of k, the final step may need to be divided into several cases, posing challenges in implementing a constant-time algorithm. Although combining additional steps may yield speed improvements, the resulting performance gains may be mitigated by the added complexity of the process. We expect the proposed combined-constant-time algorithm can substitute the existing NTRU constant-time algorithm because the gain from the reduced running time is much more significant, even after considering the increased cost from complexity and memory usage. Also, both proposed combined-variable-time and combined-constant-time algorithms will accelerate the previous algorithms with hardware implementation. We want to indicate that the simulation source code has been published in [31].

Acknowledgements We extend our sincere appreciation to Kwang-Sik Kim of the DGIST for his valuable contribution in simulating the algorithms on the ARM Cortex M4 platform.

Author Contributions E. Seo. and Y. Kim. wrote the main manuscript text, and J. No. discussed the main idea and revised the text. All three authors reviewed and devised the proposed algorithms.

Funding Open Access funding enabled and organized by DGIST (Daegu Gyeongbuk Institute of Science &amp; Technology). This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (RS-2024-00399401, Development of Quantum-Safe Infrastructure Migration and Quantum Security Verification Technologies).

Declarations

Conflicts of Interest The authors have no competing interests to declare that are relevant to the content of this article.

Competing interests The authors declare no competing interests.

References

1. Silverman, J.H.: Almost inverses and fast NTRU key creation,NTRU Tech Report, no. 014v1, Mar. 15, (1999)
2. Koc, C.K.: High-speed RSA implementation, RSA Laboratories, Bedford, MA, USA, Tech. Rep. TR201, (1994)
3. Chen, C., Danba, O., Hoffstein, J., Hulsing, A., Rijneveld, J., Schanck, J. M., Schwabe, P., Whyte, W., Zhang, Z., Saito, T., Yamakawa, T., Xagawa, K.: NTRU, Finalist of 3 round to the NIST PQC project, vol. 3, (2018)
4. Fouque, P.-A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Prest, T., Ricosset, T., Seiler, G., Whyte, W., Zhang, Z.: Falcon: Fast-fourier lattice-based compact signatures over ntru, Finalist of 3 round to the NIST PQC project, vol. 3, (2018)
5. Aragon, N., Barreto, P., Bettaieb, S., Bidoux, L., Blazy, O., Deneuville, J., Gaborit, P., Gueron, S., Guneysu, T., Melchor, C.A., Misoczki, R., Persichetti, E., Sendrier, N., Tillich, J., Zemor, G., Vasseur, V., Ghosh, S., Richter-Brokmann, J.: BIKE: Bit flipping key encapsulation, Alternate Candidate of 3 round to the NIST PQC project
6. Bernstein, D.J., Brumley, B.B., Chen, M., Chuengsatiansup, C., Lange, T., Marotzke, A., Peng, B., Tuveri, N., Vredendaal, C.V., Yang, B.: NTRU Prime, Alternate Candidate of 3 round to the NIST PQC project
7. Brent, R.P., Rung, H.T.: A systolic algorithm for integer GCD computation, 1985 IEEE 7th Symposium on Computer Arithmetic (ARITH), Urbana, IL, USA, pp. 118-125, (1985)
8. Bojanczyk, A.W., Brent, R.P.: A systolic algorithm for extended GCD computation. Computers & Mathematics with Application **14**(4), 233–238 (1987)
9. Goupil, A., Palicot, J.: Variation on Euclid's algorithm. IEEE Signal Processing Lett. **11**(5), 457–458 (2004)
10. Itoh, T., Tsujii, S.: A fast algorithm for computing multiplicative inverse $GF(s^m)$ using normal bases. Inf. Comput. **78**(3), 171–177 (1988)
11. Hu, J., Guo, W., Wei, J., Cheung, R.C.C.: Fast and generic inversion architectures over GF(2^m) using modified Itoh–Tsujii algorithms. IEEE Trans. on Circuits and Systems II:Express Briefs **62**(4), 367–371 (2015)
12. Vollala, S., Geetha, K., Ramasubramanian, N.: Efficient modular exponential algorithms compatible with hardware implementation of public key cryptography. Secur. Commun. Netw. **9**(16), 3105–3115 (2016)
13. Kim, Y.: Efficient algorithm for multi-nit Montgomery inverse using refined multiplicative inverse modular 2^K. IEEE Access **7**, 906–918 (2018)
14. O'Rourke, C., Sunar, B.: Achieving NTRU with Montgomery multiplication. IEEE Trans. Computers **52**(4), 440–448 (2003)
15. Ramasamy, R., Muniyandi, A.P.: Computing the modular inverse of a polynomial function over $GF(2^P)$ using bit wise operation. Int. J. Network Security **10**(2), 107–113 (2010)
16. Schroeppel, R., Orman, H., O'Malley, S., Spatscheck, O.: Fast Key Exchange with Elliptic Curve Systems, Lecture Notes in Computer Science, vol 963. Springer, Berlin, Heidelberg CRYPT0" 95, (1995)
17. Bernstein, D.J., Yang, B.: Fast constant-time gcd computation and modular inversion. IACR Transactions on Cryptographic Hardware and Embedded Systems **2019**(3), 340–398 (2019)
18. Venier, D., Cheung, R.C.C.: A Highly Parallel Constant-Time Almost-Inverse Algorithm, 2020 IEEE International Conference on Signal Processing, Communications and Computing(ICSPCC), Macau, China, pp. 1–6 (2020)
19. Zhao, N., Su, S.: An improvement and a new design of algorithms for seeking the inverse of an NTRU polynomial, 2011 Seventh Int. Conf. Computational Intelligence and Security, Sanya, China, 891-895 (2011)
20. Gaithuru, J. N., Salleh, M., Mohamad, I.: NTRU inverse polynomial algorithm based on the LU decomposition method of matrix inversion, 2017 IEEE Conf. Application, Inf. and Network Security(AINS), Miri, Malaysia, (1-6), (2017)
21. Karampetakis, N.P., Tzekis, P.: On the computation of the generalized inverse of a polynomial matrix. IMA J. Math. Control. Inf. **18**(1), 83–97 (2001)
22. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex Fourier series. Math. Comput. **19**(90), 297–301 (1965)
23. Bos, J.W.: Constant time modular inversion. J. Cryptographic Engineering **4**, 275–281 (2014)
24. Barenghi, A., Pelosi, G.: A comprehensive analysis of constant-time polynomial inversion for post-quantum cryptosystems, in Proc. the 17th ACM Int. Conf. Computing Frontiers, 269-276, (2020)
25. Nath, K., Sarkar, P.: Efficient inversion in (pseudo-)Mersenne prime order fields, IACR Cryptol. ePrint Arch. 985 (2018)
26. Hülsing, A., Rijneveld, J., Schanck, J., Schwabe, P.: High-speed key encapsulation from NTRU, Proc. Cryptographic Hardware and Embedded Systems – CHES 2017, LNCS, vol. 10529, Springer-Verlag, 232-252, (2017)
27. Richter-Brockmann, J., Chen, M., Ghosh, S., Güneysu, T.: Racing BIKE: Improved polynomial multiplication and inversion in hardware. IACR Transactions on Cryptographic Hardware and Embedded Systems **2022**(1), 557–588 (2022)
28. Drucker, N., Gueron, S., Kostic, D.: Fast polynomial inversion for post quantum QC-MDPC cryptography. In: Dolev, S., Kolesnikov, V., Lodha, S., Weiss, G. (eds.) CSCML 2020. LNCS, vol. 12161, pp. 110–127. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-49785-9_8
29. Itoh, T., Tsujii, S.: A fast algorithm for computing multiplicative inverses in GF(2m) using normal bases. Inf. Comput. **78**(3), 171–177 (1988). https://doi.org/10.1016/0890-5401(88)90024-7
30. Boothby, T.J., Bradshaw, R.W.: Bitslicing and the method of four Russians over larger finite fields (2009)
31. https://github.com/eyseo00/InversePolynomial/